

Itzik Ben-Gan

Podstawy języka T-SQL

Microsoft SQL Server 2016
i Azure SQL Database

Przekład: Leszek Biolik, Marek Włodarz

APN Promise, Warszawa 2016

Podstawy języka T-SQL: Microsoft SQL Server 2016 i Azure SQL Database

Authorized Polish translation of the English language edition entitled
T-SQL Fundamentals, Third Edition, by Itzik Ben-Gan
ISBN: 978-1-5093-0200-0

Copyright © 2016 by Itzik Ben-Gan

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by APN PROMISE SA Copyright © 2016

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego:
T-SQL Fundamentals, Third Edition, by Itzik Ben-Gan
ISBN: 978-1-5093-0200-0

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiegokolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Książka ta przedstawia poglądy i opinie autora. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiegokolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Microsoft oraz znaki towarowe wymienione na stronie <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> są zastrzeżonymi znakami towarowymi grupy Microsoft. Wszystkie inne znaki towarowe są własnością ich odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.
APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-305-2

Przekład: Leszek Biolik, Marek Włodarz
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Dla Dato

*Żyć w sercach tych, których zostawiliśmy,
to nie umrzeć*

– Thomas Campbell

Spis treści

<i>Wprowadzenie</i>	xiii
<i>Podziękowania</i>	xvii
1 Podstawy zapytań i programowania T-SQL	1
Podstawy teoretyczne	1
SQL	3
Teoria zbiorów	4
Logika predykatów	5
Model relacyjny	6
Typy systemów bazodanowych	13
Architektura SQL Server	15
Odmiany ABC produktu SQL Server	15
Instancje produktu SQL Server	18
Bazy danych	19
Schematy i obiekty	23
Tworzenie tabel i definiowanie integralności danych	24
Tworzenie tabel	25
Definiowanie integralności danych	27
Podsumowanie	31
2 Zapytania do pojedynczej tabeli	33
Elementy instrukcji <i>SELECT</i>	33
Klauzula <i>FROM</i>	36
Klauzula <i>WHERE</i>	38
Klauzula <i>GROUP BY</i>	39
Klauzula <i>HAVING</i>	43
Klauzula <i>SELECT</i>	44
Klauzula <i>ORDER BY</i>	49
Filtry <i>TOP</i> i <i>OFFSET-FETCH</i>	52
Szybki przegląd funkcji okna	56
Predykaty i operatory	58
Wyrażenia <i>CASE</i>	61
Znacznik <i>NULL</i>	64

Operacje jednoczesne – „all-at-once”	69
Stosowanie danych znakowych	71
Typy danych	71
Opcje sortowania (<i>collation</i>)	72
Operatory i funkcje	75
Predykat <i>LIKE</i>	84
Posługiwanie się danymi typu daty i czasu	87
Typy danych dotyczące daty i czasu	87
Literały	88
Rozdzielne stosowanie daty i czasu	92
Filtrowanie zakresów danych	94
Funkcje daty i godziny	95
Zapytania dotyczące metadanych	106
Widoki katalogowe	106
Informacyjne widoki schematu	107
Systemowe procedury składowane i funkcje	108
Podsumowanie	109
Ćwiczenia	110
Rozwiązania	115
3 Złączenia	121
Złączenia krzyżowe	122
Składnia ISO/ANSI SQL-92	122
Składnia ISO/ANSI SQL-89	123
Samo-złączenie krzyżowe (Self Cross Join)	123
Tworzenie tabel liczb	124
Złączenia wewnętrzne	126
Składnia ISO/ANSI SQL-92	126
Składnia ISO/ANSI SQL-89	127
Bezpieczeństwo złączenia wewnętrznego	128
Dodatkowe rodzaje złączeń	129
Złączenia złożone	129
Złączenie nierównościowe (Non-Equi Join)	130
Złączenia wielokrotne (multi-join)	132
Złączenia zewnętrzne	133
Podstawy złączeń zewnętrznych	133
Złączenia zewnętrzne – zagadnienia zaawansowane	136
Podsumowanie	144

Ćwiczenia	144
Ćwiczenie 1-2 (zaawansowane ćwiczenie opcjonalne)	145
Ćwiczenie 7 (zaawansowane ćwiczenie opcjonalne)	147
Ćwiczenie 8 (zaawansowane ćwiczenie opcjonalne)	148
Ćwiczenie 9 (zaawansowane ćwiczenie opcjonalne)	148
Rozwiązania	149
4 Podzapytania	155
Podzapytania niezależne	155
Przykłady skalarnych podzapytań niezależnych	156
Podzapytania niezależne o wielu wartościach	158
Podzapytania skorelowane	162
Predykat <i>EXISTS</i>	165
Zaawansowane aspekty podzapytań	167
Zwracanie poprzednich lub kolejnych wartości	167
Agregacje bieżące	168
Postępowanie w przypadku nieprawidłowo działających podzapytań	169
Podsumowanie	174
Ćwiczenia	175
Ćwiczenie 2 (zaawansowane ćwiczenie opcjonalne)	175
Ćwiczenie 7 (zaawansowane ćwiczenie opcjonalne)	177
Ćwiczenie 8 (zaawansowane ćwiczenie opcjonalne)	178
Ćwiczenie 10 (zaawansowane ćwiczenie opcjonalne)	178
Rozwiązania	179
5 Wyrażenia tablicowe	185
Tabele pochodne	185
Przypisywanie aliasów kolumn	187
Stosowanie argumentów	189
Zagnieżdżanie	190
Wielokrotne odwołania	191
Wspólne wyrażenia tablicowe	192
Przypisywanie aliasów kolumn w wyrażeniach CTE	192
Stosowanie argumentów w wyrażeniach CTE	193
Definiowanie wielu wyrażeń CTE	193
Wielokrotne odwołania w wyrażeniach CTE	194
Rekurencyjne wyrażenia CTE	195
Widoki	198

Widoki i klauzula <i>ORDER BY</i>	199
Opcje widoku	202
Wbudowane funkcje zwracające tabele	206
Operator <i>APPLY</i>	207
Podsumowanie	211
Ćwiczenia	211
Ćwiczenie 4 (zaawansowane ćwiczenie opcjonalne)	213
Ćwiczenie 5-2 (zaawansowane ćwiczenie opcjonalne)	215
Rozwiązania	216
6 Operatory zbiorowe	221
Operator <i>UNION</i>	222
Operator wielozbioru <i>UNION ALL</i>	222
Operator zbiorowy <i>UNION</i> z niejawną opcją <i>Distinct</i>	223
Operator <i>INTERSECT</i>	224
Operator <i>INTERSECT</i> (z ukrytą opcją <i>Distinct</i>)	225
Operator wielozbioru <i>INTERSECT ALL</i>	225
Operator <i>EXCEPT</i>	228
Operator zbiorowy <i>EXCEPT</i> (z opcją <i>Distinct</i>)	228
Operator wielozbioru <i>EXCEPT ALL</i>	229
Pierwszeństwo	230
Omijanie nieobsługiwanych faz logicznych	232
Podsumowanie	234
Ćwiczenia	234
Ćwiczenie 6 (zaawansowane ćwiczenie opcjonalne)	236
Rozwiązania	237
7 Zaawansowane zagadnienia tworzenia zapytań	241
Funkcje okna	241
Rankingowe funkcje okna	244
Offsetowe funkcje okna	248
Agregujące funkcje okna	251
Przestawianie danych	254
Przestawianie danych przy użyciu zapytania grupującego	256
Przestawianie danych przy użyciu operatora <i>PIVOT</i>	257
Odwrotne przestawianie danych	260
Odwrotne przestawianie danych przy użyciu operatora <i>APPLY</i>	261
Odwrotne przestawianie danych za pomocą operatora <i>UNPIVOT</i>	264

Zbiory grupujące.....	265
Klauzula pomocnicza <i>GROUPING SETS</i>	267
Klauzula pomocnicza <i>CUBE</i>	267
Klauzula pomocnicza <i>ROLLUP</i>	268
Funkcje <i>GROUPING</i> i <i>GROUPING_ID</i>	269
Podsumowanie	272
Ćwiczenia	272
Rozwiązania	277
8 Modyfikowanie danych	281
Wstawianie danych.....	281
Wyrażenie <i>INSERT VALUES</i>	281
Instrukcja <i>INSERT SELECT</i>	283
Instrukcja <i>INSERT EXEC</i>	284
Instrukcja <i>SELECT INTO</i>	285
Instrukcja <i>BULK INSERT</i>	286
Właściwość <i>Identity</i> i obiekt sekwencji.....	286
Usuwanie danych	296
Instrukcja <i>DELETE</i>	297
Instrukcja <i>TRUNCATE</i>	297
<i>DELETE</i> oparte na złączeniu	299
Aktualizowanie danych	300
Instrukcja <i>UPDATE</i>	301
<i>UPDATE</i> oparte na złączeniu	302
<i>UPDATE</i> z przypisaniem	305
Scalanie danych.....	306
Modyfikowanie danych przy użyciu wyrażeń tablicowych	311
Modyfikacje przy użyciu opcji <i>TOP</i> i <i>OFFSET-FETCH</i>	313
Klauzula <i>OUTPUT</i>	316
<i>INSERT</i> z klauzulą <i>OUTPUT</i>	316
<i>DELETE</i> z klauzulą <i>OUTPUT</i>	318
<i>UPDATE</i> z klauzulą <i>OUTPUT</i>	319
<i>MERGE</i> z klauzulą <i>OUTPUT</i>	320
Zagnieżdżone wyrażenia DML.....	321
Podsumowanie	323
Ćwiczenia	323
Rozwiązania	327

9	Tabele temporalne	333
	Tworzenie tabel	334
	Modyfikowanie danych	338
	Odpytywanie danych	341
	Podsumowanie	348
	Ćwiczenia	348
	Rozwiązania	351
10	Transakcje i współbieżność	357
	Transakcje	357
	Blokowanie	361
	Blokady	361
	Rozwiązywanie problemów związanych z blokadami	364
	Poziomy izolacji	372
	Poziom izolacji <i>READ UNCOMMITTED</i>	374
	Poziom izolacji <i>READ COMMITTED</i>	375
	Poziom izolacji <i>REPEATABLE READ</i>	377
	Poziom izolacji <i>SERIALIZABLE</i>	378
	Poziomy izolacji oparte na wersjonowaniu wierszy	380
	Podsumowanie poziomów izolacji	387
	Zakleszczenia	388
	Podsumowanie	391
	Ćwiczenia	391
11	Obiekty programowalne	403
	Zmienne	403
	Wsady	406
	Wsad jako jednostka analizy	406
	Wsady i zmienne	407
	Instrukcje, których nie można łączyć w tym samym wsadzie	408
	Wsad jako jednostka rozpoznawania	408
	Opcja <i>GO n</i>	409
	Elementy kontroli przepływu wykonania	410
	Element kontroli przepływu <i>IF ... ELSE</i>	410
	Element kontroli przepływu <i>WHILE</i>	411
	Kursory	413
	Tabele tymczasowe	417
	Lokalne tabele tymczasowe	417

Globalne tabele tymczasowe	419
Zmienne tablicowe	421
Typy tablicowe	422
Dynamiczny kod SQL	423
Polecenie <i>EXEC</i>	424
Procedura składowana <i>sp_executesql</i>	424
<i>PIVOT</i> w dynamicznym kodzie SQL	426
Procedury	427
Funkcje definiowane przez użytkownika	428
Procedury składowane	429
Wyzwalacze	432
Obsługa błędów	436
Podsumowania	440
A Rozpoczynamy	441
Rozpoczynamy pracę w Azure SQL Database	442
Instalowanie produktu SQL Server w wersji dla siedziby	442
Ćwiczenie 1. Uzyskanie produktu SQL Server	442
Ćwiczenie 2. Instalowanie silnika bazy danych	443
Pobieranie i instalowanie SQL Server Management Studio	448
Pobieranie kodu źródłowego i instalowanie przykładowej bazy danych	448
Posługiwanie się programem SQL Server Management Studio	451
Korzystanie z SQL Server Books Online	457
<i>Informacje o autorze</i>	460
<i>Indeks</i>	461

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 i ANSI. Poznamy teorię konstruowania zapytań i programowania w języku T-SQL oraz sposoby projektowania kodu T-SQL w celu uzyskiwania i modyfikowania danych, a także ogólny 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 zapoznavanie 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. Fragmenty uważane za bardziej zaawansowane są w tekście zaznaczone jako opcjonalne.

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 2016. 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 jako usługa chmurowa o nazwie Windows Azure SQL Database (w skrócie SQL Database). Przykłady kodu przytaczane w książce były testowane zarówno w lokalnych instalacjach SQL Server, jak i w Azure SQL Database. Powiązana z książką witryna sieci Web (<http://aka.ms/T-SQLFund3e/downloads>) udostępnia informacje dotyczące problemów zgodności pomiędzy tymi rozwiązaniami.

W celu usprawnienia procesu nauczania książka zawiera ćwiczenia, które pozwalają poznaną informację utrwalić w praktyce. Od czasu do czasu pojawiają się ćwiczenia opcjonalne, które są bardziej zaawansowane. Ćwiczenia te przeznaczone są dla

Czytelników, którzy dobrze poznali omawiany materiał i chcą sami sprawdzić swoje umiejętności, rozwiązując trudniejsze problemy. Ćwiczenia opcjonalne dla Czytelników zaawansowanych są odpowiednio oznaczone.

Dla kogo przeznaczona jest ta książka

Niniejsza książka skierowana jest dla 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 Transact-SQL.

Założenia

Największe korzyści książka ta przyniesie osobom, które mają już doświadczenie w pracy z systemami Windows i aplikacjami opartymi na tych systemach. Ponadto osoby te powinny znać podstawowe pojęcia dotyczące systemów zarządzania relacyjnymi bazami danych. Przydatne będzie też podstawowe doświadczenie w tworzeniu oprogramowania.

Kto nie powinien czytać tej książki

Nie każda książka nadaje się dla każdego czytelnika. W książce omówiono podstawy języka i głównie skierowana jest ona do osób w praktyce korzystających z języka T-SQL, które nie mają w tym wielkiego doświadczenia. Nie będzie zapewne zbyt interesująca dla doświadczonych praktyków, od lat posługujących się tym językiem. Tym niemniej, wielu czytelników poprzedniego wydania tej książki uważa, że pomimo doświadczeń zdobytych w kolejnych latach pracy książka ta nadal jest przydatna i uzupełnia brakującą wiedzę.

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 9 poruszane są różnorodne aspekty uzyskiwania i modyfikowania danych. Rozdział 10 zawiera omówienie współbieżności i transakcji. Na koniec rozdział 11 stanowi 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 i operatora *APPLY*.
- Rozdział 6 „Operatory zbiorowe” – omówienie operatorów *UNION*, *INTERSECT* i *EXCEPT*.
- Rozdział 7 „Zaawansowane zagadnienia tworzenia zapytań” – omówienie funkcji okien, operatorów *PIVOT* i *UNPIVOT* oraz praca z operatorami *GROUPING SETS*.
- 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 „Obiekty programowalne” – 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 *TSQLV4*, rozpoczęcie pisania kodu dla SQL Server oraz poznanie sposobów uzyskania pomocy dzięki dokumentacji SQL Server Books Online.

Wymagania systemowe

Dodatek „Rozpoczynamy” zawiera informacje, których wersji produktu SQL Server 2016 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 SQL Server Books Online w sekcji „Hardware and Software Requirements for Installing SQL Server 2016” pod adresem <https://msdn.microsoft.com/en-us/library/ms143506.aspx>. W Dodatku wyjaśniono również, jak korzystać z dokumentacji SQL Server Books Online.

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

Do uruchamiania przykładów kodu, zarówno w przypadku lokalnej instancji SQL Server 2016, jak i Azure SQL Database, konieczne jest zainstalowanie oprogramowania SQL Server Management Studio (SSMS). Oprogramowanie to można pobrać z witryny firmy Microsoft pod adresem <https://msdn.microsoft.com/en-us/library/mt238290.aspx>.

Instalowanie i korzystanie z kodu źródłowego

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 w zakładce Dodatkowe Informacje:

<http://www.książki.promise.pl/asp/produkt.aspx?pid=112055>

Dodatek „Rozpoczynamy” zawiera szczegółowe informacje na temat instalowania kodów źródłowych.

Podziękowania

Wiele osób przyczyniło się do powstania tej książki, czy to bezpośrednio, czy pośrednio i wszystkim im należą się serdeczne podziękowania i uznanie.

Dla Lilach, za zrozumienie i wsparcie wszystkich moich poczyniń oraz za wyrozumiałość dla niekończących się godzin spędzonych nad SQL.

Dla mojej mamy Mila oraz rodzeństwa Mickey i Ina, za stałe wsparcie i akceptowanie nieobecności, co teraz jest trudniejsze niż kiedykolwiek. Mamo, wszyscy wierzymy, że będzie dobrze, dzięki Twojej sile i determinacji. Tato, dzięki, że jesteś tak pomocny.

Dla recenzenta technicznego książki, Boba Beauchemina: jesteś częścią społeczności SQL Server od tak wielu lat, ale nadal jestem pod wrażeniem twojej wiedzy; byłem szczęśliwy, gdy zgodziłeś się współpracować przy tej książce.

Dla Steve'a Kassa, Dejana Sarka, Gianluca Hotza, i Herberta Alberta: Dzięki za wasze cenne rady podczas planowania i pisanie tej książki. Musiałem podjąć kilka trudnych decyzji, co uwzględnić, a czego nie dołączać do książki i wasze wskazówki były bardzo pomocne.

Dla SolidQ, mojej firmy od przeszło dziesięciu lat: to wielka satysfakcja pracować w tak wspólnie prowadzonej firmie. Pracownicy firmy to nie tylko koledzy – to partnerzy, przyjaciele i rodzina. Fernando G. Guerrero, Douglas McDowell, Herbert Albert, Dejan Sarka, Gianluca Hotz, Jeanne Reeves, Glenn McCain, Fritz Lechnitz, Eric Van Soldt, Joelle Budd, Jan Taylor, Marilyn Templeton, Berry Walker, Alberto Martin, Lorena Jimenez, Ron Talmage, Andy Kelly, Rushabh Mehta, Eladio Rincón, Erik Veerman, Jay Hackney, Richard Waymire, Carl Rabeler, Chris Randall, Johan Åhlén, Raoul Illyés, Peter Larsson, Peter Myers, Paul Turley – dziękuję Wam i wielu innym osobom.

Dla zespołu redakcyjnego produktu SQL Server Pro, czyli Megan Keller, Lavon Peters, Michele Crockett, Mike Otey i z pewnością jeszcze wiele innych osób; pisałem dla tego magazynu od ponad dekady i jestem wdzięczny, że mogłem podzielić się moją wiedzą z czytelnikami magazynu.

Dla osób z grupy MVP produktu SQL Server – Alejandro Mesa, Erland Sommarskog, Aaron Bertrand, Tibor Karaszi, Paul White i wielu innych oraz dla prowadzącego program MVP, Simona Tien; jest to wspaniały program i jestem wdzięczny, że mogłem w nim uczestniczyć. Niespotykany jest poziom umiejętności w tej grupie i zawsze cieszyłem się na nasze spotkania, zarówno by podzielić się pomysłami, jak i po prostu pogadać przy piwie. Jestem przekonany, że w dużej mierze dodanie przez Microsoft funkcjonalności T-SQL do produktu SQL Server zostało zainspirowane przez członków

MVP i całą społeczność produktu SQL Server. To wspaniale przekonać się, że ta synergia przekształca się w tak znaczące i ważne rezultaty.

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

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. Dla mnie T-SQL to coś więcej niż tylko język programowania – to sposób myślenia. Często wykladałem 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. Mogłbym skrótkowo 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 poświęcę 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 oparciu o produkcie zarządzania danymi – swoim systemie RDBMS, czyli Microsoft SQL Server.

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 pewien dialekt SQL, czego przykładem jest integracja CLR w SQL Server, pozwalająca obsłużyć zadania historycznie realizowane w języku T-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 branżowym 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 I (*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 zapytań oraz 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. 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 spójenie 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 pokazać 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 twórców 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 jest identyfikowany przez nazwę, a nie przez numer porządkowy.

Podobnie rozważmy zbiór krotek (w SQL nazywanych *wierszami*), który stanowi treść relacji; element 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. Dr 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. Codd’a, a następnie rozbudowany i opracowany przez Chrisa Date, Hugh’a Darwena i innych. Pierwsza wersja modelu relacyjnego została zaproponowana przez Codd’a 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 Codd’a 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 złączenie dwóch relacji).

UWAGA Model relacyjny rozróżnia pojęcie *relacji* i *zmiennej relacji*, ale dla uproszczenia nie będę ich rozróżniać – w obu przypadkach będę używać pojęcia *relacji*. Ponadto 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 odnosił się do tabeli jak do zbioru wierszy.



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 Itzik Ben-Gan urodził się 12 lutego 1971 roku i pracuje w dziale IT”. 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 1971 (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 – kapsułującej 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 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, a wartość *nieznany*.

Dla przykładu weźmy atrybut telefonu komórkowego 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 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 Codd'a 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, 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.

Innymi przykładami ograniczeń są *klucze kandydujące* (*candidate keys*), zwane 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 więc 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*.

* 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.

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

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 zależy 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 arkusza kalkulacyjnego narusza pierwszą postać normalną. Przykładem może być zdefiniowanie relacji *YearlySales* (sprzedaż roczna) za pomocą następujących atrybutów: *salesperson* (sprzedawca), *qty2013* (ilość w 2013), *qty2014* i *qty2015*. W tym przykładzie jednak tak naprawdę nie naruszamy pierwszej postaci normalnej; po prostu nakładamy ograniczenie – ograniczamy dane do trzech określonych lat: 2013, 2014 i 2015.

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

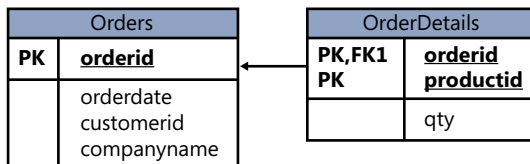
być w pełni funkcjonalnie zależny od całego klucza kandydującego. Inaczej mówiąc, atrybut niebędący częścią klucza nie może być w pełni funkcjonalnie zależny od części klucza kandydującego. Aby to trochę uprościć, jeśli potrzebujemy uzyskać wartość pewnego atrybutu niebędącego częścią klucza, trzeba 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 założmy, że zdefiniowaliśmy relację nazwaną *Orders* (zamówienia), która reprezentuje informacje o zamówieniach i jego pozycjach (rysunek 1-1). 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*.

Orders	
PK	<u>orderid</u>
PK	<u>productid</u>
	orderdate qty customerid companyname

RYСУNEK 1-1 Model danych przed zastosowaniem postaci 2NF

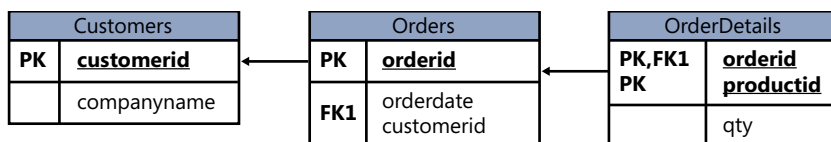
Relacja pokazana na rysunku 1-1 nie spełnia drugiej postaci normalnej, ponieważ istnieją atrybuty niebędące częścią klucza, które są zależne tylko od części klucza kandydującego (w tym przykładzie od klucza głównego). Na przykład, możemy znaleźć atrybut *orderdate* zamówienia, a także atrybuty *customerid* 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 1-2). 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*.



RYСУNEK 1-2 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 1-3) wraz z atrybutami *customerid* (klucz główny) i *companyname*. Następnie możemy usunąć atrybut *companyname* z relacji *Orders*.



RYСУNEK 1-3 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 Coddę, które obejmują złożone klucze główne i tymczasowe bazy danych, ale tematyka ta wykracza poza zakres książki.

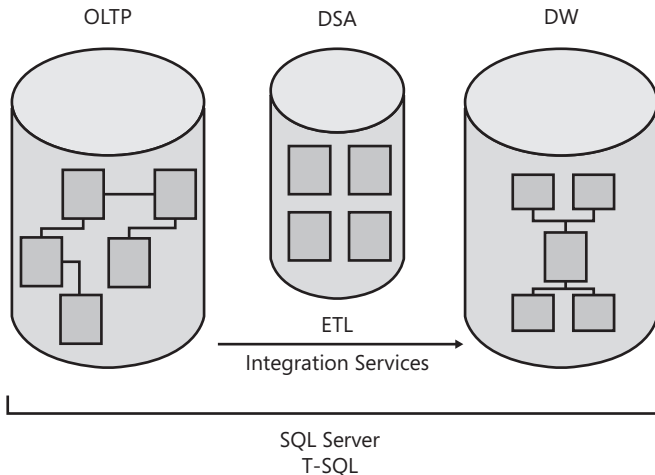


UWAGA Warto zauważyć, że SQL, a także T-SQL, pozwalają na naruszanie wszystkich form normalnych w rzeczywistych tabelach. To na twórcy modelu danych spoczywa odpowiedzialność za zaprojektowanie modelu znormalizowanego.

Typy systemów bazodanowych

Dwa główne typy systemów, czy też obciążeń, które wykorzystują SQL Server jako mechanizm bazodanowy oraz T-SQL do zarządzania i manipulowania danymi, to przetwarzanie transakcyjne w trybie online oraz hurtownie danych. Rysunek 1-4 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 1-4 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 systemie SQL Server 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* (*mini-hurtownia*, *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 systemu SQL Server były głównie ukierunkowane na środowiska OLTP, jednak ostatecznie system SQL Server zaczął również

obsługiwać systemy hurtowni danych i zapewniać odpowiednią analizę. Możemy implementować hurtownię danych jako bazę danych systemu SQL Server oraz 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). Do obsługi procesu ETL system SQL Server udostępnia narzędzie Microsoft SQL Server Integration Services (SSIS).

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 i jest używana jako obszar czyszczenia danych. Warstwa DSA nie jest dostępna dla użytkowników końcowych.

Architektura SQL Server

Niniejszy podrozdział jest wprowadzeniem do architektury systemu SQL Server, jego odmian i używanych jednostek – instancji SQL Server, baz danych, schematów i obiektów danych – oraz opisuje przeznaczenie wszystkich tych elementów.

Odmiany ABC produktu SQL Server

Przez wiele lat system SQL Server był dostępny tylko w jednej odmianie – w siedzibie (*on-premises*), czyli w wersji pudełkowej (*box*). Niedawno firma Microsoft zdecydowała się na udostępnienie kilku odmian, by klienci mogli wybrać taką, która im najbardziej odpowiada. Obecnie firma Microsoft oferuje trzy odmiany produktu SQL Server, które wewnętrznie są nazywane ABC: A dla Appliance, B dla Box oraz C dla Cloud.

Wersja Appliance

Pomysł związany z wyróżnieniem odmiany A wywodzi się z chęci udostępnienia kompletnego rozwiązania „pod klucz”, które obejmuje sprzęt, oprogramowanie i usługi. Szybkość działania jest osiągana dzięki spójnemu rozmieszczeniu komponentów z magazynem danych umieszczonym blisko jednostki przetwarzającej. Wersja ta jest utrzymywana lokalnie po stronie klienta. Firma Microsoft współpracuje z dostawcami sprzętu, takimi jak firmy Dell czy HP w celu oferowania takiego produktu. Ekspertki z firmy Microsoft i dostawcy sprzętu zapewniają klientowi wydajność, bezpieczeństwo i dostępność jego systemu.

Obecnie dostępnych jest kilka rozwiązań klasy Appliance, z których wyróżnić można Microsoft Analytics Platform System (APS), skoncentrowany na hurtowniach danych i przetwarzaniu wielkich ilości danych (*big data*). Rozwiązanie to obejmuje silnik hurtowni danych o nazwie Parallel Data Warehouse (PDW), wykorzystujące technologię masowego przetwarzania równoległego (*massively parallel processing* – MPP).

Zawiera ono również mechanizm HDInsight, stanowiący opracowaną przez firmę Microsoft dystrybucję silnika bazodanowego Hadoop. APS zawiera również technologię zapytań o nazwie PolyBase, która pozwala na wykorzystanie zapytań T-SQL wobec danych relacyjnych zawartych w PDW i nierelacyjnych danych z HDInsight.

Wersja Box

Odmiana Box produktu SQL Server, formalnie nazywana systemem SQL Server w siedzibie (*on-premises*), jest tradycyjnym sposobem 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 systemu SQL Server w siedzibie. W Dodatku znaleźć można informacje szczegółowe na temat uzyskania i instalowania wersji testowej SQL Server, a także na temat tworzenia przykładowej bazy danych.

Wersja Cloud

Przetwarzanie w chmurze polega na zapewnieniu zasobów obliczeniowych 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. Zazwyczaj 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 pudełkowa (zatem ten sam język T-SQL jest używany do zarządzania i manipulowania danymi).

W przypadku *chmury publicznej* usługi udostępniane są poprzez sieć i dostępne dla każdego (gotowego zapłacić). Firma Microsoft udostępnił 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. Początkowo możemy wybrać spośród wielu wstępnie skonfigurowanych maszyn wirtualnych, które już zawierają

zainstalowaną określoną wersję i wydanie SQL Server (silnik typu *box*). 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 wariantcie PaaS. Na potrzeby systemów OLTP oferowana jest usługa Azure SQL Database. Często jest ona skrótowo określana terminem SQL Database. Klient może mieć wiele baz danych na serwerze w chmurze (jest to oczywiście serwer „konceptualistyczny” – w istocie nie wiemy, czy baza jest utrzymywana przez konkretną maszynę, a przede wszystkim nie jest to w ogóle istotne), ale nie może przełączać się pomiędzy tymi bazami w ramach pojedynczej sesji.

Co interesujące, firma Microsoft wykorzystuje ten sam bazowy kod dla SQL Database i SQL Server. 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. Występujące różnice zostały opisane w dokumencie dostępnym pod adresem <https://azure.microsoft.com/en-us/documentation/articles/sql-database-transact-sql-information>. Warto też zauważyć, że tempo aktualizacji i wdrożeń nowych wersji SQL Database jest większe, niż w przypadku SQL Server w wersji pudełkowej. Oznacza to, że niektóre nowe funkcje T-SQL mogą być dostępne w SQL Database, zanim pojawią się w wersji SQL Server dla siedziby.

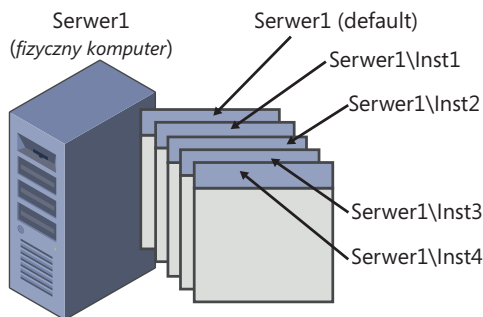
Microsoft udostępnia również ofertę typu PaaS dla systemów hurtowni danych, nazywaną Microsoft Azure SQL Data Warehouse (w skrócie SQL Data Warehouse). Usługa ta jest zasadniczo rozwiązaniem PDW/APS w chmurze. Firma Microsoft używa tego samego kodu bazowego dla wersji Appliance i usługi chmurowej. Do zarządzania i manipulowania danymi w APS i SQL Data Warehouse wykorzystywany jest język T-SQL, ale trzeba zauważyć, że nie jest to ta sama powłoka T-SQL, co w przypadku SQL Server i SQL Database.

Firma Microsoft udostępnia również inne chmurowe usługi danych, takie jak Data Lake dla rozwiązań klasy *big data*, Azure DocumentDB dla usług dokumentów NoSQL i inne.

Skomplikowane? Jeśli to kogoś pocieszy, nie jest osamotniony. Jak powiedziałem, firma Microsoft udostępnia przytłaczającą różnorodność technologii związanych z bazami danych. Osobliwe jest to, że jedynym wątkiem wspólnym dla większości z nich jest język T-SQL.

Instancje produktu SQL Server

W przypadku produktu pudełkowego *instancja* (wystąpienie) systemu SQL Server, jak ilustruje to rysunek 1-5, jest pełną, odrębną instalacją silnika bazy danych lub usługi SQL Server. 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.



RYСУNEK 1-5 Wiele instancji SQL Server na tym samym komputerze

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 dwie instancje SQL Server zainstalowane na komputerze *Serwer1*. Jedna z tych instancji została zainstalowana jako domyślna, a druga jako instancja nazwana – *Inst1*. 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 systemu SQL Server na tym samym komputerze, jednak wspomnę tylko o niektórych z nich. Jedną z nich 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 systemu SQL Server, które imitują środowisko

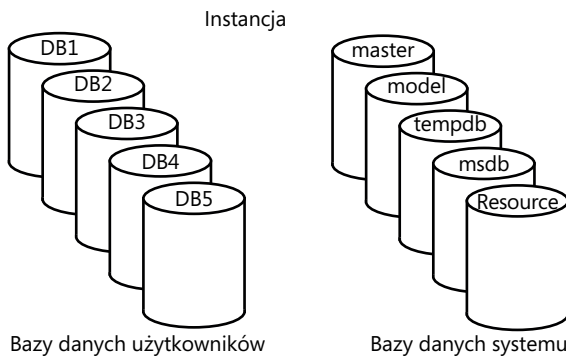
produkcyjne pod względem wersji, wydania i zainstalowanych pakietów serwisowych produktu SQL Server. Jeśli w organizacji jest wiele środowisk użytkowników, dział techniczny potrzebuje wielu instalacji SQL Server. Zamiast utrzymywania wielu komputerów, na których instalowane są różne instalacje systemu SQL Server i które muszą być obsługiwane oddzielnie, można mieć jeden komputer z zainstalowanymi wieloma instancjami. Oczywiście ten sam efekt można uzyskać przy użyciu maszyn wirtualnych, ale nadal mamy konieczność utrzymywania większej liczby samych systemów operacyjnych oraz pewnym dodatkowym obciążeniem narzucanym przez hiperwizor.

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 systemu 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 utrzymywanych jest 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 1-6). Podczas instalowania wersji SQL Server dla siedziby 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.



RYСУNEK 1-6 Przykład wielu baz danych w instancji systemu 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.
- **Resource** Baza danych *Resource* jest ukryta, przeznaczona tylko do odczytu i przechowuje definicje wszystkich obiektów systemu. Podczas wykonywania zapytań dotyczących obiektów systemowych w bazie danych wydaje się, że znajdują się one w schemacie *sys* lokalnej bazy danych, jednak w rzeczywistości umieszczone są w bazie danych *Resource*.
- **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 stworzymy, 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.
- **tempdb** Baza danych *tempdb* to miejsce przechowywania przez system SQL Server danych tymczasowych, takich jak tabele robocze, obszary sortowania, informacje dotyczące wersji wiersza itp. System SQL Server pozwala tworzyć tabele tymczasowe do naszego własnego użytku, a ich fizyczna lokalizacja to właśnie baza danych *tempdb*. Trzeba pamiętać, że ta baza danych jest usuwana i ponownie tworzona jako kopia bazy danych *model*, ilekroć ponownie uruchamiana jest instancja systemu SQL Server.
- **msdb** Baza danych *msdb* to miejsce, w którym swoje dane przechowuje usługa SQL Server Agent. Usługa SQL Server Agent odpowiada za automatyzację obejmującą takie jednostki, jak zadania, harmonogramy i alerty. Usługa SQL Server Agent odpowiada także za replikację. Baza danych *msdb* przechowuje również informacje dotyczące innych funkcji systemu SQL Server, takich jak Database Mail, Service Broker, kopie zapasowe i inne.

W instalacji systemu SQL Server w wersji dla siedziby możemy bezpośrednio łączyć się z systemowymi bazami danych *master*, *model*, *tempdb* i *msdb*. W produkcie Azure SQL Database możemy bezpośrednio łączyć się tylko z bazą danych *master*. Jeśli tworzymy tymczasowe tabele lub deklarujemy tymczasowe zmienne tablicowe (więcej informacji na ten temat znaleźć można w rozdziale 11 „Obiekty programowalne”), są one tworzone w bazie danych *tempdb*, ale nie możemy bezpośrednio łączyć się z bazą *tempdb* i wprost w niej tworzyć obiektów.

Wewnątrz instancji możemy tworzyć wiele baz danych użytkownika (aż do 32 767). Baza danych użytkownika przechowuje obiekty i dane przeznaczone dla aplikacji.

Na poziomie bazy danych można zdefiniować właściwość nazwaną *collation*, która będzie określać obsługę języka, wrażliwość na wielkość znaków i kolejność sortowania znaków w tej bazie danych. Jeśli podczas tworzenia bazy danych nie zostanie wyspecyfikowana właściwość *collation*, nowa baza danych posługuje się domyślną właściwością *collation* instancji (wybraną podczas instalacji).

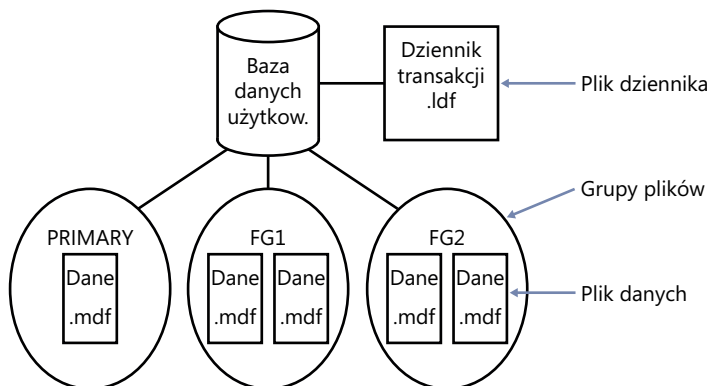
Aby uruchomić kod języka T-SQL w konkretnej bazie danych, aplikacja kliencka musi połączyć się z instancją SQL Server i znaleźć się w kontekście użycia właściwej bazy danych. Aplikacja nadal może uzyskiwać dostęp do obiektów z innych baz danych, dołączając prefiks nazwy bazy danych.

Z punktu widzenia zabezpieczeń, aby można było połączyć się z instancją SQL Server, administrator bazy danych (DBA) musi dla nas utworzyć login. W przypadku instancji SQL Server w wersji dla siedziby login może być powiązany z naszymi poświadczeniami w systemie Windows i w takim przypadku jest to *login uwierzytelniony w systemie Windows*. Przy użyciu uwierzytelnienia systemu Windows nie musimy przedstawiać loginu i hasła podczas łączenia się z systemem SQL Server, ponieważ poświadczenia te zostały już dostarczone podczas logowania się w systemie Windows. W obu produktach, SQL Server w siedzibie i Azure SQL Database, login może być niezależny od poświadczeń dla systemu Windows i w takim przypadku nazywany jest to *login uwierzytelniony w systemie SQL Server*. Podczas łączenia się z systemem SQL Server przy użyciu uwierzytelnienia SQL Server trzeba przedstawić zarówno nazwę logowania, jak i hasło.

DBA musi przypisać nasz login do bazy danych użytkownika w każdej bazie danych, do której chcemy uzyskać dostęp. Baza danych użytkownika jest jednostką, której przydzielane są uprawnienia do obiektów bazy danych.

SQL Server obsługuje funkcję nazwaną *contained databases* (zawarte w sobie, izolowane bazy danych), która zrywa związek pomiędzy użytkownikiem bazy danych a loginem na poziomie serwera. Użytkownik całkowicie „zawiera się” wewnątrz określonej bazy danych i nie jest powiązany z loginem na poziomie serwera. Podczas tworzenia użytkownika, administrator bazy danych dostarcza również hasło. Podczas łączenia się z systemem SQL Server użytkownik musi wyspecyfikować bazę danych, z którą się łączy, a także nazwę użytkownika i hasło – później nie może przełączać się do innych baz danych.

Do tej pory omawiane były logiczne aspekty baz danych. Przy korzystaniu z produktu SQL Database zajmujemy się tylko tą logiczną warstwą. Nie interesuje nas warstwa fizyczna danych bazy i plików dziennika, baza *tempdb* itp. Jeśli jednak korzystamy z wersji SQL Server dla siedziby, jesteśmy odpowiedzialni także za warstwę fizyczną. Na rysunku 1-7 przedstawiono schemat fizycznego układu bazy danych.



RYSUNEK 1-7 Układ bazy danych

Baza danych zbudowana jest z plików danych i plików dzienników transakcji oraz opcjonalnych plików kontrolnych przechowujących dane zoptymalizowane pamięciowo (jako część funkcjonalności o nazwie In-Memory OLTP, omówionej skrótowo w dalszej części rozdziału). Przy tworzeniu bazy danych definiujemy różne właściwości każdego pliku, a w tym nazwę pliku, lokalizację, początkowy rozmiar, rozmiar maksymalny i wielkość automatycznego przyrostu. Każda baza danych musi mieć co najmniej jeden plik danych i co najmniej jeden plik dziennika (ustawienie domyślne w SQL Server). Pliki danych przechowują dane o obiektach, a pliki dzienników przechowują informacje, które potrzebne są systemowi SQL Server do utrzymywania transakcji.

Chociaż system SQL Server potrafi równolegle zapisywać wiele plików danych, pliki dziennika zapisywane są w trybie sekwencyjnym – w danym momencie może być zapisywany tylko jeden plik dziennika. Z tego względu, inaczej niż w przypadku plików danych, posiadanie wielu plików dzienników nie jest korzystne z punktu widzenia wydajności. Nowe pliki dzienników dodajemy, jeśli zajęty zostaje dysk, na którym przechowywany jest plik dziennika.

Pliki danych są uporządkowane logicznie w *grupie plików* (*filegroup*). Grupa plików jest miejscem docelowym tworzenia obiektów, takich jak tabela czy indeks. Dane obiektu znajdują się w plikach należących do docelowej grupy plików. Grupy plików to mechanizm kontrolowania fizycznej lokalizacji obiektów. Baza danych musi posiadać co najmniej jedną grupę plików nazwaną *PRIMARY*, a opcjonalnie może zawierać także inne grupy plików. Grupa plików *PRIMARY* zawiera podstawowy plik danych (o rozszerzeniu *mdf*) bazy danych oraz katalog systemowy bazy danych. W grupie *PRIMARY* opcjonalnie możemy dodawać pomocnicze pliki danych (pliki o rozszerzeniu *ndf*). Grupy plików użytkownika zawierają tylko pomocnicze pliki danych. Możemy określić, która grupa plików zostanie oznaczona jako domyślna. Obiekty tworzone są w domyślnej grupie plików, jeśli polecenie tworzenia obiektu nie specyfikuje wprost innej, docelowej grupy plików.

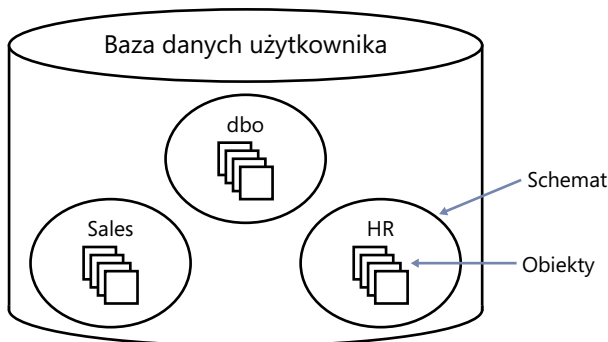
Silnik bazodanowy SQL Server zawiera dodatkowy mechanizm o nazwie In-Memory OLTP. Funkcjonalność ta pozwala zintegrować z bazą danych obiekty zoptymalizowane pamięciowo, takie jak zoptymalizowane tabele lub natywnie skompilowane procedury. W tym celu konieczne jest utworzenie w bazie danych grupy plików oznakowanej jako zawierającej dane zoptymalizowane pamięciowo, a następnie zdefiniować w tej grupie przynajmniej jedną ścieżkę do folderu. SQL Server w tym folderze będzie przechowywał pliki punktów kontrolnych dla danych zoptymalizowanych pamięciowo. Pliki te są używane do odtwarzania danych za każdym razem, gdy instancja SQL Server jest uruchamiana ponownie.

Rozszerzenia nazw plików MDF, LDF i NDF

Rozszerzenia nazw plików bazy danych MDF i LDF są dość zrozumiałe. Rozszerzenie MDF oznacza Master Data File (nie należy mylić z wzorcową bazą danych – chodzi o główny plik danych), a rozszerzenie LDF oznacza Log Data File (plik danych dziennika). Zgodnie z pewną anegdotą, podczas dyskusji dotyczącej pomocniczych plików danych jeden z projektantów zasugerował żartobliwie, by użyć rozszerzenia NDF, które oznacza „Not Master Data File” – i tak już pozostało.

Schematy i obiekty

Wcześniejsze stwierdzenie, że baza danych jest kontenerem obiektów, było pewnym uproszczeniem. Jak pokazuje rysunek 1-8, baza danych zawiera *schematy*, a dopiero schematy zawierają obiekty. Schemat możemy traktować jak kontener obiektów, takich jak tabele, widoki, procedury składowane itd.



RYСУNEK 1-8 Baza danych, schematy i obiekty bazy danych

Uprawnienia można kontrolować na poziomie schematu. Na przykład możemy użytkownikowi przydzielić uprawnienie SELECT do schematu, co pozwoli użytkownikowi

wykonywać zapytania do wszystkich obiektów tego schematu. Tak więc zabezpieczenia są jednym z czynników określania, jaki powinien być porządek obiektów w schemacie.

Schemat jest również przestrzenią nazw – jest używany jako prefiks nazwy obiektu. Załóżmy dla przykładu, że mamy tabelę nazwaną *Orders* w schemacie nazwanym *Sales*. Nazwa obiektu kwalifikowana schematem (nazywana również *dwuczęściową nazwą obiektu*) to *Sales.Orders*. Jeśli odwołując się do obiektu pominiemy nazwę schematu, system SQL Server zastosuje proces rozpoznawania nazwy schematu – sprawdzenie, czy obiekt istnieje w domyślnym schemacie użytkownika, a jeśli nie, czy istnieje w schemacie *dbo*. Firma Microsoft zaleca, by podczas odwoływania się do obiektów w kodzie zawsze używać dwuczęściowych nazw obiektów. Jeśli nazwa nie jest wyspecyfikowana wprost, powstają pewne względnie niewielkie dodatkowe koszty związane z rozpoznawaniem nazwy obiektu. Wprawdzie koszty te nie są duże, jednak dlaczego w ogóle mielibyśmy je ponosić? Ponadto, jeśli w różnych schematach istnieją obiekty o tych samych nazwach, możemy ostatecznie uzyskać nie ten obiekt, który jest potrzebny.

Tworzenie tabel i definiowanie integralności danych

W tym podrozdziale omówię podstawy związane z tworzeniem tabel i definiowaniem integralności danych przy użyciu poleceń języka T-SQL. Przytaczane w książce przykłady kodu bez obaw możemy uruchamiać w naszym środowisku.



INFORMACJE DODATKOWE Informacje o uruchamianiu kodu w systemie SQL Server można znaleźć w Dodatku.

Jak wspomniałem wcześniej, książka skupia się raczej na języku DML, a nie na DDL. Ważne jednak pozostaje zrozumienie sposobów tworzenia tabel i definiowania integralności danych. Nie będę zajmować się tu bardzo szczegółowymi tematami, a jedynie przedstawię skrótowy opis kwestii zasadniczych.

Zanim przyjrzymy się kodowi tworzenia tabeli, zapamiętajmy, że tabele znajdują się wewnątrz schematów, a schematy wewnątrz baz danych. W przykładach w książce używana jest przykładowa baza danych *TSQV4* oraz schemat nazwany *dbo*.



INFORMACJE DODATKOWE Szczegółowe wskazówki na temat tworzenia przykładowej bazy danych zamieszczone są w Dodatku.

Przytoczone przykłady korzystają ze schematu nazwanego *dbo*, który jest automatycznie tworzony w każdej bazie danych i jest również używany jako schemat domyślny dla użytkowników, którzy nie zostali jawnie powiązani z innym schematem.

Tworzenie tabel

Poniższy kod tworzy tabelę nazwaną *Employees* w schemacie *dbo* w bazie danych *TSQLV4*.

```
USE TSQLV4;

DROP TABLE IF EXISTS dbo.Employees;

CREATE TABLE dbo.Employees
(
    Empid      INT          NOT NULL,
    firstname  VARCHAR(30)  NOT NULL,
    lastname   VARCHAR(30)  NOT NULL,
    hiredate   DATE         NOT NULL,
    mgrid      INT          NULL,
    ssn        VARCHAR(20)  NOT NULL,
    salary     MONEY        NOT NULL
);
```

Polecenie *USE* ustawia kontekst bieżącej bazy danych definiując, że jest to baza *TSQLV4*. Ważne jest umieszczanie polecenia *USE* w skryptach, które tworzą obiekty, by zapewnić, że SQL Server utworzy obiekty w określonej bazie danych. W przypadku implementacji systemu SQL Server w siedzibie instrukcja *USE* może tak naprawdę zmienić kontekst bazy danych z jednej na inną. W produkcji SQL Database nie możemy przełączać się pomiędzy różnymi bazami danych, ale instrukcja *USE* będzie działać prawidłowo, o ile jesteśmy już połączeni z docelową bazą danych. Tak więc nawet w przypadku SQL Database zalecam umieszczanie polecenia *USE*, by zagwarantować, że podczas tworzenia obiektów jesteśmy połączeni z właściwą bazą danych.

Wyrażenie *DROP IF EXISTS* powoduje usunięcie tabeli, o ile już istnieje. Składnia ta została wprowadzona w wersji SQL Server 2016. Przy korzystaniu z wcześniejszych wersji SQL Server należy zamiast niego użyć następującego wyrażenia:

```
IF OBJECT_ID(N'dbo.Employees', 'U') IS NOT NULL DROP TABLE dbo.Employees;
```

Instrukcja *IF* wywołuje funkcję *OBJECT_ID*, by sprawdzić, czy tabela *Employees* już istnieje w bieżącej bazie danych. Funkcja *OBJECT_ID* jako dane wejściowe akceptuje nazwę i typ obiektu. Typ *'U'* reprezentuje tabelę użytkownika. Funkcja ta zwraca identyfikator obiektu wewnętrznego, jeśli istnieje obiekt o określonej nazwie i typie, w przeciwnym razie zwraca wartość *NULL*. Jeśli funkcja zwraca wartość *NULL*, wiemy, że obiekt nie istnieje. W naszym przypadku kod usuwa tabelę, jeśli już istniała, a następnie tworzy nową. Oczywiście możemy wybrać inne działanie; na przykład, jeśli obiekt już istnieje, możemy po prostu go nie tworzyć.

Instrukcja `CREATE TABLE` jest odpowiedzialna za definiowanie tego, co poprzednio nazywaliśmy nagłówkiem relacji. W tym miejscu specyfikujemy nazwę tabeli oraz definicje atrybutów (w nawiasach), czyli definicje kolumn.

Zwróćmy uwagę na używanie dwuczęściowej nazwy tabeli `dbo.Employees`, zgodnie z wcześniejszymi zaleceniami. Jeśli pominiemy nazwę schematu, system SQL Server przy uruchamianiu kodu założy, że z bazą danych powiązany jest schemat domyślny.

Dla każdego atrybutu specyfikujemy nazwę atrybutu, typ danych i to, czy może przyjmować wartość `NULL` (cecha nazywana *nullability*).

W tabeli `Employees` atrybuty `empid` (identyfikator pracownika) i `mgrid` (identyfikator menedżera) są zdefiniowane jako typ danych `INT` (czterobajtowa liczba całkowita); atrybuty `firstname` (imię), `lastname` (nazwisko) i `ssn` (numer ubezpieczenia) zostały zdefiniowane jako `VARCHAR` (ciąg znaków o zmiennej długości z wyspecyfikowaną maksymalną liczbą obsługiwanych znaków); atrybut `hiredate` (data zatrudnienia) został zdefiniowany jako typ `DATE` (data), a atrybut `salary` (pensja) jako `MONEY`.

Jeśli nie określimy wprost, czy kolumna dopuszcza bądź nie dopuszcza stosowanie znaczników `NULL`, system SQL Server musi oprzeć się na ustawieniach domyślnych. Standard SQL wymaga, by w przypadku braku tego określenia przyjmować możliwość stosowania wartości `NULL` (dopuszczenie znaczników `NULL`), jednak SQL Server zawiera ustawienie, które pozwala zmienić to działanie. Usilnie zalecam, by być precyzyjnym i nie polegać na ustawieniach domyślnych. Ponadto równie mocno zalecam definiowanie kolumn jako `NOT NULL`, chyba że istnieją dobre powody, dla których trzeba obsługiwać znaczniki `NULL`. Jeśli kolumna nie powinna dopuszczać wartości `NULL`, ale nie wymusiliśmy tego za pomocą ograniczenia `NOT NULL`, można się założyć, że znaczniki `NULL` będą się pojawiać. W tabeli `Employees` wszystkie kolumny zostały zdefiniowane jako `NOT NULL`, za wyjątkiem kolumny `mgrid`. Wartość `NULL` atrybutu `mgrid` reprezentuje fakt, że pracownik nie ma menedżera, jak w przypadku dyrektora organizacji.

Styl kodowania

Trzeba zdawać sobie sprawę z kilku ogólnych uwag związanych ze stylem kodowania, używaniem białych znaków (spacji, tabulatorów, złamań wiersza itp.) oraz średników. Osobiście nie znam żadnego formalnie obowiązującego stylu kodowania i nie wydaje mi się, aby takowy kiedykolwiek powstał (w przeszłości bądź przyszłości). Moją radą jest stosowanie takiego stylu, który będzie wygodny w użyciu dla nas samych i naszych współpracowników. Ostatecznie największe znaczenie ma spójność, czytelność i łatwość utrzymywania kodu. Starałem się odzwierciedlić te aspekty w kodzie przytaczanym w książce.

Język T-SQL pozwala używać białych znaków w kodzie w niemal dowolny sposób. Znaki te poprawiają czytelność. Kod z poprzedniego podrozdziału mógłbym napisać w jednym wierszu. Nie byłoby to jednak tak czytelne, jak w przypadku podzielenia go na wiele wierszy, w których dodatkowo użyłem wcięć tekstu.

Praktyka kończenia poleceń średnikiem jest powszechna, a w istocie jest to wymóg standardu i jest stosowana w kilku innych platformach bazodanowych. SQL Server wymaga użycia średnika jedynie w szczególnych przypadkach, jednak wstawienie średnika tam, gdzie nie jest on wymagany, nie powoduje problemów. Zalecam więc, by przyswoić sobie zwyczaj kończenia wszystkich poleceń średnikiem. Średnik nie tylko poprawia czytelność, ale w niektórych sytuacjach może nas ustrzec przed kłopotami (w sytuacji, kiedy średnik jest wymagany, a nie został wyspecyfikowany, komunikaty o błędach generowane przez system SQL Server nie zawsze są wystarczająco pomocne).

UWAGA Dokumentacja systemu SQL Server wskazuje, że niekończenie poleceń T-SQL średnikiem jest funkcjonalnością przestarzałą. Oznacza to, że w przyszłych wersjach produktu znajdzie się funkcja wymuszania używania średnika. Jest to jeszcze jeden powód, dla którego warto przyzwyczaić się do średników kończących polecenia, nawet jeśli aktualnie nie są wymagane.



Definiowanie integralności danych

Jak już wspomniałem wcześniej, jedną z wielkich zalet modelu relacyjnego jest to, że jego częścią jest mechanizm zapewniający integralność danych. Integralność danych wymuszana jako część modelu, a mianowicie jako część definicji tabeli, jest nazywana deklaratywną integralnością danych. Integralność danych wymuszana za pomocą kodu, jak na przykład przy użyciu procedur składowanych czy wyzwalaczy, jest nazywana proceduralną integralnością danych.

Typ danych, możliwość wyboru obsługi znaczników *NULL* dla atrybutów, a nawet sam model danych są przykładami ograniczeń deklaratywnej integralności danych. W tym podrozdziale opiszemy inne przykłady deklaratywnych ograniczeń: klucza głównego, unikatowości, klucza obcego, sprawdzania (check) i wartości domyślnych. Tego typu ograniczenia możemy definiować podczas tworzenia tabeli, jako część instrukcji *CREATE TABLE*, albo narzucić je dla już utworzonych tabel przy użyciu instrukcji *ALTER TABLE*. Wszystkie typy ograniczeń, za wyjątkiem domyślnych, mogą być definiowane jako *ograniczenia złożone* – to znaczy bazujące na więcej niż jednym atrybucie.

Ograniczenia klucza głównego

Ograniczenie klucza głównego wymusza unikatowość wierszy, a oprócz tego zabrania stosowania znaczników *NULL* w ograniczanych atrybutach. Każdy unikatowy zbiór wartości w atrybutach ograniczeń może w tabeli wystąpić tylko raz – inaczej mówiąc, tylko w jednym wierszu. Próba zdefiniowania ograniczenia klucza głównego dla kolumny, która dopuszcza znaczniki *NULL*, zostanie odrzucona przez system RDBMS. Każda tabela może mieć tylko jeden klucz główny.

Poniżej zamieszczono przykład definiowania ograniczenia klucza głównego dla atrybutu *empid* w utworzonej wcześniej tabeli *Employees*.

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT PK_Employees
  PRIMARY KEY(empid);
```

Dzięki ustanowieniu tego klucza głównego mamy pewność, że wszystkie wartości *empid* będą unikatowe i znane. Próba wstawienia lub zaktualizowania wiersza w taki sposób, że naruszone zostaje ograniczenie, zostanie odrzucona przez system RDBMS i spowoduje wygenerowanie komunikatu o błędzie.

W celu wymuszenia unikatowości logicznego klucza głównego system SQL Server utworzy w tle unikatowy indeks. Jest on mechanizmem fizycznym używanym przez SQL Server do wymuszania jednoznaczności. Indeksy (niekoniecznie unikatowe) są również używane do przyspieszania zapytań poprzez eliminowanie niepotrzebnego przeszukiwania całej tabeli (działanie podobne do indeksu w książce).

Ograniczenia unikatowości

Ograniczenia *UNIQUE* wymuszają unikatowość wierszy, co pozwala zaimplementować w naszej bazie danych koncepcję alternatywnych kluczy modelu relacyjnego. Inaczej niż w przypadku kluczy głównych, wewnątrz tej samej tabeli możemy definiować wiele ograniczeń unikatowości. Ponadto ograniczenie nie jest limitowane tylko do kolumn zdefiniowanych jako *NOT NULL*. Poniższy kod definiuje ograniczenie Unique dla kolumny *ssn* w tabeli *Employees*.

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT UNQ_Employees_ssn
  UNIQUE(ssn);
```

Podobnie jak w przypadku ograniczenia klucza głównego, system SQL Server w tle utworzy unikatowy indeks jako mechanizm fizyczny wymuszania logicznego ograniczenia unikatowości.

Zgodnie ze standardem SQL zakłada się, że kolumna z ograniczeniem unikatowości dopuszcza stosowanie wielu znaczników *NULL* (tak jakby dwa znaczniki *NULL* różniły się od siebie). Jednakże implementacja systemu SQL Server odrzuca powielone znaczniki *NULL* (tak jakby dwa znaczniki *NULL* były identyczne). Jeśli chcemy emulować działanie ograniczenia unikatowości zgodne ze standardem w SQL Server,

możemy użyć unikatowego filtrowanego indeksu, który uwzględni (odfiltruje) tylko wartości nie-*NULL*. Dla przykładu założmy, że kolumna *ssn* zezwala na *NULL* i chcemy utworzyć taki indeks zamiast ograniczenia unikatowości. Możemy w tym celu użyć następującego kodu:

```
CREATE UNIQUE INDEX idx_ssn_notnull
ON dbo.Employees(ssn)
WHERE ssn IS NOT NULL;
```

Indeks jest zdefiniowany jako unikatowy, zaś filtr wyklucza z niego znaczniki *NULL*, zatem duplikujące się znaczniki *NULL* będą dozwolone, ale wartości faktyczne (nie-*NULL*) nie mogą być powielane.

Ograniczenia klucza obcego

Klucz obcy wymusza integralność referencyjną. Ograniczenie to jest zdefiniowane dla jednego lub wielu atrybutów za pomocą konstrukcji nazywanej tabelą odwołującą się (*referencing*) i wskazuje atrybuty klucza kandydującego (klucz główny lub ograniczenie unikatowe) w czymś, co nazywane jest tabelą odwołania (*referenced*). Zwróćmy uwagę, że obie tabele mogą być jedną i tą samą tabelą. Celem klucza obcego jest ograniczenie wartości w kolumnach klucza obcego do tych, które już istnieją w kolumnach odwołania.

Poniższy kod tworzy tabelę nazwaną *Orders* wraz z kluczem głównym zdefiniowanym w oparciu o kolumnę *orderid*.

```
DROP TABLE IF EXISTS dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid    INT           NOT NULL,
    empid      INT           NOT NULL,
    custid     VARCHAR(10)  NOT NULL,
    orderts    DATETIME2     NOT NULL,
    qty        INT           NOT NULL,
    CONSTRAINT PK_Orders
        PRIMARY KEY(orderid)
);
```

Założmy, że chcemy wymusić regułę integralności, która ogranicza wartości obsługiwane przez kolumnę *empid* tabeli *Orders* do wartości, które istnieją w kolumnie *empid* w tabeli *Employees*. Zadanie to możemy osiągnąć, definiując ograniczenie klucza obcego dla kolumny *empid* tabeli *Orders*, wskazując kolumnę *empid* tabeli *Employees*, jak w poniższym przykładzie:

```
ALTER TABLE dbo.Orders
ADD CONSTRAINT FK_Orders_Employees
FOREIGN KEY(empid)
REFERENCES dbo.Employees(empid);
```

Analogicznie, jeśli chcemy ograniczyć wartości obsługiwane przez kolumnę *mgrid* w tabeli *Employees* do wartości, które istnieją w kolumnie *empid* tej samej tabeli, można to zrealizować poprzez dodanie poniższego klucza obcego:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT FK_Employees_Employees
  FOREIGN KEY(mgrid)
  REFERENCES dbo.Employees(empid);
```

Zwróćmy uwagę, że znaczniki *NULL* są dopuszczone w kolumnach klucza obcego (*mgrid* w ostatnim przykładzie), nawet jeśli nie istnieją znaczniki *NULL* w kolumnach klucza kandydującego, do których następuje odwołanie.

Poprzednie dwa przykłady są podstawowymi definicjami kluczy obcych, które wymuszają referencyjne działanie nazwane *no action* (brak akcji). Brak działania oznacza, że próby usunięcia wierszy z tabeli odwołania lub zaktualizowania atrybutów klucza kandydującego odwołania zostaną odrzucone, jeśli powiązane wiersze istnieją w tabeli odwołującej się. Na przykład, jeśli spróbujemy usunąć wiersz pracownika z tabeli *Employees*, a istnieje powiązane zamówienie w tabeli *Orders*, system RDBMS odrzuci taką próbę i wygeneruje błąd.

Możemy zdefiniować klucz obcy z działaniem, które będzie kompensować takie próby (usunięcia wiersza z tabeli odwołania lub zaktualizowanie atrybutów klucza kandydującego odwołania, jeśli powiązane wiersze istnieją w tabeli odwołującej się). Możemy zdefiniować opcję *ON DELETE* i *ON UPDATE* za pomocą akcji, takich jak *CASCADE*, *SET DEFAULT* i *SET NULL*, jako część definicji klucza obcego. Opcja *CASCADE* oznacza, że operacja (usuwanie lub aktualizacja) będą miała charakter kaskadowy dla powiązanych wierszy. Na przykład, wyrażenie *ON DELETE CASCADE* oznacza, że podczas usuwania wiersza z tabeli odwołania system RDBMS usunie powiązane wiersze z tabeli odwołującej się. Opcje *SET DEFAULT* i *SET NULL* oznaczają, że działanie będzie odpowiednio ustawiać atrybuty klucza obcego powiązanych wierszy za pomocą domyślnych wartości kolumny lub za pomocą wartości *NULL*. Zwróćmy uwagę, że niezależnie od wyboru akcji tabela odwołująca się będzie miała „osierocone” wiersze tylko w przypadku wyjątku ze znacznikami *NULL*, o których była mowa wcześniej. Rodzice bez dzieci są zawsze dozwoleni.

Ograniczenia Check

Ograniczenie Check (sprawdzenie) pozwala zdefiniować predykat, tak by do wprowadzenia wiersza w tabeli lub do zmodyfikowania wiersza wymagane było jego spełnienie. Na przykład poniższe ograniczenie Check zapewnia, że kolumna *salary* w tabeli *Employees* będzie obsługiwać jedynie wartości dodatnie.

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT CHK_Employees_salary
  CHECK(salary > 0.00);
```

Próba wstawienia lub zaktualizowania wiersza przy użyciu wartości ujemnych zostanie odrzucona przez system RDBMS. Zwróćmy uwagę, że ograniczenie Check odrzuca próbę wstawienia lub aktualizacji wiersza, jeśli predykat ma wartość *FALSE*. Modyfikacje będą akceptowane, jeśli predykat ma wartość *TRUE* lub *UNKNOWN*. Na przykład, pensja (salary) równa -1000 zostanie odrzucona, zaakceptowane natomiast będą wartości 50000 i *NULL*.

Podczas dodawania ograniczeń Check i klucza obcego możemy wyspecyfikować opcję nazwaną *WITH NOCHECK*, która informuje system RDBMS, że chcemy ominąć ograniczenie sprawdzania dla istniejących danych. Podejście takie nie jest uważane za dobrą praktykę, ponieważ nie możemy mieć pewności, że nasze dane są spójne. Możemy również wyłączyć lub włączyć istniejące ograniczenia Check i klucz obcy.

Ograniczenia Default

Ograniczenie Default (domyślne) powiązane jest z konkretnym atrybutem. Jest to wyrażenie używane jako wartość domyślna, jeśli podczas wstawiania wiersza nie jest wprost wyspecyfikowana wartość atrybutu. Poniższy kod jest przykładem definiowania ograniczenia Default dla atrybutu *orders* (reprezentującego datownik zamówienia):

```
ALTER TABLE dbo.Orders
  ADD CONSTRAINT DFT_Orders_orders
  DEFAULT(SYSDATETIME()) FOR orders;
```

Wyrażenie Default wywołuje funkcję *SYSDATETIME*, która zwraca bieżącą datę i godzinę. Po zdefiniowaniu tego wyrażenia domyślnego, ilekroć wstawiany będzie wiersz w tabeli *Orders* i nie będzie wyspecyfikowana wprost wartość atrybutu *orders*, system SQL Server ustawi wartość atrybutu przy użyciu funkcji *SYSDATETIME*.

Na koniec wykonamy poniższy kod, by wyczyścić naszą bazę.

```
DROP TABLE dbo.Orders, dbo.Employees;
```

Podsumowanie

W rozdziale tym skrótowo przedstawiłem podstawy tworzenia zapytań i programowania języka T-SQL. Pokazałem teoretyczne podstawy, wyjaśniając mocne fundamenty, na których opiera się język T-SQL. Opisałem także architekturę systemu SQL Server, a na koniec pokazałem techniki używania T-SQL do tworzenia tabel i definiowania integralności danych. Mam nadzieję, że teraz bardziej widoczne jest to coś szczególnego w języku SQL i że nie jest to język, który można opanować w sposób mechaniczny. Rozdział ten wyposażył nas w podstawowe pojęcia – prawdziwa podróż dopiero przed nami.

ROZDZIAŁ 2

Zapytania do pojedynczej tabeli

W tym rozdziale poznamy podstawy polecenia *SELECT*, skupiając się na zapytaniach odnoszących się do pojedynczej tabeli. Najpierw przedstawię logikę przetwarzania zapytania, czyli ciąg logicznych faz prowadzących do wygenerowania prawidłowego zbioru wyników danego polecenia *SELECT*. Następnie omówię inne aspekty zapytań do pojedynczej tabeli, w tym predykaty i operatory, wyrażenia *CASE*, znaczniki *NULL*, operacje typu *all-at-once*, operacje na danych znakowych oraz datach i godzinach, a także tworzenie zapytań dotyczących metadanych. Wiele przykładów kodu i ćwiczeń w tej książce korzysta z przykładowej bazy danych nazwanej *TSQV4*. Instrukcje na temat pobierania i instalowania tej bazy danych znaleźć można w Dodatku „Rozpoczynamy”.

Elementy instrukcji *SELECT*

Zadanie instrukcji *SELECT* to odpytanie tabeli, zastosowanie pewnych logicznych operacji i zwrócenie wyników. W tym podrozdziale omówię fazy logiki przetwarzania zapytania. Przedstawię logiczną kolejność, zgodnie z którą przetwarzane są różne klauzule oraz co dzieje się w trakcie realizacji każdej fazy.

Poprzez określenie „logiczne przetwarzanie zapytania” rozumiem koncepcyjną metodę, poprzez którą standard SQL określa, jak zapytanie powinno być przetwarzane i jak powinny być uzyskiwane końcowe wyniki. Nie trzeba się obawiać, że niektóre opisywane logiczne fazy przetwarzania wydają się nieefektywne. Mechanizm produktu Microsoft SQL Server nie musi działać literalnie zgodnie z logicznym przetwarzaniem zapytania; raczej ma pełną swobodę w wyborze sposobu fizycznego przetwarzania, inaczej porządkując poszczególne fazy procesu, o ile końcowe wyniki będą takie same jak wyniki dyktowane przez logiczne przetwarzanie zapytania. System SQL Server w fizycznym przetwarzaniu zapytania może – w rzeczywistości często tak się dzieje – wprowadzać wiele skrótowych operacji jako rezultat optymalizacji zapytania.

Aby przedstawić logiczne przetwarzanie zapytania i różnych postaci klauzuli *SELECT*, posłużę się przykładem zawartym w listingu 2-1.

LISTING 2-1 Przykład zapytania

```
USE TSQLV4;  
  
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders  
FROM Sales.Orders  
WHERE custid = 71  
GROUP BY empid, YEAR(orderdate)  
HAVING COUNT(*) > 1  
ORDER BY empid, orderyear;
```

Zapytanie to filtruje zamówienia złożone przez klienta 71, grupuje te zamówienia według pracownika i roku złożenia zamówienia oraz filtruje tylko te grupy i lata, w których było więcej niż jedno zamówienie. Dla pozostałych po filtrowaniu grup zapytanie prezentuje identyfikator pracownika, rok zamówienia i liczbę zamówień posortowanych według ID pracownika i roku zamówienia. Jeśli działanie zapytania wydaje się niezrozumiałe, nie trzeba się martwić; po kolei objaśnię poszczególne klauzule zapytania i będziemy stopniowo je konstruować.

Kod rozpoczyna się od użycia instrukcji *USE*, która zapewnia, że w naszej sesji kontekstem jest przykładowa baza danych *TSQLV4*. Jeśli kontekst sesji byłby już odpowiednio określony, instrukcja *USE* nie jest wymagana.

Zanim przejdę do omawiania szczegółów każdej fazy instrukcji *SELECT*, chciałbym zwrócić uwagę na kolejność logicznego przetwarzania klauzul zapytania. W większości języków programowania wiersze kodu są przetwarzane w kolejności ich zapisania. W języku SQL rzeczy mają się inaczej. Choć klauzula *SELECT* występuje na początku zapytania, logicznie przetwarzana jest prawie na końcu. Kolejność w logice przetwarzania klauzul jest następująca:

1. *FROM*
2. *WHERE*
3. *GROUP BY*
4. *HAVING*
5. *SELECT*
6. *ORDER BY*

Tak więc, nawet jeśli pod względem składniowym przykładowe zapytanie (listing 2-1) rozpoczyna się od klauzuli *SELECT*, logicznie jego klauzule przetwarzane są w następującej kolejności:

```
FROM Sales.Orders  
WHERE custid = 71  
GROUP BY empid, YEAR(orderdate)  
HAVING COUNT(*) > 1  
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders  
ORDER BY empid, orderyear
```


W celu zaprezentowania zapytania w bardziej czytelny sposób poniższa lista pokazuje, co wykonuje cała instrukcja*:

1. Odpytanie wierszy z (*from*) tabeli *Sales.Orders*.
2. Filtrowanie tylko tych zamówień, gdzie (*where*) ID klienta równa się 71.
3. Grupowanie (*group*) zamówień według ID pracownika i roku zamówienia.
4. Filtrowanie (*filter*) tylko tych grup (ID pracownika i rok zamówienia), dla których było więcej niż jedno zamówienie.
5. Wybranie (*select*) dla każdej grupy ID pracownika, roku zamówienia i liczby zamówień.
6. Uporządkowanie (*sort*) wierszy w danych wyjściowych według ID pracownika i roku zamówienia.

Nie możemy napisać zapytania w prawidłowej kolejności logicznej. Musimy rozpocząć od klauzuli *SELECT*, tak jak w listingu 2-1. Istnieje powód tych rozbieżności pomiędzy kolejnością wpisywania a kolejnością logicznego przetwarzania klauzul. Projektanci języka SQL opracowali język deklaracji, w którym możemy wprowadzać żądania w sposób podobny do logiki języka angielskiego. Dla przykładu przeanalizujmy instrukcje przekazywane komuś w naturalnym języku: „Przynieś mi kluczyki do samochodu z lewej górnej szuflady w kuchni”. Najpierw określamy czynność (przynieś), później wskazujemy obiekt działań (kluczyki), a dopiero na końcu wskazujemy lokalizację obiektu (szuflada). Gdybyśmy zechcieli te same instrukcje przekazać robotowi, na przykład programowi komputerowemu, musielibyśmy rozpocząć od lokalizacji, zanim wskażemy, co ma być uzyskane w tej lokalizacji i co – ostatecznie – należy z tym zrobić. Instrukcje miałyby zapewne następującą postać „idź do kuchni; otwórz lewą górną szufladę; zabierz kluczyki; przynieś kluczyki do mnie”. Kolejność klauzul zapytania jest podobna do języka naturalnego – pierwsza jest klauzula *SELECT*, definiująca ostateczne działanie (wybór). Kolejność logicznego przetwarzania zapytania przypomina sposób wprowadzania instrukcji dla robota – pierwsza przetwarzana jest klauzula *FROM*, wskazująca lokalizację.

Teraz, gdy już znamy kolejność logicznego przetwarzania klauzul, możemy wyjaśnić szczegółowo każdą fazę.

W trakcie omawiania logicznego przetwarzania zapytania będę odwoływać się do *klauzul zapytania* i do *faz zapytania* (na przykład klauzula *WHERE* i faza *WHERE*). Klauzula zapytania to składniowy element zapytania, tak więc, jeśli omawiam składnię elementu zapytania, zazwyczaj będę używał pojęcia klauzuli – na przykład „w klauzuli *WHERE* specyfikujemy predykat”. Podczas omawiania logicznych operacji, które mają miejsce jako część logicznego przetwarzania zapytania, używam pojęcia fazy – na przykład „faza *WHERE* zwraca wiersze, dla których predykat ma wartość *TRUE*”.

* Angielskie odpowiedniki spójników i dyrektyw (wyróżnionych wytłuszczeniem) są, jak można zauważyć, klauzulami języka SQL (przyp. tłum.).

Przypomnijmy sobie z poprzedniego rozdziału zalecenie dotyczące używania średnika kończącego polecenia. W tej chwili SQL Server nie wymaga kończenia średnikiem wszystkich instrukcji. Jest to wymagane jedynie w szczególnych przypadkach, kiedy interpretacja kodu mogłaby być niejednoznaczna. Proponuję jednak, aby kończyć średnikiem wszystkie polecenia, ponieważ jest to zgodne ze standardem, poprawia czytelność, a prawdopodobne jest, że SQL Server będzie tego wymagał w przyszłości dla większej liczby przypadków, o ile nie zawsze. Obecnie dodanie średnika w miejscu, w którym nie jest wymagany, nie powoduje problemów. Tak więc warto przyzwyczaić się do stawiania średnika na końcu wszystkich poleceń.

Klauzula *FROM*

Klauzula *FROM* jest pierwszą klauzulą logicznego przetwarzania zapytania. W tej klauzuli specyfikujemy nazwy tabel, które chcemy przepytac, i operatory tabelaryczne, które działają na tych tabelach. W tym rozdziale nie będę omawiać operatorów tabelarycznych – są one opisane w rozdziałach 3, 5 i 7. Na tym etapie będziemy traktować klauzulę *FROM* jako prostą deklarację, w której określamy nazwę tabeli do odpytywania. Przykładowe zapytanie (listing 2-1) odpytuje tabelę *Orders* w schemacie *Sales*, znajdując 830 wierszy.

```
FROM Sales.Orders
```

Przypominam zalecenie z poprzedniego rozdziału, by zawsze używać nazw obiektów kwalifikowanych schematem. Jeśli nie specyfikujemy wprost nazwy schematu, system SQL Server musi ją rozpoznawać pośrednio w oparciu o swoje reguły rozwiązywania nazw. Proces ten wprowadza niewielki koszt, a może także spowodować, że SQL Server wybierze inny obiekt, niż planowaliśmy. Dzięki jednoznaczniemu formułowaniu poleceń nasz kod jest bezpieczniejszy, ponieważ zapewniamy, że uzyskujemy ten obiekt, który planowaliśmy. Dodatkowo nie ponosimy żadnych niepotrzebnych kosztów.

Aby zwrócić wszystkie wiersze tabeli bez żadnych specjalnych manipulacji, wystarczy utworzyć zapytanie z klauzulą *FROM*, w której wyspecyfikujemy tabelę, i klauzulą *SELECT* ze wskazaniem atrybutu, który chcemy odebrać. Na przykład poniższe polecenie zwraca wszystkie wiersze tabeli *Orders* w schemacie *Sales* przy wybranych atrybutach *orderid*, *custid*, *empid*, *orderdate* i *freight*.

```
SELECT orderid, custid, empid, orderdate, freight
FROM Sales.Orders;
```

Oto wyniki działania tego zapytania w skróconej formie:

orderid	custid	empid	orderdate	freight
-----	-----	-----	-----	-----
10248	85	5	2014-07-04	32.38
10249	79	6	2014-07-05	11.61
10250	34	4	2014-07-08	65.83
10251	84	3	2014-07-08	41.34

```

10252      76      4      2014-07-09  51.30
10253      34      3      2014-07-10  58.17
10254      14      5      2014-07-11  22.98
10255      68      9      2014-07-12 148.33
10256      88      3      2014-07-15  13.97
10257      35      4      2014-07-16  81.91
...
(830 row(s) affected)

```

Chociaż może wydawać się, że dane wyjściowe zapytania są zwracane w konkretnym porządku, nie jest to zagwarantowane. Temat ten omawię szczegółowo w podrozdziałach „Klauzula *SELECT*” i „Klauzula *ORDER BY*”.

Separatory nazw identyfikatorów

Dopóki identyfikatory schematów, tabel czy kolumn użyte w zapytaniu są zgodne z regułami formatu identyfikatorów regularnych, nie musimy oddzielać ich nazw żadnymi separatorami. Zasady tworzenia identyfikatorów regularnych opisane są w dokumentacji SQL Server Books Online pod adresem URL: <http://msdn.microsoft.com/en-us/library/ms175874>. W skrócie – identyfikator regularny nie może zawierać spacji ani innych znaków niedrukowalnych, cudzysłowów, nie może się zaczynać od cyfry ani nie może być zastrzeżonym słowem kluczowym (bez względu na wielkość liter – standard SQL zastrzega dowolne wersje słów kluczowych). Lista słów zastrzeżonych zależy od poziomu kompatybilności bazy danych.

Jeśli identyfikator nie jest regularny – na przykład zawiera spację lub znaki specjalne, rozpoczyna się od cyfry lub zastrzeżonego słowa kluczowego – trzeba go odseparować. Istnieje kilka sposobów oddzielania identyfikatorów w systemie SQL Server. Standardową formą zdefiniowaną w standardzie SQL jest użycie znaków podwójnego prostego cudzysłowu – na przykład *"Order Details"*. Formą specyficzną dla T-SQL jest użycie nawiasów kwadratowych – na przykład *[Order Details]*, ale standardowa forma jest również obsługiwana.

W przypadku identyfikatorów, które są zgodne z formatem identyfikatorów regularnych, separowanie ich jest opcjonalne. Na przykład do tabeli nazwanej *OrderDetails* i umieszczonej w schemacie *Sales* można odwoływać się za pomocą wyrażenia *Sales.OrderDetails*, *"Sales"."OrderDetails"* lub *[Sales].[OrderDetails]*. Osobiście polecam nieużywanie separatorów, jeśli nie są wymagane, ponieważ powoduje to „zaśmiecanie” kodu. Dodatkowo, o ile to my sami przypisujemy identyfikatory, zawsze zalecam stosowanie identyfikatorów regularnych, na przykład *OrderDetails*, a nie *Order Details*.

Klauzula *WHERE*

Klauzula *WHERE* określa predykat lub wyrażenie logiczne służące do filtrowania wierszy zwracanych przez fazę *FROM*. W fazie *WHERE*, kolejnej fazie logicznego przetwarzania zapytania, zwracane są tylko te wiersze, dla których wyrażenie logiczne ma wartość *TRUE*. W przykładzie zapytania (listing 2-1) faza *WHERE* przepuszcza tylko te zamówienia, które złożył klient 71.

```
FROM Sales.Orders
WHERE custid = 71
```

Spośród 830 wierszy zwróconych przez fazę *FROM*, faza *WHERE* przepuszcza tylko 31 wierszy, dla których identyfikator klienta (customer ID) ma wartość 71. Aby zobaczyć, które wiersze otrzymamy w odpowiedzi po zastosowaniu filtru *custid = 71*, uruchamiamy poniższe zapytanie.

```
SELECT orderid, empid, orderdate, freight
FROM Sales.Orders
WHERE custid = 71;
```

Zapytanie to generuje następujące dane wyjściowe:

orderid	empid	orderdate	freight
-----	-----	-----	-----
10324	9	2014-10-08	214.27
10393	1	2014-12-25	126.56
10398	2	2014-12-30	89.16
10440	4	2015-02-10	86.53
10452	8	2015-02-20	140.26
10510	6	2015-04-18	367.63
10555	6	2015-06-02	252.49
10603	8	2015-07-18	48.77
10607	5	2015-07-22	200.24
10612	1	2015-07-28	544.08
10627	8	2015-08-11	107.46
10657	2	2015-09-04	352.69
10678	7	2015-09-23	388.98
10700	3	2015-10-10	65.10
10711	5	2015-10-21	52.41
10713	1	2015-10-22	167.05
10714	5	2015-10-22	24.49
10722	8	2015-10-29	74.58
10748	3	2015-11-20	232.55
10757	6	2015-11-27	8.19
10815	2	2016-01-05	14.62
10847	4	2016-01-22	487.57
10882	4	2016-02-11	23.10
10894	1	2016-02-18	116.13
10941	7	2016-03-11	400.81
10983	2	2016-03-27	657.54
10984	1	2016-03-30	211.22
11002	4	2016-04-06	141.16

```

11030      7      2016-04-17      830.75
11031      6      2016-04-17      227.22
11064      1      2016-05-01       30.09
(31 row(s) affected)

```

Klauzula *WHERE* ma duże znaczenie z punktu widzenia wydajności zapytania. Na podstawie wyrażenia filtrującego SQL Server ocenia możliwość użycia indeksów w celu uzyskania dostępu do wymaganych danych. Przy użyciu indeksów SQL Server może czasem uzyskać wymagane dane przy znacznie mniejszym nakładzie pracy w porównaniu do skanowania całej tabeli. Filtry zapytania zmniejszają także ruch sieciowy generowany przez zwracanie wszystkich możliwych wierszy do źródła zapytania i filtrowanie po stronie klienta.

Poprzednio wspomniałem, że przez fazę *WHERE* zwracane są jedynie te wiersze, dla których wyrażenie logiczne ma wartość *TRUE*. Zawsze jednak należy pamiętać, że język T-SQL stosuje trójwartościową logikę predykatu, w której wyrażenie logiczne może mieć trzy wartości *TRUE*, *FALSE* lub *UNKNOWN*. W przypadku trójwartościowej logiki powiedzenie „zwraca *TRUE*” nie oznacza tego samego, co powiedzenie „nie zwraca *FALSE*”. Faza *WHERE* zwraca wiersze, dla których wyrażenie logiczne ma wartość *TRUE* i nie zwraca wierszy, dla których wyrażenie logiczne ma wartość *FALSE* lub *UNKNOWN*. Dokładniejsze omówienie tego tematu znaleźć można w dalszej części rozdziału w podrozdziale zatytułowanym „Znaczniki *NULL*”.

Klauzula *GROUP BY*

Faza *GROUP BY* pozwala uporządkować (zgrupować) wiersze zwrócone przez poprzednio przetwarzaną fazę logiczną. Grupy są określane przez elementy wyspecyfikowane w klauzuli *GROUP BY*. Na przykład klauzula *GROUP BY* w zapytaniu z listingu 2-1 zawiera elementy *empid* i *YEAR(orderdate)*.

```

FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)

```

Oznacza to, że faza *GROUP BY* tworzy grupę dla każdego unikatowego połączenia wartości identyfikatora pracownika (*empid*) i roku zamówienia, które pojawiają się w danych zwracanych przez fazę *WHERE*. Wyrażenie *YEAR(orderdate)* wywołuje funkcję *YEAR*, by z kolumny *orderdate* zwracać tylko część dotyczącą roku.

Faza *WHERE* zwraca 31 wierszy, w których występuje 16 niepowtarzających się kombinacji wartości ID pracownika i roku zamówienia.

```

empid      YEAR(orderdate)
-----
1          2014
1          2015
1          2016
2          2014

```

2	2015
2	2016
3	2015
4	2015
4	2016
5	2015
6	2015
6	2016
7	2015
7	2016
8	2015
9	2014

Tak więc faza *GROUP BY* tworzy 16 grup i przypisuje każdy z 31 wierszy zwracanych przez fazę *WHERE* do odpowiedniej grupy.

Jeśli zapytanie obejmuje tworzenie grup, wszystkie logiczne fazy następujące po fazie *GROUP BY* – w tym *HAVING*, *SELECT* i *ORDER BY* – muszą operować na grupach, a nie na poszczególnych wierszach. W ostatecznych wynikach zapytania każda grupa jest w końcu reprezentowana przez pojedynczy wiersz. Oznacza to, że wszystkie wyrażenia wyspecyfikowane w klauzulach przetwarzanych w kolejnych fazach (po fazie *GROUP BY*) muszą dla poszczególnych grup zagwarantować zwracanie wartości skalarnej (pojedynczej).

Wyrażenia bazujące na elementach, które znajdują się na liście *GROUP BY*, spełniają wymagania, ponieważ zgodnie z definicją każda grupa ma tylko jedno niepowtarzalne wystąpienie każdego elementu *GROUP BY*. Na przykład w grupie dla pracownika o identyfikatorze 8 i roku zamówienia 2015 istnieje tylko jedna unikatowa wartość ID pracownika i tylko jedna unikatowa wartość roku zamówienia. Z tego względu w klauzulach przetwarzanych po fazie *GROUP BY*, takich jak klauzula *SELECT*, możemy odwoływać się do wyrażeń *empid* i *YEAR(orderdate)*. Na przykład poniższe zapytanie zwraca 16 wierszy dla 16 grup wartości ID pracownika i roku zamówienia.

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

Zapytanie to zwraca następujące wyniki:

empid	orderyear
-----	-----
1	2014
1	2015
1	2016
2	2014
2	2015
2	2016
3	2015
4	2015
4	2016

```

5          2015
6          2015
6          2016
7          2015
7          2016
8          2015
9          2014
(16 row(s) affected)

```

Elementy, które nie znajdują się na liście *GROUP BY*, są dopuszczone jedynie jako dane wejściowe dla funkcji agregujących, takich jak *COUNT*, *SUM*, *AVG*, *MIN* czy *MAX*. Na przykład poniższe zapytanie zwraca łączną wartość frachtu za towary (*totalfreight*) i liczbę zamówień w odniesieniu do każdego pracownika i roku zamówienia.

```

SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    SUM(freight) AS totalfreight,
    COUNT(*) AS numorders
FROM Sales.Orders WHERE custid = 71
GROUP BY empid, YEAR(orderdate);

```

Zapytanie to generuje następujące dane wyjściowe:

empid	orderyear	totalfreight	numorders
1	2014	126.56	1
2	2014	89.16	1
9	2014	214.27	1
1	2015	711.13	2
2	2015	352.69	1
3	2015	297.65	2
4	2015	86.53	1
5	2015	277.14	3
6	2015	628.31	3
7	2015	388.98	1
8	2015	371.07	4
1	2016	357.44	3
2	2016	672.16	2
4	2016	651.83	3
6	2016	227.22	1

(15 row(s) affected)

Wyrażenie *SUM(freight)* zwraca sumę wszystkich wartości frachtu za towary w każdej grupie, a funkcja *COUNT(*)* zwraca liczbę wierszy w każdej grupie – co w tym przypadku oznacza liczbę zamówień. Jeśli spróbujemy odwołać się do atrybutu, który nie znajduje się na liście *GROUP BY* (takiego jak *freight*) i odwołanie to nie pełni roli danych wejściowych dla funkcji agregującej w klauzuli przetwarzanej po klauzuli *GROUP BY*, otrzymamy błąd – w takim przypadku nie ma gwarancji, że wyrażenie zwróci pojedynczą wartość dla grupy. Na przykład następujące zapytanie nie zadziała:

```
SELECT empid, YEAR(orderdate) AS orderyear, freight
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

SQL Server zwróci następujący błąd:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.Orders.freight' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause
```

(Kolumna 'Sales.Orders.freight' jest nieprawidłowa na liście select, ponieważ nie jest zawarta w funkcji agregującej lub klauzuli GROUP BY.)

Zwróćmy uwagę na to, że funkcje agregujące ignorują znaczniki *NULL*, z jednym wyjątkiem – *COUNT(*)*. Przeanalizujmy na przykład grupę pięciu wierszy o wartościach 30, 10, *NULL*, 10, 10 w kolumnie nazwanej *qty*. Wyrażenie *COUNT(*)* zwróci wartość 5, ponieważ w grupie istnieje pięć wierszy, natomiast wyrażenie *COUNT(qty)* zwróci 4, ponieważ są tam cztery znane wartości. Jeśli chcemy obsługiwać tylko różne wystąpienia znanych wartości, specyfikujemy słowo kluczowe *DISTINCT* dla argumentów funkcji agregującej (w nawiasach). Na przykład wyrażenie *COUNT(DISTINCT qty)* zwróci 2, ponieważ istnieją dwie różne znane wartości. Słowo kluczowe *DISTINCT* może być używane także z innymi funkcjami agregującymi. Przykładowo wyrażenie *SUM(qty)* zwróci 60, natomiast wyrażenie *SUM(DISTINCT qty)* zwróci wartość 40. Wyrażenie *AVG(qty)* zwróci 15, natomiast wyrażenie *AVG(DISTINCT qty)* zwróci 20. Jako przykład działania opcji *DISTINCT* z funkcją agregującą, poniższy kod zwraca liczbę różnych klientów obsługiwanych przez każdego pracownika w każdym roku.

```
SELECT empid,
YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

Zapytanie to generuje następujące dane wyjściowe (skrótowe):

empid	orderyear	numcusts
-----	-----	-----
1	2014	22
2	2014	15
3	2014	16
4	2014	26
5	2014	10
6	2014	15
7	2014	11
8	2014	19
1	2015	40
2	2015	35
3	2015	46
4	2015	57
5	2015	13
6	2015	24


```

7          2015          30
8          2015          36
9          2015          16
...
(27 row(s) affected)

```

Klauzula *HAVING*

Podczas gdy klauzula *WHERE* filtruje wiersze, za pomocą klauzuli *HAVING* możemy wyspecyfikować predykat filtrowania grup. Do następnej fazy logicznego przetwarzania zapytania zwracane są tylko te grupy, dla których wyrażenie logiczne klauzuli *HAVING* ma wartość *TRUE*. Grupy, dla których logiczne wyrażenie ma wartość *FALSE* lub *UNKNOWN*, zostają odrzucone.

Ponieważ klauzula *HAVING* jest przetwarzana po pogrupowaniu wierszy, w wyrażeniu logicznym możemy odnosić się do funkcji agregujących. Na przykład w kwerendzie z listingu 2-1 klauzula *HAVING* zawiera wyrażenie logiczne *COUNT(*) > 1*, oznaczające, że faza *HAVING* filtruje tylko te grupy (kombinacje pracowników i roku zamówienia), którym odpowiada więcej niż jeden wiersz. Poniższy fragment listingu 2-1 pokazuje kroki, które zostały przetworzone do tej pory.

```

FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1

```

Pamiętamy, że w fazie *GROUP BY* powstało 16 grup identyfikatorów pracownika oraz roku zamówienia. Siedem z tych grup zawiera tylko jeden wiersz, tak więc po przetworzeniu klauzuli *HAVING* pozostanie dziewięć grup. Aby uzyskać te 9 grup, uruchamiamy następujące zapytanie:

```

SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;

```

Zapytanie to zwraca następujące dane wyjściowe.

```

empid      orderyear
-----
1          2015
3          2015
5          2015
6          2015
8          2015
1          2016
2          2016
4          2016
7          2016
(9 row(s) affected)

```

Klauzula *SELECT*

Klauzula *SELECT* to miejsce specyfikowania atrybutów (kolumn), które chcemy otrzymać w tabeli wyników zapytania. Wyrażenia na liście *SELECT* mogą bazować na atrybutach odpytanych tabel, bez lub przy zastosowaniu dodatkowych operacji. Na przykład lista *SELECT* w przykładowym listingu 2-1 zawiera następujące wyrażenia: *empid*, *YEAR(orderdate)* i *COUNT(*)*. Jeśli wyrażenie odnosi się do atrybutu bez dodatkowych operacji, jak na przykład *empid*, nazwa atrybutu docelowego jest taka sama jak nazwa atrybutu źródłowego. Do atrybutu docelowego możemy opcjonalnie przypisać własne nazwy przy użyciu klauzuli *AS* (jako) – na przykład *empid AS employee_id*. Wyrażenia, które wykonują operacje, takie jak *YEAR(orderdate)*, a także takie, które nie opierają się na atrybucie źródłowym, takie jak wywołanie funkcji *CURRENT_TIMESTAMP*, w wynikach zapytania nie mają nazwy, o ile nie utworzymy dla nich nazwy zastępczej (aliasu). W niektórych przypadkach język T-SQL pozwala kwerendzie zwracać wynikowe kolumny bez nazw, jednak model relacyjny na to nie zezwala. Zdecydowanie polecam tworzenie aliasów dla takich wyrażeń, na przykład *YEAR(orderdate) AS orderyear*, tak by wszystkie atrybuty wynikowe miały nazwy. Przy spełnieniu tego warunku tabela wyników zwrócona przez kwerendę będzie uważana za relacyjną.

Alternatywne metody tworzenia aliasów

Oprócz klauzuli *AS*, język T-SQL obsługuje kilka innych konstrukcji, za pomocą których możemy nazywać wyrażenia, jednak dla mnie klauzula *AS* wydaje się najbardziej czytelną i intuicyjną metodą i dlatego sugeruję jej stosowanie. Przedstawię pozostałe konstrukcje dla zachowania pełnego obrazu, a także by opisać trudne do wykrycia błędy związane z jedną z nich. Poza konstrukcją *<wyrażenie> AS <alias>* język T-SQL obsługuje także konstrukcje *<alias> = <wyrażenie>* („*alias równa się wyrażenie*”) oraz *<wyrażenie> <alias>* („*wyrażenie spacja alias*”). Przykładem pierwszej konstrukcji jest *orderyear = YEAR(orderdate)*, a przykładem drugiej *YEAR(orderdate) orderyear*. Ta druga forma, w której po wyrażeniu jest spacja, a po niej alias, jest szczególnie mało czytelna i zdecydowanie zalecam unikanie jej stosowania. Niestety, jest to często spotykane w kodzie tworzonemu przez różnych użytkowników.

Warto zwrócić uwagę, że jeśli przez pomyłkę nie wpisujemy przecinka pomiędzy dwiema nazwami kolumn na liście *SELECT*, nie zostanie zgłoszony błąd – zamiast tego SQL Server założy, że druga nazwa jest aliasem pierwszej nazwy kolumny. Dla przykładu założmy, że chcemy napisać kwerendę, która wybiera kolumny *orderid* i *orderdate* z tabeli *Sales.Orders* i omyłkowo nie wpisaliśmy przecinka pomiędzy nazwami kolumn, jak w poniższej instrukcji:

```
SELECT orderid orderdate
FROM Sales.Orders;
```

Zapytanie to pod względem składniowym jest uważane za prawidłowe, tak jakbyśmy chcieli utworzyć alias kolumny *orderid* przy użyciu nazwy *orderdate*. W danych wyjściowych otrzymamy tylko jedną kolumnę, w której umieszczone są identyfikatory zamówień (*orderId*), a kolumna nazwana jest *orderdate*.

```
orderdate
-----
10248
10249
10250
10251
10252
...
(830 row(s) affected)
```

Błąd taki trudno później zauważyć, więc lepiej pamiętać o tym przy pisaniu kodu!

Po dołączeniu fazy *SELECT*, do tej pory przetworzone zostały następujące klauzule zapytania z listingu 2-1.

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
```

Klauzula *SELECT* generuje tabelę wyników zapytania. W przypadku zapytania z listingu 2-1 nagłówkami tabeli wyników są atrybuty *empid*, *orderyear* i *numorders*, a treść składa się z dziewięciu wierszy (po jednym dla każdej grupy). Aby uzyskać te 9 wierszy, uruchamiamy następujące zapytanie:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

Zapytanie to generuje następujące dane wyjściowe:

empid	orderyear	numorders
----	-----	-----
1	2015	2
3	2015	2
5	2015	3
6	2015	3
8	2015	4

```

1          2016          3
2          2016          2
4          2016          3
7          2016          2
(9 row(s) affected)

```

Przypomnijmy, że klauzula *SELECT* jest przetwarzana *później*, niż klauzule *FROM*, *WHERE*, *GROUP BY* i *HAVING*. Oznacza to, że aliasy przypisane wyrażeniom w klauzuli *SELECT* jeszcze nie istnieją, gdy przetwarzane są klauzule należące do wcześniejszych faz. Bardzo typowym błędem programistów, którzy nie przyswoili sobie dobrze prawidłowej kolejności przetwarzania logicznego klauzul, jest odnoszenie się do aliasów wyrażen w klauzulach, które są przetwarzane przed klauzulą *SELECT*. Poniżej przytoczony został przykład takiej nieprawidłowej próby w klauzuli *WHERE*.

```

SELECT orderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE orderyear > 2014;

```

Z pozoru zapytanie to wygląda na poprawne, jeśli jednak uwzględnimy fakt, że aliasy kolumn są tworzone w fazie *SELECT* – która jest przetwarzana po fazie *WHERE* – widzimy, że odwołanie do aliasu *orderyear* w klauzuli *WHERE* jest nieprawidłowe. I rzeczywiście, system SQL Server wygeneruje następujący błąd:

```

Msg 207, Level 16, State 1, Line 3
Invalid column name 'orderyear'.
(Nieprawidłowa nazwa kolumny 'orderyear')

```

Co zabawne, jeden z moich studentów zupełnie poważnie zapytał mnie, kiedy Microsoft zamierza naprawić ten błąd. Jak można zauważyć po lekturze tego rozdziału, zachowanie takie nie jest błędem, ale wynika z samego projektu. Co więcej, nie zostało ono zdefiniowane przez firmę Microsoft; jest to element standardu SQL.

Jeden ze sposobów obejścia tego problemu polega na powtarzaniu wyrażenia *YEAR(orderdate)* w obu klauzulach *WHERE* i *SELECT*.

```

SELECT orderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE YEAR(orderdate) > 2014;

```

Warto zauważyć, że optymalizator SQL Server potrafi zidentyfikować powtórne użycie tego samego wyrażenia w jednym zapytaniu – w tym przypadku *YEAR(orderdate)*. Wyrażenie będzie oszacowane lub obliczone tylko raz.

Poniższe zapytanie jest innym przykładem nieprawidłowego odniesienia do aliasu kolumny. Zapytanie próbuje odwołać się do aliasu kolumny w klauzuli *HAVING*, która również jest przetwarzana przed klauzulą *SELECT*.

```

SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71

```

```
GROUP BY empid, YEAR(orderdate)
HAVING numorders > 1;
```

Zapytanie to generuje błąd informujący, że nazwa kolumny *numorders* jest nieprawidłowa. I tu też trzeba w obu klauzulach powtórzyć wyrażenie *COUNT(*)*.

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

W modelu relacyjnym operacje na relacjach bazują na algebrze relacyjnej i dają wyniki w postaci relacji (zbioru). Przypomnijmy, że ciałem relacji jest zbiór krotek, a zbiór nie zawiera duplikatów. W odróżnieniu od modelu relacyjnego, opartego na matematycznej teorii mnogości, SQL opiera się na teorii wielozbiorów. Matematyczne pojęcie *wielozbioru* (ang. *multiset* lub *bag*) pod wieloma aspektami jest podobne do zbioru, ale pozwala na istnienie duplikatów. Tabela w SQL nie musi zawierać klucza. Bez kluczy unikatowość wierszy nie jest zagwarantowana, a w takiej sytuacji tabela nie jest zbiorem. Co więcej, jeśli nawet odpytywane tabele posiadają klucze i są kwalifikowane jako zbiory, zapytanie *SELECT* dotyczące tych tabel nadal może zwrócić wyniki zawierające duplikaty wierszy. Pojęcie „zbiór wyników” jest często używane do określenia danych wyjściowych zapytania *SELECT*, ale zestaw wyników niekoniecznie można kwalifikować jako zbiór w sensie matematycznym. Na przykład, chociaż tabela *Orders* jest zbiorem, gdyż niepowtarzalność wymuszona jest za pomocą klucza, zapytanie z listingu 2-2 dotyczące tabeli *Orders* zwraca duplikaty wierszy.

LISTING 2-2 Zapytanie zwracające duplikaty wierszy

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;
```

Zapytanie to generuje następujące dane wyjściowe (wydruk skrócony, zduplikowane wiersze zostały wyróżnione wytłuszczeniem):

empid	orderyear
9	2014
1	2014
2	2014
4	2015
8	2015
6	2015
6	2015
8	2015
...	
1	2016

(31 row(s) affected)

Język SQL zapewnia środki gwarantujące unikatowość wyników polecenia *SELECT* w postaci klauzuli *DISTINCT*, która usuwa duplikaty wierszy, co ilustruje listing 2-3.

LISTING 2-3 Zapytanie z klauzulą *DISTINCT*

```
SELECT DISTINCT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;
```

Zapytanie to generuje następujące dane wyjściowe:

empid	orderyear
-----	-----
1	2014
1	2015
1	2016
2	2014
2	2015
2	2016
3	2015
4	2015
4	2016
5	2015
6	2015
6	2016
7	2015
7	2016
8	2015
9	2014

(16 row(s) affected)

Spośród 31 wierszy wielozbioru zwróconego przez zapytanie z listingu 2-2, po usunięciu duplikatów w zbiorze zwróconym przez zapytanie z listingu 2-3 znalazło się 16 wierszy.

Język SQL zezwala na użycie na liście *SELECT* gwiazdki (*), oznaczającej żądanie zwrócenia wszystkich atrybutów odpytywanych tabel – nie trzeba wymieniać ich wszystkich wprost, co ilustruje poniższy przykład.

```
SELECT *
FROM Sales.Shippers;
```

Użycie gwiazdki to w większości przypadków nienajlepsze rozwiązanie praktyczne, poza nielicznymi wyjątkami, i uważane jest za wadliwy wzorec programowania. Zaleca się specyfikowanie wprost listy atrybutów, nawet jeśli potrzebne są wszystkie atrybuty odpytywanej tabeli. Istnieje wiele powodów, by stosować to zalecenie. Inaczej niż w modelu relacyjnym, język SQL utrzymuje liczby porządkowe kolumn, opierając się na kolejności specyfikowania kolumn w instrukcji *CREATE TABLE*. Użycie polecenia *SELECT ** gwarantuje, że uzyskamy kolumny w kolejności zgodnej z ich liczbą porządkową. Aplikacje klienckie mogą odnosić się do kolumn w wynikach

zgodnie z ich przyporządkowaniem, a nie według nazw (samo w sobie jest to niedobłą praktyką). Jakakolwiek zmiana schematu tabeli – na przykład dodanie lub usunięcie kolumn, zmiana ich kolejności itp. – może spowodować błędy w aplikacji klienckiej lub, co gorsza, błędy logiczne, które przejdą niezauważone. Poprzez wyspecyfikowanie wprost potrzebnych atrybutów zawsze uzyskujemy odpowiednie atrybuty, o ile w tabeli istnieją takie kolumny. Jeśli kolumna, do której się odwołujemy, została usunięta z tabeli, otrzymamy komunikat o błędzie i wiemy, że musimy odpowiednio poprawić nasz kod.

Niektórzy dziwią się, skąd biorą się różnice wydajności pomiędzy użyciem gwiazdki a wymienieniem wprost nazw kolumn. W tym pierwszym przypadku muszą być podjęte pewne dodatkowe działania w celu rozpoznania nazw kolumn. Zazwyczaj koszt ten jest niewielki w porównaniu do innych kosztów związanych z zapytaniem i trudno go zauważyć. Jeśli pojawia się jakakolwiek różnica wydajności, nawet niewielka, najprawdopodobniej jest to zasługa wymieniania wprost nazw kolumn, a ponieważ jest to zalecane rozwiązanie praktyczne, sytuacja jest podwójnie korzystna.

Wewnątrz klauzuli *SELECT* nadal nie możemy odnosić się do aliasu kolumny, który został utworzony w tej samej klauzuli *SELECT*, niezależnie od tego, czy wyrażenie przypisujące alias pojawia się z lewej, czy z prawej strony wyrażenia, do którego próbujemy się odwołać. Na przykład poniższa próba będzie nieudana.

```
SELECT orderid,  
       YEAR(orderdate) AS orderyear,  
       orderyear + 1 AS nextyear  
FROM Sales.Orders;
```

Przyczyny tego ograniczenia wyjaśnię później, w podrozdziale „Operacje jednoczesne – *all-at-once*”. Jak już wcześniej wspomniałem, jednym ze sposobów ominięcia tego problemu jest powtórzenie wyrażenia.

```
SELECT orderid,  
       YEAR(orderdate) AS orderyear,  
       YEAR(orderdate) + 1 AS nextyear  
FROM Sales.Orders;
```

Klauzula **ORDER BY**

Klauzula *ORDER BY* pozwala posortować wiersze w danych wyjściowych w celu odpowiedniej prezentacji. W kontekście logicznego przetwarzania zapytania klauzula *ORDER BY* jest przetwarzana jako ostatnia. Przykładowe zapytanie z listingu 2-4 sortuje wiersze danych wyjściowych według identyfikatora pracownika i roku zamówienia.

LISTING 2-4 Zapytanie ilustrujące działanie klauzuli *ORDER BY*

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders  
FROM Sales.Orders  
WHERE custid = 71
```

```
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid, orderyear;
```

Zapytanie to generuje następujące dane wyjściowe:

empid	orderyear	numorders
1	2015	2
1	2016	3
2	2016	2
3	2015	2
4	2016	3
5	2015	3
6	2015	3
7	2016	2
8	2015	4

(9 row(s) affected)

Tym razem zagwarantowana będzie kolejność prezentowania danych wyjściowych – inaczej niż w przypadku zapytań, które nie zawierają klauzuli *ORDER BY*.

Jedną z najważniejszych kwestii związanych z językiem SQL jest pamiętanie o tym, że tabela nie gwarantuje kolejności, ponieważ tabela jest traktowana jako reprezentacja zbioru (lub wielozbioru, jeśli zawiera duplikaty), a w zbiorze nie mamy określonego porządku. Oznacza to, że kiedy odpytujemy tabelę bez wyspecyfikowania klauzuli *ORDER BY*, system SQL Server ma zupełną dowolność, w jakiej kolejności wynikowe wiersze pojawią się w danych wyjściowych. Jedyną metodą zagwarantowania uporządkowania wierszy jest jawne użycie klauzuli *ORDER BY*. Jeśli jednak użyjemy klauzuli *ORDER BY*, wyniki nie mogą być kwalifikowane jako tabela, ponieważ zagwarantowana jest kolejność wierszy wyników. Zapytanie z klauzulą *ORDER BY* generuje coś, co w standardowym języku SQL nazywane jest kursor – nierelacyjny wynik z zagwarantowaną kolejnością wierszy.

Można się zastanawiać, dlaczego w ogóle przedmiotem rozważań jest kwestia, czy zapytanie zwraca tabelę, czy kursor. Niektóre elementy i operacje języka SQL wymagają, by zapytanie dostarczało wyniki w formie tabeli, a nie kursora; przykładami mogą być wyrażenia tablicowe i operatory zbiorów, omawiane odpowiednio w rozdziałach 5 „Wyrażenia tablicowe” i 6 „Operatory zbiorów”.

Zwróćmy uwagę, że klauzula *ORDER BY* odnosi się do aliasu kolumny *orderyear*, który został utworzony w fazie *SELECT*. Faza *ORDER BY* jest w rzeczywistości jedyną fazą, w której możemy odwoływać się do aliasów kolumn utworzonych w fazie *SELECT*, ponieważ jest to jedyna faza przetwarzana po fazie *SELECT*. Można również zauważyć, że jeśli zdefiniujemy alias kolumny o takiej samej nazwie, jak nazwa kolumny wejściowej, na przykład *l – coll AS coll*, po czym w klauzuli *ORDER BY* odniesiemy się do tego aliasu, przy określaniu kolejności uwzględniana będzie nowa kolumna.

Jeśli chcemy sortować według wyrażenia w kolejności rosnącej, dodajemy *ASC* zaraz za wyrażeniem, na przykład *orderyear ASC* (od *ascending* – rosnąco) lub niczego

nie wpisujemy, ponieważ *ASC* to opcja domyślna. Jeśli chcemy sortować w kolejności malejącej, trzeba wyspecyfikować opcję *DESC* (od *descending* – malejąco) po wyrażeniu, jak na przykład *orderyear DESC*.

Język T-SQL zezwala na użycie liczby porządkowej kolumn w klauzuli *ORDER BY* na podstawie kolejności, w jakiej kolumny występują na liście *SELECT*. Na przykład w zapytaniu z listingu 2-4, zamiast instrukcji:

```
ORDER BY empid, orderyear
```

możemy użyć:

```
ORDER BY 1, 2
```

Z kilku powodów nie jest to jednak zalecane postępowanie w programowaniu. Po pierwsze, w modelu relacyjnym atrybuty nie mają liczby porządkowej i trzeba się do nich odnosić poprzez nazwę. Po drugie, wprowadzając poprawki do klauzuli *SELECT* możemy zapomnieć o wprowadzeniu odpowiednich zmian w klauzuli *ORDER BY*. Kiedy stosujemy nazwy kolumn, nasz kod jest odporny na tego rodzaju pomyłki.

Język T-SQL pozwala na specyfikowanie w klauzuli *ORDER BY* elementów, które nie występują w klauzuli *SELECT*, co oznacza, że możemy sortować wyniki według czegoś, co niekoniecznie chcemy, by było zwracane w danych wyjściowych. Główną wadą takiego postępowanie jest jednak to, że nie można sprawdzić właściwego posortowania danych, przeglądając wyniki. Na przykład poniższe zapytanie sortuje wiersze pracowników według daty zatrudnienia bez zwracania atrybutu *hiredate* (data zatrudnienia).

```
SELECT empid, firstname, lastname, country  
FROM HR.Employees  
ORDER BY hiredate;
```

Jeśli jednak użyta została klauzula *DISTINCT*, w liście *ORDER BY* jesteśmy ograniczeni tylko do tych elementów, które pojawiają się na liście *SELECT*. Jest to spowodowane tym, że jeśli specyfikujemy klauzulę *DISTINCT*, pojedynczy wynik może reprezentować wiele wierszy źródłowych; z tego względu może nie być jasne, która z wielu możliwych wartości w wyrażeniu *ORDER BY* powinna zostać użyta. Przeanalizujemy następującą nieprawidłową kwerendę:

```
SELECT DISTINCT country  
FROM HR.Employees  
ORDER BY empid;
```

W tabeli *Employees* istnieje dziewięciu pracowników – pięciu z USA i czterech z Wielkiej Brytanii. Jeśli w tej kwerendzie pominiemy nieprawidłową klauzulę *ORDER BY*, zwrócone zostaną dwa wiersze – jeden dla każdego kraju. Ponieważ każdy kraj pojawia się w wielu wierszach tabeli źródłowej i każdy taki wiersz ma inne ID pracownika (employee ID), znaczenie wyrażenia *ORDER BY empid* w istocie nie jest dobrze określone.

Filtry *TOP* i *OFFSET-FETCH*

Wcześniej w rozdziale opisałem filtry oparte na predykatkach *WHERE* i *HAVING*. W tym podrozdziale omówimy filtry bazujące na liczbie wierszy i kolejności: *TOP* oraz *OFFSET-FETCH*.

Filtr *TOP*

Opcja *TOP* jest własną funkcją języka T-SQL (nie należącą do standardu SQL), która pozwala ograniczać liczbę lub procent wierszy zwracanych przez zapytanie. Funkcja jest zależna od dwóch elementów, będących częścią jej specyfikacji; jeden jest liczbą lub procentem zwracanych wierszy, a drugi określa kolejność. Na przykład, aby zwrócić z tabeli *Orders* pięć najnowszych zamówień, w klauzuli *SELECT* specyfikujemy *TOP (5)*, a w klauzuli *ORDER BY* użyjemy *orderdate DESC* (listing 2-5).

LISTING 2-5 Zapytanie ilustrujące działanie filtru *TOP*

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Zapytanie to zwraca następujące dane wyjściowe:

orderid	orderdate	custid	empid
-----	-----	-----	-----
11077	2016-05-06	65	1
11076	2016-05-06	9	4
11075	2016-05-06	68	8
11074	2016-05-06	73	7
11073	2016-05-05	58	2

(5 row(s) affected)

Pamiętajmy, że klauzula *ORDER BY* jest przetwarzana po klauzuli *SELECT*, która zawiera opcję *DISTINCT*. To samo dotyczy opcji *TOP*, która bazuje na klauzuli *ORDER BY*, dostarczającej jej narzędzi filtrowania. Oznacza to, że jeśli w klauzuli *SELECT* wyspecyfikowana jest opcja *DISTINCT*, filtr *TOP* jest oceniany po usunięciu duplikatów wierszy.

Warto także zwrócić uwagę, że jeśli użyta jest opcja *TOP*, klauzula *ORDER BY* ma podwójne przeznaczenie. Jednym celem jest zdefiniowanie kolejności prezentowania wierszy w wynikach zapytania, a drugim celem jest określenie, które wiersze mają być odfiltrowane. Na przykład zapytanie w listingu 2-5 zwraca pięć wierszy o najwyższej wartości *ordatedate* (najnowsze) i prezentuje wiersze w kolejności malejącej względem daty zamówienia (*orderdate DESC*).

Zasadniczo możemy nie mieć pewności, czy klauzula *TOP* zwraca wyniki w postaci tabeli, czy kursora. Zazwyczaj zapytanie z klauzulą *ORDER BY* zwraca kursor, a nie wyniki relacyjne. Jednak co będzie w sytuacji, kiedy potrzebujemy odfiltrować wiersze

za pomocą opcji *TOP* w oparciu o pewne uporządkowanie, ale chcemy nadal zwracać wyniki relacyjne? Ponadto, co jeśli trzeba filtrować wiersze przy użyciu opcji *TOP* bazując na jednej kolejności, a prezentować wiersze wyjściowe w innej kolejności?

Aby zrealizować takie zadania, musimy użyć wyrażenia tablicowego, ale omówienie tych kwestii zamieszczone jest w rozdziale 5 „Wyrażenia tablicowe”. Teraz chciałem jedynie powiedzieć, że jeśli konstrukcja opcji *TOP* wydaje się niejasna, to są ku temu powody – inaczej mówiąc, przyczyna leży po stronie projektu funkcji. Byłoby świetnie, gdyby opcja *TOP* pozwalała na wyspecyfikowanie porządku dla siebie samej, niezależnie od sortowania do celów prezentacji zapewnianego przez *ORDER BY*, ale tak nie jest. Ten statek już płynie.

Opcji *TOP* można użyć w połączeniu ze słowem kluczowym *PERCENT*; w tym przypadku system SQL Server oblicza liczbę zwracanych wierszy w oparciu o procent liczby zakwalifikowanych wierszy (wynik zaokrąglony w górę). Na przykład poniższe zapytanie żąda 1% najnowszych zamówień.

```
SELECT TOP (1) PERCENT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Zapytanie to generuje następujące dane wyjściowe:

orderid	orderdate	custid	empid
-----	-----	-----	-----
11074	2016-05-06	7	
11075	2016-05-06	8	
11076	2016-05-06	9	4
11077	2016-05-06	65	1
11070	2016-05-05	44	2
11071	2016-05-05	46	1
11072	2016-05-05	20	4
11073	2016-05-05	58	2
11067	2016-05-04	17	1

(9 row(s) affected)

Zapytanie zwraca 9 wierszy, ponieważ tabela *Orders* składa się z 830 wierszy, a 1% z 830 to 9 (po zaokrągleniu).

W zapytaniu z listingu 2-5 mogliśmy zauważyć, że lista *ORDER BY* nie zachowuje niepowtarzalności, ponieważ dla kolumny *orderdate* (data zamówienia) nie został zdefiniowany klucz główny lub ograniczanie unikatowe. Wiele wierszy może mieć tę samą datę zamówienia. W sytuacji, kiedy nie są wyspecyfikowane żadne dodatkowe kryteria, kolejność wierszy o tej samej dacie zamówienia nie jest określona. Fakt ten powoduje, że zapytanie jest niedeterministyczne – więcej niż jeden wynik może być prawidłowy. W przypadkach nierozstrzygniętych system SQL Server określa kolejność wierszy w oparciu o to, który wiersz pojawi się fizycznie pierwszy.

Możemy też użyć opcji *TOP* bez klauzuli *ORDER BY*, a kolejność jest wówczas zupełnie niezdefiniowana – SQL Server zwraca *n* wierszy, które fizycznie pojawiają się jako pierwsze, gdzie *n* to liczba żądanych wierszy.

Zwróćmy uwagę, że w danych wyjściowych zapytania z listingu 2-5 najstarsza data zamówienia w zwróconych wierszach to 4 maja 2016 i taki wiersz jest jeden. W tabeli mogą istnieć wiersze z tą samą datą zamówienia i przy istniejącej, nie-unikatowej liście *ORDER BY* nie mamy gwarancji, które z nich zostaną zwrócone.

Jeśli chcemy, by zapytanie było deterministyczne, trzeba spowodować, by lista *ORDER BY* była niepowtarzalna; inaczej mówiąc, trzeba dodać dodatkową regułę rozstrzygania. Na przykład możemy do listy *ORDER BY* dodać opcję *orderid DESC* (listing 2-6) tak, aby w przypadku niejednoznaczności preferowany był wiersz o większym identyfikatorze zamówienia – *order ID*.

LISTING 2-6 Zapytanie ilustrujące użycie opcji *TOP* z unikatową listą *ORDER BY*

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
```

Zapytanie zwraca następujące dane wyjściowe:

orderid	orderdate	custid	empid
-----	-----	-----	-----
11077	2016-05-06	65	1
11076	2016-05-06	9	4
11075	2016-05-06	68	8
11074	2016-05-06	73	7
11073	2016-05-05	58	2

(5 row(s) affected)

Analizując wyniki zapytań z listingu 2-5 i 2-6 wydaje się, że są one takie same. Ważna różnica polega na tym, że wyniki prezentowane dla zapytania z listingu 2-5 to jedne z kilku możliwych prawidłowych wyników dla tego zapytania, natomiast wyniki pokazywane dla zapytania z listingu 2-6 to jedyny możliwy prawidłowy rezultat.

Zamiast dodawania dodatkowej reguły rozstrzygania do listy *ORDER BY*, możemy zażądać zwrócenia wszystkich nierozstrzygniętych wyników. Na przykład oprócz pięciu wierszy, które zwracane są w kwerendzie z listingu 2-5, możemy poprosić o zwrócenie wszystkich pozostałych wierszy tabeli, które mają tę samą wartość sortowania (w tym przypadku datę zamówienia) co ostatni znaleziony (5 maj 2016 w tym przypadku). Cel ten możemy zrealizować, dodając opcję *WITH TIES*, co ilustruje poniższe zapytanie.

```
SELECT TOP (5) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Zapytanie to zwraca następujące wyniki:

orderid	orderdate	custid	empid
-----	-----	-----	-----
11077	2016-05-06	65	1

11076	2016-05-06	9	4
11075	2016-05-06	68	8
11074	2016-05-06	73	7
11073	2016-05-05	58	2
11072	2016-05-05	20	4
11071	2016-05-05	46	1
11070	2016-05-05	44	2

(8 row(s) affected)

Zwróćmy uwagę, że wyniki to 8 wierszy, chociaż użyliśmy opcji *TOP (5)*. System SQL Server najpierw zwraca wiersze *TOP (5)* w oparciu o kolejność *orderdate DESC*, a następnie wszystkie pozostałe wiersze tabeli, które miały tę samą wartość *orderdate* co ostatni z pięciu wierszy, do których uzyskany był dostęp. W tym przypadku wybór wierszy jest zdeterminowany, ale ich kolejność (prezentacja) dla wierszy o tej samej dacie już nie jest.

Filtr *OFFSET-FETCH*

Opcja *TOP* to bardzo przydatny rodzaj filtru, ma jednak dwa mankamenty – nie należy do standardu oraz nie obsługuje funkcji pomijania początkowych wierszy. Standard SQL definiuje filtr podobny do filtru *TOP*, który nazwany został *OFFSET-FETCH*. Filtr ten obsługuje funkcję pomijania, co czyni go bardzo przydatnym przy wykonywaniu doraźnych zadań stronicowania wyników zapytania.

Filtr *OFFSET-FETCH*, dostępny od wersji SQL Server 2012, jest traktowany jako rozszerzenie klauzuli *ORDER BY*, która normalnie służy do określania kolejności prezentowania wyników. Używając klauzuli *OFFSET* możemy wskazać, ile początkowych wierszy chcemy pominąć, zaś w klauzuli *FETCH* określamy, ile wierszy pobieramy (po opuszczonych wierszach). Dla przykładu przeanalizujmy następujące zapytanie.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

Zapytanie porządkuje wiersze z tabeli *Orders*, bazując na atrybutach *orderdate*, *orderid* *ordering* (od najstarszego do najnowszego, gdzie *orderid* jest kryterium rozstrzygającym). Opierając się na tej kolejności klauzula *OFFSET* pomija pierwszych 50 wierszy, a klauzula *FETCH* zwraca tylko następnych 25 wierszy.

Zwróćmy uwagę, że zapytanie, które używa *OFFSET-FETCH*, musi zawierać klauzulę *ORDER BY*. Ponadto klauzula *FETCH* nie jest obsługiwana bez klauzuli *OFFSET*. Jeśli nie chcemy pomijać żadnego wiersza, ale chcemy je filtrować za pomocą klauzuli *FETCH*, musimy użyć wyrażenia *OFFSET 0 ROWS*. Dopuszczona jest jednak klauzula *OFFSET* bez klauzuli *FETCH*. W takim przypadku zapytanie pomija wskazaną liczbę wierszy i zwraca wszystkie pozostałe wiersze.

Składnia klauzul *OFFSET-FETCH* ma kilka interesujących językowo aspektów. Zamiennie można stosować formę liczby pojedynczej i mnogiej *ROW* i *ROWS*. Celem

było dopuszczenie wyrażenia filtru w postaci intuicyjnej, podobnej do języka angielskiego. Na przykład założmy, że chcemy pobrać tylko jeden wiersz; wprawdzie pod względem składniowym byłoby to prawidłowe, niemniej jednak specyfikacja wyglądałaby trochę dziwnie: *FETCH 1 ROWS **. Z tego względu możemy używać formy *FETCH 1 ROW*. To samo dotyczy klauzuli *OFFSET*. Ponadto, jeśli nie pomijamy żadnego wiersza (*OFFSET 0 ROWS*), okazuje się, że pod względem językowym pojęcie „first” (pierwsze) jest bardziej naturalne niż „next” (następne). W związku z tym można zamiennie używać słów kluczowych *FIRST* i *NEXT*.

Jak widzimy, klauzula *OFFSET-FETCH* jest bardziej elastyczna niż klauzula *TOP*, ponieważ obsługuje pomijanie początkowych wierszy. Jednakże klauzula *OFFSET-FETCH* nie obsługuje opcji *PERCENT* i *WITH TIES*, które działają z klauzulą *TOP*. Ponieważ klauzula *OFFSET-FETCH* jest standardem, a *TOP* nim nie jest, zalecam stosowanie klauzuli *OFFSET-FETCH*, chyba że potrzebne są działania, które nie są obsługiwane przez klauzulę *OFFSET-FETCH*, a można je zrealizować za pomocą *TOP*.

Szybki przegląd funkcji okna

Funkcja okna to funkcja, która dla każdego wiersza w zapytaniu podstawowym wykonuje operacje na oknie (zbiorze) wierszy i oblicza skalarne (pojedyncze) wartości wyników. Okno wierszy definiowane jest przy użyciu klauzuli *OVER*. Funkcje okna mają wiele możliwości i rozwiązują szeroki zakres zagadnień, takich jak wykonywanie obliczeń analizy danych. Funkcje okna należą do standardu SQL, zaś język T-SQL obsługuje pewien podzbiór tej funkcjonalności.

W tym miejscu książki szczegółowe zgłębianie tych funkcji byłoby jednak przedwczesne. Przedstawię teraz tylko pobieżne omówienie tej koncepcji i zaprezentuję ich działanie przy użyciu funkcji okna *ROW_NUMBER*. Później, w rozdziale 7 „Zaawansowane zagadnienia tworzenia zapytań”, przedstawię szczegółowe opisy tych funkcji.

Jak już wspomniałem, funkcja okna działa na zbiorze wierszy określonym przez klauzulę *OVER*. Dla każdego wiersza w wyjściowym zapytaniu klauzula *OVER* prezentuje funkcji podzbiór wierszy uzyskany ze zbioru wyników podstawowego zapytania. Klauzula *OVER* może ograniczać wiersze w oknie przy użyciu klauzuli pomocniczej *PARTITION BY* i może definiować kolejność obliczeń (jeśli ma to znaczenie) przy użyciu klauzuli pomocniczej *ORDER BY* (nie należy jej mylić z klauzulą *ORDER BY* służącą do prezentacji wyników zapytania).

Przeanalizujmy następujący przykład zapytania:

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(PARTITION BY custid
                          ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

* Dosłownie „dopasuj 1 wierszy” (przyp. tłum.).

Zapytanie to generuje następujące wyniki:

orderid	custid	val	rownum
-----	-----	-----	-----
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3
10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1
...			

(830 row(s) affected)

Funkcja *ROW_NUMBER* przypisuje niepowtarzające się, kolejne, inkrementowane liczby całkowite wierszom wynikowym wewnątrz relatywnej partycji, opierając się na wskazanej kolejności. Klauzula *OVER* w przykładowej funkcji dzieli okno według atrybutu *custid*, dzięki czemu nie powtarzają się numery wierszy dla każdego klienta. Klauzula *OVER* definiuje także kolejność w oknie według atrybutu *val*, tak więc kolejne numery wierszy są wewnątrz partycji inkrementowane w oparciu o atrybut *val*.

Zwróćmy uwagę, że funkcja *ROW_NUMBER* wewnątrz każdej partycji musi generować unikatowe wartości, co oznacza, że nawet jeśli wartość określająca kolejność nie zwiększa się, numer wiersza musi być zwiększony. Z tego względu, jeśli lista *ORDER BY* funkcji *ROW_NUMBER* nie jest unikatowa, jak w poprzednim przykładzie, zapytanie nie jest deterministyczne, czyli możliwy jest więcej niż jeden prawidłowy wynik. Jeśli chcemy, by obliczenia numerów wierszy były deterministyczne, do listy *ORDER BY* musimy dodać elementy, które zapewnią takie działanie. Na przykład do listy *ORDER BY* możemy dodać atrybut *orderid* jako kryterium rozstrzygania, by obliczenia numeru wierszy były deterministyczne.

Klauzuli *ORDER BY* wyspecyfikowanej w klauzuli *OVER* nie należy mylić z klauzulą określającą kolejność prezentacji; nie zmienia ona relacyjnego charakteru wyników. Jeśli zapytanie nie zawiera opisanej wcześniej klauzuli prezentacji *ORDER BY*, nie mamy żadnych gwarancji co do kolejności wierszy prezentowanych w wynikach wyjściowych. Jeśli trzeba zapewnić kolejność prezentowania wyników, musimy dodać klauzulę prezentacji *ORDER BY*, jak w pokazanym przykładzie.

Zwróćmy uwagę, że wyrażenia na liście *SELECT* są przetwarzane przed klauzulą *DISTINCT* (jeśli taka istnieje). Zasada ta stosuje się do wyrażeń opartych na funkcjach okna, które występują w liście *SELECT*. Znaczenie tego faktu wyjaśnione zostanie w rozdziale 7.

Jako podsumowanie, poniższa lista przedstawia kolejność logiczną przetwarzania wszystkich do tej pory omawianych klauzul:

- *FROM*
- *WHERE*
- *GROUP BY*
- *HAVING*
- *SELECT*
 - Wyrażenia i tworzenie aliasów dla wynikowych kolumn
 - *DISTINCT*
- *ORDER BY*
 - *TOP / OFFSET-FETCH*

Predykaty i operatory

Język T-SQL zawiera elementy, w których można specyfikować predykaty – na przykład filtry zapytania, takie jak *WHERE* i *HAVING*, ograniczenia *CHECK* i inne. Przypomnijmy, że predykaty są wyrażeniami logicznymi, które przyjmują wartość *TRUE*, *FALSE* lub *UNKNOWN*. Predykaty możemy łączyć przy użyciu operatorów logicznych, takich jak *AND* (koniunkcja, logiczne i) i *OR* (alternatywa, logiczne lub). Możemy w nich również stosować innego rodzaju operatory, na przykład operatory porównań, takich jak równość, większe lub mniejsze, a także bardziej zaawansowane.

Do predykatów zaawansowanych występujących w języku T-SQL należą *IN*, *BETWEEN-AND* oraz *LIKE*. Predykat *IN* pozwala nam sprawdzić, czy wartość, czyli wyrażenie skalarne, jest równe co najmniej jednemu elementowi wskazanego zbioru. Na przykład poniższe zapytanie zwraca zamówienia, dla których identyfikatory zamówienia (order ID) są równe 10248, 10249 lub 10250.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderid IN(10248, 10249, 10250);
```

Predykat *BETWEEN-AND* pozwala sprawdzać, czy wartość znajduje się w wyspecyfikowanym przedziale, definiowanym przez dwie wartości graniczne (włącznie z nimi). Na przykład poniższe zapytanie zwraca wszystkie zamówienia z zakresu od 10300 do 10310 (włącznie).

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderid BETWEEN 10300 AND 10310;
```


Predykat *LIKE* pozwala sprawdzać, czy wartość ciągu znaków zgodna jest z określonym wzorcem. Przykładowo, poniżej pokazane zapytanie zwraca pracowników, dla których nazwisko rozpoczyna się od litery D.

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname LIKE N'D%';
```

W dalszej części rozdziału będziemy szczegółowo omawiać wzorce zgodności i predykat *LIKE*.

Zwróćmy uwagę na użycie litery N jako prefiksu ciągu 'D%'; prefiks taki oznacza *National* i jest używany do zaznaczenia, że ciąg znaków jest typu Unicode (*NCHAR* lub *NVARCHAR*), a nie typem zwykłych znaków (*CHAR* lub *VARCHAR*). Ponieważ typ danych atrybutu *lastname* to *NVARCHAR(40)*, litera N jest używana jako prefiks ciągu. W dalszej części, w podrozdziale „Stosowanie danych znakowych” szczegółowo omówione jest używanie ciągów znakowych.

Język T-SQL obsługuje następujące operatory porównań: =, >, <, >=, <=, <>, !=, !>, !<, z których trzy ostatnie (negacje) nie należą do standardu. Ponieważ niestandardowe operatory posiadają standardowe alternatywy (na przykład <> zamiast !=), zalecam unikania stosowania niestandardowych operatorów. Na przykład poniższe zapytanie zwraca wszystkie zamówienia złożone 1 stycznia 2016 lub później.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20160101';
```

Jeśli zachodzi potrzeba połączenia wyrażeń logicznych, możemy stosować operatory logiczne *OR* i *AND*. Jeśli chcemy zanegować wyrażenie, posługujemy się operatorem *NOT*. Na przykład poniższe zapytanie zwraca zamówienia złożone 1 stycznia 2016 lub później, które były obsługiwane przez pracowników o identyfikatorach 1, 3 lub 5.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20160101'
      AND empid IN(1, 3, 5);
```

Język T-SQL obsługuje cztery oczywiste operatory arytmetyczne: +, -, * i /, a także operator % (modulo), który zwraca pozostałość z dzielenia całkowitego. Na przykład poniższe zapytanie oblicza wartość netto jako wynik arytmetycznego działania na atrybutach *quantity* (ilość), *unitprice* (cena jednostkowa) i *discount* (rabat).

```
SELECT orderid, productid, qty, unitprice, discount,
      qty * unitprice * (1 - discount) AS val
FROM Sales.OrderDetails;
```

Zauważmy, że typ danych wyrażenia skalarnego korzystających z dwóch operandów jest określany w języku T-SQL przez operand wyższy w sensie pierwszeństwa typu danych. Jeśli operandy są tego samego typu danych, również wynik wyrażenia jest tego

typu. Na przykład w wyniku dzielenia dwóch liczb całkowitych (*INT*) otrzymujemy liczbę całkowitą. Wyrażenie $5/2$ zwraca wartość całkowitą 2 typu *INT*, a nie 2,5 typu *NUMERIC*. W przypadku stosowania stałych nie stanowi to problemu, ponieważ zawsze możemy określić te wartości jako liczbowe z miejscem dziesiętnym. Jeśli jednak działania dotyczą na przykład dwóch kolumn z liczbami całkowitym, na przykład *col1/col2*, trzeba przypisać operandom odpowiednie typy, jeśli chcemy, by wynik obliczeń był liczbą ułamkową: $CAST(col1 AS NUMERIC(12, 2)) / CAST(col2 AS NUMERIC(12, 2))$. Definicja typu danych *NUMERIC(12, 2)* zawiera dwa elementy: *precision* (dokładność), w tym przypadku 12, oraz *scale* (podziałka), w tym przypadku 2, co oznacza, że typ ten zawiera łącznie 12 cyfr, z których 2 umieszczone są po przecinku dziesiętnym.

Jeśli dwa operandy mają różne typy, ten o niższym priorytecie jest „awansowany” do typu danych o wyższym priorytecie. Na przykład dla wyrażenia $5/2.0$, pierwszy operand to *INT*, a drugi to *NUMERIC*. Ponieważ operand *NUMERIC* ma wyższy priorytet niż *INT*, przed operacją arytmetyczną operand *INT* 5 jest po prostu przekształcany do postaci operandu *NUMERIC* 5.0, a w rezultacie otrzymamy 2.5.

Informacje o kolejności priorytetów dla różnych typów znaleźć można w dokumentacji SQL Server Books Online w rozdziale „Data Type Precedence”.

Jeśli w tym samym wyrażeniu występuje wiele operatorów, system SQL Server analizuje je w oparciu o reguły pierwszeństwa operatorów. Poniższa lista opisuje pierwszeństwo operatorów, od najwyższego do najniższego:

1. () (nawiasy)
2. * (mnożenie), / (dzielenie), % (modulo)
3. + (liczba dodatnia), - (liczba ujemna), + (dodawanie),
+ (konkatenacja ciągów), - (odejmowanie)
4. =, >, <, >=, <=, <>, !=, !=, != (operatory porównania)
5. NOT
6. AND
7. BETWEEN, IN, LIKE, OR
8. = (przypisanie)

Na przykład w poniższym zapytaniu operator *AND* ma pierwszeństwo w odniesieniu do operatora *OR*.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE
custid = 1 AND empid IN(1, 3, 5) OR custid = 85 AND empid IN(2, 4, 6);
```

Zapytanie zwraca zamówienia, które albo „złożył klient 1 i były obsługiwane przez pracowników 1, 3 bądź 5”, albo „złożył klient 85 i były obsługiwane przez pracowników 2,4 bądź 6”.

Nawiasy mają najwyższy priorytet, tak więc dają nam pełną kontrolę. Przez wzgląd na inne osoby przeglądające lub utrzymujące nasz kod oraz ze względu na jego czytelność dobrą praktyką jest stosowanie nawiasów, nawet jeśli nie są wymagane. Ta sama uwaga dotyczy wcięć. Na przykład poniższe zapytanie logicznie jest ekwiwalentem poprzedniego, ale jest znacznie bardziej przejrzyste.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE
    ( custid = 1
      AND empid IN(1, 3, 5) )
  OR
    ( custid = 85
      AND empid IN(2, 4, 6) );
```

Używanie nawiasów do wymuszania pierwszeństwa operatorów logicznych jest analogiczne do stosowania nawiasów dla operatorów arytmetycznych. Na przykład w poniższym wyrażeniu bez nawiasów mnożenie poprzedza dodawanie.

```
SELECT 10 + 2 * 3;
```

Dlatego też wyrażenie zwraca wartość 16. Możemy zastosować nawiasy, by dodawanie było wykonywane w pierwszej kolejności.

```
SELECT (10 + 2) * 3;
```

Tym razem wyrażenie zwróci wartość 36.

Wyrażenia CASE

CASE to wyrażenie skalarne, które zwraca wartość w oparciu o logikę warunkową. Zwróćmy uwagę, że CASE jest wyrażeniem, a nie instrukcją; z tego względu nie pozwala ono na kontrolowanie przepływu operacji lub wykonywanie działań w oparciu o logikę warunkową. Zamiast tego w oparciu o logikę warunkową zwracana jest wartość tego wyrażenia. Ponieważ CASE to wyrażenie skalarne, dopuszczane jest wszędzie tam, gdzie można stosować wyrażenia skalarne, na przykład w klauzulach *SELECT*, *WHERE*, *HAVING* i *ORDER BY* czy ograniczeniach *CHECK*.

Wyrażenie CASE ma dwa formaty: prosty (*simple*) oraz z przeszukiwaniem (*searched*). Prosty format pozwala porównywać jedną wartość lub wyrażenie skalarne z listą możliwych wartości i zwraca wartość dla pierwszej zgodności. Jeśli żadna wartość na liście nie jest równa porównywanej wartości, wyrażenie CASE zwraca wartość, która występuje w klauzuli *ELSE* (jeśli taka istnieje). Jeśli wyrażenie CASE nie zawiera klauzuli *ELSE*, domyślnie będzie to *ELSE NULL*.

Poniższe zapytanie użyte w odniesieniu do tabeli *Production.Products* w klauzuli *SELECT* stosuje wyrażenie CASE do utworzenia opisu wartości kolumny *categoryid*.

```

SELECT productid, productname, categoryid,
       CASE categoryid
         WHEN 1 THEN 'Beverages'
         WHEN 2 THEN 'Condiments'
         WHEN 3 THEN 'Confections'
         WHEN 4 THEN 'Dairy Products'
         WHEN 5 THEN 'Grains/Cereals'
         WHEN 6 THEN 'Meat/Poultry'
         WHEN 7 THEN 'Produce'
         WHEN 8 THEN 'Seafood'
       ELSE 'Unknown Category'
END AS categoryname
FROM Production.Products;

```

Zapytanie to generuje następujące dane wyjściowe (pokazane w skróconej postaci):

productid	productname	categoryid	categoryname
1	Product HHYDP	1	Beverages
2	Product RECZE	1	Beverages
3	Product IMEHJ	2	Condiments
4	Product KSBRM	2	Condiments
5	Product EPEIM	2	Condiments
6	Product VAIIV	2	Condiments
7	Product HMLNI	7	Produce
8	Product WVJFP	2	Condiments
9	Product AOZBW	6	Meat/Poultry
10	Product YHXGE	8	Seafood
...			

(77 row(s) affected)

Zauważmy, że to zapytanie jest tylko przykładem zastosowania wyrażenia *CASE*. O ile zbiór kategorii nie jest bardzo mały i niezmienny, lepszym rozwiązaniem będzie prawdopodobnie (przykładowo) utrzymywanie kategorii produktów w tabeli i połączenie tej tabeli z tabelą *Products*, jeśli zachodzi potrzeba uzyskania opisów kategorii. W istocie baza danych *TSQLV4* zawiera tego rodzaju tabelę *Categories*.

Prosty format wyrażenia *CASE* zawiera pojedynczą wartość lub wyrażenie umieszczane zaraz za słowem kluczowym *CASE*, które jest z porównywaną listą możliwych wartości klauzul *WHEN*. Format z wyszukiwaniem jest bardziej elastyczny, ponieważ pozwala na specyfikowanie predykatów, czyli wyrażeń logicznych, w klauzulach *WHEN*, co nie ogranicza nas do porównywania przy użyciu równości. Wyrażenie *CASE* z wyszukiwaniem zwraca wartość w klauzuli *THEN*, która jest skojarzona z pierwszym wyrażeniem logicznym *WHEN*, mającym wartość *TRUE*. Jeśli żadne z wyrażeń *WHEN* nie przyjmuje wartości *TRUE*, wyrażenie *CASE* zwraca wartość, która występuje w klauzuli *ELSE* (lub *NULL*, jeśli klauzula *ELSE* nie została wyspecyfikowana). Na przykład poniższe zapytanie tworzy opis kategorii w oparciu o to, czy wartość jest mniejsza od 1000,00, znajduje się pomiędzy 1000,00 a 3000,00 lub jest większa niż 3000,00.

```

SELECT orderid, custid, val,
CASE
    WHEN val < 1000.00 THEN 'Less than 1000'
    WHEN val BETWEEN 1000.00 AND 3000.00 THEN 'Between 1000 and 3000'
    WHEN val > 3000.00 THEN 'More than 3000'
    ELSE 'Unknown'
END AS valuecategory
FROM Sales.OrderValues;

```

Dane wyjściowe zapytania są następujące:

orderid	custid	val	valuecategory
10248	85	440.00	Less than 1000
10249	79	1863.40	Between 1000 and 3000
10250	34	1552.60	Between 1000 and 3000
10251	84	654.06	Less than 1000
10252	76	3597.90	More than 3000
10253	34	1444.80	Between 1000 and 3000
10254	14	556.62	Less than 1000
10255	68	2490.50	Between 1000 and 3000
10256	88	517.80	Less than 1000
10257	35	1119.90	Between 1000 and 3000
...			

(830 row(s) affected)

Można zauważyć, że każde proste wyrażenie CASE można przekształcić do formatu z wyszukiwaniem, natomiast odwrotne przekształcenie nie zawsze jest możliwe.

Język T-SQL obsługuje kilka funkcji, które można traktować jako skrócone wersje wyrażenia CASE: są to *ISNULL*, *COALESCE*, *IIF* oraz *CHOOSE*. Zwróćmy uwagę, że spośród nich tylko funkcja *COALESCE* należy do standardu. Ponadto funkcje *IIF* i *CHOOSE* są dostępne dopiero od wersji SQL Server 2012.

Funkcja *ISNULL* akceptuje dwa argumenty jako dane wejściowe i zwraca pierwszy, który nie jest *NULL* lub zwraca *NULL*, jeśli oba są *NULL*. Na przykład funkcja *ISNULL(col1, '')* zwraca wartość *col1*, jeśli to nie jest *NULL*, a w przeciwnym razie zwraca pusty ciąg. Funkcja *COALESCE* jest podobna, ale obsługuje dwa lub więcej argumentów i zwraca pierwszy, który nie przyjmuje *NULL* lub zwraca *NULL*, jeśli wszystkie argumenty są *NULL*. Jak już wspominałem wcześniej, jeśli mamy wybór, ogólnym zaleceniem jest stosowanie funkcji standardowych, tak więc zalecane jest stosowanie funkcji *COALESCE*, a nie funkcji *ISNULL*.

Niestandardowe funkcje *IIF* i *CHOOSE* zostały dodane w wersji SQL Server 2012, by ułatwić migrację z produktu Microsoft Access. Funkcja *IIF(<wyrażenie_logiczne>, <expr1>, <expr2>)* zwraca *expr1*, jeśli *wyrażenie_logiczne* ma wartość *TRUE*; w przeciwnym razie zwraca *expr2*. Na przykład wyrażenie *IIF(col2 <> 0, col2/col1, NULL)* zwraca wynik operacji *col2/col1*, jeśli *col1* nie jest zerem, w przeciwnym razie zwraca wartość *NULL*. Funkcja *CHOOSE(<index>, <expr1>, <expr2>, ..., <exprn>)* zwraca wyrażenie z listy w wyspecyfikowanym indeksie. Na przykład wyrażenie *CHOOSE(3, col1,*

col2, *col3*) zwraca wartość *col3*. Rzecz jasna, rzeczywiste wyrażenia stosujące funkcję *CHOOSE* są przeważnie bardziej dynamiczne – na przykład są zależne od danych wprowadzanych przez użytkownika.

Do tej pory użyliśmy tylko kilku przykładów do zapoznania się z wyrażeniem *CASE* i funkcjami, które mogą być traktowane jako skróty wyrażenia *CASE*. Chociaż na podstawie tych przykładów może się to nie wydawać oczywiste, wyrażenie *CASE* jest bardzo sprawnym i przydatnym elementem języka.

Znacznik *NULL*

Jak wyjaśniłem w rozdziale 1 „Podstawy zapytań i programowania T-SQL”, język SQL obsługuje znacznik *NULL* do przedstawiania brakujących wartości i stosuje logikę trójwartościową, co oznacza, że predykaty mogą przyjmować wartości *TRUE*, *FALSE* lub *UNKNOWN*. W tym względzie język T-SQL jest zgodny ze standardem. Znaczniki *NULL* i wartość *UNKNOWN* w języku SQL może wprowadzać nieporozumienia, ponieważ ludzie są przyzwyczajeni do myślenia w kategoriach logiki dwuwartościowej (*TRUE* i *FALSE*), a zamęt powiększa się jeszcze bardziej, ponieważ różne elementy języka SQL w różny sposób traktują znaczniki *NULL* i wartość *UNKNOWN*.

Rozpocznijmy od logiki trójwartościowej. Wyrażenie logiczne dotyczące tylko istniejących, czyli rzeczywiście występujących wartości, oceniane jest jako prawda lub fałsz (*TRUE* lub *FALSE*). Jeśli jednak wyrażenie logiczne odnosi się do brakującej wartości, przyjmuje wartość *nieznany* (*UNKNOWN*). Przeanalizujmy dla przykładu predykat *salary > 0* (pensja > 0). Jeśli pensja ma wartość 1000, wyrażenie oceniane jest jako *TRUE*. Jeśli pensja ma wartość -1000, wyrażenie oceniane jest jako *FALSE*. Kiedy pensja ma wartość *NULL*, wyrażenie oceniane jest jako *UNKNOWN*.

Język SQL traktuje wartości *TRUE* i *FALSE* w intuicyjny i zapewne oczekiwany sposób. Jeśli przykładowo predykat *salary > 0* pojawi się w filtrze zapytania (na przykład w klauzuli *WHERE* lub *HAVING*), zwracane są wiersze lub grupy, dla których wyrażenie przyjmuje wartość *TRUE*, natomiast odrzucane są wiersze, dla których wyrażenie przyjmuje wartość *FALSE*. Podobnie, jeśli predykat *salary > 0* występuje w tabeli w ograniczeniu *CHECK*, akceptowane są polecenia *INSERT* lub *UPDATE* dla wszystkich wierszy, dla których wyrażenie przyjmuje wartość *TRUE*, natomiast odrzucane są te wiersze, dla których wyrażenie przyjmuje wartość *FALSE*.

Różne elementy języka SQL rozmaicie traktują wartość *UNKNOWN* (i niekoniecznie w oczekiwany sposób, przynajmniej dla niektórych osób). Sposób działania stosowany przez SQL dla filtrów zapytania to „akceptowanie *TRUE*”, co oznacza, że zarówno wartość *FALSE*, jak i *UNKNOWN* są odrzucane. Jednak inaczej jest w przypadku ograniczeń *CHECK*, gdzie stosowana jest zasada „odrzuć *FALSE*”, co oznacza, że akceptowana jest zarówno wartość *TRUE*, jak i *UNKNOWN*. Gdyby język SQL używał dwuwartościowej logiki predykatów, nie byłoby różnic pomiędzy definicjami „akceptowanie *TRUE*” i „odrzuć *FALSE*”. Jednak w przypadku logiki trójwartościowej

definicja „akceptowanie *TRUE*” odrzuca wartość *UNKNOWN* (akceptuje tylko *TRUE*, czyli odrzucane są obie wartości *FALSE* i *UNKNOWN*), natomiast definicja „odrzucać *FALSE*” akceptuje wartość *UNKNOWN* (odrzucać jest tylko wartość *FALSE*, a tym samym akceptowane są obie wartości *TRUE* i *UNKNOWN*). W przypadku predykatu z poprzedniego przykładu *salary > 0*, wartość *NULL* spowoduje ocenienie wyrażenia jako *UNKNOWN*. Jeśli predykat ten występuje w klauzuli *WHERE* zapytania, wiersz z wartością *NULL* dla atrybutu *salary* zostanie odrzucony. Jeśli jednak predykat ten pojawi się w tabeli w ograniczeniu *CHECK*, wiersz z wartością *NULL* dla atrybutu *salary* zostanie zaakceptowany.

Zaskakującym aspektem wartości *UNKNOWN* jest to, że po jej zanegowaniu nadal otrzymujemy wartość *UNKNOWN*. Na przykład dla predykatu *NOT (salary > 0)*, jeśli pensja to *NULL*, wyrażenie *salary > 0* oceniane jest jako *UNKNOWN*, a wyrażenie *NOT UNKNOWN* pozostaje *UNKNOWN*.

Może się wydać dziwne, że wyrażenie porównania dwóch znaczników *NULL* (*NULL = NULL*) oceniane jest jako *UNKNOWN*. Z punktu widzenia języka SQL uzasadnieniem tego jest to, że znacznik *NULL* reprezentuje brakującą lub nieznaną wartość i tak naprawdę nie możemy powiedzieć, czy jedna nieznaną wartość jest równa drugiej nieznaną wartości. Dlatego w języku SQL udostępniono predykaty *IS NULL* oraz *IS NOT NULL*, których należy używać zamiast *= NULL* i *<> NULL*.

By przybliżyć te kwestie, pokażę wspomniane powyżej aspekty na przykładzie trójwartościowej logiki predykatu. Tabela *Sales.Customers* ma trzy atrybuty nazwane *country*, *region* i *city*, które przechowują informacje o lokalizacji klienta. Każda lokalizacja zawiera informacje o kraju i mieście. Niektóre z nich mają informacje o regionie (na przykład *country*: USA, *region*: WA, *city*: Seattle), natomiast dla niektórych lokalizacji brak jest elementu *region*, ale nie miałby on zastosowania (na przykład *country*: UK, *region*: NULL, *city*: London). Przeanalizujemy następujące zapytanie, które próbuje zwrócić wszystkich klientów, dla których atrybut *region* ma wartość WA.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = N'WA';
```

Wykonanie powyższego zapytania generuje następujące dane wyjściowe:

custid	country	region	city
-----	-----	-----	-----
43	USA	WA	Walla Walla
82	USA	WA	Kirkland
89	USA	WA	Seattle

Pośród 91 wierszy tabeli *Customers* zapytanie zwraca trzy wiersze, w których atrybut *region* ma wartość WA. Zapytanie nie zwraca wierszy, w których wartość atrybutu *region* jest podana i różni się od WA (predykat oceniony jest jako *FALSE*), ale nie zwraca także tych wierszy, w których atrybut *region* to *NULL* (predykat oceniony jest jako *UNKNOWN*).

Pokazane poniżej zapytanie próbuje uzyskać wszystkich klientów, dla których atrybut *region* ma wartość różną niż WA.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region <> N'WA';
```

Zapytanie to generuje następujące wyniki:

custid	country	region	city
10	Canada	BC	Tsawassen
15	Brazil	SP	Sao Paulo
21	Brazil	SP	Sao Paulo
31	Brazil	SP	Campinas
32	USA	OR	Eugene
33	Venezuela	DF	Caracas
34	Brazil	RJ	Rio de Janeiro
35	Venezuela	Táchira	San Cristóbal
36	USA	OR	Elgin
37	Ireland	Co. Cork	Cork
38	UK	Isle of Wight	Cowes
42	Canada	BC	Vancouver
45	USA	CA	San Francisco
46	Venezuela	Lara	Barquisimeto
47	Venezuela	Nueva Esparta	I. de Margarita
48	USA	OR	Portland
51	Canada	Québec	Montréal
55	USA	AK	Anchorage
61	Brazil	RJ	Rio de Janeiro
62	Brazil	SP	Sao Paulo
65	USA	NM	Albuquerque
67	Brazil	RJ	Rio de Janeiro
71	USA	ID	Boise
75	USA	WY	Lander
77	USA	OR	Portland
78	USA	MT	Butte
81	Brazil	SP	Sao Paulo
88	Brazil	SP	Resende

(28 row(s) affected)

Wiedząc, że tabela zawiera 91 wierszy, moglibyśmy oczekiwać uzyskania 88 wierszy (91 wierszy tabeli minus 3 zwrócone w poprzednim zapytaniu), tak więc dziwne może się wydawać uzyskanie tylko 28 wierszy. Pamiętajmy jednak, że w filtrze zapytania definicja brzmi „akceptowanie *TRUE*”, co oznacza, że odrzucane są zarówno wiersze, dla których wyrażenie logiczne ma wartość *FALSE*, jak i te, dla których ma wartość *UNKNOWN*. Tak więc zapytanie to zwraca wiersze, w których dla atrybutu *region* wartość jest określona i wartość ta jest inna niż WA. Zapytanie nie zwraca wierszy, dla których atrybut *region* ma wartość WA, ale nie zwraca też wierszy, dla których atrybut *region* to *NULL*. Te same wyniki otrzymamy, jeśli użyjemy predykatu *NOT* (*region* = N'WA'), ponieważ w wierszach, dla których wartość atrybutu *region* wynosi *NULL*,

wyrażenie `region = N'WA'` oceniane jest jako `UNKNOWN`, ale wyrażenie `NOT (region = N'WA')` również oceniane jest jako `UNKNOWN`.

Aby uzyskać wszystkie wiersze, dla których atrybut `region` ma wartość `NULL`, nie możemy użyć predykatu `region = NULL`, ponieważ wyrażenie to przyjmuje wartość `UNKNOWN` we wszystkich wierszach – zarówno w tych, dla których wartość jest wpisana, jak i w tych, dla których brak tej wartości (jest `NULL`). Poniższe zapytanie zwraca pusty zbiór.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = NULL;
```

```
custid      country      region      city
-----
(0 row(s) affected)
```

Zamiast tego powinniśmy użyć predykatu `IS NULL`.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region IS NULL;
```

Zapytanie to zwraca następujące wyniki (pokazane w skróconej postaci):

```
custid      country      region      city
-----
1           Germany      NULL         Berlin
2           Mexico        NULL         México D.F.
3           Mexico        NULL         México D.F.
4           UK             NULL         London
5           Sweden        NULL         Luleå
6           Germany      NULL         Mannheim
7           France        NULL         Strasbourg
8           Spain         NULL         Madrid
9           France        NULL         Marseille
11          UK             NULL         London
...
(60 row(s) affected)
```

Jeśli chcemy, by zapytanie zwróciło wszystkie wiersze, dla których atrybut `region` nie ma wartości `WA`, czyli te, dla których wartość jest wprowadzona i są różne od `WA` wraz z tymi, dla których brakuje informacji o regionie, trzeba jawnie dołączyć test dla znaczników `NULL`, jak w poniższym przykładzie:

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region <> N'WA'
OR region IS NULL;
```

Zapytanie to generuje następujące dane wyjściowe (pokazane w skróconej postaci):

custid	country	region	city
1	Germany	NULL	Berlin
2	Mexico	NULL	México D.F.
3	Mexico	NULL	México D.F.
4	UK	NULL	London
5	Sweden	NULL	Luleå
6	Germany	NULL	Mannheim
7	France	NULL	Strasbourg
8	Spain	NULL	Madrid
9	France	NULL	Marseille
10	Canada	BC	Tsawassen
...			

(88 row(s) affected)

Język SQL traktuje znaczniki *NULL* w sposób niespójny także przy porównaniach i sortowaniu. Niektóre elementy języka traktują dwa znaczniki *NULL* jako równe sobie, a inne jako różniące się od siebie.

Na przykład w przypadku grupowania i sortowania dwa znaczniki *NULL* są uważane za równe sobie. Tak więc klauzula *GROUP BY* umieści wszystkie znaczniki *NULL* w jednej grupie, tak samo jak występujące wartości, a klauzula *ORDER BY* sortuje razem wszystkie znaczniki *NULL*. Standard SQL pozostawia do decyzji implementacji produktu określenie kolejności, czy znaczniki *NULL* są sortowane przed występującymi wartościami, czy po nich. Język T-SQL sortuje znaczniki *NULL* przed występującymi wartościami.

Jak wspomniałem, filtry zapytania stosują definicję „akceptowanie *TRUE*”. Wyrażenie porównywania dwóch znaczników *NULL* daje wartość *UNKNOWN*; z tego względu takie wiersze zostają odfiltrowane.

Przy wymuszaniu ograniczenia *UNIQUE* standard SQL traktuje znaczniki *NULL* jako różne od siebie (dopuszczając wiele znaczników *NULL*). Odwrotnie jest w języku T-SQL, gdzie ograniczenie *UNIQUE* traktuje dwa znaczniki *NULL* jako równe sobie (dopuszczając tylko jeden znacznik *NULL*, jeśli ograniczenie zdefiniowane jest dla pojedynczej kolumny).

Trzeba pamiętać o braku spójności w traktowaniu przez SQL wartości *UNKNOWN* i *NULL* oraz o potencjalnych błędach logicznych – w każdym opracowywanym zapytaniu powinniśmy mieć jasno sprecyzowane działanie wobec znaczników *NULL* i trójwartościowej logiki. Jeśli domyślne traktowanie nie jest tym, co chcemy, trzeba to zmienić wprost; inaczej musimy upewnić się, że działanie domyślne jest w rzeczywistości tym, co zaplanowaliśmy.

Operacje jednoczesne – „all-at-once”

Język SQL obsługuje pojęcie operacji *all-at-once* (wszystko na raz), co oznacza, że wszystkie wyrażenia występujące w tej samej logicznej fazie przetwarzania zapytania logicznie oceniane są w tym samym momencie.

Pojęcie to wyjaśnia, dlaczego na przykład nie możemy odnosić się do aliasów kolumn przypisanych w klauzuli *SELECT* wewnątrz tej samej klauzuli *SELECT*. Przeanalizujmy następujące zapytanie:

```
SELECT
  orderid,
  YEAR(orderdate) AS orderyear,
  orderyear + 1 AS nextyear
FROM Sales.Orders;
```

Odniesienie do aliasu kolumny *orderyear* w trzecim wyrażeniu listy *SELECT* jest nieprawidłowe, chociaż wyrażenie odwołania pojawia się „po” wyrażeniu, w którym alias jest przypisywany. Przyczyną tego stanu rzeczy jest to, że z punktu widzenia logiki nie istnieje kolejność oceniania wyrażań na liście *SELECT* – lista jest (nieuporządkowanym) zbiorem wyrażań. Konceptyjnie wszystkie wyrażenia listy *SELECT* są oceniane w tym samym momencie. Z tego względu próba wykonania tego zapytania wygeneruje następujący komunikat o błędzie:

```
Msg 207, Level 16, State 1, Line 4
Invalid column name 'orderyear'.
(Nieprawidłowa nazwa kolumny 'orderyear')
```

Oto inny przykład znaczenia operacji jednoczesnych: Załóżmy, że mamy tabelę nazwaną *T1* z dwiema kolumnami zawierającymi liczby całkowite, nazwanymi *col1* i *col2* oraz że chcemy uzyskać wszystkie wiersze, dla których wyrażenie *col2/col1* jest większe niż 2. Ponieważ mogą istnieć wiersze tabeli, dla których *col1* ma wartość zero, trzeba zapewnić, że w takich sytuacjach operacja dzielenia nie będzie wykonywana – inaczej zapytanie wygeneruje błąd: dzielenie przez zero. Tak więc piszemy zapytanie o następującej postaci:

```
SELECT col1, col2
FROM dbo.T1
WHERE col1 <> 0 AND col2/col1 > 2;
```

Mogłoby się wydawać, że SQL Server ocenia wyrażenia od lewej do prawej i że jeśli wyrażenie *col1 <> 0* ocenione zostanie jako *FALSE*, SQL Server przejdzie dalej; nie będzie „zawracał sobie głowy” ocenianiem wyrażenia *col2/col1 > 2*, ponieważ w tym momencie już wiadomo, że całe wyrażenie ma wartość *FALSE*. Tak więc moglibyśmy sądzić, że to zapytanie nigdy nie wygeneruje błędu dotyczącego operacji dzielenia przez zero. Okazuje się jednak, że jest inaczej.

System SQL Server obsługuje takie skracanie działań, jednak ze względu na koncepcję operacji jednoczesnych (*all-at-once*) w standardzie SQL, może w dowolnej kolejności przetwarzać wyrażenia w klauzuli *WHERE*. SQL Server podejmuje decyzje takie jak powyższa, bazując na ocenie kosztów, co oznacza, że zazwyczaj pierwsze szacowane jest wyrażenie, które jest tańsze. Jak widać, jeśli optymalizator SQL Server zdecyduje przetworzyć w pierwszej kolejności wyrażenie $col2/col1 > 2$, zapytanie może nie działać, ponieważ wygeneruje błąd dzielenia przez zero.

Tego błędu możemy uniknąć kilkoma metodami. Na przykład kolejność oceniania klauzul *WHEN* w wyrażeniu *CASE* jest zagwarantowana, co pozwala poprawić zapytanie w następujący sposób:

```
SELECT col1, col2
FROM dbo.T1
WHERE
  CASE
    WHEN col1 = 0 THEN 'no' -- lub 'yes' jeśli wiersz powinien być zwrócony
    WHEN col2/col1 > 2 THEN 'yes'
    ELSE 'no'
  END = 'yes';
```

W wierszach, gdzie *col1* jest równe zero, pierwsza klauzula *WHEN* przyjmuje wartość *TRUE* i wyrażenie *CASE* zwraca ciąg 'no' (można zastąpić 'no' ciągiem 'yes', jeśli chcemy zwracać wiersze, dla których *col1* jest równe zero). Tylko jeśli pierwsze wyrażenie *CASE* nie przyjmuje wartości *TRUE* – co oznacza, że wartość *col1* nie jest równa zero – wykonywana jest druga klauzula *WHEN* sprawdzania, czy prawdziwe jest wyrażenie $col2/col1 > 2$ (czy przyjmuje wartość *TRUE*). Jeśli tak się dzieje, wyrażenie *CASE* zwraca ciąg 'yes'. We wszystkich innych sytuacjach wyrażenie *CASE* zwraca ciąg 'no'. Predykat klauzuli *WHERE* zwraca wartość *TRUE*, tylko jeśli wynik wyrażenia *CASE* jest równy ciągowi 'yes'. Oznacza to, że nigdy nie nastąpi próba dzielenia przez zero.

To rozwiązanie jest trochę zawile. W tym konkretnym przypadku możemy zastosować poprawkę matematyczną, która całkowicie eliminuje dzielenie.

```
SELECT col1, col2
FROM dbo.T1
WHERE (col1 > 0 AND col2 > 2*col1) OR (col1 < 0 AND col2 < 2*col1);
```

Przykład ten dołączyłem, by wyjaśnić unikatowe i istotne pojęcie operacji jednoczesnych, a także by podkreślić fakt, że SQL Server gwarantuje kolejność przetwarzania klauzul *WHEN* w wyrażeniu *CASE*.

Stosowanie danych znakowych

W tym podrozdziale przedstawię korzystanie z danych znakowych, w tym typy danych, opcje sortowania, operatory i funkcje oraz dopasowywanie do wzorca.

Typy danych

System SQL Server obsługuje dwa rodzaje typów danych tekstowych – zwykłe i Unicode. Typy danych zwykłych obejmują *CHAR* i *VARCHAR*, a typy danych Unicode obejmują *NCHAR* i *NVARCHAR*. Dla każdego znaku zwykłego używany jest jeden bajt pamięci, natomiast dane Unicode dla jednego znaku wymagają dwóch bajtów, a w przypadku, kiedy potrzebna jest para zastępcza, wymagane są cztery bajty (szczegółowe wyjaśnienia dotyczące par zastępczych zawiera dokument <https://msdn.microsoft.com/en-us/library/windows/desktop/dd374069>). Jeśli dla kolumny wybieramy zwykły typ znaków, oprócz angielskiego możemy używać tylko jednego języka*. Obsługa języka dla kolumny jest określona przez efektywną opcję sortowania dla kolumny, co zostanie skrótowo opisane nieco dalej. W przypadku typów danych Unicode obsługiwanych jest wiele języków. Tak więc, jeśli przechowujemy dane znakowe różnych języków, trzeba zapewnić stosowanie typów znakowych Unicode, a nie zwykłych.

Dwa rodzaje typów danych znakowych różnią się także w sposobie wyrażania literałów. Kiedy przedstawiamy literał zwykłego znaku, po prostu stosujemy pojedyncze znaki cudzysłowu: 'To jest literał ciągu znaków zwykłych'. Kiedy wyrażamy literał znaków Unicode, trzeba użyć znaku N (oznacza National) jako prefiksu: N'To jest literał ciągu znaków Unicode'.

Dowolny typ danych bez elementu *VAR* (*CHAR*, *NCHAR*) w swojej nazwie ma ustaloną długość, co oznacza, że system SQL Server rezerwuje miejsce w wierszu w oparciu o zdefiniowany rozmiar kolumny, a nie w oparciu o rzeczywistą długość ciągu znaków. Na przykład jeśli kolumna zdefiniowana jest jako *CHAR*(25), system SQL Server rezerwuje miejsce dla 25 znaków w wierszu, niezależnie od długości przechowywanego ciągu znaków. Ponieważ nie jest wymagane żadne powiększanie wiersza, jeśli ciągi są rozszerzane, typy danych o stałej długości lepiej nadają się dla systemów wykonujących liczne operacje zapisu. Ponieważ jednak ciągi o ustalonej długości nieoptymalnie zużywają przestrzeń dyskową, większe koszty ponosimy przy odczycie danych.

Typ danych z elementem *VAR* (*VARCHAR*, *NVARCHAR*) w swojej nazwie ma zmienną długość, co oznacza, że system SQL Server zużywa tyle miejsca w pamięci w wierszu,

* Warto zauważyć, że istnieje dodatkowe ograniczenie – przy użyciu kodów jednobajtowych można zapisać tylko znaki języków posługujących się alfabetami dających się przedstawić w postaci tzw. stron kodowych (należą do nich wszystkie odmiany alfabetu łacińskiego, a także grecki, cyrylica, hebrajski czy arabski). Języki wykorzystujące szerszy zestaw znaków, takie jak japoński lub chiński zawsze wymagają co najmniej dwóch bajtów do przedstawienia jednego znaku, zatem nie można ich zmieścić w typach *CHAR* ani *VARCHAR*.

ile jest wymagane do przechowywania znaków występujących w ciągu znaków, plus dwa dodatkowe bajty dla danych offsetu. Na przykład kiedy kolumna jest zdefiniowana jako typ `VARCHAR(25)`, maksymalnie obsługiwanych jest 25 znaków, ale w praktyce ilość miejsca określona jest przez rzeczywistą liczbę znaków ciągu. Ponieważ zajętość pamięci dla tych typów danych jest mniejsza niż w przypadku typów danych o ustalonej długości, szybsze są operacje odczytu. Jednakże operacje aktualizowania mogą wiązać się z poszerzeniem wiersza; może to skutkować przenoszeniem danych poza bieżącą stronę. Z tego względu aktualizacje danych posiadających typy danych o zmiennej długości są mniej efektywne, niż aktualizacje typów danych o ustalonej długości.



UWAGA Jeśli używana jest kompresja, wymagania magazynowe się zmieniają. Informacje na temat kompresji znaleźć można w artykule „Data Compression” w dokumentacji SQL Server Books Online pod adresem <http://msdn.microsoft.com/en-us/library/cc280449.aspx>.

Typy danych o zmiennej długości można definiować za pomocą określenia `MAX`, a nie za pomocą maksymalnej liczby znaków. Kiedy kolumna zdefiniowana jest za pomocą wyrażenia `MAX`, dowolna wartość o rozmiarze poniżej pewnego progu (domyślnie 8000 bajtów) może być przechowywana wewnątrz wiersza (o ile mieści się w wierszu). Dowolna wartość o rozmiarze przekraczającym ten próg jest przechowywana zewnętrznie (w odniesieniu do wiersza) jako wielki obiekt binarny (LOB).

W dalszej części, w podrozdziale „Zapytania dotyczące metadanych”, zamieszczono wyjaśnienia, jak można pobrać informacje metadanych obiektów bazy danych, wliczając w to typy danych kolumn.

Opcje sortowania (*collation*)

Collation (sortowanie*) to właściwość danych znakowych, która obejmuje kilka aspektów, w tym obsługiwany język, porządek znaków, rozróżnianie wielkości liter, rozróżnianie znaków diakrytycznych i inne. Aby uzyskać zestaw obsługiwanych właściwości *collation* wraz z ich opisami, w odniesieniu do tabeli możemy wykonać kwerendę za pomocą funkcji `fn_helpcollations`, jak w poniższym przykładzie:

```
SELECT name, description
FROM sys.fn_helpcollations();
```

Poniższa lista opisuje sortowanie `Latin1_General_CI_AS`:

* W języku polskim nie istnieje jednoznaczny odpowiednik angielskiego wyrazu „collation”, który określa całość opcji dotyczących sortowania tekstów, takich jak uwzględnianie znaków diakrytycznych, traktowanie cyfr, znaków specjalnych, wielkich i małych liter itp. Używany termin „sortowanie” jest słabym zamiennikiem, ale nie jest dostępny lepszy odpowiednik (przyp. tłum.).

- **Latin1_General** Użyta jest strona kodowa 1252 (obsługuje znaki języka angielskiego i niemieckiego, a także znaki używane przez większość krajów Europy Zachodniej).
- **Sortowanie słownikowe** Sortowanie i porównywanie znaków opiera się na kolejności słownikowej (A i $a < B$ i b).
Można zauważyć, że kolejność słownikowa jest używana, jeśli żadna inna kolejność nie jest wprost zdefiniowana, ponieważ jest to ustawienie domyślne. Bardziej dokładnie, w nazwie sortowania nie występuje wprost element *BIN*. Gdyby element *BIN* był użyty w nazwie, oznaczałoby to, że sortowanie i porównywanie znaków będzie opierać się na reprezentacji binarnej znaków ($A < B < a < b$).
- **CI** Dla danych nie jest rozróżniana wielkość liter ($a = A$).
- **AS** Dla danych rozróżniane są znaki diakrytyczne ($\grave{a} < \grave{a}$).

W przypadku implementacji systemu SQL Server dla siedziby właściwość *collation* może być definiowana na czterech różnych poziomach: instancji, bazy danych, kolumny i wyrażenia. Poziom efektywny to najniższy zdefiniowany i on zostanie użyty. W Azure SQL Database właściwość *collation* wskazywana jest na poziomie bazy danych, kolumny i wyrażenia. Istnieje też kilka dodatkowych aspektów sortowania w zawartych bazach danych. Szczegółowe objaśnienia zawiera artykuł <https://msdn.microsoft.com/en-GB/library/fff929080.aspx>.

Sortowanie instancji określane jest jako w trakcie instalacji programu i określa parametr *collation* dla wszystkich systemowych baz danych i jest używane jako ustawienie domyślne dla baz danych użytkownika.

Podczas tworzenia bazy danych użytkownika możemy jawnie wyspecyfikować *collation* bazy danych przy użyciu klauzuli *COLLATE*. Jeśli brak takiej specyfikacji, wybierane jest sortowanie domyślne.

Sortowanie bazy danych określa opcje sortowania metadanych obiektów w bazie danych i jest używane jako domyślne dla kolumn tabel użytkownika. Należy podkreślić znaczenie tego faktu, że *collation* bazy danych określa *collation* dla metadanych, wliczając w to nazwy obiektów i kolumn. Jeśli przykładowo *collation* bazy danych określa, że nie jest rozróżniana wielkość liter, nie będziemy mogli w tym samym schemacie utworzyć dwóch tabel nazwanych *T1* i *t1*.

Przy użyciu klauzuli *COLLATE* możemy wprost określić sortowanie kolumny jako część jej definicji. Jeśli brak takiej specyfikacji, zastosowane zostaną opcje sortowania bazy danych.

W konkretnym wyrażeniu można zmienić *collation* przy użyciu klauzuli *COLLATE*. Na przykład w środowisku, w którym nie jest rozróżniana wielkość liter, poniższe zapytanie wykona porównanie bez rozróżniania wielkości liter:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = N'davis';
```

Zapytanie to zwraca wiersz dla Sary Davis, choć nie jest zgodna wielkość liter, ponieważ w efekcie końcowym sprawdzanie wielkości liter nie jest aktywne.

empid	firstname	lastname
1	Sara	Davis

Jeśli chcemy, by w filtrze rozróżniana była wielkość liter, nawet jeśli *collation* kolumny definiuje nierozróżnianie wielkości liter, to w tym celu możemy zmienić opcje sortowania wyrażenia:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname COLLATE Latin1_General_CS_AS = N'davis';
```

Tym razem zapytanie zwraca pusty zbiór, ponieważ przy porównywaniu z uwzględnieniem wielkości liter nie zostaje znaleziony żaden wynik.

Identyfikatory ujęte w cudzysłów

W standardzie SQL pojedynczy cudzysłów używany jest do ograniczania literałów ciągów znaków (na przykład 'litera'), natomiast znaki podwójnego cudzysłowu służą do oddzielania nieregularnych identyfikatorów, takich jak nazwy kolumn czy tabel, które zawierają spacje lub rozpoczynają się od cyfry (na przykład "nieregularny identyfikator"). W systemie SQL Server dostępne jest ustawienie nazwane *QUOTED_IDENTIFIER*, które steruje znaczeniem podwójnego znaku cudzysłowu. Ustawienie to można zastosować albo na poziomie bazy danych przy użyciu polecenia *ALTER DATABASE*, albo na poziomie sesji przy użyciu polecenia *SET*. Kiedy ustawienie jest włączone, działanie jest zgodne ze standardem SQL, co oznacza, że podwójne cudzysłowy są używane do separacji identyfikatorów. Jeśli ustawienie jest wyłączone, działanie jest niestandardowe, a podwójne znaki cudzysłowu używane są do ograniczania ciągów znaków literałów. Zdecydowanie zaleca się stosowanie standardowego działania (ustawienie włączone). Większość interfejsów baz danych, w tym OLE DB i ODBC, domyślnie włącza to ustawienie.



WSKAZÓWKĄ Jako rozwiązanie alternatywne do użycia podwójnych znaków cudzysłowu do separacji identyfikatorów, SQL Server obsługuje także nawiasy kwadratowe (na przykład [Irregular Identifier]).

Jeśli chcemy użyć znaku pojedynczego cudzysłowu jako części ciągu, trzeba wpisać dwa kolejne znaki pojedynczego cudzysłowu. Na przykład aby przedstawić literał *abc'de*, napiszemy 'abc'de'.

Operatory i funkcje

W tym podrozdziale omówiono połączenia ciągów oraz funkcje, które operują na ciągach. Dla operacji łączenia ciągów język T-SQL udostępnia operator + i funkcję *CONCAT*. Dla innych operacji na ciągach znaków język T-SQL udostępnia kilka funkcji, a mianowicie *SUBSTRING*, *LEFT*, *RIGHT*, *LEN*, *DATALENGTH*, *CHARINDEX*, *PATINDEX*, *REPLACE*, *REPLICATE*, *STUFF*, *UPPER*, *LOWER*, *RTRIM*, *LTRIM*, *FORMAT*, *COMPRESS*, *DECOMPRESS* oraz *STRING_SPLIT*. W kolejnych podrozdziałach omówione zostaną te często używane operatory i funkcje. Warto podkreślić, że nie istnieje coś takiego, jak biblioteka funkcji należąca do standardu SQL – wszystkie one są specyficzne dla określonej implementacji.

Łączenie ciągów (operator znak plus [+] i funkcja *CONCAT*)

Do łączenia (konkatenacji) ciągów znaków język T-SQL udostępnia operator w postaci znaku plus (+) oraz funkcję *CONCAT* (od wersji SQL Server 2012). Na przykład poniższe zapytanie do tabeli *Employees* generuje kolumnę wyników w postaci imienia i nazwiska poprzez połączenie *firstname* (imię), spacji i *lastname* (nazwisko).

```
SELECT empid, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

Wykonanie zapytania daje następujące wyniki:

empid	fullname
1	Sara Davis
2	Don Funk
3	Judy Lew
4	Yael Peled
5	Sven Buck
6	Paul Suurs
7	Russell King
8	Maria Cameron
9	Patricia Doyle

Standard SQL określa, że połączenia ze znacznikiem *NULL* powinny dawać wartość *NULL* – jest to też domyślne działanie systemu SQL Server. Przeanalizujmy dla przykładu zapytanie do tabeli *Customers* (listing 2-7).

LISTING 2-7 Zapytanie ilustrujące łączenie ciągów

```
SELECT custid, country, region, city,
       country + N',' + region + N',' + city AS location
FROM Sales.Customers;
```

Niektóre wiersze w tabeli *Customers* w kolumnie *region* mają znacznik *NULL*. Dla tych wierszy SQL Server zwraca domyślnie *NULL* w kolumnie wyników dotyczącej lokalizacji.

custid	country	region	city	location
1	Germany	NULL	Berlin	NULL
2	Mexico	NULL	México D.F.	NULL
3	Mexico	NULL	México D.F.	NULL
4	UK	NULL	London	NULL
5	Sweden	NULL	Luleå	NULL
6	Germany	NULL	Mannheim	NULL
7	France	NULL	Strasbourg	NULL
8	Spain	NULL	Madrid	NULL
9	France	NULL	Marseille	NULL
10	Canada	BC	Tsawassen	Canada,BC,Tsawassen
11	UK	NULL	London	NULL
12	Argentina	NULL	Buenos Aires	NULL
13	Mexico	NULL	México D.F.	NULL
14	Switzerland	NULL	Bern	NULL
15	Brazil	SP	Sao Paulo	Brazil,SP,Sao Paulo
16	UK	NULL	London	NULL
17	Germany	NULL	Aachen	NULL
18	France	NULL	Nantes	NULL
19	UK	NULL	London	NULL
20	Austria	NULL	Graz	NULL

...

(91 row(s) affected)

Aby potraktować *NULL* jako ciąg pusty – a mówiąc ściślej, aby *zastąpić* znacznik *NULL* pustym ciągiem – możemy użyć funkcji *COALESCE*. Funkcja ta akceptuje listę wartości wejściowych i zwraca pierwszą, która nie jest znacznikiem *NULL*. Poniżej pokazano metodę zmodyfikowania zapytania z listingu 2-7 w celu programowego zastąpienia znaczników *NULL* ciągami pustymi.

```
SELECT custid, country, region, city,
country + COALESCE( N',' + region, N'' ) + N',' + city AS location
FROM Sales.Customers;
```

T-SQL od wersji SQL Server 2012 udostępnia funkcję nazwaną *CONCAT*, która akceptuje listę danych wejściowych (rozdzielaną przecinkami) i łączy je, automatycznie zastępując znaczniki *NULL* pustymi ciągami. Na przykład wyrażenie *CONCAT('a', NULL, 'b')* zwraca ciąg 'ab'. Poniższy kod pokazuje, jak zastosować funkcję *CONCAT* do łączenia elementów lokalizacji klienta, zastępując jednocześnie znaczniki *NULL* pustymi ciągami.

```
SELECT custid, country, region, city,
CONCAT(country, N',' + region, N',' + city) AS location
FROM Sales.Customers;
```

Funkcja *SUBSTRING*

Funkcja *SUBSTRING* wydobywa część ciągu z całego ciągu.

Składnia:

```
SUBSTRING(string, start, length)
```

Funkcja ta operuje na ciągu wejściowym (*string*) i wydobywa podciąg (fragment ciągu), rozpoczynając od położenia początkowego (*start*), którego liczbę znaków określa parametr (*length*). Na przykład poniższy kod zwraca dane wyjściowe 'abc'.

```
SELECT SUBSTRING('abcde', 1, 3);
```

Warto zauważyć, że znaki w ciągu numerowane są od 1, inaczej niż w niektórych innych językach programowania, w których numerowanie zaczyna się od 0. Jeśli wartość trzeciego argumentu wychodzi poza koniec ciągu wejściowego, funkcja zwraca wszystko do końca bez zgłaszania błędu. Działanie takie może być przydatne, jeśli chcemy uzyskać wszystko od pewnego punktu do końca ciągu – po prostu specyfikujemy maksymalną długość typu danych lub wartość reprezentującą pełną długość ciągu wejściowego.

Funkcje *LEFT* i *RIGHT*

Funkcje *LEFT* i *RIGHT* to skrócone funkcje *SUBSTRING*, które zwracają żadaną liczbę znaków, licząc od lewego lub prawego końca ciągu wejściowego odpowiednio.

Składnia:

```
LEFT(string, n), RIGHT(string, n)
```

Pierwszy argument (*string*) to ciąg, na którym funkcja operuje. Drugi argument (*n*) to liczba znaków wydobywanych z ciągu wejściowego, licząc od lewego lub prawego końca ciągu. Na przykład poniższy kod zwraca 'cde'.

```
SELECT RIGHT('abcde', 3);
```

Funkcje *LEN* i *DATALENGTH*

Funkcja *LEN* zwraca liczbę znaków ciągu wejściowego.

Składnia:

```
LEN(string)
```

Zwróćmy uwagę, że funkcja zwraca liczbę znaków ciągu wejściowego (*string*), a niekoniecznie liczbę bajtów. W przypadku znaków zwykłych obie liczby są takie same, ponieważ dla każdego znaku wymagany jest jeden bajt w pamięci. W przypadku znaków Unicode dla znaku wymagane są dwa bajty (co najmniej, w większości

przypadków); z tego względu liczba znaków to połowa liczby bajtów. Aby uzyskać liczbę bajtów, trzeba posłużyć się funkcją *DATALENGTH*, a nie funkcją *LEN*. Na przykład poniższy kod zwraca wynik 5.

```
SELECT LEN(N'abcde');
```

Kod pokazany poniżej zwraca wynik 10.

```
SELECT DATALENGTH(N'abcde');
```

Inna różnica pomiędzy funkcjami *LEN* i *DATALENGTH* polega na tym, że pierwsza zwraca wyniki bez spacji końcowych, a druga je uwzględnia.

Funkcja *CHARINDEX*

Funkcja *CHARINDEX* zwraca położenie pierwszego wystąpienia podciągu wewnątrz ciągu.

Składnia:

```
CHARINDEX(substring, string[, start_pos])
```

Funkcja ta zwraca położenie pierwszego argumentu (*substring*) wewnątrz drugiego argumentu (*string*). Opcjonalnie możemy wyspecyfikować trzeci argument (*start_pos*), by wskazać funkcji położenie, od którego ma rozpocząć wyszukiwanie. Jeśli trzeci argument jest pominięty, funkcja rozpoczyna wyszukiwanie od pierwszego znaku. Jeśli podciąg nie zostanie znaleziony, funkcja zwraca 0. Na przykład poniższy kod zwraca pierwsze położenie spacji w ciągu 'Itzik Ben-Gan', tak więc w wyniku pojawi się wartość 6.

```
SELECT CHARINDEX(' ', 'Itzik Ben-Gan');
```

Funkcja *PATINDEX*

Funkcja *PATINDEX* zwraca położenie pierwszego wystąpienia wzorca wewnątrz ciągu.

Składnia:

```
PATINDEX(pattern, string)
```

Argument wzorca (*pattern*) używa podobnych wzorców do tych używanych przez predykat *LIKE* w języku T-SQL. Wzorce i predykat *LIKE* wyjaśnione zostały w dalszej części rozdziału, w podrozdziale „Predykat *LIKE*”. Chociaż jeszcze nie wyjaśniłem sposobu przedstawiania wzorców w języku T-SQL, przytoczę przykład, by zilustrować wyszukiwanie położenia pierwszego wystąpienia cyfry wewnątrz ciągu.

```
SELECT PATINDEX('%[0-9]%', 'abcd123efgh');
```

Kod ten zwraca wartość 5.

Funkcja *REPLACE*

Funkcja *REPLACE* zastępuje wszystkie wystąpienia podciągu innym ciągiem.

Składnia:

```
REPLACE(string, substring1, substring2)
```

W danym ciągu funkcja zastępuje wszystkie wystąpienia podciągu 1 (*substring1*) podciągiem 2 (*substring2*). Na przykład poniższy kod w ciągu wejściowym zastępuje wszystkie wystąpienia dywizu znakiem dwukropka.

```
SELECT REPLACE('1-a 2-b', '-', ':');
```

Po wykonaniu tego kodu pojawi się wynik: '1:a 2:b'.

Funkcji *REPLACE* możemy używać do obliczania liczby wystąpień znaku wewnątrz ciągu. W tym celu zastępujemy wszystkie wystąpienia znaku ciągiem pustym (zero znaków) i obliczamy pierwotną długość ciągu minus nową długość. Na przykład poniższe zapytanie dla każdego pracownika zwraca liczbę określającą, ile razy znak *e* pojawił się w atrybucie *lastname*.

```
SELECT empid, lastname,
       LEN(lastname) - LEN(REPLACE(lastname, 'e', '')) AS numoccur
FROM HR.Employees;
```

Po wykonaniu zapytania otrzymujemy następujący wynik:

empid	lastname	numoccur
-----	-----	-----
5	Buck	0
8	Cameron	1
1	Davis	0
9	Doyle	0
2	Funk	0
7	King	0
3	Lew	1
4	Peled	2
6	Suurs	0

Funkcja *REPLICATE*

Funkcja *REPLICATE* powiela ciąg zadaną liczbę razy.

Składnia:

```
REPLICATE(string, n)
```

Poniższy kod przykładowy trzykrotnie replikuje ciąg 'abc', zwracając w wyniku ciąg 'abcabcabc'.

```
SELECT REPLICATE('abc', 3);
```

Kolejny przykład ilustruje użycie funkcji *REPLICATE* wraz z funkcją *RIGHT* i łączeniem ciągów. Pokazane poniżej zapytanie do tabeli *Production.Suppliers* generuje 10-cio cyfrowy ciąg przedstawiający identyfikator dostawcy (supplier ID) w postaci liczby całkowitej z początkowymi zerami.

```
SELECT supplierid,
       RIGHT(REPLICATE('0', 9) + CAST(supplierid AS VARCHAR(10)), 10) AS
strsupplierid
FROM Production.Suppliers;
```

Wyrażenie to, generując kolumnę wynikową *strsupplierid*, replikuje znak 0 dziewięć razy (tworząc ciąg '000000000') i łączy z przedstawieniem ciągu identyfikatora dostawcy, by utworzyć wynik. Ciąg reprezentujący liczbę całkowitą identyfikatora dostawcy tworzony jest przez funkcję *CAST*, która jest używana do przekształcania typu danych wartości wejściowych. Na koniec z uzyskanego ciągu wyrażenie pobiera 10 znaków licząc od prawej, zwracając ostatecznie dziesięciocyfrowe przedstawienie identyfikatora dostawcy wraz z początkowymi zerami. Poniżej pokazano dane wyjściowe tego zapytania (w skróconej postaci):

supplierid	strsupplierid
29	0000000029
28	0000000028
4	0000000004
21	0000000021
2	0000000002
22	0000000022
14	0000000014
11	0000000011
25	0000000025
...	

(29 row(s) affected)

Zwróćmy uwagę, że od wersji SQL Server 2012 dostępna jest funkcja *FORMAT*, która znacznie łatwiej pozwala realizować takie zadania związane z formatowaniem, choć przy wyższym koszcie wykonania. Funkcja ta opisana jest w dalszej części tego rozdziału.

Funkcja *STUFF*

Funkcja *STUFF* pozwala z ciągu usunąć podciąg i w jego miejsce wstawić nowy podciąg.

Składnia:

```
STUFF(string, pos, delete_length, insertstring)
```

Funkcja ta operuje na ciągu wskazanym jako pierwszy z parametrów wejściowych i usuwa taką liczbę znaków, jak określa to parametr *delete_length*, rozpoczynając usuwanie od znaku, którego położenie określa parametr wejściowy *pos*. Następnie wstawia

ciąg wyspecyfikowany przez parametr *insertstring* w położeniu *pos*. Na przykład poniższy kod przetwarza ciąg string 'xyz', usuwa z niego jeden znak, licząc od drugiego znaku i zamiast niego wstawia podciąg 'abc'.

```
SELECT STUFF('xyz', 2, 1, 'abc');
```

Funkcja generuje następujący wynik: 'xabcz'.

Jeśli tylko chcemy wstawić ciąg i niczego w nim nie usuwać, możemy określić wartość trzeciego argumentu jako 0.

Funkcje *UPPER* i *LOWER*

Funkcje *UPPER* i *LOWER* zwracają ciąg wejściowy (string), w którym wszystkie znaki są zamienione odpowiednio na wielkie (*UPPER*) lub małe (*LOWER*) litery.

Składnia:

```
UPPER(string), LOWER(string)
```

Na przykład poniższy kod zwraca wynik: 'ITZIK BEN-GAN'.

```
SELECT UPPER('Itzik Ben-Gan');
```

Natomiast poniższy kod zwraca: 'itzik ben-gan'.

```
SELECT LOWER('Itzik Ben-Gan');
```

Trzeba zauważyć, że w przypadku znaków diakrytycznych lub akcentowanych lub alfabetów innych niż łaciński poprawne zamienianie liter jest zależne od właściwego ustawienia parametru *Collation*. Jeśli na przykład parametr ten jest ustawiony na *Latin1*, a w przekształcanym tekście wystąpi znak diakrytyczny, taki jak *q* lub *Æ*, znaki te nie zostaną przekształcone. Ograniczenie to nie dotyczy danych Unicode, które są zawsze przekształcane poprawnie, o ile tylko dla wybranego znaku (alfabetu) istnieje rozróżnienie wielkich i małych liter (nie występujące np. w alfabecie chińskim).

Funkcje *RTRIM* i *LTRIM*

Funkcje *RTRIM* i *LTRIM* zwracają ciąg wejściowy (string) z usuniętymi spacjami początkowymi lub końcowymi odpowiednio.

Składnia:

```
RTRIM(string), LTRIM(string)
```

Jeśli chcemy usunąć zarówno początkowe, jak i końcowe spacje, używamy wyniku działania jednej funkcji jako danych wejściowych drugiej. Na przykład poniższy kod usuwa zarówno początkowe, jak i końcowe spacje z ciągu wejściowego, zwracając w wyniku 'abc'.

```
SELECT RTRIM(LTRIM('   abc   '));
```

Funkcja *FORMAT*

Funkcja *FORMAT* pozwala formatować wartości wejściowe (*input*) w postaci ciągu znaków w oparciu o ciąg formatowania (*format_string*) i opcjonalny parametr *culture*, zgodne ze standardem platformy Microsoft .NET.

Składnia:

```
FORMAT(input, format_string, [culture])
```

Istnieje wiele możliwości formatowania danych wejściowych przy użyciu standardowych lub niestandardowych ciągów formatowania. Artykuł MSDN dostępny pod adresem <http://go.microsoft.com/fwlink/?LinkId=211776> przekazuje bardziej szczegółowe informacje. Jednak jako prosty przykład przypomnijmy sobie zawile wyrażenie użyte wcześniej do sformatowania 10-cio cyfrowego ciągu z zerami na początku. Przy użyciu funkcji *FORMAT* to samo zadanie można zrealizować za pomocą niestandardowego ciągu formatu '0000000000' lub przy użyciu standardowego 'd10'. Na przykład poniższy kod zwraca wynik '0000001759'.

```
SELECT FORMAT(1759, '0000000000');
```



UWAGA Funkcja *FORMAT* jest zwykle bardziej kosztowna, niż alternatywne rozwiązania T-SQL, których możemy używać do formatowania wartości. W ogólności należy powstrzymać się od jej stosowania, chyba że jesteśmy gotowi zaakceptować spadek wydajności. W ramach testów wykonałem zapytanie do tabeli zawierającej 1 000 000 wierszy w celu przedstawienia jednej z całkowitoliczbowych kolumn jako 10-cyfrowego ciągu znakowego. Wykonanie zapytania przy użyciu funkcji *FORMAT* wymagało niemal minuty, ale trwało mniej niż sekundę, gdy zastosowałem alternatywną metodę wykorzystującą funkcje *REPLICATE* i *RIGHT*, pokazaną wcześniej w tym rozdziale.

Funkcje *COMPRESS* i *DECOMPRESS*

Funkcje *COMPRESS* i *DECOMPRESS* wykorzystują algorytm GZIP do kompresowania i dekompresowania danych wejściowych, odpowiednio. Obie funkcje zostały wprowadzone w wersji SQL Server 2016.

Składnia

```
COMPRESS(string), DECOMPRESS(string)
```

Funkcja *COMPRESS* przyjmuje ciąg znakowy lub binarny jako wejście i zwraca skompresowaną wartość typu *VARBINARY(MAX)*. Oto przykład użycia funkcji z literałem znakowym jako wejściem:

```
SELECT COMPRESS(N'To jest moje CV. Wyobraź sobie, że jest dużo dłuższe.');
```


Wynikiem będzie wartość binarna zawierająca skompresowaną postać ciągu wejściowego.

Jeśli chcemy przechować dane wejściowe w postaci skompresowanej w kolumnie tabeli, musimy zastosować funkcję *COMPRESS* do danych wejściowych i umieścić wynik w tabeli. Można to zrealizować jako część wyrażenia *INSERT* wstawiającego nowe wiersze do docelowej tabeli (więcej informacji na temat modyfikowania danych zawiera rozdział 8). Dla przykładu założmy, że w naszej bazie danych mamy tabelę o nazwie *EmployeeCVs*, z kolumnami *empid* oraz *cv*. Kolumna *cv* przechowuje skompresowaną postać życiorysu pracownika i jest zdefiniowana jako typ *VARBINARY(MAX)*. Przyjmijmy, że mamy procedurę składowaną o nazwie *AddEmpCV*, przyjmującą parametry wejściowe *@empid* oraz *@cv* (informacje na temat procedur składowanych zawiera rozdział 11). Parametr *@cv* to nieskompresowana postać życiorysu pracownika i jest zdefiniowany jako typ *NVARCHAR(MAX)*. Procedura jest odpowiedzialna za wstawienie nowego wiersza do tabeli wraz ze skompresowaną wersją życiorysu pracownika. Wyrażenie *INSERT* w tej procedurze mogłoby wyglądać następująco:

```
INSERT INTO dbo.EmployeeCVs( empid, cv ) VALUES( @empid, COMPRESS(@cv) );
```

Funkcja *DECOMPRESS* przyjmuje ciąg binarny jako wejście i zwraca zdekompresowaną wartość typu *VARBINARY(MAX)*. Zauważmy, że jeśli oryginalna wartość (przed skompresowaniem) była typu znakowego, konieczne będzie jawne rzutowanie wyniku funkcji *DECOMPRESS* na właściwy typ docelowy. Dla przykładu poniższy kod nie zwróci oryginalnej wartości wejściowej; zamiast tego otrzymamy wartość binarną:

```
SELECT DECOMPRESS(COMPRESS(N'This is my cv. Imagine it was much longer.'));
```

Aby uzyskać oryginalną postać, musimy rzutować ten wynik na docelowy typ ciągu znakowego, jak poniżej:

```
SELECT CAST(
    DECOMPRESS(COMPRESS(N'To jest moje CV. Wyobraź sobie, że jest dużo dłuższe.'))
    AS NVARCHAR(MAX));
```

Rozważmy teraz tabelę *EmployeeCVs* z poprzedniego przykładu. Aby odczytać nieskompresowaną postać życiorysu pracownika, możemy użyć poniższego zapytania (nie należy próbować wykonać tego kodu, gdyż tabela taka nie istnieje w przykładowej bazie danych):

```
SELECT empid, CAST(DECOMPRESS(cv) AS NVARCHAR(MAX)) AS cv
FROM dbo.EmployeeCVs;
```

Funkcja *STRING_SPLIT*

Funkcja tablicowa *STRING_SPLIT* dzieli wejściowy ciąg znakowy zawierający rozdzielaną listę wartości na indywidualne elementy. Funkcja ta została wprowadzona w wersji SQL Server 2016.

Składnia

```
SELECT value FROM STRING_SPLIT(string, separator);
```

W odróżnieniu od opisanych dotąd funkcji operujących na ciągach, które wszystkie są funkcjami skalarnymi (zwracającymi pojedynczą wartość), funkcja *STRING_SPLIT* jest funkcją tablicową (zwracającą tabelę). Jako wejście przyjmuje listę wartości rozdzielanych separatorem oraz sam separator i zwraca tabelę z kolumną typu ciągowego o nazwie *value*, zawierającą indywidualne elementy listy w kolejnych wierszach. Jeśli chcemy, aby elementy były zwracane jako inny typ danych, konieczne jest rzutowanie kolumny *value* na typ docelowy. Na przykład poniższy kod przyjmuje ciąg wejściowy '10248,10249,10250' oraz separator ',' i zwraca tabelę zawierającą poszczególne elementy listy:

```
SELECT CAST(value AS INT) AS myvalue FROM STRING_SPLIT('10248,10249,10250', ',') AS S;
```

W tym przykładzie lista wejściowa zawiera wartości reprezentujące identyfikatory zamówień. Ponieważ identyfikatory powinny być liczbami całkowitymi, zapytanie konwertuje kolumnę *value* na typ danych *INT*. Oto wynik działania tego kodu:

```
myvalue
-----
10248
10249
10250
```

Typowym zastosowaniem tego typu rozdzielania jest przekazanie rozdzielanej jakimś separatorem listy wartości reprezentujących klucze, takie jak identyfikatory zamówień, do procedury składowanej lub funkcji zdefiniowanej przez użytkownika i zwrócenie odpowiednich wierszy z pewnej tabeli, takiej jak *Orders*, które mają pasujące klucze. Można to osiągnąć poprzez złączenie funkcji *STRING_SPLIT* z tabelą docelową i dopasowaniem kluczy z obu stron złączenia.

Predykat *LIKE*

Język T-SQL udostępnia predykat *LIKE*, który pozwala sprawdzać, czy ciąg znaków jest zgodny z wyspecyfikowanym wzorcem. Podobne wzorce używane są przez opisaną wcześniej funkcję *PATINDEX*. W tym podrozdziale pokażę symbole wieloznaczne obsługiwane we wzorcach i ich użycie na przykładach.

Symbol wieloznaczny % (procent)

Znak procentu zastępuje ciąg znaków o dowolnej długości, w tym ciąg pusty. Na przykład poniższe zapytanie zwraca pracowników, których nazwisko rozpoczyna się od litery D.

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'D%';
```

Zapytanie to zwraca następujące dane wyjściowe:

empid	lastname
1	Davis
9	Doyle

Zauważmy, że często możemy posłużyć się funkcjami *SUBSTRING* i *LEFT*, zamiast predykatu *LIKE*, aby uzyskać ten sam rezultat. Wydaje się jednak, że predykat *LIKE* jest lepiej zoptymalizowany – zwłaszcza wtedy, gdy wzór rozpoczyna się ustalonym prefiksem.

Symbol wieloznaczny _ (podkreślenie)

Znak podkreślenia zastępuje dowolny pojedynczy znak. Na przykład poniższe zapytanie zwraca pracowników, których druga litera nazwiska to litera *e*.

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'_e%';
```

Wykonanie zapytania daje następujące dane wyjściowe:

empid	lastname
3	Lew
4	Peled

Symbol wieloznaczny [<lista znaków>]

Nawias kwadratowy z listą znaków (na przykład [ABC]) zastępuje pojedynczy znak, który musi być jednym ze znaków wyspecyfikowanych na liście. Na przykład poniższe zapytanie zwraca listę pracowników, których pierwsza litera nazwiska to A, B lub C.

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[ABC]%';
```

Zapytanie to zwraca następujący wynik:

empid	lastname
5	Buck
8	Cameron

Symbol wieloznaczny [<znak>-<znak>]

Nawias kwadratowy z zakresem znaków (na przykład [A-E]) zastępuje pojedynczy znak, który musi być jednym ze znaków podanego zakresu. Na przykład poniższe zapytanie zwraca listę pracowników, dla których pierwsza litera nazwiska należy do zakresu A do E (włącznie).

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[A-E]%' ;
```

Po wykonaniu zapytania otrzymujemy następujące dane:

empid	lastname
5	Buck
8	Cameron
1	Davis
9	Doyle

Symbol zastępczy [^<lista znaków lub zakres>]

Nawias kwadratowy ze znakiem daszka (^), po którym umieszczona jest lista znaków lub zakres (na przykład [^ABC] lub [^A-E]), reprezentuje pojedynczy znak, który *nie* jest znakiem wyspecyfikowanym za pośrednictwem listy lub zakresu. Na przykład poniższe zapytanie zwraca listę pracowników, dla których pierwsza litera nazwiska nie jest literą z zakresu A do E.

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'^[A-E]%' ;
```

Zapytanie generuje następujące dane wyjściowe:

empid	lastname
2	Funk
7	King
3	Lew
4	Peled
6	Suurs

Znak ucieczki

Jeśli chcemy wyszukać znak, który jest używany jako symbol wieloznaczny (taki jak %, _, [lub]), posługujemy się „znakiem ucieczki” (*escape*). Przed wyszukiwanym znakiem definiujemy znak, dla którego mamy pewność, że nie występuje w danych, jako znak ucieczki i zaraz po wzorcu podajemy słowo kluczowe *ESCAPE*, po którym znajduje się znak ucieczki. Na przykład aby sprawdzić, czy kolumna nazwana *col1* zawiera znak podkreślenia, używamy wyrażenia *col1 LIKE '%!_%' ESCAPE '!'.*

Dla symboli wieloznacznych %, _ i [możemy użyć nawiasów kwadratowych zamiast znaku ucieczki. Na przykład zamiast wyrażenia `col1 LIKE '%!_%' ESCAPE '!'` możemy użyć `col1 LIKE '%[_]%'`.

Posługiwanie się danymi typu daty i czasu

Stosowanie dat i czasu w systemie SQL Server nie jest zadaniem banalnym. Można tu napotkać rozmaite trudności, takie jak tworzenie literałów w sposób niezależny od języka czy rozdzielne używanie dat (dni) i czasu (godzin, minut i sekund).

W tym podrozdziale najpierw omówię typy danych dotyczące czasu, które są obsługiwane w systemie SQL Server, a następnie przedstawię zalecany sposób używania tych typów danych. Na koniec opiszę funkcje obsługujące daty oraz czas.

Typy danych dotyczące daty i czasu

Język T-SQL obsługuje sześć typów danych dla dat i czasu – starsze *DATETIME* i *SMALLDATETIME*, które są dostępne od najwcześniejszych wersji SQL Server, oraz wprowadzone w wersji SQL Server 2008 typy danych *DATE* i *TIME*, a także zespolony typ *DATETIME2*, obsługujący większy zakres dat i zapewniający większą dokładność niż typ danych *DATETIME*, oraz typ *DATETIMEOFFSET*, który dodatkowo zawiera składnik przesunięcia (strefy czasowej). Typy *DATE* i *TIME* reprezentują oddzielnie samą datę lub samą godzinę (odpowiednio). Pozostałe typy reprezentują łącznie datę i czas (liczony od północy). Tabela 2-1 zawiera szczegółowe informacje na temat tych typów danych, w tym wymagania dotyczące pamięci, obsługiwane zakresy dat, dokładność i zalecany format wprowadzania.

TABELA 2-1 Typy danych dotyczące daty i czasu

Typ danych	Rozmiar (bajty)	Obsługiwany zakres dat	Dokładność	Zalecany format wpisu i przykład
<i>DATETIME</i>	8	Od 1 stycznia 1753 do 31 grudnia 9999	3 1/3 ms (1/300 sekundy)	'YYYYMMDD hh:mm:ss.nnn' '20090212 12:30:15.123'
<i>SMALLDATETIME</i>	4	Od 1 stycznia 1900 do 6 czerwca 2079	1 minuta	'YYYYMMDD hh:mm' '20090212 12:30'
<i>DATE</i>	3	Od 1 stycznia 0001 do 31 grudnia 9999	1 dzień	'YYYY-MM-DD' '2009-02-12'
<i>TIME</i>	3-5	nie dotyczy	100 ns	'hh:mm:ss.nnnnnnn' '12:30:15.1234567'

Ciąg dalszy na stronie następnej

TABELA 2-1 Typy danych dotyczące daty i czasu

Typ danych	Rozmiar (bajty)	Obsługiwany zakres dat	Dokładność	Zalecany format wpisu i przykład
<i>DATETIME2</i>	6-8	Od 1 stycznia 0001 do 31 grudnia 9999	100 ns	'YYYY-MM-DD hh:mm:ss.nnnnnnnn' '2009-02-12 12:30:15.1234567'
<i>DATETIMEOFFSET</i>	8-10	Od 1 stycznia 0001 do 31 grudnia 9999	100 ns	'YYYY-MM-DD hh:mm:ss.nnnnnnnn [+ -]hh:mm' '2009-02-12 12:30:15.1234567 +02:00'

Wymagania dotyczące pamięci dla trzech ostatnich typów danych w tabeli 2-1 (*TIME*, *DATETIME2* i *DATETIMEOFFSET*) zależne są od użytej dokładności. Dokładność specyfikowana jest w postaci liczby całkowitej z zakresu od 0 do 7 i reprezentuje wielkość ułamkowej części sekundy (liczbę cyfr po przecinku). Na przykład *TIME(0)* oznacza brak ułamka, czyli dokładność jednosekundową. *TIME(3)* oznacza czas określony z dokładnością jednej milisekundy, a *TIME(7)* oznacza czas z dokładnością 100 nanosekund. Jeśli dokładność nie zostanie wyspecyfikowana, domyślnie system SQL Server zakłada wartość 7 dla wszystkich trzech wspomnianych typów danych. Przy konwertowaniu wartości do typu danych o niższej precyzji dane są zaokrąglane do najbliższej wartości dającej się wyrazić w docelowym typie.

Literały

Jeśli zachodzi potrzeba wyspecyfikowania literału (stałej) typu danych daty i godziny, trzeba uwzględnić kilka kwestii. Po pierwsze, co może zabrzmieć trochę dziwnie, system SQL Server nie zapewnia środków do jawnego wyrażenia literału daty i godziny; zamiast tego pozwala użyć literału innego typu, który można przekształcić – wprost lub niejawnie – na typ danych daty i godziny. Najlepszym rozwiązaniem praktycznym jest stosowanie ciągów znaków do wyrażania wartości daty i godziny, co ilustruje poniższy przykład.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20150212';
```

System SQL Server rozpoznaje napis '20150212' jako ciąg znaków, a nie jako literał daty i godziny, ponieważ jednak wyrażenie stosuje operandy dwóch różnych typów, jeden operand trzeba niejawnie przekształcić na drugi typ. Niejawna konwersja pomiędzy typami opiera się na mechanizmie nazwanym *pierwszeństwo typów danych*. System SQL Server definiuje pierwszeństwo typów danych i zazwyczaj będzie niejawnie przekształcał operand, którego typ danych posiada niższy priorytet, na typ danych o wyższym priorytecie.

W tym przykładzie literal ciągu znaków zostaje przekształcony w typ danych kolumny (*DATETIME*), ponieważ ciągi znaków w kontekście priorytetu typu danych są na niższym poziomie niż typy danych daty i godziny. Reguły przekształcania niejawnego nie zawsze są tak proste i w rzeczywistości różne reguły są stosowane do filtrów i do innych wyrażeń, ale dla potrzeb omawianych zagadnień nie będę komplikować opisu. Kompletny opis priorytetów typów danych znaleźć można w artykule „Data Type Precedence” w dokumentacji SQL Server Books Online.

W poprzednim przykładzie próbowałem pokazać, że przekształcenie niejawne ma miejsce w tle. Przytoczone zapytanie jest logicznym ekwiwalentem kolejnego, które jawnie przeprowadza konwersję ciągów znaków na typ danych *DATETIME*.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = CAST('20150212' AS DATETIME);
```

Warto pamiętać, że niektóre postaci ciągów znaków dla literalów daty i godziny są zależne od języka, co oznacza, że jeśli poddamy je konwersji na typy danych daty i godziny, system SQL Server może je różnie interpretować w zależności od używanych ustawień regionalnych sesji. Każdy login zdefiniowany przez administratora bazy danych ma przypisany do siebie domyślny język i o ile nie zostanie zmieniony wprost, język ten staje się czynnym językiem sesji. Domyślny język sesji możemy zastąpić przy użyciu polecenia *SET LANGUAGE*, ale ogólnie rzecz biorąc, podejście takie nie jest zalecane, ponieważ niektóre aspekty kodu mogą zależeć od domyślnego języka użytkownika.

W tle czynny język sesji definiuje kilka ustawień związanych z danym językiem, a w tym ustawienie nazwane *DATEFORMAT*, które określa, jak system SQL Server interpretuje literały wprowadzane podczas konwersji z typu ciągu znaków na typ daty i godziny. Ustawienie *DATEFORMAT* jest wyrażone jako połączenie znaków *d*, *m* i *y* (dzień, miesiąc, rok). Na przykład ustawienie języka *us_english* określa *DATEFORMAT* jako *mdy*, natomiast ustawienia dla brytyjskiej wersji języka angielskiego określa to ustawienia jako *dmy*. Możemy zastępować ustawienie *DATEFORMAT* w sesji przy użyciu polecenia *SET DATEFORMAT*, ale jak już wspomniano, zmiana ustawień związanych z językiem generalnie nie jest zalecana.

Przeanalizujmy dla przykładu literal '02/12/2015'. System SQL Server interpretuje datę jako 12 lutego 2015 lub jako 2 grudnia 2015 podczas konwersji tego literalu na jeden z następujących typów: *DATETIME*, *DATE*, *DATETIME2* lub *DATETIMEOFFSET*. Czynnikiem determinującym jest czynne ustawienie *LANGUAGE/DATEFORMAT*. W celu zilustrowania różnych interpretacji tego samego literalu ciągu znaków uruchamiamy następujący kod:

```
SET LANGUAGE British;
SELECT CAST('02/12/2015' AS DATETIME);
SET LANGUAGE us_english;
SELECT CAST('02/12/2015' AS DATETIME);
```

Przeglądając się danym wyjściowym możemy zauważyć, że literal jest odmiennie interpretowany w różnych środowiskach.

Ustawienie języka zmienione na British:

2015-12-02

Ustawienie języka zmienione na us_english:

2015-02-12

Ustawienie *LANGUAGE/DATEFORMAT* wpływa jedynie na sposób interpretowania wprowadzanych wartości; ustawienia te nie mają wpływu na format używany do prezentowania danych wyjściowych, który jest określony przez interfejs bazy danych używany przez narzędzie klienckie (takie jak ODBC), a nie przez ustawienie *LANGUAGE/DATEFORMAT*. Na przykład OLEDB i ODBC prezentują wartości *DATETIME* w formacie 'YYYY-MM-DD hh:mm:ss.nnn'.

Ponieważ napisany kod może być używany przez międzynarodowych użytkowników przy różnych ustawieniach języka dla loginu, zasadnicze znaczenie ma zrozumienie, że niektóre formaty literalów są zależne od języka. Zdecydowanie zalecane jest pisanie literalów w sposób niezależny od języka. Formaty obojętne w kontekście języka są zawsze interpretowane przez system SQL Server w ten sam sposób i nie wpływają na nie ustawienia związane z językiem. W tabeli 2-2 przedstawiono formaty literalów, które są neutralne językowo dla wszystkich typów dat i godzin.

TABELA 2-2 Formaty typu danych daty i godziny neutralne językowo

Typ danych	Format	Przykłady
<i>DATETIME</i>	'YYYYMMDD hh:mm:ss.nnn'	'20090212 12:30:15.123'
	'YYYY-MM-DDThh:mm:ss.nnn'	'2009-02-12T12:30:15.123'
	'YYYYMMDD'	'20090212'
<i>SMALLDATETIME</i>	'YYYYMMDD hh:mm'	'20090212 12:30'
	'YYYY-MM-DDThh:mm'	'2009-02-12T12:30'
	'YYYYMMDD'	'20090212'
<i>DATE</i>	'YYYYMMDD'	'20090212'
	'YYYY-MM-DD'	'2009-02-12'
<i>DATETIME2</i>	'YYYYMMDD hh:mm:ss.nnnnnnn'	'20090212 12:30:15.1234567'
	'YYYY-MM-DD hh:mm:ss.nnnnnnn'	'2009-02-12 12:30:15.1234567'
	'YYYY-MM-DDThh:mm:ss.nnnnnnn'	'2009-02-12T12:30:15.1234567'
	'YYYYMMDD'	'20090212'
	'YYYY-MM-DD'	'2009-02-12'

TABELA 2-2 Formaty typu danych daty i godziny neutralne językowo

Typ danych	Format	Przykłady
<i>DATETIMEOFFSET</i>	'YYYYMMDD hh:mm:ss.nnnnnnnn [+ -] hh:mm'	'20090212 12:30:15.1234567 +02:00'
	'YYYY-MM-DD hh:mm:ss.nnnnnnnn [+ -] hh:mm'	'2009-02-12 12:30:15.1234567 +02:00'
	'YYYYMMDD'	'20090212'
	'YYYY-MM-DD'	'2009-02-12'
<i>TIME</i>	'hh:mm:ss.nnnnnnnn'	'12:30:15.1234567'

W tabeli 2-2 warto zwrócić uwagę na kilka spraw. W przypadku wszystkich typów zawierających oba składniki, datę i czas, jeśli w literale nie wyspecyfikujemy części dotyczącej czasu, SQL Server przyjmie północ. Jeśli nie określimy przesunięcia strefy czasowej, SQL Server założy 00:00 (czas UTC). Istotne jest również to, że formaty 'YYYY-MM-DD' i 'YYYY-MM-DD hh:mm:...' są zależne od języka podczas konwertowania do typu *DATETIME* lub *SMALLDATETIME*, a są obojętne dla języka podczas przekształcania do typów *DATE*, *DATETIME2* i *DATETIMEOFFSET*.

Na przykład w poniższym kodzie zauważymy, że ustawienie języka nie wpływa na sposób interpretowania literału podczas konwersji do typu *DATETIME*, jeśli przedstawiony jest w formacie 'YYYYMMDD'.

```
SET LANGUAGE British;
SELECT CAST('20150212' AS DATETIME);
SET LANGUAGE us_english;
SELECT CAST('20150212' AS DATETIME);
```

Dane wyjściowe pokazują, że literał został w obu przypadkach zinterpretowany jako 12 lutego 2015.

Zmiana ustawienia języka na British:

```
2015-02-12
```

Zmiana ustawienia języka na us_english:

```
2015-02-12
```

Prawdopodobnie nie położyłem wystarczającego nacisku, by przekonać, że najlepszym rozwiązaniem praktycznym jest stosowanie formatów neutralnych językowo, takich jak 'YYYYMMDD', ponieważ takie formaty są interpretowane w ten sam sposób niezależnie od ustawień *LANGUAGE/DATEFORMAT*.

Jeśli jednak do przedstawienia literałów koniecznie chcemy użyć formatu zależnego od języka, mamy dwie możliwości. Jedną jest użycie funkcji *CONVERT* do bezpośredniego przekształcenia literału ciągu znakowego na wymagany typ danych, specyfikując w trzecim argumencie liczbę reprezentującą użyty styl. W dokumentacji SQL Server

Books Online, w artykule „The CAST and CONVERT Functions” zamieszczona jest tabela ze wszystkimi numerami stylów i formatami, które reprezentują. Jeśli przykładowo chcemy wyspecyfikować literał '02/12/2015' za pomocą formatu mm/dd/yyyy, użyjemy numeru stylu 101, jak w poniższym kodzie.

```
SELECT CONVERT(DATETIME, '02/12/2015', 101);
```

Literał będzie interpretowany jako 12 lutego 2015 niezależnie od czynnych ustawień języka. Jeśli chcemy użyć formatu dd/mm/yyyy, posłużymy się numerem stylu 103.

```
SELECT CONVERT(DATETIME, '02/12/2015', 103);
```

W tej sytuacji literał jest interpretowany jako 2 grudnia 2015.

Drugą możliwością jest użycie funkcji *PARSE*, dostępnej od wersji SQL Server 2012. Funkcja ta pozwala przeanalizować wartość jako żądany typ i wskazać właściwość *culture*. Na przykład poniżej pokazany kod jest ekwiwalentem użycia funkcji *CONVERT* ze stylem 101 (US English).

```
SELECT PARSE('02/12/2015' AS DATETIME USING 'en-US');
```

Poniższy kod jest równoważny użyciu funkcji *CONVERT* ze stylem 103 (British English):

```
SELECT PARSE('02/12/2015' AS DATETIME USING 'en-GB');
```



UWAGA Użycie funkcji *PARSE* jest znacząco bardziej kosztowne, niż funkcji *CONVERT*. Z tego względu raczej należy powstrzymać się przed jej stosowaniem.

Rozdzielne stosowanie daty i czasu

Jeśli potrzebujemy pracować tylko z datami lub tylko z czasem (godziną), zalecane jest stosowanie typów danych *DATE* oraz *TIME* odpowiednio. Stosowanie się do tej wskazówki może stać się wyzwaniem, jeśli potrzebujemy ograniczyć się tylko do stosowania starszych typów danych *DATETIME* i *SMALLDATETIME*, na przykład ze względu na kompatybilność ze starszymi systemami. Problem polega na tym, że te przestarzałe typy zawierają zarówno składowe daty, jak i czasu. Najlepsze praktyki zalecają, aby w sytuacji, gdy potrzebujemy jedynie dat, zapisywać datę z północą (0:00:00) jako składową czasową. W przypadku, gdy potrzebne są nam tylko godziny i minuty, zapisujemy je jako datę bazową 1 stycznia 1900.

Aby zademonstrować oddzielne posługiwanie się datą i czasem, wykorzystam tabelę o nazwie *Sales.Orders2*, która zawiera kolumnę *orderdate* typu *DATETIME*. Wykonanie poniższego kodu tworzy tabelę *Sales.Orders2*, kopiując dane z tabeli *Sales.Orders* i rzutując źródłową kolumnę *orderdate* o typie *DATE* na typ *DATETIME*:

```
DROP TABLE IF EXISTS Sales.Orders2;
```

```
SELECT orderid, custid, empid, CAST(orderdate AS DATETIME) AS orderdate
INTO Sales.Orders2
FROM Sales.Orders;
```

Wyrażenie *SELECT INTO* nie pojawiło się dotąd w tej książce. Jego szczegółowy opis znajduje się w rozdziale 8, „Modyfikowanie danych”.

Jak wspomniałem, kolumna *orderdate* w tabeli *Sales.Orders2* ma typ danych *DATETIME*, ale ponieważ zawiera w rzeczywistości tylko składową daty, wszystkie wartości jako składową czasową zawierają północ. Tak więc, gdy potrzebujemy odfiltrować tylko zamówienia dla konkretnej daty, nie musimy używać zakresu wartości, ale możemy posłużyć się operatorem równości, jak poniżej:

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders2
WHERE orderdate = '20160212';
```

Podczas konwersji literału ciągu znakowego na typ *DATETIME* system SQL Server dla składnika czasu przyjmuje północ, jeśli czas nie jest wyspecyfikowany. Ponieważ wszystkie wartości w kolumnie *orderdate* zostały zapisane z zerem jako składnikiem czasu, zwrócone zostaną wszystkie zamówienia złożone o wskazanej dacie. Zwróćmy uwagę, że możemy używać ograniczenia *CHECK*, by zapewnić, że dla składnika czasu użyta została tylko północ (wartość 0).

```
ALTER TABLE Sales.Orders2
ADD CONSTRAINT CHK_Orders2_orderdate
CHECK( CONVERT(CHAR(12), orderdate, 114) = '00:00:00:000' );
```

Funkcja *CONVERT* wydobywa tylko część czasową wartości *orderdate* jako ciąg znakowy używający stylu 114. Ograniczenie *CHECK* weryfikuje, że ciąg ten reprezentuje północ.

Jeśli składnik czasu zawiera inne wartości (różne od zera), musimy stosować filtr zakresu, jak w poniższym przykładzie.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20150212'
AND orderdate < '20150213';
```

Jeśli chcemy posługiwać się tylko danymi czasu przy użyciu starszych typów danych, możemy przechowywać wszystkie wartości przy użyciu daty bazowej 1 stycznia 1900. Podczas konwersji literału ciągu znakowego zawierającego tylko składnik czasu na typ *DATETIME* lub *SMALLDATETIME* system SQL Server założy, że data jest datą bazową. Dla przykładu uruchomimy następujący kod:

```
SELECT CAST('12:30:15.123' AS DATETIME);
```

Polecenie generuje następujący rezultat:

```
-----
1900-01-01 12:30:15.123
```

Załóżmy, że mamy tabelę z kolumną nazwaną *tm*, której typ danych to *DATETIME* i że wszystkie dane przechowujemy przy użyciu daty bazowej. I ponownie, możemy to wymusić stosując ograniczenie *CHECK*. Aby zwrócić wszystkie wiersze, dla których wartość czasu wynosi 12:30:15.123, posłużymy się filtrem *WHERE tm = '12:30:15.123'*. Ponieważ składnik daty nie został wyspecyfikowany, podczas niejawnej konwersji ciągu znakowego na typ danych *DATETIME* system SQL Server założy, że data jest datą bazową.

Jeśli korzystamy tylko z dat lub tylko z czasu, ale wartości wejściowe zawierają zarówno składnik daty, jak i czasu, trzeba zastosować pewne operacje na wartościach wejściowych, by „wyzerować” nieistotną część. Tak więc, ustawiamy składnik czasu za pomocą wartości północ, jeśli chcemy korzystać tylko z dat lub ustawiamy składnik daty za pomocą daty bazowej, jeśli chcemy korzystać wyłącznie ze składnika czasu. W podrozdziale „Funkcje daty i godziny” znaleźć można krótkie wyjaśnienie, jak zrealizować te zadania.

Filtrowanie zakresów danych

Jeśli zachodzi potrzeba filtrowania zakresu dat, na przykład cały rok czy miesiąc, naturalne wydaje się użycie takich funkcji, jak *YEAR* i *MONTH*. Na przykład poniższe zapytanie zwraca wszystkie zamówienia złożone w roku 2015.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE YEAR(orderdate) = 2015;
```

Trzeba jednak zdawać sobie sprawę, że w większości przypadków, kiedy stosujemy operacje dla przefiltrowanych kolumn, SQL Server nie może użyć indeksu w wydajny sposób. Prawdopodobnie niełatwo to zrozumieć bez pewnej wiedzy podstawowej na temat działania indeksów i wydajności, co wykracza poza zakres tej książki, ale na razie wystarczy zapamiętać to jako ogólną zasadę: aby można było sprawnie wykonywać indeks, trzeba poprawić predykat tak, by nie zawierał operacji na przefiltrowanych kolumnach, jak w poniższym kodzie:

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20150101' AND orderdate < '20160101';
```

Podobnie, zamiast stosowania funkcji filtrowania zamówień złożonych w danym miesiącu, jak w poniższym kodzie:

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE YEAR(orderdate) = 2015 AND MONTH(orderdate) = 2;
```

stosujemy filtr zakresu:

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20150201' AND orderdate < '20150301';
```

Funkcje daty i godziny

W tym podrozdziale opisane zostaną funkcje wykonujące operacje na typach danych daty i czasu, wliczając w to typy *GETDATE*, *CURRENT_TIMESTAMP*, *GETUTCDATE*, *SYSDATETIME*, *SYSUTCDATETIME*, *SYSDATETIMEOFFSET*, *CAST*, *CONVERT*, *SWITCHOFFSET*, *TODATETIMEOFFSET*, *DATEADD*, *DATEDIFF*, *DATEPART*, *YEAR*, *MONTH*, *DAY*, *DATENAME*, różne funkcje *FROMPARTS* oraz *EOMONTH*.

Bieżąca data i godzina

Następujące funkcje bezargumentowe (bez parametrów) zwracają bieżącą datę i godzinę w systemie, w którym działa instancja systemu SQL Server: *GETDATE*, *CURRENT_TIMESTAMP*, *GETUTCDATE*, *SYSDATETIME*, *SYSUTCDATETIME* i *SYSDATETIMEOFFSET*. W tabeli 2-3 zamieszczono opis tych funkcji.

TABELA 2-3 Funkcje, które zwracają bieżącą datę i godzinę

Funkcja	Zwracany typ	Opis
<i>GETDATE</i>	<i>DATETIME</i>	Bieżąca data i godzina
<i>CURRENT_TIMESTAMP</i>	<i>DATETIME</i>	To samo co <i>GETDATE</i> , lecz zgodne z ANSI SQL
<i>GETUTCDATE</i>	<i>DATETIME</i>	Bieżąca data i godzina w strefie UTC
<i>SYSDATETIME</i>	<i>DATETIME2</i>	Bieżąca data i godzina
<i>SYSUTCDATETIME</i>	<i>DATETIME2</i>	Bieżąca data i godzina w strefie UTC
<i>SYSDATETIMEOFFSET</i>	<i>DATETIMEOFFSET</i>	Bieżąca data i godzina, a w tym strefa czasowa

Trzeba pamiętać, by wpisać puste nawiasy dla wszystkich funkcji, które powinny być specyfikowane bez parametrów, z wyjątkiem funkcji standardu ANSI – *CURRENT_TIMESTAMP*. Ponadto, ponieważ funkcja *CURRENT_TIMESTAMP* i *GETDATE* zwracają te same wyniki, ale tylko pierwsza jest standardem, lepiej posługiwać się właśnie nią. Jest to rozwiązanie praktyczne, którego staram się przestrzegać jako zasady ogólnej – jeśli mam kilka opcji, które wykonują te same działania przy braku różnic funkcjonalnych czy wydajnościowych, a jedna z nich jest standardem (a pozostałe nie), zawsze stosuję opcję standardową.

Poniższy kod ilustruje stosowanie funkcji zwracających bieżącą datę i godzinę.

```
SELECT
    GETDATE()           AS [GETDATE],
    CURRENT_TIMESTAMP   AS [CURRENT_TIMESTAMP],
    GETUTCDATE()        AS [GETUTCDATE],
    SYSDATETIME()       AS [SYSDATETIME],
```

```
SYSUTCDATETIME() AS [SYSUTCDATETIME],
SYSDATETIMEOFFSET() AS [SYSDATETIMEOFFSET];
```

Jak widać, żadna z funkcji nie zwraca tylko bieżącej daty systemowej lub tylko bieżącego czasu systemu. Zadanie to jednak można łatwo zrealizować poprzez konwersję funkcji *CURRENT_TIMESTAMP* lub *SYSDATETIME* na *DATE* lub *TIME*, jak w poniższym przykładzie:

```
SELECT
  CAST(SYSDATETIME() AS DATE) AS [current_date],
  CAST(SYSDATETIME() AS TIME) AS [current_time];
```

Funkcje *CAST*, *CONVERT* i *PARSE* oraz ich odpowiedniki *TRY_*

Funkcje *CAST*, *CONVERT* i *PARSE* służą do przekształcania wartości wejściowej na pewien typ docelowy. Jeśli konwersja powiedzie się, funkcja zwraca przekształconą wartość; w przeciwnym razie zapytanie zakończy się błędem. Funkcje te mają swoje odpowiedniki nazwane *TRY_CAST*, *TRY_CONVERT* i *TRY_PARSE*. Każda funkcja z prefiksem *TRY_* akceptuje te same wartości wejściowe, co jej odpowiednik i wykonuje te same działania; różnica polega na tym, że jeśli dane wejściowe nie mogą być przekształcone na typ docelowy, funkcja zwraca znacznik *NULL* zamiast zgłoszenia błędu.

Składnia:

```
CAST(value AS datatype)
TRY_CAST(value AS datatype)
CONVERT (datatype, value [, style_number])
TRY_CONVERT (datatype, value [, style_number])
PARSE (value AS datatype [USING culture])
TRY_PARSE (value AS datatype [USING culture])
```

Wszystkie trzy funkcje przekształcają wartość wejściową (*value*) na określony docelowy typ danych (*datatype*). W niektórych przypadkach funkcja *CONVERT* wykorzystuje trzeci opcjonalny argument, który służy do specyfikowania stylu konwersji (*style_number*). Na przykład jeśli przekształcamy ciąg znakowy na jeden z typów danych daty i godziny (lub odwrotnie), numer stylu wskazuje format ciągu. Na przykład styl 101 wskazuje postać 'MM/DD/YYYY', a styl 103 wskazuje postać 'DD/MM/YYYY'. Pełna lista numerów stylów i ich znaczenie zamieszczona jest w dokumentacji SQL Server Books Online w artykule „CAST and CONVERT”. Podobnie tam, gdzie ma to zastosowanie, funkcja *PARSE* obsługuje parametr *culture* – na przykład 'en-US' dla U.S. English i 'en-GB' dla British English.

Jak już wspominałem, podczas konwersji ciągu znakowego na jeden z typów danych daty i godziny niektóre formaty ciągu są zależne od języka. Zaleca się stosowanie jednego z formatów neutralnych językowo lub używanie funkcji *CONVERT/PARSE*

i specyfikowanie wprost numeru stylu czy parametru *culture*. W ten sposób kod jest zawsze interpretowany w ten sam sposób niezależnie od języka loginu.

Zwróćmy uwagę, że funkcja *CAST* jest standardem ANSI, natomiast nie są nimi funkcje *CONVERT* i *PARSE*, tak więc, o ile nie musimy stosować numeru stylu czy parametru *culture*, zaleca się używanie funkcji *CAST*; w ten sposób zapewniamy naszemu kodowi możliwie wysoki poziom standaryzacji.

Poniżej przedstawiam kilka przykładów użycia funkcji *CAST*, *CONVERT* i *PARSE* za pomocą typu danych daty i godziny. Poniższy kod przekształca literal ciągu znakowego '20160212' na typ danych *DATE*.

```
SELECT CAST('20160212' AS DATE);
```

Poniższy kod przekształca wartość bieżącej daty i godziny systemu na typ danych *DATE*, w praktyce uzyskując tylko bieżącą datę systemową.

```
SELECT CAST(SYSDATETIME() AS DATE);
```

Kod pokazany poniżej przekształca wartość bieżącej daty i godziny systemu na typ danych *TIME*, w praktyce uzyskując tylko bieżący czas systemu.

```
SELECT CAST(SYSDATETIME() AS TIME);
```

Zgodnie z wcześniejszą sugestią, jeśli trzeba korzystać z typów *DATETIME* lub *SMALLEDATETIME* (na przykład dla zachowania kompatybilności ze starszymi systemami) i chcemy tylko uzyskać datę lub tylko godzinę, „zerujemy” część, która nie jest istotna. Inaczej mówiąc, by korzystać tylko z dat, ustawiamy czas na północ. Aby używać tylko czasu, ustawiamy datę na datę bazową, czyli 1 stycznia 1900.

Poniższy kod przekształca wartość bieżącej daty i godziny na *CHAR(8)* przy użyciu stylu 112 ('YYYYMMDD').

```
SELECT CONVERT(CHAR(8), CURRENT_TIMESTAMP, 112);
```

Jeśli przykładowo bieżąca data to 12 lutego 2009, kod zwróci wartość '20090212'. Pamiętajmy, że ten styl jest neutralny pod względem języka, tak więc, jeśli kod jest przekształcany z powrotem na typ *DATETIME*, uzyskamy bieżącą datę o północy.

```
SELECT CAST(CONVERT(CHAR(8), CURRENT_TIMESTAMP, 112) AS DATETIME);
```

Podobnie, by „wyzerować” część dotyczącą daty za pomocą wartości daty bazowej, najpierw przeprowadzamy konwersję bieżącej daty i godziny na typ *CHAR(12)* przy użyciu stylu 114 ('hh:mm:ss.nnn').

```
SELECT CONVERT(CHAR(12), CURRENT_TIMESTAMP, 114);
```

Po powrotnej konwersji kodu na typ *DATETIME* uzyskamy bieżący czas dla daty bazowej.

```
SELECT CAST(CONVERT(CHAR(12), CURRENT_TIMESTAMP, 114) AS DATETIME);
```

Działania funkcji *PARSE* prezentuje kilka przykładów, które były już wcześniej analizowane w tym rozdziale.

```
SELECT PARSE('02/12/2015' AS DATETIME USING 'en-US');
SELECT PARSE('02/12/2015' AS DATETIME USING 'en-GB');
```

Pierwsze polecenie analizuje ciąg wejściowy przy użyciu parametru *culture* U.S. English, a drugi przy użyciu British English.

Warto przypomnieć, że funkcja *PARSE* wiąże się ze znacząco zwiększonym kosztem wykonania, niż funkcja *CONVERT* i z tego względu zalecane jest stosowanie tej ostatniej.

Funkcja *SWITCHOFFSET*

Funkcja *SWITCHOFFSET* dostosowuje wartość wejściową typu *DATETIMEOFFSET* do określonej strefy czasowej.

Składnia:

```
SWITCHOFFSET(datetimeoffset_value, time_zone)
```

Na przykład poniższy kod dostosowuje bieżącą wartość funkcji *SYSDATETIMEOFFSET* do strefy czasowej -05:00.

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '-05:00');
```

Jeśli więc bieżąca wartość czasu systemowego to 12 lutego 2016 10:00:00.0000000 -08:00, kod ten zwróci wartość 12 lutego 2016 13:00:00.0000000 -05:00.

Poniższy kod dostosowuje bieżącą wartość czasu systemowego do strefy UTC.

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '+00:00');
```

Zakładając użycie wartości z poprzedniego przykładu, kod ten zwróci wartość 12 lutego 2016 18:00:00.0000000 +00:00.

Funkcja *TODATETIMEOFFSET*

Funkcja *TODATETIMEOFFSET* ustawia przesunięcie strefy czasowej (*time_zone*) dla wejściowej wartości daty i godziny (*date_and_time_value*).

Składnia:

```
TODATETIMEOFFSET(date_and_time_value, time_zone)
```

Funkcja ta różni się od funkcji *SWITCHOFFSET* tym, że jej pierwsza wartość wejściowa jest typem daty i godziny, w którym brak informacji o przesunięciu. Funkcja ta po prostu łączy wejściowe wartości daty i godziny z wyspecyfikowanym przesunięciem strefy czasowej, by utworzyć nową wartość typu *datetimeoffset*.

Zazwyczaj posługujemy się tą funkcją podczas przeprowadzania migracji danych, w których brak jest informacji o strefie czasowej do danych, gdzie informacje te są uwzględniane. Wyobraźmy sobie tabelę przechowującą dane wartości daty i godziny w atrybucie nazwanym *dt*, którego typ danych to *DATETIME2* lub *DATETIME*, a strefę czasową w atrybucie nazwanym *theoffset*. Następnie decydujemy się na połączenie tych dwóch atrybutów do postaci jednego, który uwzględnia przesunięcie i nazywa się *dto*. Możemy zmienić tabelę i dodać nowy atrybut. Następnie aktualizujemy go za pomocą wyrażenia *TODATETIMEOFFSET(dt, theoffset)*, a na koniec usuwamy dwa istniejące atrybuty *dt* i *theoffset*.

Funkcja *AT TIME ZONE*

Funkcja *AT TIME ZONE* przyjmuje jako wejście wartość daty i czasu i konwertuje ją na typ *datetimeoffset* odpowiadającą docelowej strefie czasowej. Funkcja ta została wprowadzona w wersji SQL Server 2016.

Składnia

```
dt_val AT TIME ZONE time_zone
```

Wejściowa wartość *dt_val* może być jednego z następujących typów danych: *DATETIME*, *SMALLDATETIME*, *DATETIME2* lub *DATETIMEOFFSET*. Wartość *time_zone* może być jedną z obsługiwanych w systemie Windows nazw stref czasowych, tak jak wyświetlane są w kolumnie *name* widoku systemowego *sys.time_zone_info*. Poniższe zapytanie pozwala wyświetlić dostępne strefy czasowe, ich przesunięcie względem UTC, a także to, czy jest w niej aktualnie stosowany czas letni (*Daylight Savings Time* – DST):

```
SELECT name, current_utc_offset, is_currently_dstFROM sys.time_zone_info;
```

Co do *dt_val*: przy używaniu dowolnego z trzech typów pozbawionych informacji o strefie czasowej (*DATETIME*, *SMALLDATETIME* lub *DATETIME2*), funkcja *AT TIME ZONE* zakłada, że wartość wejściowa pochodzi z docelowej strefy czasowej. W rezultacie zachowuje się analogicznie do funkcji *TODATETIMEOFFSET* z wyjątkiem tego, że przesunięcie nie musi być stałe – może ono zależeć od tego, czy stosowany jest czas letni. Dla przykładu posłużmy się strefą Pacific Standard Time. Gdy nie obowiązuje czas letni, przesunięcie względem UTC wynosi -08:00; w czasie obowiązywania czasu letniego przesunięcie to -07:00. Poniższy kod demonstruje użycie tej funkcja dla danych wyjściowych nie zawierających informacji o strefie czasowej:

```
SELECT
  CAST('20160212 12:00:00.0000000' AS DATETIME2)
  AT TIME ZONE 'Pacific Standard Time' AS val1,
  CAST('20160812 12:00:00.0000000' AS DATETIME2)
  AT TIME ZONE 'Pacific Standard Time' AS val2;
```

Kod ten generuje następujący wynik:

val1	val2
2016-02-12 12:00:00.0000000 -08:00	2016-08-12 12:00:00.0000000 -07:00

Pierwsza wartość dotyczy okresu, gdy czas letni nie jest stosowany; tym samym przesunięcie wynosi $-08:00$, zgodnie z oczekiwaniem. Druga wartość wypada podczas obowiązywania czasu letniego i przesunięcie zostało ustawione na $-07:00$. Nie ma tu żadnej niejednoznaczności.

Istnieją dwa szczególne przypadki: gdy następuje przełączenie na czas letni oraz gdy następuje powrót do czasu standardowego (zimowego). Podczas przełączania się na czas letni zegar jest przestawiany o jedną godzinę do przodu, zatem jest taka godzina, która nie istnieje (zazwyczaj przesunięcie następuje o godzinie 2:00 w nocy, co oznacza, że w istocie nie pojawiają się takie wartości, jak 2:30). Jeśli jako daną wejściową podamy czas nieistniejący (należący do tej pominiętej godziny), przyjmowane jest przesunięcie obowiązujące przed dokonaniem zmiany (w naszym przykładzie $-08:00$). Przy przechodzeniu z czasu letniego na zimowy zegar jest cofany o godzinę, zatem istnieje godzina, która zostanie powtórzona dwukrotnie. W przypadku podania na wejściu czasu należącego do tej godziny przyjmowany jest czas zimowy (czyli w naszym przykładzie ponownie zostanie użyte przesunięcie $-08:00$).

Gdy wartość wejściowa *dt_val* jest typu *datetimeoffset*, funkcja *AT TIME ZONE* zachowuje się podobnie do funkcji *SWITCHOFFSET*. Ponownie jednak docelowe przesunięcie nie musi być stałe. T-SQL używa reguł konwersji stref czasowych usługi Windows Time do wykonania przekształcenia. Poniższy kod demonstruje wykorzystanie funkcji z danymi wejściowymi *datetimeoffset*:

```
SELECT
  CAST('20160212 12:00:00.0000000 -05:00' AS DATETIMEOFFSET)
  AT TIME ZONE 'Pacific Standard Time' AS val1,
  CAST('20160812 12:00:00.0000000 -04:00' AS DATETIMEOFFSET)
  AT TIME ZONE 'Pacific Standard Time' AS val2;
```

Dane wejściowe odnoszą się do strefy czasowej Eastern Standard Time. W obydwu jako składowa czasowa występuje południe. Pierwsza wartość dotyczy okresu, gdy czas letni nie jest stosowany (przesunięcie wynosi $-05:00$), zaś drugie następuje, gdy czas letni obowiązuje (zatem przesunięcie to $-04:00$). Obie wartości są konwertowane na wartość strefy czasowej Pacific Standard Time (przesunięcie $-08:00$ przy braku czasu letniego i $-07:00$, gdy on obowiązuje). W obydwu przypadkach czas wymaga cofnięcia o trzy godziny. Przykład ten zwraca poniższe dane wyjściowe:

val1	val2
2016-02-12 09:00:00.0000000 -08:00	2016-08-12 09:00:00.0000000 -07:00

Funkcja **DATEADD**

Funkcja *DATEADD* pozwala na dodawanie do wejściowej wartości daty i godziny określonej liczby jednostek wyspecyfikowanej części daty.

Składnia:

`DATEADD(part, n, dt_val)`

Element wejściowy *part* przyjmuje następujące wartości *year*, *quarter*, *month*, *dayofyear*, *day*, *week*, *weekday*, *hour*, *minute*, *second*, *millisecond*, *microsecond* i *nanosecond* (czyli odpowiednio rok, kwartał, dzień roku, dzień, tydzień, dzień tygodnia, godzina, minuta, sekunda, milisekunda, mikrosekunda i nanosekunda). Element *part* możemy również specyfikować w skróconej postaci, jak na przykład *yy* dla *year*. Informacje szczegółowe na ten temat znajdziemy w dokumentacji SQL Server Books Online.

Zwracanym typem dla danych wejściowych daty i godziny jest ten sam typ co typ danych wejściowych. Jeśli do tej funkcji przekazywany jest na wejściu literał znakowy, dane wyjściowe są typu *DATETIME*.

Na przykład poniższy kod dodaje jeden rok do daty 12 lutego 2016.

```
SELECT DATEADD(year, 1, '20160212');
```

Kod zwraca następujący wynik:

```
2017-02-12
```

Funkcje **DATEDIFF** i **DATEDIFF_BIG**

Funkcje *DATEDIFF* i *DATEDIFF_BIG* zwracają różnicę pomiędzy dwoma wartościami daty i godziny (*dt_val*) w odniesieniu do wyspecyfikowanej części daty (*part*).

Składnia:

`DATEDIFF(part, dt_val1, dt_val2)`
`DATEDIFF_BIG(part, dt_val1, dt_val2)`

Na przykład poniżej pokazany kod zwraca różnicę w dniach pomiędzy dwoma wartościami.

```
SELECT DATEDIFF(day, '20160212', '20170212');
```

Kod ten zwróci wartość 366.

Istnieją takie sytuacje, gdy obliczane różnice przekraczają maksymalną wartość typu *INT* (czyli 2 147 483 647). Dla przykładu różnica w milisekundach pomiędzy 1 stycznia 0001 (1 roku naszej ery) a 12 lutego 2016 wynosi 63 590 832 000 000. Nie można użyć funkcji *DATEDIFF* do obliczenia takiej różnicy, gdyż zwraca ona wartość typu *INT*. Z tego względu SQL Server został uzupełniony o funkcję *DATEDIFF_BIG*:

```
SELECT DATEDIFF_BIG(millisecond, '00010101', '20160212');
```

Załóżmy, że chcemy uzyskać wartość czasową wskazującą początek dnia, do którego należy podana na wejściu wartość daty i czasu. W takim przypadku możemy po prostu rzutować wejściową wartość na typ *DATE* (w praktyce usuwając składnik czasowy), a następnie rzutować ten wynik na docelowy typ danych. Jednak przy nieco bardziej wyrafinowanym wykorzystaniu funkcji *DATEADD* i *DATEDIFF* możemy wyznaczyć początki lub końce innych części daty (dnia, miesiąca, kwartału lub roku), które odpowiadają wartości wejściowej. Dla przykładu poniższy kod pozwala wyznaczyć początek dnia odpowiadającego wejściowej wartości daty i czasu:

```
SELECT
    DATEADD(
        day,
        DATEDIFF(day, '19000101', SYSDATETIME()), '19000101');
```

W tym przykładzie najpierw zostaje użyta funkcja *DATEDIFF* do wyliczenia różnicy w pełnych dniach pomiędzy datą bazową ('19000101' w tym przypadku) a bieżącą datą i czasem (nazwijmy tę różnicę *diff*). Następnie funkcja *DATEADD* dodaje *diff* dni do daty bazowej. W rezultacie uzyskujemy bieżącą datę systemową o północy.

Jeśli wykorzystamy to wyrażenie, ale przy użyciu składowej *month* (miesiąc) zamiast *day* i zadamy o to, aby jako daty bazowej użyć pierwszego dnia miesiąca (jak w tym przykładzie), otrzymamy w wyniku pierwszy dzień bieżącego miesiąca:

```
SELECT
    DATEADD(
        month,
        DATEDIFF(month, '19000101', SYSDATETIME()), '19000101');
```

Analogicznie możemy posłużyć się składową *year* (rok) i datą bazową będącą pierwszym dniem roku, aby uzyskać datę początkową bieżącego roku.

Jeśli potrzebujemy znaleźć ostatni dzień miesiąca lub roku, wystarczy użyć jako daty bazowej ostatniego dnia miesiąca lub roku. Poniższy przykład zwraca datę ostatniego dnia bieżącego roku:

```
SELECT
    DATEADD(
        year,
        DATEDIFF(year, '18991231', SYSDATETIME()), '18991231');
```

Przy użyciu podobnych wyrażeń ze składową *month* można uzyskać datę ostatniego dnia miesiąca, ale znacznie łatwiejsze będzie wykorzystanie funkcji *EOMONTH*. Niestety nie istnieją podobne funkcje pozwalające znaleźć koniec roku lub kwartału, zatem w takich sytuacjach konieczne jest użycie obliczeń podobnych do pokazanych.

Funkcja *DATEPART*

Funkcja *DATEPART* zwraca liczbę całkowitą reprezentującą żadaną część (*part*) wartości daty i godziny (*dt_val*).

Składnia:

```
DATEPART(part, dt_val)
```

Element wejściowy *part* przyjmuje następujące wartości *year*, *quarter*, *month*, *dayofyear*, *day*, *week*, *weekday*, *hour*, *minute*, *second*, *millisecond*, *microsecond*, *nanosecond*, *TZoffset* i *ISO_WEEK* (rok, kwartał, miesiąc, dzień roku, dzień, tydzień, dzień tygodnia, godziny, minuty, sekundy, milisekundy, mikrosekundy, nanosekundy, przesunięcie strefy czasowej oraz numer tygodnia zgodnie ze standardem ISO). Jak wspomniałem wcześniej, dla części daty i godziny możemy także użyć formy skrótowej, na przykład *yy* zamiast *year*, *mm* zamiast *month*, *dd* zamiast *day* itd.

Na przykład poniższy kod na podstawie wartości wejściowej zwraca część dotyczącą miesiąca.

```
SELECT DATEPART(month, '20160212');
```

Kod zwraca liczbę całkowitą 2.

Funkcje *YEAR*, *MONTH* i *DAY*

Funkcje *YEAR*, *MONTH* i *DAY* są skróconą postacią funkcji *DATEPART*, zwracającej liczbę całkowitą reprezentującą w wejściowej wartości daty i godziny (*dt_val*) odpowiednią część – rok, miesiąc czy dzień.

Składnia:

```
YEAR(dt_val)
```

```
MONTH(dt_val)
```

```
DAY(dt_val)
```

Na przykład pokazany poniżej kod zwraca dzień, miesiąc i rok dla podanej wartości wejściowej:

```
SELECT
    DAY('20160212') AS theday,
    MONTH('20160212') AS themonth,
    YEAR('20160212') AS theyear;
```

Kod zwraca następujące wyniki:

theday	themonth	theyear
12	2	2016

Funkcja *DATENAME*

Funkcja *DATENAME* zwraca ciąg znaków reprezentujących część (*part*) dla daty i godziny (*dt_val*).

Składnia:

```
DATENAME(dt_val, part)
```

Funkcja ta jest podobna do funkcji *DATEPART* i rzeczywiście ma takie same opcje dla wejściowego elementu *part*. Jednakże, jeśli ma to zastosowanie, funkcja zwraca nazwę żądanej części, a nie liczbę. Na przykład poniższy kod dla danej wartości wejściowej zwraca nazwę miesiąca.

```
SELECT DATENAME(month, '20160212');
```

Jak pamiętamy, funkcja *DATEPART* dla tych danych wejściowych zwróciła liczbę całkowitą 2. Funkcja *DATENAME* zwraca nazwę miesiąca, która jest zależna od języka. Jeśli dla naszej sesji wybrany jest język angielski (na przykład U.S. English czy British English), uzyskamy wartość 'February'. Jeśli językiem sesji jest włoski, uzyskamy wartość 'Febbraio'. Jeśli żądana część nie ma nazwy, a jedynie wartość liczbową (jak w przypadku *year*), funkcja *DATENAME* zwraca wartość numeryczną w postaci ciągu znakowego. Na przykład następujący kod zwraca '2016'.

```
SELECT DATENAME(year, '20160212');
```

Funkcja *ISDATE*

Funkcja *ISDATE* akceptuje na wejściu ciąg znakowy (*string*) i zwraca 1, jeśli można go przekształcić na typ danych daty i godziny, lub 0, jeśli nie jest to możliwe.

Składnia:

```
ISDATE(string)
```

Przykładowo poniższy kod zwraca wartość 1.

```
SELECT ISDATE('20160212');
```

W poniższym przypadku zwrócona zostanie wartość 0.

```
SELECT ISDATE('20160230');
```

Funkcje ...*FROMPARTS*

Funkcje *FROMPARTS* zostały wprowadzone w wersji SQL Server 2012. Funkcje te akceptują dane wejściowe reprezentujące części daty i godziny i na podstawie tych części konstruują wartość żądanego typu.

Składnia:

`DATEFROMPARTS (year, month, day)`

`DATETIME2FROMPARTS (year, month, day, hour, minute, seconds, fractions, precision)`

`DATETIMEFROMPARTS (year, month, day, hour, minute, seconds, milliseconds)`

`DATETIMEOFFSETFROMPARTS (year, month, day, hour, minute, seconds, fractions, hour_offset, minute_offset, precision)`

`SMALLDATETIMEFROMPARTS (year, month, day, hour, minute)`

`TIMEFROMPARTS (hour, minute, seconds, fractions, precision)`

gdzie *year, month, day, hour, minute, seconds, fractions, precision, hour_offset, minute_offset* to odpowiednio rok, miesiąc, dzień, godzina, minuta, sekundy, ułamkowa część sekundy, dokładność, przesunięcie godzinowe, przesunięcie minutowe.

Funkcje te ułatwiają konstruowanie wartości daty i godziny na podstawie różnych składników, a także upraszczają migrację z innych środowisk, które już obsługują podobne funkcje. Poniższy kod ilustruje użycie tych funkcji.

```
SELECT
    DATEFROMPARTS(2012, 02, 12),
    DATETIME2FROMPARTS(2012, 02, 12, 13, 30, 5, 1, 7),
    DATETIMEFROMPARTS(2012, 02, 12, 13, 30, 5, 997),
    DATETIMEOFFSETFROMPARTS(2012, 02, 12, 13, 30, 5, 1, -8, 0, 7),
    SMALLDATETIMEFROMPARTS(2012, 02, 12, 13, 30),
    TIMEFROMPARTS(13, 30, 5, 1, 7);
```

Funkcja *EOMONTH*

Funkcja *EOMONTH* została wprowadzona w systemie SQL Server 2012. Funkcja akceptuje wejściową wartość (*input*) daty i godziny oraz zwraca odpowiednią datę końca miesiąca w postaci typu danych *DATE*. Funkcja obsługuje także pomocniczy argument opcjonalny, wskazujący ilość dodawanych miesięcy (*months_to_add*).

Składnia:

`EOMONTH(input [, months_to_add])`

Na przykład poniższy kod zwraca koniec bieżącego miesiąca.

```
SELECT EOMONTH(SYSDATETIME());
```

Poniższe zapytanie zwraca zamówienia złożone w ostatnim dniu miesiąca.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate = EOMONTH(orderdate);
```

Zapytania dotyczące metadanych

SQL Server udostępnia narzędzia do uzyskiwania informacji o metadanych obiektów, takie jak informacje o tabelach w bazie danych i kolumnach w tabeli. Narzędzia te obejmują widoki katalogowe, informacyjne widoki schematu oraz systemowe procedury składowane i funkcje. Zagadnienia te są dobrze udokumentowane w SQL Server Books Online w rozdziale „Querying the SQL Server System Catalog”, tak więc w książce nie będę ich omawiać zbyt szczegółowo, ale przedstawię kilka przykładów działania każdego narzędzia metadanych, by pokazać, jakiego rodzaju narzędzia są dostępne i ułatwić rozpoczęcie ich stosowania.

Widoki katalogowe

Widoki katalogowe udostępniają bardzo szczegółowe informacje o obiektach bazy danych, wliczając w to informacje specyficzne dla systemu SQL Server. Jeśli przykładowo chcemy utworzyć listę tabel bazy danych wraz z nazwami ich schematów, uruchamiamy poniższą kwerendę widoku `sys.tables`:

```
USE TSQLV4;
```

```
SELECT SCHEMA_NAME(schema_id) AS table_schema_name, name AS table_name
FROM sys.tables;
```

Funkcja `SCHEMA_NAME` jest używana do przekształcenia identyfikatora schematu (liczby całkowitej) na jego nazwę. Zapytanie to zwraca następujący wynik:

table_schema_name	table_name
HR	Employees
Production	Suppliers
Production	Categories
Production	Products
Sales	Customers
Sales	Shippers
Sales	Orders
Sales	OrderDetails
Stats	Tests
Stats	Scores
dbo	Nums

Aby uzyskać informacje o kolumnach tabeli, możemy odpytać tabelę `sys.columns`. Na przykład poniższy kod zwraca informacje o kolumnach w tabeli `Sales.Orders`, w tym nazwy kolumn, typy danych (z systemowym identyfikatorem typu przetłumaczonym na nazwę przy użyciu funkcji `TYPE_NAME`), maksymalne długości, nazwy sortowań i informacji, czy możliwa jest obsługa znaczników `NULL`.

```
SELECT
    name AS column_name,
```



```

TYPE_NAME(system_type_id) AS column_type,
max_length,
collation_name,
is_nullable
FROM sys.columns
WHERE object_id = OBJECT_ID(N'Sales.Orders');

```

Zapytanie to zwraca następujący rezultat:

column_name	column_type	max_length	collation_name	is_nullable
orderid	int	4	NULL	0
custid	int	4	NULL	1
empid	int	4	NULL	0
orderdate	datetime	8	NULL	0
requireddate	datetime	8	NULL	0
shippeddate	datetime	8	NULL	1
shipperid	int	4	NULL	0
freight	money	8	NULL	0
shipname	nvarchar	80	Latin1_General_CI_AI	0
shipaddress	nvarchar	120	Latin1_General_CI_AI	0
shipcity	nvarchar	30	Latin1_General_CI_AI	0
shipregion	nvarchar	30	Latin1_General_CI_AI	1
shippostalcode	nvarchar	20	Latin1_General_CI_AI	1
shipcountry	nvarchar	30	Latin1_General_CI_AI	0

Informacyjne widoki schematu

Informacyjne widoki schematu to zestaw widoków, które znajdują się w schemacie nazwanym *INFORMATION_SCHEMA* i udostępniają informacje metadanych w sposób standardowy, to znaczy widoki są definiowane jako standard SQL i tym samym nie zawierają aspektów specyficznych dla implementacji SQL Server (takich jak indeksy).

Na przykład poniższe zapytanie do widoku *INFORMATION_SCHEMA.TABLES* wymienia tabele użytkownika w bieżącej bazie danych wraz z ich nazwami schematów.

```

SELECT TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = N'BASE TABLE';

```

Poniższe zapytanie do widoku *INFORMATION_SCHEMA.COLUMNS* pozwala uzyskać większość dostępnych informacji o kolumnach tabeli *Sales.Orders*.

```

SELECT
    COLUMN_NAME, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH,
    COLLATION_NAME, IS_NULLABLE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = N'Sales'
    AND TABLE_NAME = N'Orders';

```

Systemowe procedury składowane i funkcje

Systemowe procedury składowane i funkcje przepytują wewnętrznie katalog systemowy i pozwalają nam uzyskać bardziej dokładnie informacje o metadanych. I ponownie, pełną listę obiektów i ich szczegółowy opis znajdziemy w dokumentacji SQL Server Books Online, a teraz przytoczę tylko kilka przykładów. Procedura składowana *sp_tables* zwraca listę obiektów (takich jak tabele i widoki), wobec których w bieżącej bazie danych można uruchamiać zapytania.

```
EXEC sys.sp_tables;
```

Procedura *sp_help* akceptuje nazwę obiektu i zwraca wiele zbiorów wyników zawierających informacje ogólne o obiekcie, a także informacje o kolumnach, indeksach, ograniczeniach i innych kwestiach. Na przykład poniższy kod zwraca informacje szczegółowe o tabeli *Orders*.

```
EXEC sys.sp_help
    @objname = N'Sales.Orders';
```

Procedura *sp_columns* zwraca informacje o kolumnach obiektu. Na przykład uruchomienie poniższego kodu zwraca informacje o kolumnach tabeli *Orders*.

```
EXEC sys.sp_columns
    @table_name = N'Orders',
    @table_owner = N'Sales';
```

Procedura *sp_helpconstraint* zwraca informacje o ograniczeniach w obiekcie. Na przykład poniższy kod zwraca informacje o ograniczeniach w tabeli *Orders*.

```
EXEC sys.sp_helpconstraint
    @objname = N'Sales.Orders';
```

Jeden zestaw funkcji zwraca informacje o właściwościach jednostek, takich jak instancje systemu SQL Server, baza danych, obiekt, kolumna itp. Funkcja *SERVERPROPERTY* zwraca żadaną właściwość bieżącej instancji. Na przykład uruchomienie poniższego kodu zwróci informacje o poziomie produktu (takie jak RTM, SP1, SP2 itp.) bieżącej instancji.

```
SELECT
    SERVERPROPERTY('ProductLevel');
```

Funkcja *DATABASEPROPERTYEX* zwraca żadaną właściwość bazy danych o wyspecyfikowanej nazwie. Na przykład poniższy kod zwraca ustawienie *collation* bazy danych *TSQV4*.

```
SELECT
    DATABASEPROPERTYEX(N'TSQV4', 'Collation');
```

Funkcja *OBJECTPROPERTY* zwraca żadaną właściwość dla obiektu o wyspecyfikowanej nazwie. Na przykład dane wyjściowe uzyskane po uruchomieniu poniższego kodu wskazują, czy tabela *Orders* posiada klucz główny.

```
SELECT  
    OBJECTPROPERTY(OBJECT_ID(N'Sales.Orders'), 'TableHasPrimaryKey');
```

Zwróćmy uwagę na zagnieżdżenie funkcji *OBJECT_ID* wewnątrz *OBJECTPROPERTY*. Funkcja *OBJECTPROPERTY* oczekuje identyfikatora obiektu, a nie nazwy, tak więc używamy funkcji *OBJECT_ID* do zwrócenia identyfikatora tabeli *Orders*.

Funkcja *COLUMNPROPERTY* zwraca żadaną właściwość w wyspecyfikowanej kolumnie. Na przykład poniższy kod wskazuje, czy kolumna *shipcountry* w tabeli *Orders* ma możliwość obsługi wartości *NULL*.

```
SELECT  
    COLUMNPROPERTY(OBJECT_ID(N'Sales.Orders'), N'shipcountry', 'AllowsNull');
```

Podsumowanie

W niniejszym rozdziale omówiłem polecenie *SELECT*, logikę przetwarzania zapytań i różne inne aspekty zapytań odnoszących się do pojedynczych tabel. Zawarłem w nim całkiem sporo zagadnień, w tym wiele nowych i unikatowych koncepcji. Jeśli tematyka języka T-SQL jest dla kogoś zupełnie nowa, pewnie w tym momencie czuje się trochę przytłoczony, ale trzeba pamiętać, że w tym rozdziale pokazałem najbardziej istotne kwestie języka SQL, które początkowo trudno opanować. Jeśli jakieś pojęcia nie są całkiem jasne, warto będzie ponownie zajrzeć do tego fragmentu później, po „przespaniu się” z tematem.

Okazją do nabycia praktyki i lepszego przyswojenia poznanego materiału z pewnością są ćwiczenia zamieszczone w rozdziale.

Ćwiczenia

W tym podrozdziale proponuję ćwiczenia, które ułatwią lepsze przyswojenie tematyki opisanej w rozdziale 2. Rozwiązania ćwiczeń zamieszczono w kolejnym podrozdziale.

W Dodatku na końcu książki znaleźć można instrukcje pobierania i instalowania przykładowej bazy danych *TSQLV4*.

Ćwiczenie 1

Utworzyć zapytanie wobec tabeli *Sales.Orders*, które zwraca zamówienia złożone w czerwcu 2015.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

orderid	orderdate	custid	empid
-----	-----	-----	-----
10555	2015-06-02	71	6
10556	2015-06-03	73	2
10557	2015-06-03	44	9
10558	2015-06-04	4	1
10559	2015-06-05	7	6
10560	2015-06-06	25	8
10561	2015-06-06	24	2
10562	2015-06-09	66	1
10563	2015-06-10	67	2
10564	2015-06-10	65	4
...			

(30 row(s) affected)

Ćwiczenie 2

Opracować zapytanie do tabeli *Sales.Orders*, które zwraca zamówienia złożone w ostatnim dniu miesiąca.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

orderid	orderdate	custid	empid
-----	-----	-----	-----
10269	2014-07-31	89	5
10317	2014-09-30	48	6
10343	2014-10-31	44	4
10399	2014-12-31	83	8
10432	2015-01-31	75	3
10460	2015-02-28	24	8
10461	2015-02-28	46	1

```

10490      2015-03-31  35      7
10491      2015-03-31  28      8
10522      2015-04-30  44      4
...

```

```
(26 row(s) affected)
```

Ćwiczenie 3

Napisać zapytanie do tabeli *HR.Employees*, zwracające listę pracowników, których nazwisko zawiera co najmniej dwie litery *a*.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *HR.Employees*
- Oczekiwane dane wyjściowe:

```

empid      firstname      lastname
-----
9          Patricia      Doyle

```

```
(1 row(s) affected)
```

Ćwiczenie 4

Napisać zapytanie do tabeli *Sales.OrderDetails*, zwracające zamówienia o łącznej wartości (ilość * cena jednostkowa) przekraczającej 10000, posortowane według łącznej wartości.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *Sales.OrderDetails*
- Oczekiwane dane wyjściowe:

```

orderid      totalvalue
-----
10865        17250.00
11030        16321.90
10981        15810.00
10372        12281.20
10424        11493.20
10817        11490.70
10889        11380.00
10417        11283.20
10897        10835.24
10353        10741.60
10515        10588.50
10479        10495.60
10540        10191.70
10691        10164.80

```

```
(14 row(s) affected)
```

Ćwiczenie 5

W celu sprawdzenia poprawności danych napisz zapytanie do tabeli *HR.Employees*, które zwraca pracowników, których nazwisko zaczyna się od małej litery alfabetu łacińskiego, czyli z zakresu od a do z. Przypomnijmy, że ustawienie *collation* przykładowej bazy danych nie rozróżnia wielkości liter (*Latin1_General_CI_AS*):

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *HR.Employees*
- Oczekiwane dane wyjściowe:

```
empid      lastname
-----
```

```
(0 row(s) affected)
```

Ćwiczenie 6

Wyjaśnij różnicę pomiędzy poniższymi zapytaniami:

```
-- Zapytanie 1
SELECT empid, COUNT(*) AS numorders
FROM Sales.Orders
WHERE orderdate < '20160501'
GROUP BY empid;
```

```
-- Zapytanie 2
SELECT empid, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY empid
HAVING MAX(orderdate) < '20160501';
```

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *Sales.Orders*

Ćwiczenie 7

Napisać zapytanie do tabeli *Sales.Orders*, zwracające trzy kraje, do których wysyłka produktów miała najwyższą średnią wartość w roku 2015.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *Sales.Orders*
- Oczekiwane dane wyjściowe:

```
shipcountry      avgfreight
-----
```

Austria	178.3642
Switzerland	117.1775
Sweden	105.16

```
(3 row(s) affected)
```

Ćwiczenie 8

Napisać zapytanie do tabeli *Sales.Orders*, które oblicza liczbę wierszy zamówień, porządkując je według daty zamówienia (przy użyciu identyfikatora zamówienia jako kryterium rozstrzygania) oddzielnie dla każdego klienta.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

custid	orderdate	orderid	rownum
-----	-----	-----	-----
1	2015-08-25	10643	1
1	2015-10-03	10692	2
1	2015-10-13	10702	3
1	2016-01-15	10835	4
1	2016-03-16	10952	5
1	2016-04-09	11011	6
2	2014-09-18	10308	1
2	2015-08-08	10625	2
2	2015-11-28	10759	3
...			

(830 row(s) affected)

Ćwiczenie 9

Przy użyciu tabeli *HR.Employees* skonstruować polecenie *SELECT*, które zwraca płeć każdego pracownika, bazując na tytule grzecznościowym. Dla 'Ms.' i 'Mrs.' zwracana jest wartość 'Female'; dla 'Mr.' zwracana jest wartość 'Male'; a dla wszystkich pozostałych przypadków (na przykład 'Dr.') zwracana jest wartość 'Unknown'.

- Wykorzystywane tabele: baza danych *TSQLV4* i tabela *HR.Employees*
- Oczekiwane dane wyjściowe:

empid	firstname	lastname	titleofcourtesy	gender
-----	-----	-----	-----	-----
1	Sara	Davis	Ms.	Female
2	Don	Funk	Dr.	Unknown
3	Judy	Lew	Ms.	Female
4	Yael	Peled	Mrs.	Female
5	Sven	Buck	Mr.	Male
6	Paul	Suurs	Mr.	Male
7	Russell	King	Mr.	Male
8	Maria	Cameron	Ms.	Female
9	Patricia	Doyle	Ms.	Female

(9 row(s) affected)

Ćwiczenie 10

Napisać zapytanie do tabeli *Sales.Customers*, które dla każdego klienta zwraca identyfikator klienta i region. Posortować wiersze w danych wyjściowych według regionu, przy czym znaczniki *NULL* mają być umieszczone jako ostatnie (po wartościach innych niż *NULL*). Pamiętajmy, że domyślne sortowanie znaczników *NULL* w języku T-SQL zakłada sortowanie tych znaczników w pierwszej kolejności (przed wartościami innymi niż *NULL*).

- Wykorzystywane tabele: baza danych *TSQV4* i tabela *Sales.Customers*
- Oczekiwane dane wyjściowe (w skróconej postaci):

custid	region
-----	-----
55	AK
10	BC
42	BC
45	CA
37	Co. Cork
33	DF
71	ID
38	Isle of Wight
46	Lara
...	
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
...	

(91 row(s) affected)

Rozwiązania

W tym podrozdziale udostępniono rozwiązania ćwiczeń wraz z odpowiednimi wyjaśnieniami.

Ćwiczenie 1

Możemy rozważyć użycie funkcji *YEAR* i *MONTH* w klauzuli *WHERE*, na przykład:

```
USE TSQLV4;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE YEAR(orderdate) = 2015 AND MONTH(orderdate) = 6;
```

Rozwiązanie to jest prawidłowe i zwraca poprawny wynik. Jednak jak już wspominałem wcześniej w tym rozdziale, jeśli stosujemy operacje na filtrowanych kolumnach, w większości przypadków system SQL Server nie może efektywnie używać indeksu (jeśli takie operacje istnieją dla tej kolumny). Z tego względu lepiej będzie posłużyć się filtrem zakresu.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate >= '20150601'
      AND orderdate < '20150701';
```

Ćwiczenie 2

Do wykonania tego zadania możemy użyć funkcji *EOMONTH*, jak na przykład:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate = EOMONTH(orderdate);
```

Możliwe jest również rozwiązanie bardziej skomplikowane. Jako część omówienia funkcji daty i godziny, przedstawiłem następującą postać wyrażenia do obliczenia ostatniego dnia miesiąca odpowiadającego wyspecyfikowanej dacie.

```
DATEADD(month, DATEDIFF(month, '19991231', date_val), '19991231')
```

Jest to bardziej złożona technika, ale ma tę zaletę, że można jej użyć również do wyliczenia końca okresu innego rodzaju, takiego jak rok lub kwartał.

Wyrażenie to najpierw oblicza różnicę miesięcy pomiędzy ostatnim dniem pewnego miesiąca odniesienia (w tym przypadku 31 grudnia 1999) a wyspecyfikowaną datą. Różnica nazwana jest *diff*. Dodając liczbę *diff* miesięcy do daty odniesienia uzyskujemy ostatni dzień wyspecyfikowanego miesiąca daty. Poniżej znajduje się kompletne zapytanie zwracające tylko te zamówienia, dla których data zamówienia jest równa ostatniemu dniowi miesiąca.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate =
    DATEADD(month, DATEDIFF(month, '19991231', orderdate), '19991231');
```

Ćwiczenie 3

W tym ćwiczeniu skorzystamy z wzorca porównania za pomocą predykatu *LIKE*. Pamiętajmy, że znak procentu (%) reprezentuje ciąg znaków dowolnego rozmiaru, a w tym ciąg pusty. Z tego względu możemy użyć wzorca '%a%a%' do wyrażenia co najmniej dwóch wystąpień znaku *a* w dowolnym miejscu ciągu. Poniżej pokazane jest całe zapytanie:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname LIKE '%a%a%';
```

Ćwiczenie 4

To ćwiczenie wymaga zastosowania pewnych sztuczek i jeśli rozwiązáaliśmy je prawidłowo, możemy być z siebie dumni. Pewne subtelne wymaganie w tym żądaniu łatwo można pominąć lub zinterpretować nieprawidłowo. Zwróćmy uwagę, że wymaganie ma postać: „zwraca zamówienia o łącznej wartości większej niż 10000”, a nie „zamówienia o wartości większej niż 10000”. Inaczej mówiąc, wiersze poszczególnych pozycji zamówienia nie muszą spełniać wymagania, natomiast wymagania te powinna spełniać grupa wszystkich pozycji wewnątrz danego zamówienia. Oznacza to, że zapytanie nie powinno zawierać filtru z klauzulą *WHERE*, jak poniżej:

```
WHERE quantity * unitprice > 10000
```

Zapytanie powinno raczej grupować dane według identyfikatora zamówienia i stosować filtr w klauzuli *HAVING*, jak na przykład:

```
HAVING SUM(quantity*unitprice) > 10000
```

Poniżej pokazane jest całe zapytanie.

```
SELECT orderid, SUM(qty*unitprice) AS totalvalue
FROM Sales.OrderDetails
GROUP BY orderid
HAVING SUM(qty*unitprice) > 10000
ORDER BY totalvalue DESC;
```

Ćwiczenie 5

Ktoś mógłby próbować rozwiązać ten problem przy użyciu zapytania podobnego do poniższego :

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname COLLATE Latin1_General_CS_AS LIKE N'[a-z]%' ;
```

Wyrażenie w klauzuli *WHERE* używa słowa kluczowego *COLLATE* w celu zmiany bieżącego ustawienia bez rozróżniania wielkości liter dla kolumny *lastname* na ustawienie z rozróżnianiem. Predykat *LIKE* sprawdza następnie, czy nazwisko zaczyna się od litery z zakresu od *a* do *z*. Trudność w tym miejscu polega na tym, że wybrane ustawienie *collation* używa porządku słownikowego, w którym formy małych i wielkich liter występują naprzemiennie, a nie w oddzielnych grupach. Porządek sortowania wygląda następująco:

aAbBcC...xXyYzZ

Jak można zauważyć, do wyznaczonego przedziału należą nie tylko wszystkie małe litery od *a* do *z*, ale również wielkie litery od *A* do *Y* (wyłączona jest tylko wielka litera *Z*). Tym samym uruchomienie pokazanego zapytania zwróci następujący wynik:

empid	lastname
-----	-----
8	Cameron
1	Davis
9	Doyle
2	Funk
7	King
3	Lew
5	Mortensen
4	Peled
6	Suurs

Aby wyszukać tylko małe litery od *a* do *z*, jednym z możliwych rozwiązań będzie wyliczenie ich jawnie we wzorcu predykatu *LIKE*:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname COLLATE Latin1_General_CS_AS
      LIKE N'[abcdefghijklmnopqrstuvwxyz]%' ;
```

Naturalnie istnieją również inne możliwe rozwiązania.

Chciałbym podziękować Paulowi White, który mi pomógł, gdy sam kiedyś wpadłem w tę pułapkę.

Ćwiczenie 6

Klauzula *WHERE* jest filtrem wierszy, podczas gdy *HAVING* filtruje grupy. Zapytanie 1 filtruje tylko te zamówienia, które zostały złożone przed majem 2016, grupuje je według identyfikatora pracownika i zwraca liczbę zamówień, które obsłużył każdy pracownik spośród tych przefiltrowanych. Innymi słowy, oblicza ono, ile zamówień złożonych przed majem 2016 roku obsłużył każdy pracownik. Zapytanie nie uwzględnia w obliczeniach zamówień złożonych w maju 2016 lub później. Pracownik będzie pojawiać się w danych wyjściowych, o ile obsłużył (lub obsłużyła) przynajmniej jedno zamówienie przed majem 2016, niezależnie od tego, czy zajmował się jakimiś zamówieniami złożonymi później. Wynik zapytania 1 pokazany jest poniżej:

empid	numorders
-----	-----
9	43
3	127
6	67
7	70
1	118
4	154
5	42
2	94
8	101

Zapytanie 2 grupuje wszystkie zamówienia według identyfikatora pracownika, po czym filtruje tylko te grupy, w których maksymalna data aktywności następuje przed majem 2016. Następnie wylicza liczbę zamówień dla każdej grupy (pracownika). Zapytanie pomija całą grupę, jeśli pracownik obsłużył jakiekolwiek zamówienia w maju 2016 lub później. Mówiąc wprost, zapytanie zwraca tych pracowników, którzy nie obsłużyli żadnych zamówień od maja 2016 wraz z całkowitą liczbą przypadających na nich zamówień. Zapytanie to generuje następujący wynik:

empid	numorders
-----	-----
9	43
3	127
6	67
5	42

Dla przykładu zajmijmy się pracownikiem o identyfikatorze 1. Pracownik ten był aktywny zarówno przed, jak i po maju 2016. Wyniki pierwszego zapytania uwzględniają tego pracownika, ale liczba zamówień uwzględnia tylko te, które zostały obsłużone przed majem 2016. Wyniki drugiego zapytania w ogóle nie zawierają tego pracownika.

Ćwiczenie 7

Ponieważ żądanie dotyczy działania w roku 2015, zapytanie powinno zawierać klauzulę *WHERE* z odpowiednim filtrem zakresu dat (*orderdate* >= '20150101' AND *orderdate* < '20160101'). Ponieważ wymaganie dotyczy średnich wartości dostaw w odniesieniu do kraju wysyłki, a dla danego kraju tabela może zawierać wiele wierszy, zapytanie powinno grupować wiersze według kraju i obliczać średnią dostawę. Aby uzyskać trzy kraje o najwyższej średniej dostawie, zapytanie powinno użyć klauzuli *TOP* (3), opierając się na logicznej kolejności malejącej średniej dostawy. Poniżej przedstawiono całe rozwiązanie:

```
SELECT TOP (3) shipcountry, AVG(freight) AS avgfreight
FROM Sales.Orders
WHERE orderdate >= '20150101' AND orderdate < '20160101'
GROUP BY shipcountry
ORDER BY avgfreight DESC;
```

Pamiętajmy, że możemy również użyć standardowej opcji *OFFSET-FETCH* zamiast opcji *TOP*, która nie jest standardem. Poniżej pokazano poprawione rozwiązanie z użyciem opcji *OFFSET-FETCH*.

```
SELECT shipcountry, AVG(freight) AS avgfreight
FROM Sales.Orders
WHERE orderdate >= '20150101' AND orderdate < '20160101'
GROUP BY shipcountry
ORDER BY avgfreight DESC
OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;
```

Ćwiczenie 8

Ponieważ ćwiczenie wymaga, aby obliczenia liczby wierszy zostały wykonane oddzielnie dla każdego klienta, wyrażenie okna powinno zawierać *PARTITION BY custid*. Ponadto wymaganie określało zastosowanie logicznej kolejności według *orderdate*, z użyciem *orderid* jako kryterium rozstrzygania. Z tego względu klauzula *OVER* powinna zawierać *ORDER BY orderdate, orderid*. Poniżej pokazane jest całe rozwiązanie.

```
SELECT custid, orderdate, orderid,
       ROW_NUMBER() OVER(PARTITION BY custid ORDER BY orderdate, orderid) AS rownum
FROM Sales.Orders
ORDER BY custid, rownum;
```

Ćwiczenie 9

Logikę warunkową wymaganą w tym ćwiczeniu możemy obsługiwać przy użyciu wyrażenia *CASE*. Stosując prostą postać wyrażenia *CASE* specyfikujemy atrybut *titleofcourtesy* zaraz po słowie kluczowym *CASE*; wymieniamy wszystkie możliwe tytuły grzecznościowe w oddzielnej klauzuli *WHEN*, po której umieszczona jest klauzula *THEN* i pleć; a w klauzuli *ELSE* specyfikujemy wartość *'Unknown'*.

```
SELECT empid, firstname, lastname, titleofcourtesy,
       CASE titleofcourtesy
         WHEN 'Ms.' THEN 'Female'
         WHEN 'Mrs.' THEN 'Female'
         WHEN 'Mr.' THEN 'Male'
         ELSE      'Unknown'
       END AS gender
FROM HR.Employees;
```

Możemy również użyć wyrażenia *CASE* w postaci z przeszukiwaniem i z dwoma predykatami – jednym do obsługi wszystkich przypadków, gdzie pleć to kobieta i drugim do obsługi wszystkich przypadków, gdzie pleć to mężczyzna oraz klauzuli *ELSE* z wartością *'Unknown'*.

```
SELECT empid, firstname, lastname, titleofcourtesy,
       CASE
         WHEN titleofcourtesy IN('Ms.', 'Mrs.') THEN 'Female'
         WHEN titleofcourtesy = 'Mr.'           THEN 'Male'
         ELSE                                  'Unknown'
       END AS gender
FROM HR.Employees;
```

Ćwiczenie 10

Domyślnie system SQL Server sortuje znaczniki *NULL* przed innymi wartościami (innymi niż *NULL*). Aby znaczniki *NULL* były sortowane ostatnie, możemy użyć wyrażenia *CASE*, które zwraca 1, jeśli kolumna *region* ma wartość *NULL* i 0, jeśli ma wartość inną niż *NULL*. Dzięki temu wyrażeniu znaczniki inne niż *NULL* otrzymają wartość 0; z tego względu sortowane będą przed znacznikami *NULL* (które otrzymują wartość 1). To wyrażenie *CASE* jest używane jako pierwsza kolumna sortowania. Kolumna *region* powinna być wyspecyfikowana jako druga kolumna sortowania. W ten sposób prawidłowe jest sortowanie znaczników innych niż *NULL* wewnątrz ich własnej grupy. Poniżej przedstawiono całe zapytanie.

```
SELECT custid, region
FROM Sales.Customers
ORDER BY
       CASE WHEN region IS NULL THEN 1 ELSE 0 END, region;
```

ROZDZIAŁ 3

Złączenia

Klauzula *FROM* zapytania jest przetwarzana jako pierwsza, zaś wewnątrz niej działają operatory tabel wykonujące wstępne przekształcenia tabel wejściowych. Język T-SQL obsługuje cztery operatory tabel – *JOIN*, *APPLY*, *PIVOT* i *UNPIVOT*. Operator tabeli *JOIN* należy do standardu SQL, natomiast operatory *APPLY*, *PIVOT* i *UNPIVOT* są rozszerzeniami standardu dostępnymi w dialekcie T-SQL. Każdy operator tabeli wykonuje działania na tabelach wskazanych jako dane wejściowe, stosuje właściwy dla siebie zestaw faz logicznego przetwarzania zapytania i zwraca tabelę wynikową. W tym rozdziale skupię się na operatorze tabeli *JOIN*. Operator *APPLY* omówię w rozdziale 5 „Wyrażenia tablicowe”, a operatory *PIVOT* i *UNPIVOT* opisałem w rozdziale 7 „Zaawansowane zagadnienia tworzenia zapytań”.

Operator tabeli *JOIN* (złączenie) działa na dwóch tabelach wejściowych. Trzy podstawowe typy złączeń to: złączenia krzyżowe (*cross join*), wewnętrzne (*inner join*) oraz zewnętrzne (*outer join*). Te trzy typy złączeń różnią się między sobą pod względem występujących w nich faz logicznego przetwarzania kwerendy; każdy typ wykorzystuje inny zestaw. W złączeniu krzyżowym występuje tylko jedna faza – iloczyn kartezjański (*Cartesian Product*). W złączeniu wewnętrznym stosowane są dwie fazy – iloczyn kartezjański oraz filtr (*Filter*), natomiast w złączeniu zewnętrznym stosowane są trzy fazy – iloczyn kartezjański, filtr oraz dodanie wierszy zewnętrznych (*Add Outer Rows*). W tym rozdziale szczegółowo omówię każdy rodzaj złączenia i ich fazy.

Logiczne przetwarzanie zapytania opisuje ogólne kroki logicznego działania, które dla dowolnego wyspecyfikowanego zapytania utworzą prawidłowy wynik, podczas gdy *fizyczne przetwarzanie* to rzeczywiste kroki wykonywania działań przez mechanizm systemu RDBMS. Niektóre fazy logicznego przetwarzania zapytania dla złączeń mogą wydawać się niezbyt efektywne, jednak te niewydajne fazy zostaną zoptymalizowane przez implementację fizyczną. Istotne jest podkreślenie pojęcia *logiczny* w określeniu *logiczne przetwarzanie*. Etapy tego procesu wykonują działania na tabelach wejściowych w oparciu o algebrę relacyjną. Mechanizm bazy danych nie musi być dosłownie zgodny z fazami logicznego przetwarzania zapytania, o ile zagwarantuje, że wygenerowane wyniki są takie same jak wyniki narzucone przez logiczne przetwarzanie. Relacyjny mechanizm systemu SQL Server często stosuje różne skrótowe operacje w celu zoptymalizowania procesu, o ile wiadomo, że nadal zwrócone zostaną prawidłowe wyniki. W tej książce skupiam się na poznaniu logicznych aspektów wykonywania zapytań, jednak chcę podkreślić tę kwestię, by uniknąć nieporozumień.

Złączenia krzyżowe

Z punktu widzenia logiki złączenie krzyżowe (*cross join*) jest najprostszym typem złączenia. Złączenie krzyżowe stosuje tylko jedną fazę przetwarzania logicznego, czyli iloczyn kartezjański. Faza ta wykonuje działania na dwóch tabelach, które dostarczane są jako dane wejściowe złączenia i wyznacza iloczyn kartezjański tych dwóch tabel, czyli każdy wiersz jednej tabeli wejściowej jest łączony ze wszystkimi wierszami drugiej tabeli. Tak więc, jeśli mamy m wierszy w jednej tabeli i n wierszy w innej, w wyniku otrzymamy $m \times n$ wierszy.

Język T-SQL obsługuje dwie standardowe składnie dla złączeń krzyżowych – SQL-92 oraz SQL-89. Z powodów, które przedstawię w dalszej części rozdziału, zalecane jest stosowanie składni SQL-92. Z tych względów właśnie składnia SQL-92 jest traktowana jako podstawowa i jest używana w przykładach. Jednak dla kompletności opisu w tym podrozdziale przedstawię obie składnie.

Składnia ISO/ANSI SQL-92

Pokazane poniżej zapytanie wykonuje złączenie krzyżowe pomiędzy tabelą *Customers* a *Employees* (przy użyciu składni SQL-92) w bazie danych TSQLV4 i w zbiorze wyników zwraca atrybuty *custid* i *empid*.

```
USE TSQLV4;

SELECT C.custid, E.empid
FROM Sales.Customers AS C
      CROSS JOIN HR.Employees AS E;
```

Ponieważ w tabeli *Customers* jest 91 wierszy, a w *Employees* jest 9 wierszy, zapytanie generuje zestaw wyników 819 wierszy, pokazanych w skróconej postaci poniżej:

custid	empid
1	1
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9
2	1
2	2
2	3
2	4
...	

(819 row(s) affected)

Przy stosowaniu składni SQL-92 specyfikujemy słowa kluczowe *CROSS JOIN* pomiędzy dwoma złączanymi tabelami.

Zwróćmy uwagę, że w klauzuli *FROM* w tym zapytaniu tabelom *Customers* i *Employees* przypisane zostały aliasy *C* i *E*, odpowiednio. Zbiór wyników wygenerowany przez złączenie krzyżowe jest wirtualną tabelą o atrybutach wywodzących się z obu stron złączenia. Ponieważ aliasy przypisane zostały do tabel źródłowych, nazwy kolumn tabeli wirtualnej są poprzedzone aliasem tabeli (na przykład *C.custid*, *E.empid*). Jeśli w klauzuli *FROM* tabelom nie przypiszemy aliasów, nazwy kolumn w tabeli wirtualnej będą poprzedzone pełnymi nazwami tabel źródłowych (na przykład *Customers.custid*, *Employees.empid*). Celem prefiksów jest umożliwienie jednoznacznej identyfikacji kolumn, gdy takie same nazwy kolumn występują w obu tabelach. Aliasy tabel są przypisywane dla zapewnienia zwięzłości. Pamiętajmy, że użycie prefiksów kolumn jest wymagane tylko w sytuacji, kiedy występują niejednoznaczne nazwy kolumn (te same nazwy kolumn pojawiają się w więcej niż jednej tabeli); w jednoznacznych przypadkach prefiksy kolumn są opcjonalne. Jednak dobrą praktyką jest stosowanie prefiksów w każdej sytuacji dla zapewnienia przejrzystości. Pamiętajmy także, że jeśli tabeli przypiszemy alias, nieprawidłowe będzie użycie pełnej nazwy tabeli jako prefiksu kolumny; w niejednoznacznych sytuacjach jako prefiksy trzeba wtedy stosować aliasy.

Składnia ISO/ANSI SQL-89

System SQL Server obsługuje również starszą składnię złączeń krzyżowych, która została wprowadzona w standardzie SQL-89. W przypadku tej składni po prostu wpisujemy przecinek pomiędzy nazwami tabel, jak w poniższym kodzie:

```
SELECT C.custid, E.empid  
FROM Sales.Customers AS C, HR.Employees AS E;
```

Nie ma żadnej różnicy logicznej ani wydajnościowej pomiędzy tymi składniami. Obie są integralnymi częściami standardu SQL i obie są w pełni obsługiwane przez najnowszą wersję języka T-SQL. Nie są mi znane żadne plany wycofania starszej składni, a sam również nie widzę żadnych powodów takiego postępowania, skoro jest to integralna część standardu. Niezależnie od tego, zaleca się stosowanie składni SQL-92 z powodów, które staną się jasne po omówieniu działania złączeń wewnętrznych.

Samo-złączenie krzyżowe (Self Cross Join)

Możemy złączyć wiele instancji tej samej tabeli. Możliwość taka nazywana jest samo-złączeniem (*self join*) i jest obsługiwana przez wszystkie podstawowe typy złączeń (krzyżowe, wewnętrzne i zewnętrzne).

Na przykład poniższe zapytanie wykonuje samo-złączenie dwóch instancji tabeli *Employees*.

```
SELECT
    E1.empid, E1.firstname, E1.lastname,
    E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
CROSS JOIN HR.Employees AS E2;
```

Zapytanie to tworzy wszystkie możliwe połączenia par pracowników. Ponieważ tabela *Employees* ma 9 wierszy, zapytanie zwraca 81 wierszy, pokazanych poniżej w skrótovej postaci.

empid	firstname	lastname	empid	firstname	lastname
1	Sara	Davis	1	Sara	Davis
2	Don	Funk	1	Sara	Davis
3	Judy	Lew	1	Sara	Davis
4	Yael	Peled	1	Sara	Davis
5	Sven	Buck	1	Sara	Davis
6	Paul	Suurs	1	Sara	Davis
7	Russell	King	1	Sara	Davis
8	Maria	Cameron	1	Sara	Davis
9	Patricia	Doyle	1	Sara	Davis
1	Sara	Davis	2	Don	Funk
2	Don	Funk	2	Don	Funk
3	Judy	Lew	2	Don	Funk
4	Yael	Peled	2	Don	Funk
5	Sven	Buck	2	Don	Funk
6	Paul	Suurs	2	Don	Funk
7	Russell	King	2	Don	Funk
8	Maria	Cameron	2	Don	Funk
9	Patricia	Doyle	2	Don	Funk
...					

(81 row(s) affected)

Dla samo-złączenia tworzenie aliasów nie jest opcjonalne – bez aliasów tabeli wszystkie nazwy kolumn w zestawie wyników byłyby niejednoznaczne.

Tworzenie tabel liczb

Jedną z sytuacji, w której złączenia krzyżowe są przydatne, jest ich użycie do wygenerowania zbioru wyników zawierających sekwencje liczb całkowitych (1, 2, 3 itd.). Taka sekwencja liczb to bardzo przydatne narzędzie o różnych zastosowaniach. Przy użyciu złączeń krzyżowych możemy bardzo sprawnie tworzyć sekwencje liczb całkowitych.

Rozpoczynamy od utworzenia tabeli nazwanej *Digits* z kolumną nazwaną *digit* i wypełniamy tę tabelę 10 wierszami z cyframi od 0 do 9. Poniższy kod utworzy tabelę *Digits* w bazie danych TSQVL4 (dla celów testowych) i wypełnia ją dziesięcioma cyframi.

```
USE TSQVL4;
```

```

DROP TABLE IF EXISTS dbo.Digits;

CREATE TABLE dbo.Digits(digit INT NOT NULL PRIMARY KEY);

INSERT INTO dbo.Digits(digit)
VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9);

SELECT digit FROM dbo.Digits;

```

Kod ten używa polecenia *INSERT* do wypełnienia tabeli *Digits*. Jeśli czytelnik nie zna składni polecenia *INSERT*, informacje na ten temat można znaleźć w rozdziale 8 „Modyfikowanie danych”.

Kod ten generuje poniższy wynik.

```

digit
-----
0
1
2
3
4
5
6
7
8
9

```

Załóżmy, że chcemy napisać zapytanie, które generuje sekwencję liczb całkowitych w zakresie od 1 do 1000. Możemy skrzyżować trzy instancje tabeli *Digits*, każdą reprezentującą inną potęgę 10 (1, 10, 100). Poprzez utworzenie iloczynu kartezjańskiego trzech instancji tej samej tabeli o 10 wierszach uzyskamy zbiór wyników o 1000 wierszach. W celu wygenerowania rzeczywistej liczby mnożymy cyfry z każdej instancji przez potęgę 10, którą reprezentują, sumujemy wyniki i dodajemy 1. Poniżej pokazano całe zapytanie:

```

SELECT D3.digit * 100 + D2.digit * 10 + D1.digit + 1 AS n
FROM      dbo.Digits AS D1
  CROSS JOIN dbo.Digits AS D2
  CROSS JOIN dbo.Digits AS D3
ORDER BY n;

```

Zapytanie to zwraca następujące dane wyjściowe (pokazane w skróconej postaci).

```

n
-----
1
2
3
4
5
6
7
8

```

```

9
10
...
998
999
1000

```

```
(1000 row(s) affected)
```

Jest to właśnie przykład wygenerowania sekwencji 1000 liczb całkowitych. Jeśli potrzebnych jest więcej liczb, możemy dodać więcej instancji tabeli *Digits*. Na przykład aby utworzyć sekwencję 1 000 000 wierszy, trzeba złączyć 6 instancji.

Złączenia wewnętrzne

Złączenia wewnętrzne stosują dwie logiczne fazy przetwarzania – iloczyn kartezjański pomiędzy dwoma tabelami wejściowymi (jak w złączeniu krzyżowym), a następnie filtrują wiersze w oparciu o wyspecyfikowany predykat. Podobnie jak w przypadku złączenia krzyżowego, złączenia wewnętrzne mają dwie standardowe składnie: SQL-92 i SQL-89.

Składnia ISO/ANSI SQL-92

Przy korzystaniu ze składni SQL-92 specyfikujemy słowa kluczowe *INNER JOIN* pomiędzy nazwami tabel. Słowo kluczowe *INNER* jest opcjonalne, ponieważ złączenie wewnętrzne jest złączeniem domyślnym, tak więc możemy też wpisać samo słowo kluczowe *JOIN*. Predykat używany do filtrowania wierszy umieszczamy w specjalnej klauzuli nazwanej *ON*. Predykat ten jest również nazywany warunkiem złączenia.

Na przykład poniższe zapytanie wykonuje złączenie wewnętrzne pomiędzy tabelą *Employees* i *Orders* w bazie danych TSQLV4, łącząc pracowników i zamówienia w oparciu o predykat *E.empid = O.empid*.

```
USE TSQLV4;
```

```

SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
      JOIN Sales.Orders AS O
      ON E.empid = O.empid;

```

Zapytanie generuje następujące wyniki (pokazane w skróconej postaci):

empid	firstname	lastname	orderid
1	Sara	Davis	10258
1	Sara	Davis	10270
1	Sara	Davis	10275
1	Sara	Davis	10285
1	Sara	Davis	10292

```

...
2      Don      Funk      10265
2      Don      Funk      10277
2      Don      Funk      10280
2      Don      Funk      10295
2      Don      Funk      10300
...
(830 row(s) affected)

```

Większość osób traktuje tego typu złączenia wewnętrzne jako łączenie każdego wiersza pracownika z wszystkimi wierszami zamówień, które mają ten sam identyfikator pracownika co identyfikator danego pracownika. Jest to najprostszy sposób myślenia o złączeniu. Bardziej formalne podejście do złączenia bazujące na algebrze relacyjnej określa, że najpierw złączenie wykonuje iloczyn kartezjański dwóch tabel (9 wierszy pracowników \times 830 wierszy zamówień = 7470 wierszy), a następnie filtruje wiersze w oparciu o predykat $E.empid = O.empid$, co w efekcie daje 830 wierszy. Jak wspomniałem, jest to tylko opis metody logicznej przetwarzania złączenia; w praktyce fizyczne przetwarzanie zapytania przez mechanizm bazy danych może przebiegać zupełnie inaczej.

Przypomnijmy sobie z poprzedniego rozdziału opis trójwartościowej logiki predykatu używanej przez SQL. Podobnie jak w przypadku klauzuli *WHERE* i *HAVING*, klauzula *ON* również zwraca tylko te wiersze, dla których predykat zwraca wartość *TRUE* i nie zwraca wierszy, dla których predykat ma wartość *FALSE* lub *UNKNOWN*.

W bazie danych TSQLV4 wszyscy pracownicy mają odpowiednie zamówienia, tak więc w danych wyjściowych pokazani zostaną wszyscy pracownicy. Gdyby w bazie danych występował pracownicy bez zamówień, w fazie filtrowania pracownicy ci zostaliby odrzuceni.

Składnia ISO/ANSI SQL-89

Podobnie jak w przypadku złączeń krzyżowych, złączenia wewnętrzne można przedstawiać za pomocą składni SQL-89. Pomiedzy nazwami tabel wpisujemy przecinek, tak samo jak dla złączenia krzyżowego, po czym specyfikujemy warunek złączenia w klauzuli *WHERE*, jak w poniższym przykładzie:

```

SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O
WHERE E.empid = O.empid;

```

Zwróćmy uwagę, że składnia SQL-89 nie zawiera klauzuli *ON*.

I ponownie, obie składnie są standardem, w pełni obsługiwanym przez SQL Server i są w ten sam sposób interpretowane przez mechanizm bazy danych, tak więc nie powinniśmy oczekiwać żadnych różnic wydajności. Jedna składnia jest jednak bezpieczniejsza, co zostanie pokazane w kolejnym podpunkcie.

Bezpieczeństwo złączenia wewnętrznego

Zdecydowanie zaleca się, żeby przyzwyczaić się do składni SQL-92 przy tworzeniu złączeń, ponieważ jest pod wieloma względami bezpieczniejsza. Załóżmy, że chcieliśmy napisać zapytanie złączenia wewnętrznego, ale przez pomyłkę pominęliśmy specyfikację warunku złączenia. W przypadku składni SQL-92 zapytanie staje się nieprawidłowe, a mechanizm analizowania kodu wygeneruje błąd. Dla przykładu spróbujmy uruchomić następujący kod:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
      JOIN Sales.Orders AS O;
```

Po jego uruchomieniu pojawi się błąd:

```
Msg 102, Level 15, State 1, Line 3
Incorrect syntax near ';'.
```

(Nieprawidłowa składnia w pobliżu znaku ';')

Choć początkowo może nie być oczywiste, że przyczyną błędu jest brak warunku złączenia, w końcu wpadniemy na to i poprawimy kod. Jeśli jednak zapomnimy wyspecyfikować warunek złączenia posługując się składnią ANSI SQL-89, uzyskamy prawidłowe wyrażenie, które wykona złączenie krzyżowe.

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O;
```

Ponieważ wykonanie zapytania nie generuje komunikatu o błędzie, można przeoczyć błąd logiczny, a w rezultacie użytkownicy naszej aplikacji będą opierać się na nieprawidłowych wynikach. Mało prawdopodobne jest pominięcie przez programistę warunku złączenia w tak krótkim i prostym zapytaniu, jednak większość produkcyjnych zapytań jest znacznie bardziej skomplikowanych i uwzględnia wiele tabel, filtrów i innych elementów. W takich przypadkach wzrasta prawdopodobieństwo przypadkowego pominięcia warunku złączenia.

Jeśli udało mi się przekonać czytelników do stosowania składni SQL-92 w przypadku złączeń wewnętrznych, pojawi się pytanie, dlaczego zalecenie to ma dotyczyć również dla złączeń krzyżowych. Skoro w tym przypadku nie jest stosowany żaden warunek złączenia, można by sądzić, że obie składnie nadają się w tym samym stopniu. Zalecenie stosowania składni SQL-92 dla złączeń krzyżowych wynika jednak z kilku powodów, a głównym jest konsekwencja i zachowanie jednolitości kodu. Ponadto dla przykładu załóżmy, że używamy ANSI SQL-89. Nawet jeśli naprawdę zamierzaliśmy utworzyć złączenie krzyżowe, gdy inni programiści będą oceniać lub utrzymywać nasz kod, skąd mają wiedzieć, czy planowaliśmy użyć złączenia krzyżowego, czy też po prostu zapomnieliśmy wyspecyfikować warunek złączenia?

Dodatkowe rodzaje złączeń

W tym podrozdziale zamieszczam kilka przykładów złączeń występujących na tyle często, że zostały im przypisane specyficzne nazwy: złączenia złożone (*composite join*), nierównościowe (*non-equi join*) i wielokrotne (*multi-join*).

Złączenia złożone

Złączenie złożone czy wieloskładnikowe (*composite join*) to po prostu złączenie bazujące na predykcje, który korzysta z co najmniej dwóch atrybutów z każdej strony. Złączenie złożone jest najczęściej potrzebne, gdy chcemy połączyć dwie tabele w oparciu o zależność klucz główny – klucz obcy i zależność ta jest wieloskładnikowa. Załóżmy dla przykładu, że w tabeli *dbo.Table2* mamy klucz obcy zdefiniowany dla kolumn *col1*, *col2*, odnoszący się do *dbo.Table1* i kolumn *col1*, *col2* i chcemy napisać zapytanie łączące te dwie tabele w oparciu o zależność klucz główny – klucz obcy. Klauzula *FROM* takiego zapytania będzie miała następującą postać:

```
FROM dbo.Table1 AS T1
  JOIN dbo.Table2 AS T2
    ON T1.col1 = T2.col1
   AND T1.col2 = T2.col2
```

Aby sięgnąć po bardziej realistyczny przykład, załóżmy, że trzeba śledzić aktualizacje wartości kolumn w tabeli *OrderDetails* w bazie danych TSQVLV4. Tworzymy niestandardową tabelę inspekcji nazwaną *OrderDetailsAudit*.

```
USE TSQVLV4;
```

```
DROP TABLE IF EXISTS Sales.OrderDetailsAudit
```

```
CREATE TABLE Sales.OrderDetailsAudit
(
  lsn          INT NOT NULL IDENTITY,
  orderid      INT NOT NULL,
  productid    INT NOT NULL,
  dt           DATETIME NOT NULL,
  loginname    sysname NOT NULL,
  columnname   sysname NOT NULL,
  oldval       SQL_VARIANT,
  newval       SQL_VARIANT,
  CONSTRAINT PK_OrderDetailsAudit PRIMARY KEY(lsn),
  CONSTRAINT FK_OrderDetailsAudit_OrderDetails
    FOREIGN KEY(orderid, productid)
      REFERENCES Sales.OrderDetails(orderid, productid)
);
```

Każdy wiersz tabeli inspekcji przechowuje numer seryjny dziennika (*lsn*), klucz zmodyfikowanego wiersza (*orderid*, *productid*), nazwę zmodyfikowanej kolumny (*columnname*), starą wartość (*oldval*), nową wartość (*newval*), informację, kiedy zmiana miała

miejsce (*dt*) oraz kto dokonał zmiany (*loginname*). Tabela ma klucz obcy zdefiniowany w oparciu o atrybuty *orderid*, *productid*, odnoszący się do klucza głównego tabeli *OrderDetails*. Załóżmy, że mamy już przygotowany proces rejestrowania (czyli inspekcji) w tabeli *OrderDetailsAudit* wszystkich zmian wartości kolumn w tabeli *OrderDetails*.

Chcemy napisać zapytanie do tabel *OrderDetails* i *OrderDetailsAudit*, zwracające informacje o wszystkich zmianach wartości, które miały miejsce w kolumnie *qty*. W każdym wierszu wyników trzeba zwrócić bieżącą wartość z tabeli *OrderDetails* oraz wartości przed i po zmianie z tabeli *OrderDetailsAudit*. Trzeba więc złączyć dwie tabele w oparciu o relację klucz główny – klucz obcy, jak w poniższym kodzie:

```
SELECT OD.orderid, OD.productid, OD.qty,
       ODA.dt, ODA.loginname, ODA.oldval, ODA.newval
FROM Sales.OrderDetails AS OD
     JOIN Sales.OrderDetailsAudit AS ODA
       ON OD.orderid = ODA.orderid
        AND OD.productid = ODA.productid
WHERE ODA.columnname = N'qty';
```

Ponieważ zależność oparta jest na wielu atrybutach, warunek złączenia jest złożony.

Złączenie nierównościowe (Non-Equi Join)

Kiedy w warunku złączenia stosowany jest jedynie operator równości, złączenie nazywane jest równościowym (*equi join*). Jeśli zamiast lub oprócz operatora równości w warunku złączenia występuje dowolny inny operator, złączenie nazywane jest nierównościowym (*non-equi join*).



UWAGA Standard SQL zawiera pojęcie *natural join* (złączenie naturalne), które reprezentuje złączenie wewnętrzne oparte o porównanie pomiędzy kolumnami o tej samej nazwie po obu stronach. Na przykład wyrażenie *T1 NATURAL JOIN T2* łączy wiersze w *T1* i *T2* w oparciu o porównanie kolumn o tych samych nazwach. Język T-SQL nie implementuje złączenia naturalnego. Złączenie, które zawiera jawnie podany predykat oparty na operatorze binarnym (równości lub nierówności), jest w standardzie nazywane *theta join* (złączenie *theta*). Tak więc oba typy złączeń, równościowe i nierównościowe, należą do grupy złączeń *theta*.

Jako przykład złączenia nierównościowego, poniższe zapytanie łączy dwie instancje tabeli *Employees*, by utworzyć unikatowe pary pracowników.

```
SELECT
  E1.empid, E1.firstname, E1.lastname,
  E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
     JOIN HR.Employees AS E2
       ON E1.empid < E2.empid;
```


Zwróćmy uwagę na predykat wyspecyfikowany w klauzuli *ON*. Zadaniem tego zapytania jest utworzenie unikatowych par pracowników. Gdyby użyto złączenia krzyżowego, w wynikach znalazłyby się również pary utworzone z tej samej pozycji, sam z sobą (na przykład 1 z 1), a także pary lustrzane (1 z 2, ale również 2 z 1). Przy użyciu warunku złączenia określającego, że klucz z lewej strony musi być mniejszy niż klucz z prawej strony, eliminujemy te niepożądane przypadki. Pary utworzone z jednej pozycji są eliminowane, ponieważ obie strony są równe. W przypadku par lustrzanych kwalifikuje się tylko jeden z dwóch przypadków, ponieważ tylko jeden będzie miał lewy klucz mniejszy niż prawy. W tym przypadku z 81 możliwych par pracowników, które zwróciłyby złączenie krzyżowe, zapytanie zwróci 36 unikatowych par, co ilustruje poniższy wydruk.

empid	firstname	lastname	empid	firstname	lastname
1	Sara	Davis	2	Don	Funk
1	Sara	Davis	3	Judy	Lew
2	Don	Funk	3	Judy	Lew
1	Sara	Davis	4	Yael	Peled
2	Don	Funk	4	Yael	Peled
3	Judy	Lew	4	Yael	Peled
1	Sara	Davis	5	Sven	Buck
2	Don	Funk	5	Sven	Buck
3	Judy	Lew	5	Sven	Buck
4	Yael	Peled	5	Sven	Buck
1	Sara	Davis	6	Paul	Suurs
2	Don	Funk	6	Paul	Suurs
3	Judy	Lew	6	Paul	Suurs
4	Yael	Peled	6	Paul	Suurs
5	Sven	Buck	6	Paul	Suurs
1	Sara	Davis	7	Russell	King
2	Don	Funk	7	Russell	King
3	Judy	Lew	7	Russell	King
4	Yael	Peled	7	Russell	King
5	Sven	Buck	7	Russell	King
6	Paul	Suurs	7	Russell	King
1	Sara	Davis	8	Maria	Cameron
2	Don	Funk	8	Maria	Cameron
3	Judy	Lew	8	Maria	Cameron
4	Yael	Peled	8	Maria	Cameron
5	Sven	Buck	8	Maria	Cameron
6	Paul	Suurs	8	Maria	Cameron
7	Russell	King	8	Maria	Cameron
1	Sara	Davis	9	Patricia	Doyle
2	Don	Funk	9	Patricia	Doyle
3	Judy	Lew	9	Patricia	Doyle
4	Yael	Peled	9	Patricia	Doyle
5	Sven	Buck	9	Patricia	Doyle
6	Paul	Suurs	9	Patricia	Doyle
7	Russell	King	9	Patricia	Doyle
8	Maria	Cameron	9	Patricia	Doyle

(36 row(s) affected)

Jeśli działanie tego zapytania nadal wydaje się niejasne, warto spróbować przetworzyć je w jednym kroku na mniejszym zbiorze pracowników. Załóżmy na przykład że tabela *Employees* zawiera tylko pracowników 1, 2 i 3. Najpierw powstaje iloczyn kartezjański dwóch instancji tabeli.

E1.empid	E2.empid
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

Następnie wiersze są filtrowane na podstawie predykatu $E1.empid < E2.empid$ i w ten sposób zostaniemy tylko z trzema wierszami.

E1.empid	E2.empid
1	2
1	3
2	3

Złączenia wielokrotne (multi-join)

Operator złączania tabel działa tylko na dwóch tabelach, jednak pojedyncze zapytanie może zawierać wiele złączeń. Mówiąc ogólnie, jeśli w klauzuli *FROM* występuje więcej niż jeden operator tabeli, operatory te są logicznie przetwarzane od lewej do prawej. Tak więc tabela wynikowa pierwszego operatora tabeli jest traktowana jako lewa strona danych wejściowych dla drugiego operatora tabeli; wynik działania drugiego operatora tabeli jest traktowany jako lewa strona danych wejściowych dla trzeciego operatora tabeli itd. Jeśli więc w klauzuli *FROM* istnieje wiele złączeń, pierwsze złączenie działa na dwóch tabelach bazowych, ale dla wszystkich pozostałych złączeń lewa strona danych wejściowych to wynik poprzedniego złączenia. W przypadku złączeń krzyżowych i wewnętrznych mechanizm bazy danych może (i często ma to miejsce) wewnętrznie zmienić porządek operacji, by zoptymalizować proces, ponieważ nie wpływa to na poprawność wyników zapytania.

Poniższy przykład łączy tabele *Customers* i *Orders*, dopasowując klientów do ich zamówień, a następnie łączy wynik pierwszego złączenia z tabelą *OrderDetails*, by dopasować zamówienia do poszczególnych pozycji.

```
SELECT
    C.custid, C.companyname, O.orderid,
    OD.productid, OD.qty
FROM Sales.Customers AS C
```

```

JOIN Sales.Orders AS O
  ON C.custid = O.custid
JOIN Sales.OrderDetails AS OD
  ON O.orderid = OD.orderid;

```

Zapytanie zwraca następujące rezultaty (pokazane w skróconej postaci):

custid	companyname	orderid	productid	qty

85	Customer ENQZT	10248	11	12
85	Customer ENQZT	10248	42	10
85	Customer ENQZT	10248	72	5
79	Customer FAPSM	10249	14	9
79	Customer FAPSM	10249	51	40
34	Customer IBVRG	10250	41	10
34	Customer IBVRG	10250	51	35
34	Customer IBVRG	10250	65	15
84	Customer NRCSK	10251	22	6
84	Customer NRCSK	10251	57	15
...				

(2155 row(s) affected)

Złączenia zewnętrzne

W porównaniu z innymi typami, złączenia zewnętrzne są zwykle trudniejsze do wytłumaczenia. Najpierw opiszę podstawowe zasady dotyczące złączeń zewnętrznych. Jeśli na koniec podrödziału „Podstawy złączeń zewnętrznych” materiał będzie zrozumiały, można przejść do opcjonalnego podrödziału, opisującego aspekty złączeń zewnętrznych wykraczające poza informacje podstawowe. W przeciwnym razie czytelnik może spokojnie opuścić tę część i powrócić do niej, kiedy lepiej zrozumie ten materiał.

Podstawy złączeń zewnętrznych

Złączenia zewnętrzne wprowadzono w standardzie SQL-92 i, w przeciwieństwie do złączeń krzyżowych czy wewnętrznych, istnieje dla nich tylko jedna składnia standardowa – ta, w której słowo kluczowe *JOIN* jest umieszczone pomiędzy nazwami tabel, a warunek złączenia jest specyfikowany w klauzuli *ON*. W złączeniach zewnętrznych wykonywane są dwie fazy logicznego przetwarzania znane już ze złączeń wewnętrznych (iloczyn kartezjański i filtr *ON*) oraz trzecia faza nazwana *Adding Outer Rows* (dodawanie wierszy zewnętrznych), która jest unikatowa dla tego typu złączeń.

W złączeniu zewnętrznym oznaczamy tabelę wejściową jako tabelę „zachowywaną” (*preserved*) przy użyciu słów kluczowych *LEFT OUTER JOIN*, *RIGHT OUTER JOIN* lub *FULL OUTER JOIN* pomiędzy nazwami tabel. Słowo kluczowe *OUTER* jest opcjonalne. Słowo kluczowe *LEFT* oznacza, że zachowywane są wszystkie wiersze z lewej tabeli; słowo kluczowe *RIGHT* oznacza, że zachowywane są wiersze tabeli z prawej strony;

a słowo kluczowe *FULL* oznacza, że zachowywane są wiersze obu tabel, z lewej i prawej strony. Trzecia faza logicznego przetwarzania złączenia zewnętrznego identyfikuje wiersze z zachowywanej tabeli, dla których nie znaleziono pasujących wierszy w drugiej tabeli w oparciu o predykat *ON*. Faza ta dodaje te wiersze do tabeli wynikowej wygenerowanej przez dwie pierwsze fazy złączenia; w tych zewnętrznych wierszach używa znaczników *NULL* jako wypełniacza dla atrybutów pochodzących z drugiej strony złączenia.

Działanie złączeń zewnętrznych najłatwiej wyjaśnić na przykładzie. Poniższe zapytanie łączy tabele *Customers* i *Orders* na podstawie identyfikatora klienta, by zwrócić listę klientów i ich zamówienia. Złączenie zewnętrzne jest typu lewostronnego; z tego względu zapytanie zwraca także tych klientów, którzy nie złożyli żadnego zamówienia.

```
SELECT C.custid, C.companyname, O.orderid
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid;
```

Zapytanie zwraca następujące dane wyjściowe (pokazane w skróconej postaci):

custid	companyname	orderid
1	Customer NRZBB	10643
1	Customer NRZBB	10692
1	Customer NRZBB	10702
1	Customer NRZBB	10835
1	Customer NRZBB	10952
...		
21	Customer KIDPX	10414
21	Customer KIDPX	10512
21	Customer KIDPX	10581
21	Customer KIDPX	10650
21	Customer KIDPX	10725
22	Customer DTD MN	NULL
23	Customer WVFAF	10408
23	Customer WVFAF	10480
23	Customer WVFAF	10634
23	Customer WVFAF	10763
23	Customer WVFAF	10789
...		
56	Customer QNIVZ	10684
56	Customer QNIVZ	10766
56	Customer QNIVZ	10833
56	Customer QNIVZ	10999
56	Customer QNIVZ	11020
57	Customer WVAXS	NULL
58	Customer AHXHT	10322
58	Customer AHXHT	10354
58	Customer AHXHT	10474
58	Customer AHXHT	10502
58	Customer AHXHT	10995
...		

```
91      Customer CCFIZ 10792
91      Customer CCFIZ 10870
91      Customer CCFIZ 10906
91      Customer CCFIZ 10998
91      Customer CCFIZ 11044
```

(832 row(s) affected)

Dwóch klientów w tabeli *Customers* nie złożyło żadnych zamówień. Ich identyfikatory to 22 i 57. Zwróćmy uwagę, że w danych wyjściowych dla obu klientów użyta jest wartość *NULL* w atrybutach z tabeli *Orders*. Z punktu widzenia logiki, wiersze tych dwóch klientów zostały odfiltrowane przez drugą fazę złączenia (filtr oparty o predykat *ON*), jednak trzecia faza dodała te wiersze jako wiersze zewnętrzne. Gdyby złączenie było typu wewnętrznego, te dwa wiersze nie zostałyby zwrócone. Te dwa wiersze zostały dodane, by zachować wszystkie wiersze lewej tabeli.

Być może łatwiej będzie zrozumieć wyniki złączenia zewnętrznego, jeśli będziemy uważać, że mamy dwa rodzaje wierszy pochodzące z zachowywanej strony złączenia – wiersze wewnętrzne i wiersze zewnętrzne. Wiersze wewnętrzne to te, dla których zostało znalezione dopasowanie z drugiej strony w oparciu o predykat *ON*, a wiersze zewnętrzne to te, dla których takiego dopasowania nie mamy. Złączenie wewnętrzne zwraca tylko wiersze wewnętrzne, natomiast złączenie zewnętrzne zwraca zarówno wiersze wewnętrzne, jak i zewnętrzne.

Typowa kwestia związana ze złączeniami zewnętrznymi, która jest źródłem wielu nieporozumień, dotyczy tego, czy predykat specyfikować w klauzuli *ON*, czy *WHERE*. Jak można się przekonać, w odniesieniu do wierszy z zachowywanej strony złączenia zewnętrznego filtr bazujący na predykanie *ON* nie jest ostateczny. Mówiąc inaczej, predykat *ON* nie determinuje, czy wiersz będzie pokazywany w danych wyjściowych, a tylko to, czy wiersz będzie pasował do wierszy z drugiej strony złączenia. Tak więc, jeśli trzeba zdefiniować predykat, który nie jest ostatecznym, czyli predykat, który określa, które wiersze pasują do wierszy z niezachowywanej strony, to predykat specyfikujemy w klauzuli *ON*. Jeśli potrzebny jest nam filtr, który zostanie zastosowany po utworzeniu wierszy zewnętrznych i chcemy, by był to końcowy filtr, specyfikujemy predykat w klauzuli *WHERE*. Klauzula *WHERE* jest przetwarzana po klauzuli *FROM* – w szczególności po przetworzeniu wszystkich operatorów tabel i (w przypadku złączeń zewnętrznych) po utworzeniu wszystkich wierszy zewnętrznych. Dodatkowo klauzula *WHERE* jest ostateczna, jeśli chodzi o odfiltrowanie wierszy.

Żałóśmy, że trzeba uzyskać listę tylko tych klientów, którzy nie złożyli żadnego zamówienia, czyli, mówiąc bardziej technicznie, trzeba zwrócić tylko wiersze zewnętrzne. Możemy posłużyć się poprzednim zapytaniem jako podstawą i dodać klauzulę *WHERE*, która filtruje tylko wiersze zewnętrzne. Pamiętamy, że wiersze zewnętrzne są identyfikowane przez znaczniki *NULL* w atrybutach z niezachowywanej strony złączenia. Tak więc możemy filtrować tylko te wiersze, w których jeden z atrybutów niezachowywanej strony złączenia ma wartość *NULL*, jak w poniższym kodzie:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderid IS NULL;
```

Zapytanie to zwraca tylko dwa wiersze dla klientów 22 i 57.

```
custid      companyname
-----
22          Customer DTDMM
57          Customer WVAXS
```

(2 row(s) affected)

W tym zapytaniu warto zwrócić uwagę na kilka istotnych kwestii. Przypomnijmy sobie uwagi na temat znaczników *NULL* zamieszczone wcześniej w książce: podczas wyszukiwania znacznika *NULL* trzeba używać operatora *IS NULL*, a nie równości, ponieważ operator równości, porównując coś z *NULL*, zawsze zwróci *UNKNOWN* – nawet jeśli porównywane są dwa znaczniki *NULL*. Ponadto istotny jest też wybór, który atrybut ma być filtrowany po niezachowywanej stronie złączenia. Powinniśmy wybrać atrybut, który może mieć znacznik *NULL* tylko dla wierszy zewnętrznych, ale nie dla innych (nie może mieć znaczników *NULL* pochodzących z tabeli bazowej). Z tych względów istnieją trzy bezpieczne wybory – kolumna klucza głównego, kolumna używana w predykcji złączenia i kolumna zdefiniowana w tabeli wyjściowej jako *NOT NULL*. Kolumna klucza głównego nie może zawierać *NULL*; z tego względu znacznik *NULL* w takiej kolumnie może oznaczać tylko to, że wiersz jest wierszem zewnętrznym. Jeśli w kolumnie złączenia wiersz ma znacznik *NULL*, taki wiersz został odfiltrowany przez drugą fazę złączenia, tak więc może to oznaczać tylko to, że jest to wiersz zewnętrzny. I rzecz jasna znacznik *NULL* w kolumnie zdefiniowanej jako *NOT NULL* może tylko oznaczać to, że wiersz jest wierszem zewnętrznym.

Aby nabrać wprawy w używaniu złączeń zewnętrznych i lepiej poznać ich działanie, warto wykonać ćwiczenia zamieszczone na końcu tego rozdziału.

Złączenia zewnętrzne – zagadnienia zaawansowane

W tym podrozdziale omawiam bardziej zaawansowane aspekty złączeń zewnętrznych, które Czytelnik może potraktować jako lekturę opcjonalną, o ile dobrze zapoznał się z podstawowymi zagadnieniami dotyczącymi złączeń zewnętrznych.

Dołączanie brakujących wartości

Złączeń zewnętrznych możemy używać do zidentyfikowania i dołączania brakujących wartości. Załóżmy na przykład, że trzeba uzyskać wszystkie zamówienia z tabeli *Orders* w bazie danych TSQIV4. Chcemy zapewnić, że w wynikach uzyskamy co najmniej jeden wiersz dla każdej daty z zakresu od 1 stycznia 2013 do 31 grudnia 2015. Nie

chcemy robić niczego specjalnego z datami wewnątrz zakresu, w którym są zamówienia, ale chcemy, by dane wyjściowe zawierały również daty, gdzie nie było żadnych zamówień, ze znacznikami *NULL* jako wypełniaczami w atrybutach zamówienia.

Aby wykonać to zadanie, najpierw napiszemy zapytanie, które zwraca sekwencję wszystkich dat z wybranego zakresu. Następnie możemy wykonać lewostronne złączenie zewnętrzne pomiędzy tym zbiorem a tabelą *Orders*. W ten sposób wyniki będą zawierać także daty bez zamówień.

W celu utworzenia sekwencji dat w danym zakresie zazwyczaj używam pomocniczej tabeli liczb. Tworzę tabelę nazwaną *dbo.Nums* z kolumną nazwaną *n* i wypełniam ją sekwencją liczb całkowitych (1, 2, 3 itd.). Przekonałem się, że pomocnicza tabela liczb jest wyjątkowo przydatnym narzędziem ogólnego przeznaczenia, używanym do rozwiązywania wielu problemów. Tabele taką tworzę tylko raz w bazie danych i wypełniam ją potrzebną ilością liczb. Przykładowa baza danych TSQV4 zawiera już taką tabelę pomocniczą.

Jako pierwszy krok rozwiązania musimy utworzyć sekwencję wszystkich dat w wymaganym zakresie. Zadanie to możemy wykonać, tworząc zapytanie do tabeli *Nums* i filtrując tyle liczb, ile jest dni w wymaganym zakresie dat. Do obliczenia liczby dni użyjemy funkcji *DATEDIFF*. Dodając $n - 1$ dni do początkowego punktu zakresu dat (1 stycznia 2013) uzyskujemy rzeczywistą sekwencję dat. Poniżej pokazane zostało zapytanie realizujące to zadanie:

```
SELECT DATEADD(day, n-1, '20130101') AS orderdate
FROM dbo.Nums
WHERE n <= DATEDIFF(day, '20130101', '20151231') + 1
ORDER BY orderdate;
```

Zapytanie to zwraca sekwencję wszystkich dat w zakresie od 1 stycznia 2013 do 31 grudnia 2015, co w skróconej postaci ilustruje poniższy wydruk:

```
orderdate
-----
2013-01-01
2013-01-02
2013-01-03
2013-01-04
2013-01-05
...
2015-12-27
2015-12-28
2015-12-29
2015-12-30
2015-12-31
(1096 row(s) affected)
```

Kolejny krok to rozszerzenie poprzedniego zapytania poprzez dodanie lewostronnego złączenia zewnętrznego pomiędzy tabelami *Nums* i *Orders*. Warunek złączenia porównuje daty zamówień utworzonych w tabeli *Nums* z atrybutem *orderdate*

(data zamówienia) w tabeli *Orders* przy użyciu wyrażenia *DATEADD(day, Nums.n - 1, '20130101')*, co ilustruje poniższy kod:

```
SELECT DATEADD(day, Nums.n - 1, '20130101') AS orderdate,
       0.orderid, 0.custid, 0.empid
FROM   dbo.Nums
       LEFT OUTER JOIN Sales.Orders AS 0
         ON DATEADD(day, Nums.n - 1, '20130101') = 0.orderdate
WHERE  Nums.n <= DATEDIFF(day, '20130101', '20151231') + 1
ORDER BY orderdate;
```

Zapytanie generuje następujące wyniki (pokazane w skróconej postaci):

orderdate	orderid	custid	empid
2013-01-01	NULL	NULL	NULL
2013-01-02	NULL	NULL	NULL
2013-01-03	NULL	NULL	NULL
2013-01-04	NULL	NULL	NULL
...			
2013-06-29	NULL	NULL	NULL
2013-06-30	NULL	NULL	NULL
2013-07-01	NULL	NULL	NULL
2013-07-02	NULL	NULL	NULL
2013-07-03	NULL	NULL	NULL
2013-07-04	10248	85	5
2013-07-05	10249	79	6
2013-07-06	NULL	NULL	NULL
2013-07-07	NULL	NULL	NULL
2013-07-08	10250	34	4
2013-07-08	10251	84	3
2013-07-09	10252	76	4
2013-07-10	10253	34	3
2013-07-11	10254	14	5
2013-07-12	10255	68	9
2013-07-13	NULL	NULL	NULL
2013-07-14	NULL	NULL	NULL
2013-07-15	10256	88	3
2013-07-16	10257	35	4
...			
2015-12-28	NULL	NULL	NULL
2015-12-29	NULL	NULL	NULL
2015-12-30	NULL	NULL	NULL
2015-12-31	NULL	NULL	NULL

(1446 row(s) affected)

Daty, które nie występują w tabeli *Orders*, pojawiły się w danych wyjściowych ze znacznikami *NULL* w atrybutach zamówienia.

Filtrowanie atrybutów z niezachowywanej strony złączenia zewnętrznego

Przy sprawdzaniu kodu korzystającego ze złączeń zewnętrznych pod kątem błędów logicznych jedną z kwestii, którą trzeba przeanalizować, jest klauzula *WHERE*. Jeśli predykat znajdujący się w klauzuli *WHERE* odnosi się do atrybutu z niezachowywanej strony złączenia przy użyciu wyrażenia w postaci *<atrybut> <operator> <wartość>*, zazwyczaj jest to błąd. Dzieje się tak, ponieważ atrybuty niezachowywanej strony złączenia w wierszach zewnętrznych są znacznikami *NULL*, a wyrażenie typu *NULL <operator> <wartość>* daje w wyniku *UNKNOWN* (chyba że jawnie użyjemy operatora *IS NULL*). Pamiętajmy, że klauzula *WHERE* odfiltrowuje wartości *UNKNOWN*. Taki predykat w klauzuli *WHERE* spowoduje odfiltrowanie wszystkich wierszy zewnętrznych, unieważniając w efekcie złączenie zewnętrzne. Inaczej mówiąc, jest tak, jakby nastąpiła logiczna zmiana typu złączenia – na złączenie wewnętrzne. Tak więc programista albo pomylił się w wyborze typu złączenia, albo pomylił się tworząc predykat. Jeśli nadal jest to niejasne, pomocny będzie następujący przykład:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid
WHERE O.orderdate >= '20140101';
```

Zapytanie wykonuje lewostronne złączenie zewnętrzne pomiędzy tabelami *Customers* i *Orders*. Przed zastosowaniem filtru *WHERE* operator złączenia zwraca wiersze wewnętrzne dla klientów, którzy złożyli zamówienia i wiersze zewnętrzne dla klientów, którzy nie złożyli zamówień (zamieszczając znaczniki *NULL* w atrybutach zamówienia). Predykat *O.orderdate >= '20140101'* w klauzuli *WHERE* przyjmuje wartość *UNKNOWN* dla wszystkich wierszy zewnętrznych, ponieważ wiersze te mają wartość *NULL* w atrybucie *O.orderdate*. Wszystkie wiersze zewnętrzne zostają wyeliminowane przez filtr *WHERE*, o czym można się przekonać przeglądając wyniki (pokazane w skróconej postaci).

custid	companyname	orderid	orderdate

19	Customer RFNQC	10400	2014-01-01
65	Customer NYUHS	10401	2014-01-01
20	Customer THHDP	10402	2014-01-02
20	Customer THHDP	10403	2014-01-03
49	Customer CQRAA	10404	2014-01-03
...			
58	Customer AHXHT	11073	2015-05-05
73	Customer JMIKW	11074	2015-05-06
68	Customer CCKOT	11075	2015-05-06
9	Customer RTXGC	11076	2015-05-06
65	Customer NYUHS	11077	2015-05-06

(678 row(s) affected)

Oznacza to, że użycie złączenia zewnętrznego było bezcelowe. Programista albo popełnił błąd wybierając użycie złączenia zewnętrznego, albo pomylił się w predykcji *WHERE*.

Stosowanie złączeń zewnętrznych w zapytaniach złączeń wielokrotnych

Przypomnijmy zawartą w rozdziale 2 „Zapytania do pojedynczej tabeli” dyskusję na temat operacji jednorazowych (*all-at-once*). Opisałem tam fakt, że wszystkie wyrażenia występujące w tej samej fazie logicznego przetwarzania zapytania są pod względem logicznym oceniane w tym samym momencie. Ta zasada nie stosuje się jednak do przetwarzania operatorów tabel w fazie *FROM*. Operatory tabel są logicznie przetwarzane od lewej do prawej. Zmiana kolejności, w jakiej złączenia zewnętrzne są przetwarzane, może dawać różne wyniki, tak więc nie możemy dowolnie zmieniać ich porządku.

Z logiczną kolejnością przetwarzania złączeń zewnętrznych wiążą się pewne interesujące błędy. Pewien typowy błąd logiczny pojawiający się przy korzystaniu ze złączeń zewnętrznych może być postrzegany jako odmiana błędu pokazanego w poprzednim podrozdziale. Załóżmy, że piszemy wielokrotne złączenie, wykorzystujące złączenie zewnętrzne pomiędzy dwoma tabelami, po czym następuje złączenie wewnętrzne z trzecią tabelą. Jeśli predykat w klauzuli *ON* złączenia wewnętrznego porównuje atrybut niezachowywanej strony złączenia zewnętrznego z atrybutem z trzeciej tabeli, wszystkie wiersze zewnętrzne zostaną odfiltrowane. Pamiętamy, że wiersze zewnętrzne mają znaczniki *NULL* w atrybutach niezachowywanej strony złączenia, a porównanie *NULL* z czymkolwiek daje wartość nieznaną – *UNKNOWN* i zostaje odfiltrowane przez filtr *ON*. Inaczej mówiąc, taki predykat unieważnia złączenie zewnętrzne, a pod względem logicznym pozostaje złączenie wewnętrzne. Dla przykładu przeanalizjmy następujące zapytanie:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

Pierwsze złączenie jest złączeniem zewnętrznym, które zwraca klientów z ich zamówieniami, a także tych klientów, którzy nie złożyli zamówień. Wiersze zewnętrzne, reprezentujące klientów bez zamówień, mają znaczniki *NULL* w atrybutach zamówienia. Drugie złączenie porównuje pozycje zamówienia z tabeli *OrderDetails* z wierszami wyników pierwszego złączenia w oparciu o predykat *O.orderid = OD.orderid*; jednakże w wierszach, reprezentujących klientów bez zamówień, atrybut *O.orderid* to *NULL*. Z tego względu predykat przyjmuje wartość *UNKNOWN*, a te wiersze zostają odrzucone. Dane wynikowe (pokazane w skróconej postaci) nie zawierają klientów 22 i 57 – dwóch klientów, którzy nie złożyli zamówień.

custid	orderid	productid	qty

85	10248	11	12
85	10248	42	10
85	10248	72	5
79	10249	14	9
79	10249	51	40
...			
65	11077	64	2
65	11077	66	1
65	11077	73	2
65	11077	75	4
65	11077	77	2

(2155 row(s) affected)

Mówiąc ogólnie, wiersze zewnętrzne są odrzucane, ilekroć po jakimkolwiek złączeniu zewnętrznym (lewostronnym, prawostronnym czy obustronnym) występuje złączenie wewnętrzne lub prawostronne złączenie zewnętrzne. To stwierdzenie jest rzecz jasna prawdziwe, gdy warunek drugiego złączenia porównuje znaczniki *NULL* z lewej strony z czymś po prawej stronie złączenia.

Istnieje kilka metod obejścia tego problemu. Jeśli chcemy w danych wyjściowych zamieścić klientów bez zamówień, jedną z możliwości jest użycie w drugim kroku lewostronnego złączenia zewnętrznego.

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  LEFT OUTER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

W ten sposób wiersze zewnętrzne utworzone przez pierwsze złączenie nie zostaną odfiltrowane, co ilustruje poniższy wydruk pokazany w skróconej postaci:

custid	orderid	productid	qty

85	10248	11	12
85	10248	42	10
85	10248	72	5
79	10249	14	9
79	10249	51	40
...			
65	11077	64	2
65	11077	66	1
65	11077	73	2
65	11077	75	4
65	11077	77	2
22	NULL	NULL	NULL
57	NULL	NULL	NULL

(2157 row(s) affected)

To rozwiązanie zwykle nie jest jednak dobre, gdyż zachowuje wszystkie wiersze z tabeli *Orders*. Co jednak, jeśli w tabeli *Orders* mogą znajdować się wiersze, które nie mają pasujących do nich pozycji z *OrderDetails*, i chcielibyśmy odrzucić te wiersze? Tym, czego potrzebujemy, jest złączenie wewnętrzne pomiędzy *Orders* a *OrderDetails*.

Druga możliwość polega na złączeniu najpierw tabel *Orders* i *OrderDetails* przy użyciu złączenia wewnętrznego, a następnie złączeniu wyniku z tabelą *Customers* przy użyciu prawostronnego złączenia zewnętrznego.

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
      JOIN Sales.OrderDetails AS OD
        ON O.orderid = OD.orderid
      RIGHT OUTER JOIN Sales.Customers AS C
        ON O.custid = C.custid;
```

W ten sposób wiersze zewnętrzne tworzone są przez ostatnie złączenie i nie zostają odfiltrowane.

Trzecia opcja polega na użyciu nawiasów, by wydzielić złączenie wewnętrzne pomiędzy tabelami *Orders* i *OrderDetails* do niezależnej fazy logicznej. W ten sposób możemy zastosować lewostronne złączenie zewnętrzne pomiędzy tabelą *Customers* i wynikami złączenia wewnętrznego pomiędzy tabelami *Orders* i *OrderDetails*. Zapytanie będzie wyglądało następująco:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
      LEFT OUTER JOIN
        (Sales.Orders AS O
          JOIN Sales.OrderDetails AS OD
            ON O.orderid = OD.orderid)
        ON C.custid = O.custid;
```

Agregacja *COUNT* w złączeniach zewnętrznych

Inny typowy błąd logiczny dotyczy używania funkcji *COUNT* w złączeniach zewnętrznych. Po zgrupowaniu wyników złączenia zewnętrznego i użyciu funkcji *COUNT(*)* agregacja uwzględnia zarówno wiersze wewnętrzne, jak i zewnętrzne, ponieważ wiersze zliczane są niezależnie od ich zawartości. Zazwyczaj jednak przy zliczaniu nie chcemy uwzględniania wierszy zewnętrznych. Na przykład poniższe zapytanie miało na celu zwrócenie liczby zamówień każdego klienta.

```
SELECT C.custid, COUNT(*) AS numorders
FROM Sales.Customers AS C
      LEFT OUTER JOIN Sales.Orders AS O
        ON C.custid = O.custid
GROUP BY C.custid;
```

Jednakże agregacja *COUNT(*)* zlicza wiersze niezależnie od ich zawartości i klienci, którzy nie złożyli zamówień, tacy jak klient 22 i 57, mają w wynikach złączenia wiersz

zewnętrzny. Jak widzimy w danych wyjściowych pokazanych poniżej w skróconej postaci, na liście znajdują się klienci 22 i 57 z liczbą 1, chociaż rzeczywista liczba złożonych przez nich zamówień to 0.

custid	numorders
1	6
2	4
3	7
4	13
5	18
...	
22	1
...	
57	1
...	
87	15
88	9
89	14
90	7
91	7

(91 row(s) affected)

Agregacja *COUNT(*)* nie potrafi odróżnić, czy wiersz naprawdę reprezentuje zamówienie. W celu poprawienia tego błędu powinniśmy użyć funkcji *COUNT(<column>)*, a nie *COUNT(*)*, wskazując kolumnę z niezachowywanej strony złączenia. W ten sposób agregacja *COUNT()* zignoruje wiersze zewnętrzne, ponieważ w tej kolumnie mają one wartość *NULL*. Pamiętajmy, by używać kolumn, które mogą zawierać *NULL* jedynie w przypadku, jeśli wiersz jest wierszem zewnętrznym – dobrym kandydatem będzie kolumna klucza głównego (*orderid*).

```
SELECT C.custid, COUNT(O.orderid) AS numorders
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid
GROUP BY C.custid;
```

Zwróćmy uwagę w wynikach poniżej (pokazanych w skróconej postaci), że obecnie dla klientów 22 i 57 pokazywana liczba zamówień to 0.

custid	numorders
1	6
2	4
3	7
4	13
5	18
...	
22	0
...	
57	0

...

87	15
88	9
89	14
90	7
91	7

(91 row(s) affected)

Podsumowanie

W rozdziale tym omówiłem operator tabel *JOIN*. Opisałem fazy logicznego przetwarzania kwerendy wykorzystywane w trzech podstawowych typach złączeń – krzyżowych, wewnętrznych i zewnętrznych. Rozdział ten zawierał także inne przykłady złączeń, jak złączenia złożone, nierównościowe i wielokrotne. Zakończyłem opcjonalną częścią przedstawiającą bardziej zaawansowane aspekty złączeń zewnętrznych. Aby przyswoić sobie w praktyce poznany materiał, warto wykonać ćwiczenia zamieszczone w tym rozdziale.

Ćwiczenia

W tym podrozdziale proponuję ćwiczenia, które ułatwią lepsze przyswojenie tematyki opisanej w rozdziale. Wszystkie ćwiczenia korzystają z obiektów bazy danych TSQIV4.

Ćwiczenie 1-1

Napisać zapytanie, które generuje po pięć kopii każdego wiersza pracownika.

- Wykorzystywane tabele: *HR.Employees* i *dbo.Nums*
- Oczekiwane dane wyjściowe (w skrócie):

empid	firstname	lastname	n
1	Sara	Davis	1
2	Don	Funk	1
3	Judy	Lew	1
4	Yael	Peled	1
5	Sven	Buck	1
6	Paul	Suurs	1
7	Russell	King	1
8	Maria	Cameron	1
9	Patricia	Doyle	1
1	Sara	Davis	2
2	Don	Funk	2
...			
3	Judy	Lew	5
4	Yael	Peled	5

5	Sven	Buck	5
6	Paul	Suurs	5
7	Russell	King	5
8	Maria	Cameron	5
9	Patricia	Doyle	5

(45 row(s) affected)

Ćwiczenie 1-2 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie, które zwraca wiersz dla każdego pracownika i dla każdego dnia z zakresu 12 czerwca 2016 do 16 czerwca 2016.

- Wykorzystywane tabele: *HR.Employees* i *dbo.Nums*
- Oczekiwane dane wyjściowe:

empid	dt
1	2016-06-12
1	2016-06-13
1	2016-06-14
1	2016-06-15
1	2016-06-16
2	2016-06-12
2	2016-06-13
2	2016-06-14
2	2016-06-15
2	2016-06-16
...	
7	2016-06-12
7	2016-06-13
7	2016-06-14
7	2016-06-15
7	2016-06-16
8	2016-06-12
8	2016-06-13
8	2016-06-14
8	2016-06-15
8	2016-06-16
9	2016-06-12
9	2016-06-13
9	2016-06-14
9	2016-06-15
9	2016-06-16

(45 row(s) affected)

Ćwiczenie 2

Wyjaśnić, na czym polega błąd w poniższym zapytaniu i zaproponować właściwą alternatywę:

```
SELECT Customers.custid, Customers.companyname, Orders.orderid,
       Orders.orderdate
FROM Sales.Customers AS C
     INNER JOIN Sales.Orders AS O
       ON Customers.custid = Orders.custid;
```

Ćwiczenie 3

Napisać zapytanie, które zwraca klientów z USA i dla każdego klienta zwraca łączną liczbę zamówień i łączną ilość zamówionego towaru.

- Wykorzystywane tabele: *Sales.Customers*, *Sales.Orders* i *Sales.OrderDetails*
- Oczekiwane dane wyjściowe:

custid	numorders	totalqty
32	11	345
36	5	122
43	2	20
45	4	181
48	8	134
55	10	603
65	18	1383
71	31	4958
75	9	327
77	4	46
78	3	59
82	3	89
89	14	1063

(13 row(s) affected)

Ćwiczenie 4

Napisać zapytanie zwracające klientów i ich zamówienia, wliczając w to klientów, którzy nie złożyli zamówień.

- Wykorzystywane tabele: *Sales.Customers* i *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

custid	companyname	orderid	orderdate
85	Customer ENQZT	10248	2013-07-04
79	Customer FAPSM	10249	2013-07-05
34	Customer IBVRG	10250	2013-07-08
84	Customer NRCSK	10251	2013-07-08
...			

73	Customer	JMIKW	11074	2015-05-06
68	Customer	CCKOT	11075	2015-05-06
9	Customer	RTXGC	11076	2015-05-06
65	Customer	NYUHS	11077	2015-05-06
22	Customer	DTDMN	NULL	NULL
57	Customer	WVAXS	NULL	NULL

(832 row(s) affected)

Ćwiczenie 5

Napisać zapytanie zwracające klientów, którzy nie złożyli żadnego zamówienia.

- Wykorzystywane tabele: *Sales.Customers* i *Sales.Orders*
- Oczekiwane dane wyjściowe:

custid	companyname
22	Customer DTDMN
57	Customer WVAXS

(2 row(s) affected)

Ćwiczenie 6

Napisać zapytanie, które zwraca klientów z zamówieniami złożonymi 12 lutego 2016 wraz z ich zamówieniami.

- Wykorzystywane tabele: *Sales.Customers* i *Sales.Orders*
- Oczekiwane dane wyjściowe:

custid	companyname	orderid	orderdate
66	Customer LHANT	10443	2016-02-12
5	Customer HGV LZ	10444	2016-02-12

(2 row(s) affected)

Ćwiczenie 7 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie zwracające wszystkich klientów, a dla tych, którzy złożyli zamówienia 12 lutego 2016, zwróci również te zamówienia.

- Wykorzystywane tabele: *Sales.Customers* i *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

custid	companyname	orderid	orderdate
72	Customer AHP OP	NULL	NULL
58	Customer AHX HT	NULL	NULL
25	Customer AZJ ED	NULL	NULL
18	Customer BSVAR	NULL	NULL

91	Customer CCFIZ	NULL	NULL
...			
33	Customer FVXPQ	NULL	NULL
53	Customer GCJSG	NULL	NULL
39	Customer GLLAG	NULL	NULL
16	Customer GYBBY	NULL	NULL
4	Customer HFBZG	NULL	NULL
5	Customer HGVLZ	10444	2016-02-12
42	Customer IAIJK	NULL	NULL
34	Customer IBVRG	NULL	NULL
63	Customer IRRVL	NULL	NULL
73	Customer JMIKW	NULL	NULL
15	Customer JUWXX	NULL	NULL
...			
21	Customer KIDPX	NULL	NULL
30	Customer KSLQF	NULL	NULL
55	Customer KZQZT	NULL	NULL
71	Customer LCOUJ	NULL	NULL
77	Customer LCYBZ	NULL	NULL
66	Customer LHANT	10443	2016-02-12
38	Customer LJUCA	NULL	NULL
59	Customer LOLJO	NULL	NULL
36	Customer LVJSO	NULL	NULL
64	Customer LWGMD	NULL	NULL
29	Customer MDLWA	NULL	NULL
...			

(91 row(s) affected)

Ćwiczenie 8 (zaawansowane ćwiczenie opcjonalne)

Wyjaśnić, dlaczego poniższe zapytanie nie jest poprawnym rozwiązaniem dla ćwiczenia 7:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON O.custid = C.custid
WHERE O.orderdate = '20160212'
OR O.orderid IS NULL;
```

Ćwiczenie 9 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie, które zwraca wszystkich klientów i dla każdego z nich zwraca wartość Yes/No w zależności od tego, czy klient złożył zamówienie 12 lutego 2016.

- Wykorzystywane tabele: *Sales.Customers* i *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

custid	companyname	HasOrderOn20160212
-----	-----	-----
1	Customer NRZBB	No

2	Customer MLTDN	No
3	Customer KBUDE	No
4	Customer HFBZG	No
5	Customer HGVLZ	Yes
6	Customer XHXJV	No
7	Customer QXVLA	No
8	Customer QUHWH	No
9	Customer RTXGC	No
10	Customer EEALV	No
...		

(91 row(s) affected)

Rozwiązania

W tym podrozdziale zawarte są rozwiązania ćwiczeń.

Ćwiczenie 1-1

Zadanie utworzenia wielu kopii wierszy można zrealizować za pomocą złączenia krzyżowego. Jeśli chcemy utworzyć pięć kopii wiersza każdego pracownika, trzeba wykonać złączenie krzyżowe pomiędzy tabelą *Employees* i tabelą, która ma pięć wierszy; alternatywnym rozwiązaniem może być wykonanie złączenia krzyżowego pomiędzy tabelą *Employees* i tabelą, która ma więcej niż pięć wierszy, ale poprzez filtrowanie klauzuli *WHERE* pozostawionych jest tylko pięć wierszy. Tabela *Nums* jest bardzo wygodnym narzędziem do tego celu. Po prostu krzyżujemy tabelę *Employees* i *Nums* i z tabeli *Nums* filtrujemy tyle wierszy, ile musimy utworzyć kopii (w tym przypadku pięć). Poniżej zapytanie realizujące to zadanie.

```
SELECT E.empid, E.firstname, E.lastname, N.n
FROM HR.Employees AS E
CROSS JOIN dbo.Nums AS N
WHERE N.n <= 5
ORDER BY n, empid;
```

Ćwiczenie 1-2

Ćwiczenie to jest rozszerzeniem poprzedniego. Zamiast żądania utworzenia wstępnie określonej stałej liczby kopii wiersza każdego pracownika, mamy utworzyć kopię dla każdego dnia z pewnego zakresu dat. Tak więc musimy tu obliczyć liczbę dni w żądanym zakresie dat przy użyciu funkcji *DATEDIFF* i odnieść się do wyniku tego wyrażenia w klauzuli *WHERE*, a nie do stałej. W celu utworzenia dat po prostu dodajemy *n - 1* dni do daty początkowej żądanego zakresu, jak w poniższym kodzie:

```
SELECT E.empid,
DATEADD(day, D.n - 1, '20160612') AS dt
FROM HR.Employees AS E
```

```
CROSS JOIN dbo.Nums AS D
WHERE D.n <= DATEDIFF(day, '20160612', '20160616') + 1
ORDER BY empid, dt;
```

Funkcja *DATEDIFF* zwraca 4, ponieważ różnica pomiędzy 12 czerwca 2016 a 16 czerwca 2016 to 4 dni. Do wyniku dodajemy 1 i otrzymujemy 5 dni zakresu. Tak więc, klauzula *WHERE* filtruje pięć wierszy z tabeli *Nums*, gdzie *n* jest mniejsze lub równe 5. Poprzez dodanie *n - 1* dni do 12 czerwca 2016 uzyskaliśmy wszystkie daty z zakresu od 12 czerwca 2016 do 16 czerwca 2016.

Ćwiczenie 2

W pierwszym kroku przetwarzania operator *JOIN* przypisuje tabelom *Customers* i *Orders* krótsze aliasy *C* i *O*, odpowiednio. Użycie aliasów powoduje zmianę nazw tabel na użytek zapytania. We wszystkich późniejszych fazach logicznego przetwarzania zapytania oryginalne nazwy tabel nie są dostępne i należy posługiwać się zdefiniowanymi aliasami. Zapytanie można poprawić dwoma sposobami. Pierwszy polega na unikaniu aliasów i użyciu oryginalnych nazw tabel jako prefiksów, jak poniżej:

```
SELECT Customers.custid, Customers.companyname, Orders.orderid,
       Orders.orderdate
FROM Sales.Customers
     INNER JOIN Sales.Orders
       ON Customers.custid = Orders.custid;
```

Alternatywne wyjście to zachowanie aliasów i użycie ich jako prefiksów w pozostałej części zapytania, jak poniżej:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
     INNER JOIN Sales.Orders AS
       O ON C.custid = O.custid;
```

Ćwiczenie 3

Ćwiczenie to wymaga napisania zapytania, które łączy trzy tabele: *Customers*, *Orders* i *OrderDetails*. Zapytanie powinno użyć klauzuli *WHERE* do filtrowania tylko tych wierszy, w których kraj klienta to USA. Ponieważ naszym zadaniem jest zwrócenie posumowanych wartości dla poszczególnych klientów, zapytanie powinno grupować wiersze według identyfikatora klienta. Musimy więc rozwiązać pewien problem, by dla każdego klienta zwrócić prawidłową liczbę zamówień. Poprzez złączenie tabel *Orders* i *OrderDetails* nie uzyskamy w wyniku tylko jednego wiersza dla zamówienia – otrzymamy po jednym wierszu dla każdej pozycji zamówienia. Tak więc, jeśli w liście *SELECT* użyjemy *COUNT(*)*, dla każdego klienta otrzymamy liczbę pozycji zamówienia, a nie liczbę zamówień. Aby rozwiązać ten problem, każde zamówienie trzeba uwzględnić tylko raz. Możemy to zrobić przy użyciu funkcji *COUNT(DISTINCT*

O.orderid), a nie *COUNT(*)*). Obliczenie łącznej ilości zamówionych towarów nie sprawia szczególnego problemu, ponieważ jest ona powiązana z pozycją zamówienia, a nie z zamówieniem. Poniżej przedstawiono rozwiązanie ćwiczenia:

```
SELECT C.custid, COUNT(DISTINCT O.orderid) AS numorders, SUM(OD.qty) AS
totalqty
FROM Sales.Customers AS C
      JOIN Sales.Orders AS O
          ON O.custid = C.custid
      JOIN Sales.OrderDetails AS OD
          ON OD.orderid = O.orderid
WHERE C.country = N'USA'
GROUP BY C.custid;
```

Ćwiczenie 4

Aby w wynikach pokazać zarówno klientów, którzy złożyli zamówienia, jak i klientów, którzy zamówień nie złożyli, trzeba użyć złączenia zewnętrznego, jak w poniższym kodzie:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
      LEFT OUTER JOIN Sales.Orders AS O
          ON O.custid = C.custid;
```

Zapytanie to zwraca 832 wiersze (w tym klientów 22 i 57, którzy nie złożyli zamówień). Złączenie wewnętrzne pomiędzy tymi tabelami zwróci tylko 830 wierszy, bez tych dwóch klientów.

Ćwiczenie 5

Ćwiczenie to jest rozszerzeniem poprzedniego. Aby zwrócić listę tylko tych klientów, którzy nie złożyli zamówień, trzeba do zapytania dodać klauzulę *WHERE*, która filtruje tylko wiersze zewnętrzne, czyli te, które reprezentują klientów bez żadnych zamówień. Wiersze zewnętrzne mają znaczniki *NULL* w atrybutach z niezachowywanej strony (*Orders*). Aby jednak mieć pewność, że znacznik *NULL* jest wypełniaczem dla wiersza zewnętrznego, a nie pochodzi z pierwotnej tabeli, zalecane jest odwołanie się do atrybutu, który jest kluczem głównym, kolumną złączenia lub takim atrybutem, którego definicja nie zezwala na używanie znaczników *NULL*. Poniższe rozwiązanie w klauzuli *WHERE* odwołuje się do klucza głównego tabeli *Orders*.

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
      LEFT OUTER JOIN Sales.Orders AS O
          ON O.custid = C.custid
WHERE O.orderid IS NULL;
```

Zapytanie to zwraca tylko dwa wiersze – dla klientów 22 i 57, którzy nie złożyli zamówień.

Ćwiczenie 6

Ćwiczenie to wymaga napisania zapytania, które wykonuje złożenie wewnętrzne pomiędzy tabelami *Customers* i *Orders* i filtruje tylko te wiersze, w których data zamówienia to 12 lutego 2016.

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
      JOIN Sales.Orders AS O
        ON O.custid = C.custid
WHERE O.orderdate = '20140212';
```

Klauzula *WHERE* odfiltrowała klientów, którzy nie złożyli zamówień 12 lutego 2016.

Ćwiczenie 7

Rozwiązanie tego ćwiczenie opiera się na poprzednim. Istotną kwestią jest uświadomienie sobie dwóch spraw. Po pierwsze, potrzebne jest złączenie zewnętrzne, ponieważ planujemy uzyskać listę klientów, którzy nie spełniają pewnego kryterium. Po drugie, filtr według daty zamówienia musi pojawić się w klauzuli *ON*, a nie w klauzuli *WHERE*. Pamiętajmy, że filtr *WHERE* jest stosowany po dodaniu wierszy zewnętrznych i jest to końcowe działanie. Naszym celem jest dopasowanie zamówień do klientów, tylko jeśli zamówienie było złożone przez klienta 12 lutego 2016. W danych wyjściowych dalej chcemy uzyskać klientów, którzy tego dnia nie złożyli zamówień; inaczej mówiąc, filtr według daty zamówienia powinien tylko determinować dopasowania, a nie stanowić ostatecznego filtru. W związku z tym klauzula *ON* powinna dopasowywać klientów i zamówienia zarówno w oparciu o równość pomiędzy identyfikatorem klienta w tabeli klientów i identyfikatorem klienta w tabeli zamówień, jak i datą zamówienia, czyli 12 lutego 2016. Poniższy kod realizuje te zadania:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
      LEFT OUTER JOIN Sales.Orders AS O
        ON O.custid = C.custid
        AND O.orderdate = '20140212';
```

Ćwiczenie 8

Zewnętrzne złączenie dopasowuje wszystkich klientów do ich zamówień i zachowuje także tych klientów, którzy nie złożyli żadnych zamówień. Klienci bez zamówień mają znaczniki *NULL* w atrybutach zamówienia. Następnie filtr *WHERE* zachowuje tylko te wiersze, w których data zamówienia to 12 lutego 2016 lub identyfikator zamówienia to *NULL* (czyli klient bez żadnych zamówień). Filtr odrzuca wszystkich klientów,

którzy nie złożyli zamówienia 12 lutego 2016, ale składali je w innych terminach, zaś warunki ćwiczenia 7 wymagały zwrócenia wszystkich klientów. Oto dane wyjściowe tego nieprawidłowego zapytania:

custid	companyname	orderid	orderdate
48	Customer DVFMB	10883	2016-02-12
45	Customer QXPPT	10884	2016-02-12
76	Customer SFOGW	10885	2016-02-12
22	Customer DTDMM	NULL	NULL
57	Customer WVAXS	NULL	NULL

(5 row(s) affected)

Pierwsze trzy wiersze reprezentują zamówienia złożone 12 lutego 2016. Dwa ostatnie odpowiadają klientom, którzy nie złożyli żadnych zamówień. Można zauważyć, że brakuje tu większości spośród 91 klientów z tabeli *Customers*. Jak wspomniałem, są to ci klienci, którzy nie złożyli zamówień 12 lutego 2016, ale dokonali zamówień w innych dniach.

Ćwiczenie 9

Ćwiczenie to jest rozszerzeniem ćwiczenia 7. Tutaj, zamiast zwracać zgodne zamówienia, trzeba zwrócić wartość Yes/No wskazującą, czy istnieje pasujące zamówienie. Pamiętamy, że w złączeniu zewnętrznym brak dopasowania jest identyfikowany jako wiersz zewnętrzny ze znacznikami *NULL* w atrybutach niezachowywanej strony. Tak więc możemy użyć prostego wyrażenia *CASE*, które sprawdza, czy bieżący wiersz jest takim wierszem wewnętrznym i w takim przypadku zwracana jest wartość Yes, a w przeciwnym No. Ponieważ z technicznego punktu widzenia w odniesieniu do klienta mamy więcej niż jedno dopasowanie, do listy *SELECT* powinniśmy dodać klauzulę *DISTINCT*. W ten sposób dla każdego klienta uzyskamy tylko jeden wiersz. Poniższy kod stanowi rozwiązanie ćwiczenia.

```
SELECT DISTINCT C.custid, C.companyname,  
    CASE WHEN O.orderid IS NOT NULL THEN 'Yes' ELSE 'No' END AS  
[HasOrderOn20140212]  
FROM Sales.Customers AS C  
    LEFT OUTER JOIN Sales.Orders AS O  
        ON O.custid = C.custid  
        AND O.orderdate = '20140212';
```


ROZDZIAŁ 4

Podzapytania

Język SQL obsługuje tworzenie zapytań, które zawierają w sobie inne zapytania, czyli *zagnieżdżanie* zapytań. Najbardziej zewnętrznym zapytaniem jest to, którego wyniki zwracane są do wywołującego i jest ono nazywane *zewnętrznym*. Zapytanie wewnętrzne to zapytanie, którego wyniki są używane przez zapytanie zewnętrzne i jest nazywane *podzapytaniem*. Zapytanie wewnętrzne zastępuje wyrażenie oparte na stałych lub zmiennych i jest oceniane w trakcie działania. Inaczej niż w przypadku wyników wyrażań, które używają stałych, wynik podzapytania może zmieniać się ze względu na możliwość zmian w przepytanych tabelach. Przy stosowaniu podzapytań unikamy konieczności rozdzielania etapów rozwiązania i przechowywania pośrednich wyników zapytania w zmiennych.

Podzapytanie może być albo skorelowane, albo niezależne. W przypadku podzapytania niezależnego nie występują zależności od zapytania zewnętrznego, do którego należy to podzapytanie, natomiast w przypadku podzapytania skorelowanego istnieją takie zależności. Podzapytanie może zwracać pojedynczą wartość (skalarną), może zwracać wiele wartości lub całą tabelę.

W tym rozdziale skupię się na podzapytaniach, które zwracają pojedynczą wartość (podzapytania skalarne) oraz na takich, które zwracają wiele wartości (podzapytania wielowartościowe). Podzapytania zwracające całe tabele (podzapytania tablicowe) omówione zostaną w dalszej części książki w rozdziale 5 „Wyrażenia tablicowe”.

Zarówno podzapytania niezależne, jak i skorelowane mogą zwracać pojedyncze wartości (skalarne) lub mogą zwracać wiele wartości. Najpierw omówię podzapytania niezależne, ilustrując je przykładami dla wartości pojedynczych i dla wielu wartości. Pokażę także wprost, jak odróżniać, które podzapytania są skalarne, a które wielowartościowe. Następnie opiszę podzapytania skorelowane.

I ponownie, ćwiczenia na końcu rozdziału będą bardzo pomocne w przyswojeniu poznanego materiału.

Podzapytania niezależne

Każde podzapytanie należy do pewnego zapytania zewnętrznego. W przypadku podzapytań niezależnych brak jest zależności od tabel występujących w zapytaniu zewnętrznym. W podzapytaniach niezależnych łatwo usuwać błędy, ponieważ zawsze możemy wydzielić kod podzapytania, uruchomić go i upewnić się, że działa zgodnie z przewidywaniem. Z punktu widzenia logiki kod podzapytania przetwarzany jest tylko raz,

zanim zostanie przetworzone zapytanie zewnętrzne, przy czym to ostatnie korzysta z wyników podzapytania. Kolejne podrozdziały omawiają konkretne przykłady podzapytań niezależnych.

Przykłady skalarnych podzapytań niezależnych

Skalarne podzapytanie zwraca pojedynczą wartość – niezależnie od tego, czy jest to zapytanie niezależne, czy skorelowane. Podzapytanie takie może występować wszędzie tam, gdzie może pojawiać się wyrażenie o pojedynczej wartości (takie jak *WHERE* lub *SELECT*).

Załóżmy na przykład, że potrzebujemy utworzyć zapytanie do tabeli *Orders* w bazie danych *TSQLV4*, zwracające informacje o zamówieniu, które ma maksymalny identyfikator zamówienia. Zadanie to możemy wykonać przy użyciu zmiennej. Kod może uzyskać maksymalny identyfikator *orderid* z tabeli *Orders* i przechować ten wynik w zmiennej. Następnie można odpytać tabelę *Orders* i wyfiltrować zamówienie, dla którego identyfikator jest równy wartości przechowywanej w zmiennej. Poniższy kod jest ilustracją tej metody:

```
USE TSQLV4;

DECLARE @maxid AS INT = (SELECT MAX(orderid)
                        FROM Sales.Orders);

SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = @maxid;
```

Zapytanie to zwraca następujące wyniki.

orderid	orderdate	empid	custid
11077	2016-05-06	1	65

Możemy zastąpić metodę korzystającą ze zmiennej przez podzapytanie zagnieżdżone. Zrealizujemy to poprzez zastąpienie zmiennej skalarnej podzapytaniem niezależnym, które zwraca maksymalny identyfikator zamówienia. W ten sposób nasze rozwiązanie będzie pojedynczym zapytaniem, a nie dwuetapowym procesem.

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = (SELECT MAX(0.orderid)
                FROM Sales.Orders AS 0);
```

Aby podzapytanie skalarne było prawidłowe, musi zwracać tylko jedną wartość. Jeśli skalarne podzapytanie potrafi zwrócić więcej niż jedną wartość, w czasie działania może pojawić się błąd. Przy użyciu przykładowych danych zawartych w bazie *TSQLV4* poniższe zapytanie działa bez błędu.

```

SELECT orderid
FROM Sales.Orders
WHERE empid =
    (SELECT E.empid
     FROM HR.Employees AS E
     WHERE E.lastname LIKE N'C%');

```

Celem tego zapytania jest zwrócenie identyfikatorów zamówień złożonych przez pracowników, których nazwiska zaczynają się od litery C. Podzapytanie zwraca identyfikatory pracowników, których pierwsza litera nazwiska to C, a zapytanie zewnętrzne zwraca identyfikatory zamówień, dla których identyfikator pracownika jest równy wynikowi podzapytania. Ponieważ operator równości wymaga wyrażeń o pojedynczej wartości z obu stron, podzapytanie uważane jest za skalarne. Jednak ponieważ podzapytanie może potencjalnie zwracać więcej niż jedną wartość, stosowanie w predykacie operatora równości jest złym wyborem. Jeśli podzapytanie zwróci więcej niż jedną wartość, całe zapytanie zakończy się błędem.

Zapytanie to zadziała bez błędu, ponieważ obecnie tabela Employees zawiera tylko jednego pracownika, którego nazwisko rozpoczyna się od litery C (Maria Cameron o identyfikatorze pracownika równym 8). Zapytanie to zwraca następujące wyniki (pokazane w skróconej postaci):

```

orderid
-----
10262
10268
10276
10278
10279
...
11054
11056
11065
11068
11075

(42 row(s) affected)

```

Oczywiście, jeśli podzapytanie zwróci więcej niż jedną wartość, całe zapytanie się nie powiedzie. Dla przykładu spróbujmy uruchomić je dla pracowników, których nazwisko rozpoczyna się od litery D.

```

SELECT orderid
FROM Sales.Orders
WHERE empid =
    (SELECT E.empid
     FROM HR.Employees AS E
     WHERE E.lastname LIKE N'D%');

```

Jednak nazwiska dwóch pracowników rozpoczynają się od litery D (Sara Davis i Patricia Doyle). Z tego względu po uruchomieniu zapytania generowany jest następujący błąd:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery
follows =, !=, <, <=, >, >= or when the subquery is used as an expression.
```

(Podzapytanie zwróciło więcej niż 1 wartość. Działanie takie nie jest dopuszczone, jeśli podzapytanie następuje po =, !=, <, <=, >, >= lub jeśli podzapytanie jest używane jako wyrażenie.)

Jeśli podzapytanie skalarne nie znajdzie żadnej wartości, zwraca znacznik *NULL*. Pamiętamy, że porównanie ze znacznikiem *NULL* daje wartość *UNKNOWN*, a filtry zapytania nie zwrócą wiersza, dla którego wyrażenie zostało ocenione jako *UNKNOWN*. Na przykład, aktualnie tabela *Employees* nie zawiera pracowników, których nazwiska rozpoczynają się od litery A; z tego względu poniższe zapytanie zwraca pusty zbiór.

```
SELECT orderid
FROM Sales.Orders
WHERE empid =
    (SELECT E.empid
     FROM HR.Employees AS E
     WHERE E.lastname LIKE N'A%');
```

Podzapytania niezależne o wielu wartościach

Podzapytanie o wielu wartościach to takie podzapytanie, które w postaci pojedynczej kolumny zwraca wiele wartości bez względu na to, czy jest to podzapytanie niezależne. Istnieją predykaty, takie jak *IN*, które działają na podzapytaniach o wielu wartościach.



UWAGA Istnieją jeszcze inne predykaty, które działają na podzapytaniach wielowartościowych – są to *SOME*, *ANY* i *ALL*. Są one bardzo rzadko używane i dlatego nie są opisywane w tej książce.

Postać predykatu *IN* jest następująca:

<wyrażenie skalarne> IN (<podzapytanie wielowartościowe>)

Predykat przyjmuje wartość *TRUE*, jeśli wyrażenie skalarne równe jest dowolnej wartości zwróconej przez podzapytanie. Przypomnijmy żądanie omawiane w poprzednim podrozdziale – zwrócenie identyfikatorów zamówień obsługiwanych przez pracowników, których nazwisko rozpoczyna się od pewnej litery. Ponieważ więcej niż jeden pracownik może mieć nazwisko rozpoczynające się od danej litery, żądanie to powinno być obsługiwane przez predykat *IN* i podzapytanie wielowartościowe, a nie przez operator równości i podzapytanie skalarne. Na przykład poniższe zapytanie zwraca identyfikatory zamówień złożonych przez pracowników o nazwisku rozpoczynającym się na literę D.

```
SELECT orderid
FROM Sales.Orders
WHERE empid IN
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'D%');
```

Ponieważ używany jest predykat *IN*, zapytanie to jest prawidłowe przy dowolnej liczbie zwracanych wartości – żadna wartość nie jest zwracana, zwracana jest jedna wartość lub zwracanych jest wiele wartości. Uruchomienie zapytania daje następujące wyniki (pokazane w skróconej postaci):

```
orderid
```

```
-----
```

```
10258
10270
10275
10285
10292
...
10978
11016
11017
11022
11058
```

```
(166 row(s) affected)
```

Ktoś mógłby zapytać, dlaczego przy rozwiązywaniu tego zadania nie skorzystać ze złączenia zamiast podzapytań, jak poniżej:

```
SELECT O.orderid
FROM HR.Employees AS E
JOIN Sales.Orders AS O
  ON E.empid = O.empid
WHERE E.lastname LIKE N'D%';
```

Będziemy napotykać wiele innych problemów, które można rozwiązać albo za pomocą podzapytań, albo złączeń. Nie znam uniwersalnej reguły, która podpowie, kiedy podzapytanie jest lepsze niż złączenie. W niektórych przypadkach mechanizm bazy danych interpretuje oba typy zapytań w ten sam sposób. Czasami złączenia mają lepszą wydajność niż podzapytania, a czasami jest odwrotnie. Moja metoda polega na napisaniu zapytania rozwiązującego określone zadanie w sposób intuicyjny, a jeśli wydajność nie jest satysfakcjonująca, wypróbowanie innych wariantów zapytania. Takie poprawki zapytania polegają między innymi właśnie na użyciu złączeń zamiast podzapytań lub podzapytań zamiast złączeń. Warto też zachować alternatywne warianty zapytań – modyfikacje silnika bazodanowego w przyszłości mogą sprawić, że inne formy zapytań okażą się wydajniejsze.

Jako inny przykład użycia podzapytań wielowartościowych założmy, że potrzebujemy zapytania zwracającego zamówienia złożone przez klientów z USA. Możemy

napisać zapytanie do tabeli *Orders* zwracające zamówienia, dla których identyfikator klienta należy do zbioru identyfikatorów klientów z USA. Ostatnią część możemy zrealizować w postaci niezależnego podzapytania wielowartościowego, jak w poniższym kodzie:

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
    (SELECT C.custid
     FROM Sales.Customers AS C
     WHERE C.country = N'USA');
```

Zapytanie to zwraca następujące rezultaty (pokazane w skróconej postaci):

custid	orderid	orderdate	empid
65	10262	2014-07-22	8
89	10269	2014-07-31	5
75	10271	2014-08-01	6
65	10272	2014-08-02	6
65	10294	2014-08-30	4
...			
32	11040	2016-04-22	4
32	11061	2016-04-30	4
71	11064	2016-05-01	1
89	11066	2016-05-01	7
65	11077	2016-05-06	1

(122 row(s) affected)

Podobnie jak w przypadku każdego innego predykatu, predykat *IN* można zanegować przy użyciu operatora logicznego *NOT*. Na przykład poniższe zapytanie zwraca klientów, którzy nie złożyli żadnego zamówienia.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN
    (SELECT O.custid
     FROM Sales.Orders AS O);
```

Zwróćmy uwagę, że najlepszym rozwiązaniem praktycznym jest uściślenie podzapytania, by wykluczane były znaczniki *NULL*. W tym przykładzie, dla zachowania prostoty, znaczniki *NULL* nie są wykluczane, ale w dalszej części, w podrozdziale „Problemy dotyczące znaczników *NULL*”, szczegółowo omówię to zalecenie.

Niezależne podzapytanie wielowartościowe zwraca wszystkie identyfikatory klientów, które występują w tabeli *Orders*. Oczywiście w tabeli *Orders* występują tylko identyfikatory tych klientów, którzy złożyli zamówienia. Zapytanie zewnętrzne zwraca klientów z tabeli *Customers*, dla których identyfikator klienta nie znajduje się w zbiorze wartości zwracanych przez podzapytanie – inaczej mówiąc, zwraca klientów, którzy nie złożyli zamówień. Zapytanie to zwraca następujące wyniki:

custid	companyname
22	Customer DTDMM
57	Customer WVAXS

Można zastanawiać się, czy wyspecyfikowanie klauzuli *DISTINCT* w podzapytaniu może poprawić wydajność ze względu na to, że w tabeli *Orders* identyfikator klienta może pojawiać się więcej niż raz. Jednak mechanizm bazy danych jest na tyle inteligentny, by uwzględniać usuwanie duplikatów bez konieczności specyfikowania tego wprost, tak więc nie jest to coś, czym musimy się przejmować.

Ostatni przykład w tym podrozdziale pokazuje stosowanie wielu podzapytań niezależnych w tym samym zapytaniu – zarówno podzapytań skalarnych, jak i wielowartościowych. Zanim opiszę zadanie, należy uruchomić poniższy kod, by w bazie danych TSQLV4 utworzyć tabelę nazwaną *dbo.Orders* (na potrzeby testu) i wypełnić ją identyfikatorami zamówień z tabeli *Sales.Orders* o parzystych numerach identyfikatorów.

```
USE TSQLV4;
DROP TABLE IF EXIST dbo.Orders;
CREATE TABLE dbo.Orders(orderid INT NOT NULL CONSTRAINT PK_Orders PRIMARY KEY);

INSERT INTO dbo.Orders(orderid)
SELECT orderid
FROM Sales.Orders
WHERE orderid % 2 = 0;
```

Polecenie *INSERT* zostanie opisane szczegółowo w rozdziale 8 „Modyfikowanie danych”; na razie wystarczy wiedza, że powoduje ono wstawianie do wskazanej tabeli danych przekazanych jako wyniki dalszego kodu.

Zadanie polega na zwróceniu wszystkich indywidualnych identyfikatorów zamówień, których brakuje pomiędzy minimalną a maksymalną wartością obecną w tabeli. Zadanie takie będzie trudno rozwiązać bez użycia tabel pomocniczych. Bardzo przydatna tutaj będzie tabela *Nums* opisana w rozdziale 3 „Złączenia”. Przypomnijmy, że tabela *Nums* zawiera sekwencję liczb całkowitych rozpoczynając od 1 (bez przerw). Aby zwrócić wszystkie brakujące identyfikatory zamówień z tabeli *Orders*, odpytujemy tabelę *Nums* i filtrujemy tylko te numery, które znajdują się pomiędzy minimum a maksimum w tabeli *dbo.Orders* i jednocześnie nie występują w zbiorze identyfikatorów zamówień w tabeli *Orders*. Możemy użyć skalarnych podzapytań niezależnych do zwrócenia minimalnego i maksymalnego identyfikatora zamówienia oraz wielowartościowego podzapytania niezależnego do zwrócenia zbioru wszystkich istniejących identyfikatorów zamówień. Poniżej pokazano całe rozwiązanie:

```
SELECT n
FROM dbo.Nums
WHERE n BETWEEN (SELECT MIN(O.orderid) FROM dbo.Orders AS O)
AND (SELECT MAX(O.orderid) FROM dbo.Orders AS O)
AND n NOT IN (SELECT O.orderid FROM dbo.Orders AS O);
```

Ponieważ kod, który wypełniał tabelę *dbo.Orders*, przepuszczał tylko parzyste numery identyfikatorów zamówień, zapytanie to zwraca wszystkie nieparzyste numery z zakresu pomiędzy minimalnym a maksymalnym identyfikatorem występującym w tabeli *Orders*. Dane wyjściowe tego zapytania pokazane są poniżej, w skróconej postaci:

```
n
-----
10249
10251
10253
10255
10257
...
11067
11069
11071
11073
11075
```

(414 row(s) affected)

Po wykonaniu zadania uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXISTS dbo.Orders;
```

Podzapytania skorelowane

Podzapytania skorelowane to takie, które odnoszą się do atrybutów tabeli występujących w zapytaniu zewnętrznym. Oznacza to, że podzapytanie jest zależne od zapytania zewnętrznego i nie może być wywoływane niezależnie. Z punktu widzenia logiki przetwarzania jest to sytuacja, w której podzapytanie jest oceniane oddzielnie dla każdego wiersza zewnętrznego. Na przykład zapytanie z listingu 4-1 zwraca zamówienia o maksymalnym identyfikatorze dla każdego klienta.

LISTING 4-1 Podzapytanie skorelowane

```
USE TSQLV4;

SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderid =
    (SELECT MAX(O2.orderid)
     FROM Sales.Orders AS O2
     WHERE O2.custid = O1.custid);
```

Zapytanie zewnętrzne jest wykonywane względem instancji tabeli *Orders* o aliasie *O1*; zapytanie filtruje zamówienia, dla których identyfikator *ID* jest równy wartości zwróconej przez podzapytanie. To z kolei filtruje zamówienia z drugiej instancji tabeli *Orders*

o aliasie *O2*, gdzie wewnętrzny identyfikator klienta jest równy zewnętrznemu identyfikatorowi klienta, i z odfiltrowanych zamówień zwraca maksymalny identyfikator zamówienia. Mówiąc prościej, dla każdego wiersza w tabeli *O1* zadaniem podzapytania jest zwrócenie maksymalnego identyfikatora zamówienia dla bieżącego klienta. Jeśli identyfikator zamówienia w *O1* i identyfikator zamówienia zwrócony przez podzapytanie są zgodne, identyfikator zamówienia w *O1* jest maksymalnym identyfikatorem dla bieżącego klienta, i w takim przypadku wiersz z tabeli *O1* jest zwracany w zbiorze wyników. Zapytanie to zwraca następujące wyniki (pokazane w skróconej postaci):

custid	orderid	orderdate	empid
91	11044	2016-04-23	4
90	11005	2016-04-07	2
89	11066	2016-05-01	7
88	10935	2016-03-09	4
87	11025	2016-04-15	6
...			
5	10924	2016-03-04	3
4	11016	2016-04-10	9
3	10856	2016-01-28	3
2	10926	2016-03-04	4
1	11011	2016-04-09	3

(89 row(s) affected)

Zrozumienie działania zapytań skorelowanych jest zwykle trudniejsze, niż zapytań niezależnych. Dla uproszczenia sugeruję skupienie się na pojedynczym wierszu tabeli zewnętrznej i rozważenie, jakie logiczne przetwarzanie zapytania wewnętrznego odbywa się dla tego wiersza. Dla przykładu weźmy zamówienie o identyfikatorze 10248.

custid	orderid	orderdate	empid
85	10248	2014-07-04	5

Kiedy podzapytanie jest wykonywane dla tego wiersza zewnętrznego, korelacja lub odwołanie do *O1.custid* oznacza 85. Po zastąpieniu korelacji wartością 85 otrzymujemy następujący kod podzapytania:

```
SELECT MAX(O2.orderid)
FROM Sales.Orders AS O2
WHERE O2.custid = 85;
```

Zapytanie to zwraca identyfikator zamówienia 10739. Identyfikator zamówienia wiersza zewnętrznego – 10248 – jest porównywany z identyfikatorem wiersza wewnętrznego – 10739 – i ponieważ w tym przypadku nie ma zgodności, wiersz zewnętrzny zostaje odfiltrowany. Podzapytanie zwraca tę samą wartość dla wszystkich wierszy w tabeli *O1* o tym samym identyfikatorze klienta i tylko w jednym przypadku istnieje zgodność – kiedy dla bieżącego klienta identyfikator zamówienia wiersza zewnętrznego jest maksymalny. Tego rodzaju analiza powinna ułatwić zrozumienie koncepcji podzapytań skorelowanych.

Z faktu, że podzapytania skorelowane są zależne od zapytania zewnętrznego, wynika też, że w porównaniu do zapytań niezależnych trudniej jest usuwać z nich błędy. Nie możemy po prostu wyróżnić części dotyczącej podzapytania i jej uruchomić. Jeśli przykładowo spróbujemy wyróżnić i uruchomić fragment dotyczący podzapytania w listingu 4-1, pojawi się następujący komunikat o błędzie:

```
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier "01.custid" could not be bound.
```

(Nie można było powiązać wieloczęściowego identyfikatora "01.custid")

Błąd ten wskazuje, że identyfikator *01.custid* nie może być powiązany z obiektem zapytania, ponieważ tabela *01* nie została zdefiniowana. Tabela ta jest zdefiniowana jedynie w kontekście zapytania zewnętrznego. W celu usunięcia błędów podzapytań skorelowanych trzeba zastępować korelację przez stałą i po upewnieniu się, że kod jest poprawny, ponownie zamienić stałą na korelację.

W celu przeanalizowania innego przykładu podzapytania skorelowanego założmy, że potrzebne jest zapytanie do widoku *Sales.OrderValues* i uzyskanie dla każdego zamówienia części procentowej wynikającej z podzielenia wartości bieżącego zamówienia przez łączną wartość wszystkich zamówień klienta. W rozdziale 7 „Zaawansowane zagadnienia tworzenia zapytań” przedstawię rozwiązanie tego problemu wykorzystujące funkcje okna; tutaj zajmiemy się rozwiązaniem problemu przy użyciu podzapytań. Zawsze dobrym podejściem jest przeanalizowanie kilku różnych rozwiązań problemu, ponieważ odmienne rozwiązania zwykle różnią się wydajnością i skomplikowaniem.

Możemy utworzyć zapytanie zewnętrzne do instancji widoku *OrderValues* nazwanego *01*; w liście *SELECT* wykonamy dzielenie bieżącej wartości przez wynik podzapytania skorelowanego, które dla bieżącego klienta zwraca łączną wartość zamówień z drugiej instancji *OrderValues* nazwanej *02*. Poniżej pokazano całe rozwiązanie:

```
SELECT orderid, custid, val,
       CAST(100. * val / (SELECT SUM(02.val)
                        FROM Sales.OrderValues AS 02
                        WHERE 02.custid = 01.custid)
           AS NUMERIC(5,2)) AS pct
FROM Sales.OrderValues AS 01
ORDER BY custid, orderid;
```

Funkcja *CAST* została użyta w celu przekształcenia typu danych wyrażenia na typ *NUMERIC* o dokładności 5 (całkowita liczba cyfr) i skalę na 2 (liczba cyfr po przecinku dziesiętnym).

Zapytanie zwraca następujące wyniki:

orderid	custid	val	pct
10643	1	814.50	19.06
10692	1	878.00	20.55
10702	1	330.00	7.72
10835	1	845.80	19.79

10952	1	471.20	11.03
11011	1	933.50	21.85
10308	2	88.80	6.33
10625	2	479.75	34.20
10759	2	320.00	22.81
10926	2	514.40	36.67
...			

(830 row(s) affected)

Predykat *EXISTS*

Język T-SQL obsługuje predykat *EXISTS* (istnieje), który jako dane wejściowe akceptuje podzapytanie i zwraca wartość *TRUE*, jeśli podzapytanie zwraca jakikolwiek wiersz, a w przeciwnym razie zwraca wartość *FALSE*. Na przykład poniższe zapytanie zwraca klientów z Hiszpanii, którzy złożyli zamówienia.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE country = N'Spain'
AND EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

Zapytanie zewnętrzne odwołujące się do tabeli *Customers* filtruje tylko tych klientów z Hiszpanii, dla których predykat *EXISTS* ma wartość *TRUE*. Predykat *EXISTS* zwraca wartość *TRUE*, jeśli bieżący klient ma powiązane z nim zamówienie w tabeli *Orders*.

Jedną z korzyści stosowania predykatu *EXISTS* polega na tym, że pozwala on na budowanie intuicyjnych zapytań podobnych do wyrażen języka naturalnego. Zapytanie to może zostać odczytane tak samo, jakby zostało wypowiedziane po angielsku: *select the customer ID and company name attributes from the Customers table, where the country is equal to Spain, and at least one order exists in the Orders table with the same customer ID as the customer's customer ID* (wybierz identyfikator klienta i atrybuty nazwy firmy z tabeli *Customer*, gdzie kraj to Hiszpania i w tabeli *Orders* istnieje co najmniej jedno zamówienie o tym samym identyfikatorze klienta).

Zapytanie to zwraca następujące wyniki:

custid	companyname
8	Customer QUHWH
29	Customer MDLWA
30	Customer KSLQF
69	Customer SIUIH

Podobnie jak w przypadku innych predykatów, predykat *EXISTS* można zanegować za pomocą operatora logicznego *NOT*. Na przykład poniższe zapytanie zwraca klientów z Hiszpanii, którzy nie złożyli zamówień.

```

SELECT custid, companyname
FROM Sales.Customers AS C
WHERE country = N'Spain'
      AND NOT EXISTS
      (SELECT * FROM Sales.Orders AS O
       WHERE O.custid = C.custid);

```

Wyniki zapytania są następujące:

custid	companyname
22	Customer DTDMM

Wprawdzie ta książka skupia się na logicznym przetwarzaniu zapytań, a nie na ich wydajności, dobrze będzie jednak wiedzieć, że użycie predykatu *EXISTS* prowadzi do dobrej optymalizacji. Silnik bazy danych wie, że wystarczy określić, czy podzapytanie zwraca co najmniej jeden wiersz lub nie zwraca żadnego i nie musi przetwarzać wszystkich zakwalifikowanych wierszy. Możemy tę funkcję traktować jak rodzaj skrótovej oceny. Tę samą uwagę można odnieść do predykatu *IN*.

Inaczej niż w większości innych sytuacji, w tym przypadku nie jest złym rozwiązaniem stosowanie symbolu wieloznacznego gwiazdki (*) na liście *SELECT* podzapytania w kontekście predykatu *EXISTS*. Predykat *EXISTS* sprawdza jedynie istnienie zgodnych wierszy, niezależnie od atrybutów wyspecyfikowanych na liście *SELECT*, tak jakby cała klauzula *SELECT* była zbyteczna. Silnik bazy danych wie o tym i – na przykład ze względu na możliwość użycia indeksu – ignoruje listę *SELECT* podzapytania. Mogą pojawić się pewne niewielkie koszty dodatkowe związane z procesem rozpoznawania symbolu wieloznacznego * na informacje metadanych. Te dodatkowe koszty rozpoznawania są jednak tak niewielkie, że prawdopodobnie ich nie zauważymy. Moim zdaniem zapytania powinny być naturalne i intuicyjne, chyba że istnieje uzasadniona przyczyna, by dla niej poświęcić ten aspekt kodu. Jak sądzę, postać *EXISTS (SELECT * FROM ...)* jest znacznie bardziej intuicyjna niż wyrażenie *EXISTS (SELECT 1 FROM ...)*. Nie warto oszczędzać na niewielkich kosztach związanych z interpretowaniem gwiazdki, pogarszając w ten sposób czytelność kodu.

Na koniec innym interesującym aspektem predykatu *EXISTS* jest to, że w przeciwieństwie do większości predykatów języka T-SQL, *EXISTS* stosuje logikę dwuwartościową, a nie trójwartościową. Gdy się nad tym zastanowimy, faktycznie nie istnieją sytuacje, w których nie wiadomo, czy zapytanie zwraca jakieś wiersze, czy nie.

Zaawansowane aspekty podzapytań

W podrozdziale tym omówię takie aspekty podzapytań, które można uważać za zaawansowane, wykraczające poza podstawy. Zapoznanie się z tym materiałem jest opcjonalne i zalecane, jeśli zagadnienia omawiane do tej pory są już dobrze opanowane.

Zwracanie poprzednich lub kolejnych wartości

Załóżmy, że potrzebne jest zapytanie do tabeli *Orders* w bazie danych TSQVL4, które dla każdego zamówienia zwróci informacje o bieżącym zamówieniu, a także o identyfikatorze zamówienia poprzedniego. Pojęcie „poprzedni” sugeruje logiczną kolejność, ale ponieważ wiemy, że w tabeli wiersze nie mają kolejności, musimy zastosować logiczny ekwiwalent pojęcia „poprzedniości”, który można opisać za pomocą wyrażeń języka T-SQL. Przykładem takiego logicznego ekwiwalentu jest „maksymalna wartość, która jest mniejsza niż wartość bieżąca”. Określenie to można przedstawić w języku T-SQL za pomocą skorelowanego podzapytania, jak w poniższym kodzie:

```
SELECT orderid, orderdate, empid, custid,
       (SELECT MAX(O2.orderid)
        FROM Sales.Orders AS O2
        WHERE O2.orderid < O1.orderid) AS prevorderid
FROM Sales.Orders AS O1;
```

Zapytanie to generuje następujące wyniki (pokazane w skróconej postaci):

orderid	orderdate	empid	custid	prevorderid
10248	2014-07-04	5	85	NULL
10249	2014-07-05	6	79	10248
10250	2014-07-08	4	34	10249
10251	2014-07-08	3	84	10250
10252	2014-07-09	4	76	10251
...				
11073	2016-05-05	2	58	11072
11074	2016-05-06	7	73	11073
11075	2016-05-06	8	68	11074
11076	2016-05-06	4	9	11075
11077	2016-05-06	1	65	11076

(830 row(s) affected)

Zwróćmy uwagę, że ponieważ przed pierwszym zamówieniem nie istnieje żadne inne, dla pierwszego zamówienia zapytanie zwróciło *NULL*.

Podobnie możemy przedstawić definicję pojęcia „następny” – „minimalna wartość, która jest większa, niż wartość bieżąca”. Poniżej pokazano zapytanie T-SQL, które dla każdego zamówienia zwraca następny identyfikator zamówienia.

```
SELECT orderid, orderdate, empid, custid,
       (SELECT MIN(O2.orderid)
        FROM Sales.Orders AS O2
        WHERE O2.orderid > O1.orderid) AS nextorderid
FROM Sales.Orders AS O1;
```

```
FROM Sales.Orders AS O2
WHERE O2.orderid > O1.orderid) AS nextorderid
FROM Sales.Orders AS O1;
```

Zapytanie to generuje następujące wyniki (pokazane w skróconej postaci):

orderid	orderdate	empid	custid	nextorderid
10248	2014-07-04	5	85	10249
10249	2014-07-05	6	79	10250
10250	2014-07-08	4	34	10251
10251	2014-07-08	3	84	10252
10252	2014-07-09	4	76	10253
...				
11073	2016-05-05	2	58	11074
11074	2016-05-06	7	73	11075
11075	2016-05-06	8	68	11076
11076	2016-05-06	4	9	11077
11077	2016-05-06	1	65	NULL

(830 row(s) affected)

Zwróćmy uwagę na to, że ze względu na brak zamówienia po ostatnim zamówieniu dla ostatniego zamówienia podzapytanie zwraca *NULL*.

Język T-SQL udostępnia funkcje okna nazwane *LAG* i *LEAD*, które pozwalają zwracać elementy z „poprzednich” lub „następnych” wierszy w oparciu o wyspecyfikowaną kolejność. Te i inne funkcje okien zostały opisane w rozdziale 7.

Agregacje bieżące

Agregacje obliczane na bieżąco (*running aggregates*) kumulują wartości w czasie działania. W tym podrozdziale użyty zostanie widok *Sales.OrderTotalsByYear* do zilustrowania metody obliczania takiego sumowania. Widok pokazuje łączną ilość zamówień w ciągu roku. Wykonamy zapytanie do widoku, by przeanalizować jego zawartość.

```
SELECT orderyear, qty
FROM Sales.OrderTotalsByYear;
```

Po uruchomieniu uzyskujemy następujące dane wyjściowe:

orderyear	qty
2015	25489
2016	16247
2014	9581

Załóżmy, że dla każdego roku chcemy uzyskać rok zamówienia, ilość i skumulowaną łączną ilość do tego roku włącznie. Tak więc dla każdego roku zwracana ma być suma ilości towarów zamówionych do tego roku. Dla najwcześniejszego roku zarejestrowanego w widoku (2014) kumulowana łączna ilość jest równa ilości zgromadzonej w tym

roku. Dla drugiego roku (2015) kumulowana łączna ilość jest równa sumie pierwszego roku i drugiego roku itd.

Zadanie to możemy wykonać, uruchamiając zapytanie do jednej instancji widoku (nazwanej *O1*), by dla każdego roku zwrócić rok zamówienia i ilość, a następnie przy użyciu podzapytania skorelowanego do drugiej instancji widoku (nazwanej *O2*) obliczyć skumulowaną łączną ilość. Podzapytanie powinno filtrować wszystkie lata w widoku *O2*, które są równe lub mniejsze niż rok bieżący w widoku *O1* oraz sumować ilości z widoku *O2*. Poniżej przedstawiono rozwiązanie:

```
SELECT orderyear, qty,
       (SELECT SUM(O2.qty)
        FROM Sales.OrderTotalsByYear AS O2
        WHERE O2.orderyear <= O1.orderyear) AS runqty
FROM Sales.OrderTotalsByYear AS O1
ORDER BY orderyear;
```

Zapytanie to zwraca następujące wyniki:

orderyear	qty	runqty
2014	9581	9581
2015	25489	35070
2016	16247	51317

Zwróćmy uwagę, że język T-SQL zawiera agregujące funkcje okna, które można wykorzystać do znacznie wydajniejszego i prostszego obliczania sum ruchomych. Funkcje okien omówiane są w rozdziale 7.

Postępowanie w przypadku nieprawidłowo działających podzapytań

W tym podrozdziale omawiam przypadki, w których użycie podzapytania generuje błędy, a także najlepsze praktyki pozwalające unikać takich błędów logicznych.

Problemy dotyczące znaczników NULL

Pamiętamy, że język T-SQL stosuje trójwartościową logikę. W tym podrozdziale przedstawię problemy, które powstają, gdy używamy podzapytań i zapomnimy o obsłudze znaczników *NULL* i uwzględnianiu logiki trójwartościowej.

Przeanalizujemy następujące zapytanie, które wydaje się intuicyjne i którego zadaniem jest zwrócenie listy klientów, którzy nie złożyli zamówień.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O);
```

Przy użyciu przykładowych danych z tabeli *Orders* w bazie danych TSQLV4 wydaje się, że zapytanie działa zgodnie z oczekiwaniami; rzeczywiście zwraca ono dwa wiersze klientów, którzy nie złożyli zamówień.

custid	companyname
22	Customer DTDN
57	Customer WVXS

Następnie uruchomimy poniższy kod, by wstawić do tabeli *Orders* nowe zamówienie z *NULL* jako identyfikatorem klienta.

```
INSERT INTO Sales.Orders
(custid, empid, orderdate, requireddate, shippeddate, shipperid,
 freight, shipname, shipaddress, shipcity, shipregion,
 shippostalcode, shipcountry)
VALUES(NULL, 1, '20090212', '20090212',
       '20090212', 1, 123.00, N'abc', N'abc', N'abc',
       N'abc', N'abc', N'abc');
```

Teraz ponownie uruchomimy zapytanie zwracające klientów, którzy nie złożyli zamówień.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O);
```

Tym razem zapytanie zwraca pusty zbiór. Pamięając o tym, czego dowiedzieliśmy na temat znaczników *NULL* w rozdziale 2, spróbujmy wyjaśnić, dlaczego zapytanie zwraca pusty zbiór. Ponadto zastanowimy się, jak w wynikach uzyskać klientów 22 i 57.

Oczywiście przyczyną jest identyfikator klienta *NULL*, który został dodany do tabeli *Orders*. Jest on teraz zwracany przez podzapytanie wraz ze znanymi identyfikatorami klientów.

Rozpocznijmy od części, która działa zgodnie z oczekiwaniami. Predykat *IN* zwraca wartość *TRUE* dla klienta, który złożył zamówienia (na przykład klient 85), ponieważ taki klient jest zwracany przez podzapytanie. Operator *NOT* jest używany do zanegowania predykatu *IN*; a ponieważ wartość *NOT TRUE* to *FALSE*, klient nie jest zwracany przez kwerendę zewnętrzną. Oczekiwane zachowanie jest takie, że jeśli identyfikator klienta pojawi się w tabeli *Orders*, możemy z pewnością powiedzieć, że klient złożył zamówienia, a zatem nie chcemy, by znalazł się w wynikach. Jeśli jednak (tu weźmy głęboki wdech) identyfikator klienta z tabeli *Customers* nie występuje w zbiorze różnych od *NULL* identyfikatorów w tabeli *Orders*, a zarazem istnieje identyfikator klienta *NULL* w tabeli *Orders*, nie możemy z pewnością stwierdzić, czy klient tu jest – i analogicznie nie możemy z pewnością powiedzieć, że go tam nie ma. Zaskoczenie? Mam nadzieję, że uda mi się to wyjaśnić na przykładzie.

Predykat *IN* zwraca wartość *UNKNOWN* dla takiego klienta jak klient 22, który nie występuje w zbiorze znanych identyfikatorów klientów w tabeli *Orders*. Predykat *IN* zwraca wartość *UNKNOWN* dla tego klienta, ponieważ porównanie go ze wszystkimi znanymi identyfikatorami klientów daje wartość *FALSE*, a porównanie go z wartością *NULL* daje wartość *UNKNOWN*. Wyrażenie *FALSE OR UNKNOWN* daje wartość *UNKNOWN*. Dla przeanalizowania bardziej zrozumiałego przykładu, rozważmy wyrażenie *22 NOT IN (1, 2, NULL)*. Wyrażenie to można by zapisać jako *NOT 22 IN (1, 2, NULL)*. Ostatnie wyrażenie można rozwinąć do postaci *NOT (22 = 1 OR 22 = 2 OR 22 = NULL)*. Podstawiając prawdziwe wartości poszczególnych wyrażeń w nawiasach otrzymujemy *NOT (FALSE OR FALSE OR UNKNOWN)*, co przekłada się na *NOT UNKNOWN*, czyli *UNKNOWN*.

Logiczne znaczenie *UNKNOWN* w tym miejscu, przed zastosowaniem operatora *NOT*, jest takie, że nie można określić, czy identyfikator klienta pojawił się w zbiorze, ponieważ *NULL* może reprezentować zarówno ten identyfikator klienta, jak i jakiś inny. Problem polega na tym, że negowanie wartości *UNKNOWN* za pomocą operatora *NOT* nadal daje w wyniku *UNKNOWN*, a wartość *UNKNOWN* jest odrzucana w filtrze zapytania. Oznacza to, że jeśli nie wiemy, czy identyfikator klienta pojawia się w zbiorze, nie wiemy również, czy się nie pojawia.

W skrócie, kiedy zastosujemy predykat *NOT IN* w odniesieniu do podzapytania, które zwraca co najmniej jeden *NULL*, zapytanie zewnętrzne zawsze zwróci pusty zbiór. Jakie działania praktyczne można przedsięwziąć, by unikać takich problemów?

Po pierwsze, jeśli oczekujemy, że w kolumnie nie powinny się pojawiać znaczniki *NULL*, istotna jest jej definicja jako *NOT NULL*. Wymuszanie integralności danych to znacznie ważniejsza kwestia, niż sądzi wiele osób.

Po drugie, we wszystkich pisanych zapytaniach powinniśmy uwzględniać trzy możliwe wartości trójwartościowej logiki (*TRUE*, *FALSE* i *UNKNOWN*). Trzeba jawnie rozważyć, czy zapytanie może przetwarzać znaczniki *NULL* i jeśli tak, czy domyślne traktowanie znaczników *NULL* jest odpowiednie dla naszych potrzeb. Jeśli nie, trzeba to poprawić. I tak – w poprzednim przykładzie – zapytanie zewnętrzne zwraca pusty zbiór, czego przyczyną jest porównywanie ze znacznikiem *NULL*. Jeśli chcemy sprawdzić, czy identyfikator klienta występuje w zbiorze znanych wartości i pomijać znaczniki *NULL*, powinniśmy wykluczyć znaczniki *NULL* – wprost lub pośrednio. Jedną z metod wykluczenia wprost znaczników *NULL* polega na dodaniu do podzapytania predykatu *O.custid IS NOT NULL*, jak w poniższym kodzie:

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                     FROM Sales.Orders AS O
                     WHERE O.custid IS NOT NULL);
```

Znaczniki *NULL* możemy również wykluczyć pośrednio przy użyciu predykatu *NOT EXISTS*, a nie predykatu *NOT IN*, jak w poniższym przykładzie:

```

SELECT custid, companyname
FROM Sales.Customers AS C
WHERE NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid);

```

Pamiętajmy, że w przeciwieństwie do predykatu *IN*, predykat *EXISTS* stosuje logikę dwuwartościową. Predykat *EXISTS* zawsze zwraca *TRUE* lub *FALSE* i nigdy nie zwraca *UNKNOWN*. Kiedy podzapytanie napotyka wartość *NULL* w *O.custid*, wyrażenie przyjmuje wartość *UNKNOWN*, a wiersz jest odrzucany. O ile uwzględniany jest predykat *EXISTS*, przypadki znaczników *NULL* są eliminowane w sposób naturalny, tak jakby ich nie było. W efekcie końcowym predykat *EXISTS* obsługuje tylko znane identyfikatory klientów. I z tego względu lepiej jest stosować predykat *NOT EXISTS*, zamiast *NOT IN*.

Po przeprowadzeniu tych eksperymentów warto uruchomić poniższy kod, by wyczyścić bazę danych.

```
DELETE FROM Sales.Orders WHERE custid IS NULL;
```

Błędy podstawień w nazwach kolumn podzapytania

Błędy logiczne w kodzie mogą być czasami bardzo nieuchwytnie. W tym podrozdziale opisuję trudny do wychwycenia bug, który wynika z niewinnego błędu w nazwie kolumny podzapytania. Po wyjaśnieniu, na czym polega taki błąd, omówię najlepsze praktyki, które ułatwiają unikanie takich błędów.

Przykłady w tym podrozdziale odpytują tabelę nazwaną *MyShippers* w schemacie *Sales*. Uruchamiamy poniższy kod, by utworzyć i wypełnić tę tabelę.

```

DROP TABLE IF EXISTS Sales.MyShippers;

CREATE TABLE Sales.MyShippers
(
  shipper_id INT NOT NULL,
  companyname NVARCHAR(40) NOT NULL,
  phone NVARCHAR(24) NOT NULL,
  CONSTRAINT PK_MyShippers PRIMARY KEY(shipper_id)
);

INSERT INTO Sales.MyShippers(shipper_id, companyname, phone)
VALUES(1, N'Shipper GVSUA', N'(503) 555-0137'),
      (2, N'Shipper ETYNR', N'(425) 555-0136'),
      (3, N'Shipper ZHISN', N'(415) 555-0138');

```

Przeanalizujemy poniższe zapytanie mające zwracać spedystów, którzy wysłali zamówienia do klienta 43.

```

SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN

```

```
(SELECT shipper_id
FROM Sales.Orders
WHERE custid = 43);
```

Zapytanie to generuje następujące wyniki.

shipper_id	companyname
1	Shipper GVSUA
2	Shipper ETYNR
3	Shipper ZHISN

Tylko spedytorzy 2 i 3 wysyłali zamówienia do klienta 43, ale z jakiegoś powodu zapytanie to zwróciło wszystkich spedytorów z tabeli *MyShippers*. Spróbujmy uważnie przeanalizować zapytanie, a także wykorzystywane schematy tabel, aby się przekonać, czy potrafimy znaleźć przyczynę.

Okazuje się, że nazwa kolumny w tabeli *Orders*, przechowująca identyfikator spedytora, nie nazywa się *shipper_id*, ale *shipperid* (bez znaku podkreślenia). Kolumna w tabeli *MyShippers* jest nazwana *shipper_id*, zawiera podkreślenie. Rozpoznawanie nazw kolumn bez prefiksów działa w kontekście podzapytania na zewnątrz od bieżącego/wewnętrznego zakresu. W naszym przykładzie, SQL Server najpierw szuka kolumny *shipper_id* w tabeli *Orders*. Nie znajduje tam takiej kolumny, więc SQL Server wyszukuje ją w tabeli określonej w zapytaniu zewnętrznym, *MyShippers*. Ponieważ ją znajduje, to właśnie ona zostaje użyta.

Jak widzimy, to co miało być podzapytaniem niezależnym, niezamierzenie stało się podzapytaniem skorelowanym. O ile tabela *Orders* zawiera co najmniej jeden wiersz, wszystkie wiersze z tabeli *MyShippers* znajdą dopasowanie przy porównywaniu zewnętrznego identyfikatora spedytora z tym samym identyfikatorem.

Ktoś mógłby utrzymywać, że takie działanie to słaby punkt SQL. Tak nie jest. Zachowanie takie wynika z samego projektu standardu SQL, zaś implementacja firmy Microsoft jest po prostu zgodna ze standardem. Założenie przyjęte w standardzie pozwala na odnoszenie się do nazw kolumn z tabeli zewnętrznej bez prefiksu tak długo, jak długo jest to jednoznaczne (innymi słowy, o ile występują one tylko w jednej z zaangażowanych tabel).

Problem ten można często napotkać w środowiskach, w których nie są używane spójne nazwy atrybutów. Czasami nazwy te różnią się nieznacznie, jak w tym przypadku – *shipperid* w jednej tabeli i *shipper_id* w drugiej. To wystarczyło, aby błąd się ujawnił.

W celu uniknięcia tego typu problemów można postępować zgodnie z następującymi najlepszymi praktykami:

- Stosowanie spójnych nazw atrybutów we wszystkich tabelach.
- Poprzedzanie nazw kolumn w podzapytaniu prefiksami wskazującymi nazwę tabeli źródłowej lub jej alias (jeśli go użyliśmy).

W ten sposób proces rozwiązywania wyszukuje kolumnę tylko we wskazanej tabeli. Jeśli kolumna taka nie istnieje, od razu otrzymamy komunikat o błędzie. Dla przykładu spróbujmy uruchomić następujący kod:

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
    (SELECT 0.shipper_id
     FROM Sales.Orders AS 0
     WHERE 0.custid = 43);
```

Pojawi się pokazany poniżej błąd dotyczący rozpoznawania nazw:

```
Msg 207, Level 16, State 1, Line 4
Invalid column name 'shipper_id'.
(Nieprawidłowa nazwa kolumny 'shipper_id')
```

Po pojawieniu się takiego błędu możemy łatwo zidentyfikować problem i poprawić zapytanie.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
    (SELECT 0.shipperid
     FROM Sales.Orders AS 0
     WHERE 0.custid = 43);
```

Tym razem zapytanie zwraca oczekiwane wyniki.

```
shipper_id  companyname
-----
2           Shipper ETYNR
3           Shipper ZHISN
```

Po wykonaniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
IF OBJECT_ID('Sales.MyShippers', 'U') IS NOT NULL
    DROP TABLE Sales.MyShippers;
```

Podsumowanie

W rozdziale tym przedstawiłem problematykę podzapytań. Omówiłem podzapytania niezależne, które nie zależą od zapytania zewnętrznego i podzapytania skorelowane, które są zależne od swoich zapytań zewnętrznych. W odniesieniu do wyników podzapytań, omawiane były podzapytania skalarne i wielowartościowe. Jako opcjonalny materiał, zamieściłem także fragmenty o bardziej zaawansowanej tematyce, dotyczące zwracania poprzednich i następnych wartości, wykonywanie agregacji bieżących oraz typowych problemów dotyczących błędnego działania podzapytań. W podzapytaniach trzeba zawsze pamiętać o trójwartościowej logice i znaczeniu dodawania do nazw kolumn prefiksów zbudowanych z aliasów tabel źródłowych.

W kolejnym rozdziale zajmiemy się podzapytaniami zwracającymi tabelę, nazywanymi również *wyrażeniami tablicowymi*.

Ćwiczenia

Podrozdział ten zawiera ćwiczenia, które ułatwiają lepsze przyswojenie tematyki opisanej w rozdziale. We wszystkich ćwiczeniach używana jest przykładowa baza danych *TSQV4*.

Ćwiczenie 1

Napisać zapytanie zwracające wszystkie zamówienia złożone ostatniego dnia aktywności, który można znaleźć w tabeli *Orders*.

- Wykorzystywane tabele: *Sales.Orders*
- Oczekiwane dane wyjściowe:

orderid	orderdate	custid	empid
-----	-----	-----	-----
11077	2016-05-06	65	1
11076	2016-05-06	9	4
11075	2016-05-06	68	8
11074	2016-05-06	73	7

Ćwiczenie 2 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie zwracające wszystkie zamówienia złożone przez klienta lub klientów, którzy złożyli najwięcej zamówień. Zwróćmy uwagę, że kilku klientów może mieć taką samą (maksymalną) liczbę zamówień.

- Wykorzystywane tabele: *Sales.Orders*
- Oczekiwane dane wyjściowe (w skróconej postaci):

custid	orderid	orderdate	empid
-----	-----	-----	-----
71	10324	2014-10-08	9
71	10393	2014-12-25	1
71	10398	2014-12-30	2
71	10440	2015-02-10	4
71	10452	2015-02-20	8
71	10510	2015-04-18	6
...			
71	10984	2016-03-30	1
71	11002	2016-04-06	4
71	11030	2016-04-17	7
71	11031	2016-04-17	6
71	11064	2016-05-01	1
(31 row(s) affected)			

Ćwiczenie 3

Napisać zapytanie zwracające pracowników, którzy nie złożyli zamówień 1 maja 2016 lub później.

- Wykorzystywane tabele: *HR.Employees* i *Sales.Orders*
- Oczekiwane dane wyjściowe:

empid	FirstName	lastname
3	Judy	Lew
5	Sven	Buck
6	Paul	Suurs
9	Patricia	Doyle

Ćwiczenie 4

Napisać zapytanie zwracające kraje, w których są klienci, ale nie ma pracowników.

- Wykorzystywane tabele: *Sales.Customers* i *HR.Employees*
- Oczekiwane dane wyjściowe:

```
country
-----
Argentina
Austria
Belgium
Brazil
Canada
Denmark
Finland
France
Germany
Ireland
Italy
Mexico
Norway
Poland
Portugal
Spain
Sweden
Switzerland
Venezuela
(19 row(s) affected)
```

Ćwiczenie 5

Napisać zapytanie zwracające dla każdego klienta wszystkie zamówienia złożone w ostatnim dniu aktywności klienta.

- Wykorzystywane tabele: *Sales.Orders*

- Oczekiwane dane wyjściowe:

custid	orderid	orderdate	empid
1	11011	2016-04-09	3
2	10926	2016-03-04	4
3	10856	2016-01-28	3
4	11016	2016-04-10	9
5	10924	2016-03-04	3
...			
87	11025	2016-04-15	6
88	10935	2016-03-09	4
89	11066	2016-05-01	7
90	11005	2016-04-07	2
91	11044	2016-04-23	4

(90 row(s) affected)

Ćwiczenie 6

Napisać zapytanie zwracające klientów, którzy złożyli zamówienia w roku 2015, ale nie złożyli ich w roku 2016.

- Wykorzystywane tabele: *Sales.Customers* i *Sales.Orders*
- Oczekiwane dane wyjściowe:

custid	companyname
21	Customer KIDPX
23	Customer WVFAF
33	Customer FVXPQ
36	Customer LVJSO
43	Customer UISOJ
51	Customer PVDZC
85	Customer ENQZT

(7 row(s) affected)

Ćwiczenie 7 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie zwracające klientów, którzy zamówili produkt 12.

- Wykorzystywane tabele: *Sales.Customers*, *Sales.Orders* i *Sales.OrderDetails*
- Oczekiwane dane wyjściowe:

custid	companyname
48	Customer DVFMB
39	Customer GLLAG
71	Customer LCOUJ
65	Customer NYUHS
44	Customer OXFRU
51	Customer PVDZC

```

86      Customer SNX0J
20      Customer THHDP
90      Customer XBBVR
46      Customer XPNIK
31      Customer YJCBX
87      Customer ZHY0S

```

```
(12 row(s) affected)
```

Ćwiczenie 8 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie, które dla każdego klienta i miesiąca oblicza skumulowaną łączną ilość zamówionych towarów.

- Wykorzystywane tabele: *Sales.CustOrders*
- Oczekiwane dane wyjściowe:

custid	ordermonth	qty	runqty
-----	-----	-----	-----
1	2015-08-01	38	38
1	2015-10-01	41	79
1	2016-01-01	17	96
1	2016-03-01	18	114
1	2016-04-01	60	174
2	2014-09-01	6	6
2	2015-08-01	18	24
2	2015-11-01	10	34
2	2016-03-01	29	63
3	2014-11-01	24	24
3	2015-04-01	30	54
3	2015-05-01	80	134
3	2015-06-01	83	217
3	2015-09-01	102	319
3	2016-01-01	40	359
...			

```
(636 row(s) affected)
```

Ćwiczenie 9

Wyjaśnij różnicę pomiędzy predykatami *IN* i *EXISTS*.

Ćwiczenie 10 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie, które dla każdego zamówienia zwraca liczbę dni, które upłynęły od poprzedniego zamówienia tego samego klienta. Aby ustalić kolejność zamówień, wykorzystaj kolumnę *orderdate* jako główny element porządkujący oraz *orderid* jako argument dopełnienia:

- Wykorzystywana tabela: *Sales.Orders*

■ Oczekiwane dane wyjściowe:

custid	orderdate	orderid	diff
1	2015-08-25	10643	NULL
1	2015-10-03	10692	39
1	2015-10-13	10702	10
1	2016-01-15	10835	94
1	2016-03-16	10952	61
1	2016-04-09	11011	24
2	2014-09-18	10308	NULL
2	2015-08-08	10625	324
2	2015-11-28	10759	112
2	2016-03-04	10926	97
...			

(830 row(s) affected)

Rozwiązania

W tym podrozdziale udostępniono rozwiązania ćwiczeń wraz z odpowiednimi wyjaśnieniami.

Ćwiczenie 1

Możemy napisać podzapytanie niezależne, które zwraca ostatnią (maksymalną) datę zamówienia z tabeli *Orders*. Do podzapytania tego możemy odwołać się w klauzuli *WHERE* zapytania zewnętrznego, by zwrócić wszystkie zamówienia złożone w ostatnim dniu aktywności. Poniżej przedstawiono rozwiązanie zapytania:

```
USE TSQLV4;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate =
    (SELECT MAX(0.orderdate) FROM Sales.Orders AS 0);
```

Ćwiczenie 2

Problem ten najlepiej rozwiązać w kilku krokach. Najpierw napiszemy zapytanie zwracające klienta lub klientów, którzy złożyli najwięcej zamówień. Zadanie to można zrealizować grupując zamówienia według klientów, porządkując klientów malejąco przy użyciu funkcji *COUNT(*)* i opcji *TOP(1) WITH TIES*, by zwrócić identyfikatory klientów, którzy złożyli największą liczbę zamówień. Szczegóły dotyczące opcji *TOP* znaleźć można w rozdziale 2. Poniżej zamieszczono zapytanie realizujące pierwszy etap.

```
SELECT TOP (1) WITH TIES 0.custid
FROM Sales.Orders AS 0
```

```
GROUP BY 0.custid
ORDER BY COUNT(*) DESC;
```

Zapytanie to zwraca wartość 71, będącą identyfikatorem klienta, który złożył najwięcej zamówień – 31. W przypadku danych przykładowych przechowywanych w tabeli *Orders* tylko jeden klient złożył maksymalną liczbę zamówień. Zapytanie używa jednak opcji *WITH TIES*, by zwrócić wszystkie identyfikatory klientów, którzy złożyli maksymalną liczbę zamówień w sytuacji, kiedy jest więcej niż jeden taki klient.

Następnym krokiem jest napisanie zapytania do tabeli *Orders*, zwracającego wszystkie zamówienia, dla których identyfikator klienta znajduje się w zbiorze identyfikatorów klientów zwróconych przez zapytanie pierwszego etapu.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
    (SELECT TOP (1) WITH TIES 0.custid
     FROM Sales.Orders AS 0
     GROUP BY 0.custid
     ORDER BY COUNT(*) DESC);
```

Ćwiczenie 3

Możemy napisać zapytanie niezależne do tabeli *Orders*, które filtruje zamówienia złożone 1 maja 2016 lub później i tylko z tych zamówień zwraca identyfikatory pracowników. Następnie tworzymy zapytanie zewnętrzne do tabeli *Employees*, zwracające pracowników, których identyfikatory *nie* znajdują się w zbiorze identyfikatorów pracowników zwróconych przez podzapytanie. Poniżej zaprezentowano całe rozwiązanie:

```
SELECT empid, FirstName, lastname
FROM HR.Employees
WHERE empid NOT IN
    (SELECT 0.empid
     FROM Sales.Orders AS 0
     WHERE 0.orderdate >= '20160501');
```

Ćwiczenie 4

Możemy napisać podzapytanie niezależne do tabeli *Employees*, które z każdego wiersza zwraca atrybut kraju. Napiszemy następnie zapytanie zewnętrzne do tabeli *Customers*, które filtruje tylko te wiersze klientów, których kraj nie występuje w zbiorze krajów zwróconych przez podzapytanie. W liście *SELECT* zapytania zewnętrznego wyspecyfikujemy wyrażenie *DISTINCT*, by zwrócić tylko różne wystąpienia krajów, ponieważ w tym samym kraju może być więcej niż jeden klient. Poniżej pokazano całe rozwiązanie:

```
SELECT DISTINCT country
FROM Sales.Customers
```

```
WHERE country NOT IN
  (SELECT E.country FROM HR.Employees AS E);
```

Ćwiczenie 5

Ćwiczenie to jest podobne do ćwiczenia 1 z wyjątkiem tego, że w tym przypadku chcemy zwrócić zamówienia złożone w ostatnim dniu aktywności tego klienta, a nie w ogóle. Rozwiązania obu ćwiczeń są podobne, ale tutaj trzeba użyć podzapytania skorelowanego w celu porównania wewnętrznego identyfikatora klienta z zewnętrznym identyfikatorem klienta, jak w poniższym kodzie:

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderdate =
  (SELECT MAX(O2.orderdate)
   FROM Sales.Orders AS O2
   WHERE O2.custid = O1.custid)
ORDER BY custid;
```

Nie porównujemy daty zamówienia wiersza zewnętrznego z ostatnią (maksymalną) datą wszystkich zamówień, lecz z maksymalną datą zamówienia dla bieżącego klienta.

Ćwiczenie 6

Problem ten możemy rozwiązać, tworząc zapytanie do tabeli *Customers* oraz stosując predykaty *EXISTS* i *NOT EXISTS* wraz z podzapytaniem skorelowanym, by zapewnić, że klient złożył zamówienie w roku 2015, ale nie w roku 2016. Predykat *EXISTS* zwraca wartość *TRUE* tylko wtedy, gdy w tabeli *Orders* istnieje co najmniej jeden wiersz z tym samym identyfikatorem klienta co w wierszu zewnętrznym, wewnątrz zakresu dat reprezentujących rok 2015. Predykat *NOT EXISTS* zwraca wartość *TRUE* tylko wtedy, gdy w tabeli *Orders* nie istnieje wiersz o tym samym identyfikatorze klienta co w wierszu zewnętrznym, wewnątrz zakresu dat reprezentujących rok 2016. Poniżej przedstawiono całe rozwiązanie:

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
        AND O.orderdate >= '20150101'
        AND O.orderdate < '20160101')
AND NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
        AND O.orderdate >= '20160101'
        AND O.orderdate < '20170101');
```

Ćwiczenie 7

Zadanie to można wykonać, zagnieżdżając predykaty *EXISTS* za pomocą podzapytań skorelowanych. Najbardziej zewnętrzne zapytanie piszemy w odniesieniu do tabeli *Customers*. W klauzuli *WHERE* zapytania zewnętrznego możemy użyć predykatu *EXISTS* z podzapytaniem skorelowanym do tabeli *Orders*, by filtrować tylko bieżące zamówienia klienta. W filtrze podzapytania dotyczącego tabeli *Orders* możemy użyć zagnieżdżonego predykatu *EXISTS* z podzapytaniem dotyczącym tabeli *OrderDetails*, które filtruje tylko pozycje zamówienia dla identyfikatora produktu równego 12. W ten sposób zwracani są tylko ci klienci, którzy złożyli zamówienia zawierające produkt 12. Poniżej zaprezentowano całe rozwiązanie:

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
   AND EXISTS
     (SELECT *
      FROM Sales.OrderDetails AS OD
      WHERE OD.orderid = O.orderid
      AND OD.ProductID = 12));
```

Ćwiczenie 8

Kiedy muszę rozwiązać problem dotyczący zapytania, często staram się przeformułować oryginalne żądanie w bardziej technicznej postaci, tak by łatwiej było je przetłumaczyć na zapytanie T-SQL. Aby zbudować potrzebne zapytanie, najpierw spróbujmy wyrazić bardziej precyzyjnie pierwotne żądanie „dla każdego klienta i miesiąca zwrócić skumulowaną łączną ilość”. Innymi słowy, dla każdego klienta trzeba zwrócić identyfikator klienta, miesiąc, sumę ilości w tym miesiącu i sumę ilości we wszystkich miesiącach równych lub mniejszych niż bieżący miesiąc.

Zadanie to można obsłużyć za pomocą podzapytania skorelowanego. Użyjemy zewnętrznego zapytania do tabeli *CustOrders* (z aliasem *O1*) zwracającego bieżącego klient, miesiąc i informacje o ilości. Zapytanie skorelowane odnosi się do drugiej instancji tabeli *CustOrders* (z aliasem *O2*). Wykonamy w nim agregację ilości występujących w *O2* dla bieżącego klienta z *O1*, gdzie miesiąc z *O2* jest mniejszy lub równy bieżącemu miesiącowi z *O1*. Oto pełne rozwiązanie:

```
SELECT custid, ordermonth, qty,
  (SELECT SUM(O2.qty)
   FROM Sales.CustOrders AS O2
   WHERE O2.custid = O1.custid
   AND O2.ordermonth <= O1.ordermonth) AS runqty
FROM Sales.CustOrders AS O1
ORDER BY custid, ordermonth;
```

Ćwiczenie 9

Podczas gdy predykat *IN* posługuje się logiką trójwartościową, w predykanie *EXISTS* używana jest logika dwuwartościowa. Jeśli dane nie zawierają znaczników *NULL*, predykaty *IN* i *EXISTS* dają te same wyniki zarówno w postaci pozytywnej, jak i zanegowanej (z *NOT*). Gdy występują znaczniki *NULL*, predykaty *IN* i *EXISTS* mają takie same znaczenie w formie pozytywnej, ale różnią się w postaci zanegowanej. W postaci pozytywnej, gdy szukamy wartości występującej w zbiorze znanych wartości zwracanych przez podzapytanie, obydwa zwracają *TRUE*, zaś w przypadku wartości niewystępującej w wynikach obydwa zwracają *FALSE*. W postaci zanegowanej przy wyszukiwaniu wartości należącej do zbioru znanych wartości obydwa predykaty zwracają *FALSE*; jednak przy wyszukiwaniu wartości, która nie występuje w zbiorze znanych wartości, predykat *NOT IN* zwraca *UNKNOWN* (zewnątrzny wiersz zostaje odrzucony), podczas gdy *NOT EXISTS* zwróci *TRUE* (zewnątrzny wiersz zostaje zachowany).

Najłatwiej zrozumieć to na przykładzie. Dobry przykład prezentuje podrozdział „Problemy dotyczące znaczników *NULL*” wcześniej w tym rozdziale.

Ćwiczenie 10

Zadanie to można rozwiązać w dwóch krokach:

1. Napisać zapytanie wyliczające datę poprzedniego zamówienia złożonego przez klienta.
2. Wyliczyć różnicę pomiędzy datą zwróconą w pierwszym kroku a datą bieżącego zamówienia.

Oto zapytanie realizujące pierwszy krok:

```
SELECT custid, orderdate, orderid,  
  (SELECT TOP (1) O2.orderdate  
   FROM Sales.Orders AS O2  
   WHERE O2.custid = O1.custid  
         AND ( O2.orderdate = O1.orderdate AND O2.orderid < O1.orderid  
              OR O2.orderdate < O1.orderdate )  
   ORDER BY O2.orderdate DESC, O2.orderid DESC) AS prevdate  
FROM Sales.Orders AS O1  
ORDER BY custid, orderdate, orderid;
```

Aby uzyskać datę poprzedniego zamówienia, rozwiązanie wykorzystuje skolerowane podzapytanie z filtrem *TOP*. Podzapytanie wybiera tylko zamówienia, dla których identyfikator klienta jest równy zewnętrznemu identyfikatorowi. Dodatkowo filtruje tylko zamówienia, w których albo „wewnętrzna data jest równa dacie zewnętrznego zamówienia i wewnętrzny identyfikator zamówienia jest mniejszy od zewnętrznego”, albo „wewnętrzna data zamówienia jest wcześniejsza od daty zewnętrznej”. Uzyskane zamówienia są tymi, które można uznać za wcześniejsze od bieżącego zamówienia

klienta. Przy użyciu filtra *TOP* (1) opartego na porządkowaniu według *orderdate DESC*, *orderid DESC* uzyskujemy ostatnie z tych wcześniejszych zamówień, a zatem bezpośrednio poprzedzające. Ten krok zwraca następujący wynik:

custid	orderdate	orderid	prevdate
1	2015-08-25	10643	NULL
1	2015-10-03	10692	2015-08-25
1	2015-10-13	10702	2015-10-03
1	2016-01-15	10835	2015-10-13
1	2016-03-16	10952	2016-01-15
1	2016-04-09	11011	2016-03-16
2	2014-09-18	10308	NULL
2	2015-08-08	10625	2014-09-18
2	2015-11-28	10759	2015-08-08
2	2016-03-04	10926	2015-11-28
...			

(830 row(s) affected)

Jeśli pojawi się pytanie, dlaczego nie możemy polegać tylko na porządkowaniu według *orderid*, przyczyną jest to, że wiele firm obsługuje koncepcję *późnego nadejścia*. Jest to przypadek, gdy zamówienie zostało złożone w przeszłości, ale w systemie zostało zarejestrowane w późniejszym terminem. W chwili dodania zamówienia do systemu otrzymuje wyższy numer zamówienia, ale nie ma najnowszej daty. Tak więc kolejność jest ustalana najpierw na podstawie daty zamówienia, a następnie identyfikator zamówienia jest używany jako element ujednoznaczniający. Z tego powodu klauzula *WHERE* podzapytania jest tak złożona i porządek filtra *TOP* opiera się na *orderdate DESC*, *orderid DESC*, a nie po prostu na *orderid DESC*.

W drugim kroku użyjemy funkcji *DATEDIFF* do wyliczenia różnicy w dniach pomiędzy datą wcześniejszego zamówienia zwróconą przez podzapytanie, a datą bieżącego zamówienia. Oto gotowe rozwiązanie:

```
SELECT custid, orderdate, orderid,
       DATEDIFF(day,
                (SELECT TOP (1) O2.orderdate
                 FROM Sales.Orders AS O2
                 WHERE O2.custid = O1.custid
                   AND ( O2.orderdate = O1.orderdate AND O2.orderid < O1.orderid
                       OR O2.orderdate < O1.orderdate )
                 ORDER BY O2.orderdate DESC, O2.orderid DESC), orderdate) AS diff
FROM Sales.Orders AS O1
ORDER BY custid, orderdate, orderid;
```

ROZDZIAŁ 5

Wyrażenia tablicowe

Wyrazenie tablicowe to nazwane wyrażenie zapytania, które reprezentuje prawidłową tabelę relacyjną. Wyrażenia tablicowego można używać w poleceniach związanych z operacjami na danych tak samo, jak dowolnej innej tabeli. Język T-SQL obsługuje cztery typy wyrażeń tablicowych: tabele pochodne, wspólne wyrażenia tablicowe (*Common Table Expression* – CTE), widoki i wbudowane (*inline*) funkcje zwracające tabele (*inline Table Valued Functions* – TVF). Wszystkie te rodzaje wyrażeń zostaną omówione szczegółowo w tym rozdziale. Tematyka rozdziału skupia się na stosowaniu zapytań *SELECT* w odniesieniu do wyrażeń tablicowych; modyfikacje wyrażeń tablicowych zostaną omówione w rozdziale 8 „Modyfikowanie danych”.

Wyrażenia tablicowe w żadnym miejscu nie materializują się fizycznie – są to wyrażenia wirtualne. Podczas wykonywania zapytania wyrażenia tablicowego zapytanie wewnętrzne przestaje być zagnieżdżone. Inaczej mówiąc, zapytanie zewnętrzne i wewnętrzne zostają scalone do postaci jednego zapytania dotyczącego bezpośrednio obiektów bazowych. Korzyści ze stosowania wyrażeń tablicowych zazwyczaj dotyczą aspektów logicznych kodu, a nie jego wydajności. Na przykład wyrażenia tablicowe ułatwiają upraszczanie rozwiązań przy użyciu podejścia modularnego. Wyrażenia tablicowe ułatwiają również omijanie pewnych ograniczeń języka, takich jak brak możliwości odnoszenia się do aliasów kolumn przypisanych w klauzuli *SELECT* w klauzulach zapytania, które są logicznie przetwarzane przed klauzulą *SELECT*.

Rozdział ten wprowadza także operator tabeli *APPLY* używany w połączeniu z wyrażeniem tablicowym. Pokażę, jak posługiwać się tym operatorem do zastosowania wyrażenia tablicowego do każdego wiersza innej tabeli.

Tabele pochodne

Tabele pochodne (nazywane również *podzapytaniami zwracającymi tabelę*) definiowane są w klauzuli *FROM* zapytania zewnętrznego. Zakres istnienia tabeli pochodnej to zapytanie zewnętrzne. Gdy tylko zapytanie zewnętrzne zostanie zakończone, znika tabela pochodna.

Zapytanie definiujące tabelę pochodną specyfikujemy w nawiasach, po których umieszczana jest klauzula *AS* i nazwa (alias) tabeli pochodnej. Na przykład poniższy kod definiuje tabelę pochodną o nazwie *USACusts* w oparciu o zapytanie, które zwraca wszystkich klientów z USA, a zapytanie zewnętrzne wybiera wszystkie wiersze tabeli pochodnej.

```
USE TSQLV4;

SELECT *
FROM (SELECT custid, companyname
      FROM Sales.Customers
      WHERE country = N'USA') AS USACusts;
```

W tym konkretnym przypadku, który jest prostą ilustracją podstawowej składni, tabela pochodna nie jest potrzebna, ponieważ zapytanie zewnętrzne nie wykonuje żadnych operacji.

Kod tego podstawowego przykładu zwraca następujące wyniki:

custid	companyname
32	Customer YSIQX
36	Customer LVJS0
43	Customer UIS0J
45	Customer QXPPT
48	Customer DVFMB
55	Customer KZQZT
65	Customer NYUHS
71	Customer LCOUJ
75	Customer XOJYP
77	Customer LCYBZ
78	Customer NLTYP
82	Customer EYHKM
89	Customer YBQTI

Dla wszystkich typów wyrażeń tablicowych zapytanie definiujące musi spełniać trzy wymagania:

- 1. Kolejność nie jest zagwarantowana** Wyrażenie tablicowe ma reprezentować tabelę relacyjną, a wiersze w tabeli relacyjnej nie są uporządkowane. Przypomnijmy sobie z teorii zbiorów ten aspekt relacji. Z tego powodu standard SQL zabrania stosowania klauzuli *ORDER BY* w zapytaniach używanych do definiowania wyrażenia tablicowego, chyba że klauzula *ORDER BY* spełnia inną rolę niż prezentacja. Przykładem takiego wyjątku może być sytuacja, kiedy zapytanie używa filtru *OFFSET-FETCH*. Język T-SQL wymusza podobne ograniczenia i podobne wyjątki – jeśli wyspecyfikowany jest również filtr *TOP* lub *OFFSET-FETCH*. W kontekście zapytania z filtrem *TOP* lub *OFFSET-FETCH* klauzula *ORDER BY* stanowi część specyfikacji filtru. Jeśli używamy zapytania z filtrem *TOP* lub *OFFSET-FETCH* i klauzulą *ORDER BY* do definiowania wyrażenia tablicowego, klauzula *ORDER BY* służy jedynie do filtrowania, a nie jest stosowana w typowy sposób do prezentowania wyników. Jeśli zapytanie zewnętrzne do wyrażenia tablicowego nie zawiera klauzuli *ORDER BY* służącej do prezentacji danych, wyniki są zwracane w przypadkowej kolejności. Dodatkowe informacje na ten temat znaleźć można w dalszej części w podrozdziale „Widoki i klauzula *ORDER BY*”.

2. **Wszystkie kolumny muszą mieć nazwy** Wszystkie kolumny tabeli muszą mieć nazwy; z tego względu musimy przypisywać aliasy kolumn do wszystkich wyrażeń na liście *SELECT* zapytania, które są używane do definiowania wyrażenia tablicowego.
3. **Nazwy kolumn muszą być unikatowe** Wszystkie nazwy kolumn tabeli muszą być unikatowe; z tego względu wyrażenie tablicowe, które buduje wiele kolumn o tej samej nazwie, jest nieprawidłowe. Może się tak zdarzyć, jeśli zapytanie definiujące wyrażenie tablicowe łączy dwie tabele (o ile te tabele mają tak samo nazwane kolumny). Jeśli zachodzi potrzeba użycia obu kolumn w wyrażeniu tablicowym, kolumny muszą mieć różne nazwy. Możemy to zapewnić, przypisując tym kolumnom różne aliasy.

Te trzy wymagania wynikają z założenia, że wyrażenie reprezentuje relację. Wszystkie atrybuty relacji muszą mieć nazwy, wszystkie nazwy atrybutów relacji muszą być unikatowe, a ciało relacji jest zbiorem krotek, w którym nie ma uporządkowania.

Przypisywanie aliasów kolumn

Jedną z zalet używania wyrażeń tablicowych polega na tym, że w dowolnej klauzuli zapytania zewnętrznego możemy odnosić się do aliasów kolumn, które zostały przypisane w klauzuli *SELECT* zapytania wewnętrznego. Dzięki temu możemy ominąć ograniczenie wynikające z tego, że nie można odnosić się do aliasów kolumn przypisanych w klauzuli *SELECT* w tych klauzulach zapytania, które są przetwarzane logicznie przed klauzulą *SELECT* (na przykład *WHERE* lub *GROUP BY*).

Żałóśmy przykładowo, że chcemy napisać zapytanie do tabeli *Sales.Orders* i zwrócić liczbę różnych klientów obsługiwanych w każdym roku. Poniższa próba jest nieprawidłowa, ponieważ klauzula *GROUP BY* odnosi się do aliasu kolumny, który został przypisany w klauzuli *SELECT*, a klauzula *GROUP BY* jest przetwarzana logicznie przed klauzulą *SELECT*.

```
SELECT
    YEAR(orderdate) AS orderyear,
    COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY orderyear;
```

Jeśli spróbujemy uruchomić to zapytanie, pojawi się następujący komunikat o błędzie:

```
Msg 207, Level 16, State 1, Line 5
Invalid column name 'orderyear'.
```

(Nieprawidłowa nazwa kolumny 'orderyear'.)

Problem ten możemy rozwiązać używając wyrażenia *YEAR(orderdate)* w obu klauzulach *GROUP BY* i *SELECT*, ale jest to przykład krótkiego wyrażenia. Co się stanie, jeśli wyrażenie jest znacznie dłuższe i na przykład wykorzystuje wiele różnych kolumn?

Utrzymywanie dwóch kopii tego samego wyrażenia obniża czytelność kodu i w takiej sytuacji łatwiej o pomyłki. W celu rozwiązania problemu przy użyciu tylko jednej kopii wyrażenia możemy użyć wyrażenia tablicowego, ilustrowanego listingiem 5-1.

LISTING 5-1 Zapytanie z tabelą pochodną przy użyciu wewnętrznego formatu tworzenia aliasów

```
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders) AS D
GROUP BY orderyear;
```

Zapytanie to zwraca następujące wyniki:

orderyear	numcusts
2014	67
2015	86
2016	81

Kod ten definiuje tabelę pochodną nazwaną *D* w oparciu o zapytanie do tabeli *Orders*, które dla wszystkich wierszy zwraca lata zamówień i identyfikator klienta. Lista *SELECT* zapytania wewnętrznego tworzy alias *orderyear* dla wyrażenia *YEAR(orderdate)*. Zapytanie zewnętrzne może odnosić się do aliasu kolumny *orderyear* w obu klauzulach *GROUP BY* i *SELECT*, ponieważ, o ile bierzemy pod uwagę zapytanie zewnętrzne, zapytanie dotyczy tabeli nazwanej *D* z kolumnami nazwanymi *orderyear* i *custid*.

Jak już nadmieniałem, SQL Server rozwija definicję wyrażenia tablicowego i bezpośrednio uzyskuje dostęp do obiektów bazowych. Po rozwinięciu zapytanie z listingu 5-1 wygląda następująco:

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

Warto podkreślić, że wyrażen tablicowych używamy do uproszczenia logiki, a nie poprawy wydajności. Mówiąc ogólnie, wyrażenia tablicowe nie mają wpływu na wydajność.

Kod z listingu 5-1 używa wewnętrznego (*inline*) formatu tworzenia aliasów do przypisywania wyrażeniom aliasów kolumn. Składnia wewnętrznego tworzenia aliasów jest następująca <wyrażenie> [AS] <alias>. Zwróćmy uwagę, że słowo *AS* jest opcjonalne w składni wewnętrznego tworzenia aliasów, jednakże ze względu na czytelność kodu zalecane jest umieszczanie tego słowa.

W niektórych przypadkach lepszy może być drugi obsługiwany format przypisywania aliasów kolumn, który możemy traktować jak format zewnętrzny. Przy korzystaniu z tego formatu nie przypisujemy aliasów kolumn po wyrażeniach na liście *SELECT* – specyfikujemy w nawiasach wszystkie nazwy kolumn docelowych po nazwie wyrażenia tablicowego, jak w poniższym przykładzie:

```
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate), custid
```

```
FROM Sales.Orders) AS D(orderyear, custid)
GROUP BY orderyear;
```

Ogólnie, z kilku powodów, zaleca się stosowanie aliasów wewnętrznych. Jeśli zachodzi potrzeba debugowania kodu przy używanym formacie wewnętrznym, po wyróżnieniu zapytania definiującego wyrażenie tablicowe i uruchomieniu go, kolumny w wynikach pokazywane są przy użyciu przypisanych aliasów. W przypadku formatu zewnętrznego nie możemy dołączyć nazw docelowych kolumn po wyróżnieniu zapytania wyrażenia tablicowego, tak więc wyniki pojawiają się bez nazw kolumn w przypadku nienazwanego wyrażenia. Ponadto, jeśli zapytanie wyrażenia tablicowego jest długie, używanie formatu zewnętrznego może utrudniać jego zrozumienie i określenie, którego wyrażenia dotyczą poszczególne aliasy kolumn.

Chociaż zalecanym rozwiązaniem jest stosowanie wewnętrznego formatu aliasów, w niektórych przypadkach bardziej wygodny może okazać się format zewnętrzny. Na przykład, kiedy zapytanie definiujące wyrażenie tablicowe nie podlega żadnym dalszym poprawkom i chcemy je traktować jak „czarną skrzynkę” – czyli podczas przeglądania zapytania zewnętrznego chcemy skupić naszą uwagę na nazwie wyrażenia tablicowego, po której jest lista kolumn docelowych. Używając terminologii tradycyjnych języków programowania, format ten pozwala wyspecyfikować interfejs pomiędzy zapytaniem zewnętrznym a wyrażeniem tablicowym.

Stosowanie argumentów

W zapytaniu definiującym tabelę pochodną możemy odwoływać się do argumentów. Argumentami mogą być zmienne lokalne i parametry wejściowe do procedur, takich jak procedury składowane czy funkcje. Na przykład poniższy kod deklaruje i inicjuje zmienną lokalną nazwaną *@empid*, a zapytanie definiujące tabelę pochodną *D* odnosi się do tej zmiennej lokalnej w klauzuli *WHERE*.

```
DECLARE @empid AS INT = 3;

SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders
      WHERE empid = @empid) AS D
GROUP BY orderyear;
```

Zapytanie to zwraca liczbę różnych klientów dla danego roku, których zamówienia były obsługiwane przez pracownika określonego na wejściu (pracownika, którego identyfikator przechowywany jest w zmiennej *@empid*). Poniżej pokazane są wyniki działania tego zapytania:

orderyear	numcusts
2014	16
2015	46
2016	30

Zagnieżdżanie

Jeśli zachodzi potrzeba zdefiniowania tabeli pochodnej przy użyciu zapytania, które samo odwołuje się do tabeli pochodnej, w efekcie końcowym otrzymujemy zagnieżdżanie tabel pochodnych. Ogólnie mówiąc, zagnieżdżanie jest niezalecanym stylem programowania, ponieważ powoduje komplikowanie kodu i obniża jego czytelność.

Na przykład kod z listingu 5-2 zwraca lata zamówień i liczbę klientów obsługiwanych w każdym roku jedynie dla lat, w których obsługanych było więcej niż 70 klientów.

LISTING 5-2 Zapytanie z zagnieżdżonymi tabelami pochodnymi

```
SELECT orderyear, numcusts
FROM (SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
      FROM (SELECT YEAR(orderdate) AS orderyear, custid
            FROM Sales.Orders) AS D1
      GROUP BY orderyear) AS D2
WHERE numcusts > 70;
```

Kod ten zwraca następujące wyniki:

orderyear	numcusts
2015	86
2016	81

Zadaniem najbardziej wewnętrznej tabeli pochodnej, *D1*, jest przypisanie aliasu kolumny *orderyear* do wyrażenia *YEAR(orderdate)*. Zapytanie do tabeli *D1* odwołuje się do kolumny *orderyear* w obu klauzulach *GROUP BY* i *SELECT* i przypisuje alias kolumny *numcusts* do wyrażenia *COUNT(DISTINCT custid)*. Zapytanie do tabeli *D1* służy do zdefiniowania tabeli pochodnej *D2*. Zewnętrzne zapytanie do tabeli *D2* odwołuje się do kolumny *numcusts* w klauzuli *WHERE*, by filtrować lata zamówień, w których obsłużono ponad 70 klientów.

Celem użycia wyrażeń tablicowych w tym przykładzie było uproszczenie rozwiązania poprzez ponowne użycie aliasów, zamiast powtarzania wyrażeń. Jednakże uwzględniając złożoność związaną z zagnieżdżaniem tabel pochodnych nie jestem pewny, czy to rozwiązanie jest prostsze niż rozwiązanie alternatywne, w którym nie są używane żadne tabele pochodne, a zamiast tego powtarzane są wyrażenia.

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR(orderdate)
HAVING COUNT(DISTINCT custid) > 70;
```

Mówiąc krótko, zagnieżdżanie to problematyczny aspekt tabel pochodnych.

Wielokrotne odwołania

Inny kłopotliwy aspekt tabel pochodnych wynika z faktu, że tabele te są definiowane w klauzuli *FROM* zapytania zewnętrznego, a nie przed nim. Jeśli rozpatrujemy klauzulę *FROM* zapytania zewnętrznego, tabela pochodna jeszcze nie istnieje; nie można więc odwoływać się do wielu instancji tabeli pochodnej, na przykład w celu wykonania samo-złączenia. Zamiast tego musimy definiować wiele tabel pochodnych, bazując na tym samym zapytaniu. Listing 5-3 jest przykładem takiego zapytania.

LISTING 5-3 Wiele tabel pochodnych w oparciu o to samo zapytanie

```
SELECT Cur.orderyear,
       Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
       Cur.numcusts - Prv.numcusts AS growth
FROM (SELECT YEAR(orderdate) AS orderyear,
             COUNT(DISTINCT custid) AS numcusts
      FROM Sales.Orders
      GROUP BY YEAR(orderdate)) AS Cur
LEFT OUTER JOIN
  (SELECT YEAR(orderdate) AS orderyear,
           COUNT(DISTINCT custid) AS numcusts
   FROM Sales.Orders
   GROUP BY YEAR(orderdate)) AS Prv
ON Cur.orderyear = Prv.orderyear + 1;
```

Zapytanie to łączy dwa wyrażenia tablicowe oparte na tym samym zapytaniu. Pierwsza tabela pochodna, *Cur*, reprezentuje lata zamówień, a druga tabela pochodna, *Prv*, reprezentuje lata poprzednie. Warunek złączenia *Cur.orderyear = Prv.orderyear + 1* zapewnia, że każdy wiersz z pierwszej tabeli pochodnej pasuje do roku poprzedniego drugiej tabeli. Ponieważ jest to lewe zewnętrzne złączenie, pierwszy rok, dla którego nie ma poprzedniego roku, jest również zwracany z tabeli *Cur*. Klauzula *SELECT* zapytania zewnętrznego oblicza różnicę pomiędzy liczbą klientów obsługiwanych w bieżącym i poprzednim roku.

Kod z listingu 5-3 generuje następujące dane wyjściowe:

orderyear	curnumcusts	prvnumcusts	growth
2014	67	NULL	NULL
2015	86	67	19
2016	81	86	-5

Niemożliwość odwoływania się do wielu instancji tej samej tabeli pochodnej wymusza utrzymywanie wielu kopii tej samej definicji zapytania. Prowadzi to do wydłużania kodu, który trudniej jest utrzymywać, a także zwiększa podatność na pomyłki.

Wspólne wyrażenia tablicowe

Wspólne wyrażenia tablicowe CTE (Common Table Expression) są inną standardową postacią wyrażenia tablicowego, bardzo podobnego do tabel pochodnych, mają jednak kilka ważnych zalet.

Wyrażenia CTE definiowane są przy użyciu polecenia *WITH*, a ich ogólna postać jest następująca:

```
WITH <nazwa_CTE>[(<lista_kolumn_docelowych>)] AS
(
    <zapytanie_wewnetrzne_definiujace_CTE>
)
<zapytanie_zewnetrzne_odwo_lujace_sie_do_CTE>;
```

Zapytanie wewnętrzne definiujące wyrażenie CTE musi spełniać wszystkie wymagania wymienione wcześniej dla wyrażeń tablicowych. Jako prosta ilustracja, poniższy kod definiuje wyrażenie CTE nazwane *USACusts* w oparciu o zapytanie zwracające wszystkich klientów z USA, a zapytanie zewnętrzne wybiera wszystkie wiersze wyrażenia CTE.

```
WITH USACusts AS
(
    SELECT custid, companyname
    FROM Sales.Customers
    WHERE country = N'USA'
)
SELECT * FROM USACusts;
```

Podobnie jak w przypadku tabel pochodnych, zaraz po zakończeniu zapytania zewnętrznego wyrażenie CTE znika z zakresu.



UWAGA W języku T-SQL klauzula *WITH* ma kilka różnych zastosowań. Na przykład może być użyta jako wskazówka (*hint*) tabeli w celu wymuszenia określonej opcji optymalizacyjnej lub poziomu izolacji. Aby uniknąć nieporozumień, w przypadku użycia *WITH* do definiowania wyrażenia CTE poprzednie polecenie w tym samym wsadzie (*batch*) poleceń (o ile jakieś istnieje) musi być zakończone średnikiem. A co dość dziwne, średnik kończący całe wyrażenie CTE nie jest wymagany. Jednak zawsze zalecam jego wpisanie – tak jak do zamykania wszystkich poleceń języka T-SQL, również tych definiujących CTE.

Przypisywanie aliasów kolumn w wyrażeniach CTE

Wyrażenia CTE również obsługują dwa formaty tworzenia aliasów kolumn – wewnętrzny i zewnętrzny. W przypadku wewnętrznego formatu używamy składni *<wyrażenie> AS <alias_kolumny>*; przy korzystaniu z formatu zewnętrznego specyfikujemy w nawiasach listę kolumn docelowych zaraz po nazwie wyrażenia CTE.

Poniżej przedstawiono przykład formatu wewnętrznego.

```
WITH C AS
(
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

A to z kolei jest przykład formatu zewnętrznego.

```
WITH C(orderyear, custid) AS
(
    SELECT YEAR(orderdate), custid
    FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

Zalecenia dotyczące używania jednego czy drugiego formatu są podobne jak w przypadku tabel pochodnych. Warto jednak zauważyć, że tym razem przy formacie zewnętrznym lista aliasów występuje na początku, a nie pod koniec wyrażenia definiującego, zatem czytelność tego formatu jest większa.

Stosowanie argumentów w wyrażeniach CTE

Podobnie jak w przypadku tabel pochodnych, w zapytaniu użytym do definiowania wyrażenia CTE możemy również zastosować argumenty, jak w poniższym przykładzie:

```
DECLARE @empid AS INT = 3;

WITH C AS
(
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
    WHERE empid = @empid
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

Definiowanie wielu wyrażeń CTE

Z pozoru wydaje się, że różnice pomiędzy tabelami pochodnymi a wyrażeniami CTE są w zasadzie semantyczne. Jednak to, że najpierw definiujemy wyrażenie CTE, a następnie go używamy, przynosi kilka istotnych zalet w porównaniu do tabel pochodnych. Jedną z tych zalet polega na tym, że jeśli trzeba w jednym wyrażeniu CTE odwołać się do innego, nie zagnieżdżamy ich, jak miało to miejsce w przypadku tabel

pochodnych. Zamiast tego po prostu definiujemy wiele wyrażeń CTE oddzielonych przecinkami w ramach tego samego polecenia *WITH*. Każde wyrażenie CTE może odwoływać się do wszystkich poprzednio zdefiniowanych wyrażeń CTE, a zapytanie zewnętrzne może odwoływać się do wszystkich wyrażeń CTE. Na przykład poniższy kod jest skonstruowaną w oparciu o wyrażenia CTE alternatywą dla zagnieżdżonych tabel pochodnych z listingu 5-2.

```
WITH C1 AS
(
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
),
C2 AS
(
    SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
    FROM C1
    GROUP BY orderyear
)
SELECT orderyear, numcusts
FROM C2
WHERE numcusts > 70;
```

Ponieważ wyrażenie CTE zdefiniowaliśmy przed jego użyciem, nie prowadzi to do zagnieżdżania wyrażeń CTE. Każde wyrażenie CTE występuje oddzielnie w kodzie w sposób modularny. W porównaniu do metody tabel pochodnych to modularne podejście istotnie poprawia czytelność kodu i ułatwia jego utrzymywanie.

Można zauważyć, że nie można zagnieżdżać wyrażeń CTE ani nie możemy zdefiniować wyrażenia CTE wewnątrz nawiasów tabeli pochodnej. Jednak zagnieżdżanie nie jest dobrą praktyką i należy traktować te ograniczenia jako pomoc w poprawie czytelności, a nie jako przeszkodę.

Wielokrotne odwołania w wyrażeniach CTE

Z faktu, że wyrażenie CTE jest najpierw nazwane i zdefiniowane, a dopiero potem realizowane, wynika inna zaleta: dla klauzuli *FROM* zapytania zewnętrznego wyrażenie CTE już istnieje; z tego względu możemy odwoływać się do wielu instancji tego samego wyrażenia CTE. Na przykład poniżej pokazany kod jest logicznym ekwiwalentem kodu z listingu 5-3, lecz zamiast tabel pochodnych użyte zostały wyrażenia CTE.

```
WITH YearlyCount AS
(
    SELECT YEAR(orderdate) AS orderyear,
           COUNT(DISTINCT custid) AS numcusts
    FROM Sales.Orders
    GROUP BY YEAR(orderdate)
)
SELECT Cur.orderyear,
       Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
```



```
Cur.numcusts - Prv.numcusts AS growth
FROM YearlyCount AS Cur
LEFT OUTER JOIN YearlyCount AS Prv
ON Cur.orderyear = Prv.orderyear + 1;
```

Jak widzimy, wyrażenie CTE *YearlyCount* zostało zdefiniowane raz, a w klauzuli *FROM* zapytania zewnętrznego dostęp do niego uzyskiwany jest dwukrotnie – raz jako rok bieżący *Cur* i drugi raz jako rok poprzedni *Prv*. Musimy utrzymywać tylko jedną kopię zapytania wyrażenia CTE, a nie wiele kopii, jak ma to miejsce w przypadku tabel pochodnych. Prowadzi to do znacznego uproszczenia zapytania, dzięki czemu kod jest mniej podatny na błędy.

Z punktu widzenia wydajności warto przypomnieć sobie wcześniejsze informacje, że wyrażenia tablicowe zazwyczaj nie mają wpływu na wydajność, ponieważ nie są materializowane fizycznie. Oba odwołania do wyrażenia CTE w poprzednim przykładzie oczekują na rozwinięcie. Wewnętrznie zapytanie to stanowi samo-złączenie pomiędzy dwoma instancjami tabeli *Orders*, a każda z nich skanuje dane tabeli i agreguje je przed złączeniem – taki sam proces fizyczny jak w przypadku tabel pochodnych. Jeśli działania związane z odwołaniem są bardzo kosztowne i chcemy unikać ich wielokrotnego powtarzania, powinniśmy zachować wyniki zapytania wewnętrznego w tabeli tymczasowej lub w zmiennej tablicowej. Tutejszy opis skupia się na aspektach związanych z kodowaniem, a nie na wydajności, więc warto podkreślić, że możliwość tylko jednokrotnego specyfikowania zapytania wewnętrznego i wielokrotne odwołanie się do nazwy wyrażenia CTE jest sporą zaletą w porównaniu do podejścia stosowanego przy tabelach pochodnych.

Rekurencyjne wyrażenia CTE

Ten fragment jest opcjonalny, ponieważ omawia zagadnienia wykraczające poza tematykę podstawową.

Pośród wyrażeń tablicowych wyrażenia CTE są unikatowe, ponieważ mają możliwość wywoływania rekurencyjnego. Rekurencyjne wyrażenie CTE jest definiowane przez co najmniej dwa zapytania (może być ich więcej) – co najmniej jedno zapytanie nazywane jest składnikiem zakotwicającym, a co najmniej jedno zapytanie nazywane jest składnikiem rekurencyjnym. Postać ogólna podstawowego rekurencyjnego wyrażenia CTE jest następująca:

```
WITH <nazwa_CTE>[(<docelowa_lista_kolumn>)]
AS
(
  <składnik_zakotwicający>
  UNION ALL
  <składnik_rekurencyjny>
)
<zapytanie_zewnętrzne_odwołujące_się_do_CTE>;
```

Składnik zakotwiczący to zapytanie zwracające poprawną relacyjną tabelę wyników – podobnie jak zapytanie używane do zdefiniowania nierekurencyjnego wyrażenia tablicowego. Składnik zakotwiczący jest wywoływany tylko raz.

Składnik rekurencyjny to zapytanie, które odwołuje się do nazwy wyrażenia CTE. Odwołanie do nazwy wyrażenia CTE reprezentuje z punktu widzenia logiki to, co jest poprzednim zbiorem wyników w sekwencji wykonywania. Przy pierwszym użyciu składnika rekurencyjnego poprzedni zbiór wyników reprezentuje to, co zwrócił składnik zakotwiczący. W każdym kolejnym użyciu składnika rekurencyjnego odwołanie do nazwy wyrażenia CTE reprezentuje zbiór wyników zwrócony przez poprzednie wywołanie składnika rekurencyjnego. Składnik rekurencyjny wprost nie zawiera żadnego sprawdzania zakończenia rekurencji – sprawdzenie zakończenia jest dokonywane pośrednio. Składnik rekurencyjny wywoływany jest cyklicznie, aż zwróci pusty zbiór lub przekroczony zostanie pewien limit, ustalany na poziomie silnika bazy danych.

Obydwa zapytania muszą być zgodne w kwestii liczby zwracanych kolumn i typów danych odpowiednich kolumn.

Odwołanie do nazwy wyrażenia CTE w kwerendzie zewnętrznej reprezentuje zuniifikowany zbiór wyników składnika zakotwiczącego i wszystkich wywołań składnika rekurencyjnego.

Jeśli jest to pierwsze zetknięcie się z rekurencyjnymi wyrażeniami CTE, opis może wydawać się mało zrozumiały. Najlepiej będzie więc zilustrować działanie przykładem. Poniższy kod pokazuje, jak używać rekurencyjnego wyrażenia CTE, by zwrócić informacje o pracowniku (Don Funk, identyfikator pracownika równy 2) i wszystkich jego podwładnych na wszystkich poziomach (bezpośrednich i pośrednich).

```
WITH EmpsCTE AS
(
    SELECT empid, mgrid, firstname, lastname
    FROM HR.Employees
    WHERE empid = 2

    UNION ALL

    SELECT C.empid, C.mgrid, C.firstname, C.lastname
    FROM EmpsCTE AS P
    JOIN HR.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

Składnik zakotwiczący odpytuje tabelę *HR.Employees* i po prostu zwraca wiersz pracownika 2.

```
SELECT empid, mgrid, firstname, lastname
FROM HR.Employees
WHERE empid = 2
```

Składnik rekurencyjny łączy wyrażenie CTE – reprezentujące poprzedni zbiór wyników – z tabelą *Employees*, by zwrócić bezpośrednich podwładnych pracowników zwróconych w poprzednim zbiorze wyników.

```
SELECT C.empid, C.mgrid, C.firstname, C.lastname
FROM EmpsCTE AS P
JOIN HR.Employees AS C
ON C.mgrid = P.empid
```

Inaczej mówiąc, składnik rekurencyjny jest wywoływany cyklicznie, a każde jego wywołanie zwraca podwładnych na następnym poziomie zależności. Przy pierwszym wywołaniu składnika rekurencyjnego zwracani są bezpośredni podwładni – pracownicy 3 i 5. Przy drugim wywołaniu składnika rekurencyjnego zwracani są bezpośredni podwładni pracowników 3 i 5, czyli pracownicy 4, 6, 7, 8 i 9. Przy trzecim wywołaniu składnika rekurencyjnego zapytanie nie znajduje już żadnych podwładnych; składnik rekurencyjny zwraca pusty zbiór i z tego względu rekurencja zostaje zatrzymana.

Odwołanie do nazwy wyrażenia CTE w zapytaniu zewnętrznym reprezentuje zunifikowane zbiory wyników; mówiąc inaczej, pracownika 2 i wszystkich jego podwładnych.

Poniżej przedstawiono dane wyjściowe omawianego kodu:

empid	mgrid	firstname	lastname
2	1	Don	Funk
3	2	Judy	Lew
5	2	Sven	Buck
6	5	Paul	Suurs
7	5	Russell	King
9	5	Patricia	Doyle
4	3	Yael	Peled
8	3	Maria	Cameron

W przypadku błędu w predykcji złączenia w składniku rekurencyjnym lub problemów z danymi prowadzących do tworzenia cykli (sytuacja, gdy podwładny jest wskazany jako zwierzchnik swojego zwierzchnika – niekoniecznie bezpośredniego) składnik rekurencyjny może potencjalnie być wywoływany nieskończoną liczbą razy. Jako zabezpieczenie, SQL Server domyślnie ogranicza liczbę wywołań składnika rekurencyjnego do 100. Po stu wywołaniach składnika rekurencyjnego kod przestanie działać i zwróci komunikat o błędzie. Limit ten można zmienić przy użyciu wyrażenia *OPTION(MAXRECURSION n)* na końcu zapytania zewnętrznego, gdzie *n* to liczba całkowita z zakresu 0 do 32767, reprezentująca maksymalną liczbę rekurencji. Jeśli chcemy w ogóle wyeliminować to ograniczenie, wpisujemy *MAXRECURSION 0*. Zwróćmy uwagę, że system SQL Server przechowuje pośrednie zbiory wyników zwracane przez składnik zakotwiczący i rekurencyjny w tabeli roboczej w bazie *tempdb*; jeśli ograniczenie zostanie usunięte i uruchomimy takie nieskończone zapytanie, tabela robocza bardzo szybko osiągnie wielkie rozmiary, a zapytanie nie zakończy się inaczej, niż błędem przekroczenia miejsca na dysku.

Widoki

Dwa typy wyrażeń tablicowych omawianych do tej pory – tabele pochodne i wyrażenia CTE – mają bardzo ograniczony zakres, czyli zakres pojedynczego polecenia. Wyrażenia te znikają zaraz po ukończeniu zapytania odwołującego do tych wyrażeń, co oznacza, że nie jest możliwe ponowne użycie tabel pochodnych czy wyrażeń CTE.

Widoki i wbudowane funkcje zwracające tabele (*inline Table-Valued Function* – TVF) stanowią rodzaj wyrażeń tablicowych, które mogą być ponownie użyte; ich definicje są przechowywane jako obiekty bazy danych. Po ich utworzeniu obiekty te stają się permanentną częścią bazy danych i są z niej usuwane tylko wtedy, gdy wprost zadeklarujemy takie polecenie. Pod innymi względami widoki i wbudowane funkcje TVF są traktowane jak tabele pochodne i wyrażenia CTE. Na przykład podczas odpytywania widoku lub TVF system SQL Server rozwija definicje wyrażenia tablicowego i bezpośrednio odpytuje obiekty bazowe, tak jak w przypadku tabel pochodnych i wyrażeń CTE.

W tym podrozdziale opiszę widoki, a w kolejnym zajmiemy się wbudowanymi funkcjami TVF.

Jak już wspomniałem wcześniej, widok (*view*) jest wyrażeniem tablicowym, które może być ponownie użyte i którego definicja przechowywana jest w bazie danych. Na przykład poniższy kod tworzy widok nazwany *USACusts* w schemacie *Sales* bazy danych *TSQLV4*, reprezentujący wszystkich klientów z USA.

```

DROP VIEW IF EXISTS Sales.USACusts;
GO

CREATE VIEW Sales.USACusts
AS
SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO

```

Zwróćmy uwagę, że tak samo jak w przypadku tabel pochodnych i wyrażeń CTE, zamiast wewnętrznego tworzenia aliasów kolumn (pokazanego w przykładowym kodzie), możemy użyć formatu zewnętrznego tworzenia aliasów poprzez wyspecyfikowanie w nawiasach nazw kolumn docelowych zaraz po nazwie widoku.

Po utworzeniu tego widoku możemy wykonywać zapytania do widoku, tak jak do innych tabel w bazie danych.

```

SELECT custid, companyname
FROM Sales.USACusts;

```

Ponieważ widok jest obiektem bazy danych, możemy kontrolować dostęp do widoku za pomocą uprawnień tak samo jak w przypadku tabel (uprawnienia te obejmują

SELECT, *INSERT*, *UPDATE* i *DELETE*). Możemy nawet odmówić bezpośredniego dostępu do obiektu bazowego i przydzielić dostęp do widoku.

Warto pamiętać, że ogólne zalecenie unikania wyrażenia *SELECT ** ma szczególne znaczenie w kontekście widoków. W skompilowanej postaci widoku kolumny są wyliczone, a nowe kolumny nie zostaną automatycznie dodane do widoku. Załóżmy dla przykładu, że definiujemy widok w oparciu o zapytanie *SELECT * FROM dbo.T1*, a w momencie tworzenia widoku tabela *T1* ma kolumny *col1* i *col2*. SQL Server w metadanych widoku przechowuje informacje dotyczące tylko tych dwóch kolumn. Jeśli zmieni się definicja tabeli, na przykład poprzez dodanie nowych kolumn, te nowe kolumny nie zostaną dodane do widoku. Metadane widoku możemy odświeżać przy użyciu procedury składowanej *sp_refreshview* lub *sp_refreshsqlmodule*, ale w celu uniknięcia nieporozumień najlepiej wprost wymienić nazwy kolumn, które są potrzebne w definicji widoku. Jeśli w tabelach bazowych pojawią się nowe kolumny i są one potrzebne w widoku, korzystamy z polecenia *ALTER VIEW*, by odpowiednio poprawić definicję widoku.

Widoki i klauzula **ORDER BY**

Zapytanie używane do definiowania widoku musi spełniać wszystkie wymagania wymienione poprzednio dla wyrażeń tablicowych przy omawianiu tabel pochodnych. Widok nie powinien gwarantować uporządkowania wierszy, wszystkie kolumny muszą mieć nazwy i wszystkie nazwy kolumn muszą być unikatowe. W tym podrozdziale zajmiemy się trochę kwestiami uporządkowania, które mają zasadnicze znaczenie dla zrozumienia materiału.

Pamiętamy, że klauzula prezentacji *ORDER BY* nie jest dozwolona w zapytaniu definiującym wyrażenie tablicowe, ponieważ relacja nie zawiera pojęcia uporządkowania. Próba utworzenia uporządkowanego widoku nie ma sensu, ponieważ narusza podstawowe właściwości relacji zdefiniowane przez model relacyjny. Jeśli dla celów prezentacji chcemy zwrócić posortowane wiersze widoku, nie powinniśmy tworzyć widoku w niedozwolony sposób. Zamiast tego należy wyspecyfikować klauzulę prezentacji *ORDER BY* w zapytaniu zewnętrznym do widoku, jak w poniższym przykładzie:

```
SELECT custid, companyname, region
FROM Sales.USACusts
ORDER BY region;
```

Spróbujmy uruchomić poniższy kod, by utworzyć widok przy użyciu klauzuli prezentacji *ORDER BY*.

```
ALTER VIEW Sales.USACusts
AS
SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
```

```
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

Próba uruchomienia tego kodu wygeneruje następujący komunikat o błędzie:

```
Msg 1033, Level 15, State 1, Procedure USACusts, Line 9
The ORDER BY clause is invalid in views, inline functions, derived tables,
subqueries, and common table expressions, unless TOP, OFFSET or FOR XML is also
specified.
```

(Klauzula *ORDER BY* jest nieprawidłowa w widokach, funkcjach wewnętrznych, tabelach pochodnych, podkwerendach i wspólnych wyrażeniach tablicowych, chyba że wyspecyfikowane są również opcje *TOP*, *OFFSET* lub *FOR XML*.)

Komunikat o błędzie wskazuje, że SQL Server dopuszcza klauzulę *ORDER BY* w trzech sytuacjach – gdy została użyta opcja *TOP*, *OFFSET-FETCH* lub *FOR XML*. We wszystkich tych przypadkach klauzula *ORDER BY* ma inne przeznaczenie niż jej typowe zadanie, czyli prezentacja. Standard SQL ma również podobne ograniczenie i podobny wyjątek związany z użyciem standardowej opcji *OFFSET-FETCH*.

Ponieważ język T-SQL zezwala na stosowanie klauzuli *ORDER BY* w widoku, jeśli dodatkowo wyspecyfikowana jest opcja *TOP* lub *OFFSET-FETCH*, niektórzy sądzą, że można utworzyć „uporządkowane widoki”. Jednym ze sposobów zrealizowania takiego pomysłu jest użycie opcji *TOP (100) PERCENT*, jak w poniższym przykładzie:

```
ALTER VIEW Sales.USACusts
AS
SELECT TOP (100) PERCENT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

Chociaż kod z punktu widzenia technicznego jest poprawny i widok zostanie utworzony, powinniśmy zdawać sobie sprawę, że ponieważ zapytanie zostało użyte do definiowania wyrażenia tablicowego, klauzula *ORDER BY* w tym miejscu może tylko służyć do filtrowania logicznego dla opcji *TOP*. Jeśli wykonamy zapytanie widoku i nie wyspecyfikujemy klauzuli *ORDER BY*, nie będzie zagwarantowana kolejność prezentowania wyników.

Dla przykładu uruchomimy poniższe zapytanie do tego widoku.

```
SELECT custid, companyname, region
FROM Sales.USACusts;
```

Pokazane poniżej wyniki jednego z uruchomień pokazują, że wiersze nie są posortowane według regionu.

custid	companyname	region
32	Customer YSIQX	OR
36	Customer LVJSO	OR
43	Customer UISOJ	WA
45	Customer QXPPT	CA
48	Customer DVFMB	OR
55	Customer KZQZT	AK
65	Customer NYUHS	NM
71	Customer LCOUJ	ID
75	Customer X0JYP	WY
77	Customer LCYBZ	OR
78	Customer NLTYP	MT
82	Customer EYHKM	WA
89	Customer YBQTI	WA

Jeśli zewnętrzne zapytanie nie zawiera *ORDER BY*, wyniki mogą, ale nie muszą być zwracane w określonym porządku. Jeśli wyniki są uporządkowane, przyczyną mogą być mechanizmy optymalizacji, ale zachowanie to nie jest powtarzalne. Jedynym sposobem zagwarantowania określonego porządku wyników jest użycie klauzuli *ORDER BY* w zapytaniu zewnętrznym.

W starszych wersjach SQL Server, gdy zapytanie wewnętrzne zawierało kombinację klauzul *TOP (100) PERCENT* i *ORDER BY*, zaś zapytanie zewnętrzne nie używało klauzuli *ORDER BY*, otrzymywaliśmy uporządkowane wiersze. Nie było to zachowanie gwarantowane, ale występowało w wyniku sposobu obsługi takich zapytań przez optymalizator. W którymś momencie firma Microsoft dodała sprytniejszą metodę optymalizacji, która uwzględniała taką kombinację. Niestety optymalizator nie bierze pod uwagę sytuacji, gdy zapytanie wewnętrzne używa klauzuli *OFFSET* z wartością 0 *ROWS* i bez klauzuli *FETCH*, jak w poniższym przykładzie:

```
ALTER VIEW Sales.USACusts
AS

SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region
OFFSET 0 ROWS;
GO
```

W pewnych sytuacjach, gdy przy odpytywaniu widoku nie użyjemy klauzuli *ORDER BY* w zapytaniu zewnętrznym, wiersze w wynikach mogą być posortowane według regionu. Ale trzeba wyraźnie podkreślić – *nie zakładajmy*, że tak będzie zawsze. Może zdarzyć się to dzięki mechanizmom bieżącej optymalizacji. Jeśli chcemy zagwarantować, że zapytanie widoku zwróci posortowane wiersze, w zapytaniu zewnętrznym trzeba zastosować klauzulę *ORDER BY*.

Nie należy mylić działania zapytania użytego do definiowania wyrażenia tablicowego z działaniem innego zapytania. Zapytanie z klauzulą *ORDER BY* i opcją *TOP* lub *OFFSET-FETCH* nie gwarantują kolejności prezentowania tylko w kontekście wyrażenia tablicowego. W przypadku zapytania nie używanego do definiowania wyrażenia tablicowego klauzula *ORDER BY* służy zarówno do filtrowania dla opcji *TOP* lub *OFFSET-FETCH*, jak i do prezentowania wyników.

Opcje widoku

Podczas tworzenia lub modyfikowania widoku możemy wskazać atrybuty i opcje widoku jako część jego definicji. W nagłówku widoku, poniżej klauzuli *WITH*, możemy wyspecyfikować takie atrybuty, jak *ENCRYPTION* i *SCHEMABINDING*, a na końcu zapytania możemy wyspecyfikować opcję *WITH CHECK OPTION*. Kolejne podpunkty zawierają przeznaczenie tych opcji.

Opcja *ENCRYPTION*

Opcja *ENCRYPTION* jest dostępna przy tworzeniu lub zmienianiu widoków, procedur składowanych, wyzwalaczy i funkcji definiowanych przez użytkownika (UDF). Opcja *ENCRYPTION* wskazuje, że system SQL Server będzie wewnętrznie przechowywał tekst z definicjami obiektu w zamaskowanym formacie. Zamaskowany tekst nie jest widoczny bezpośrednio dla użytkowników przy użyciu dowolnego obiektu katalogu – widoczny jest tylko dla użytkowników uprzywilejowanych przy użyciu specjalnych środków.

Zanim przyjrzymy się opcji *ENCRYPTION*, uruchomimy poniższy kod, by zmienić definicję widoku *USACusts* i przywrócić mu jego pierwotną postać.

```
ALTER VIEW Sales.USACusts
AS

SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

W celu uzyskania definicji widoku wywołujemy funkcję *OBJECT_DEFINITION*, jak w poniższym kodzie.

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

Tekst definicji widoku jest dostępny, ponieważ widok został utworzony bez opcji *ENCRYPTION*. Uzyskujemy następujące dane wyjściowe.

```
CREATE VIEW Sales.USACusts
AS
```



```

SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';

```

Następnie zmieniamy definicję widoku – tym razem dołączając opcję *ENCRYPTION*.

```

ALTER VIEW Sales.USACusts WITH ENCRYPTION
AS

SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO

```

Próbujemy ponownie uzyskać tekst definicji widoku.

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

Tym razem otrzymujemy *NULL*.

Jako alternatywnej metody dla funkcji *OBJECT_DEFINITION*, możemy użyć procedury składowanej *sp_helptext*, by uzyskać definicje obiektów. Na przykład poniższy kod żąda definicji obiektu dla widoku *USACusts*.

```
EXEC sp_helptext 'Sales.USACusts';
```

Ponieważ w tym przypadku widok był utworzony przy użyciu opcji *ENCRYPTION*, nie otrzymamy definicji obiektu, a poniższy komunikat:

```
The text for object 'Sales.USACusts' is encrypted.
(tekst dla obiektu 'Sales.USACusts' jest zaszyfrowany)
```

Opcja *SCHEMABINDING*

Opcja *SCHEMABINDING* jest dostępna dla widoków i UDF; opcja wiąże schemat obiektów i kolumn odwołania ze schematem obiektu odwołującego się. Opcja wskazuje, że obiekty odwołania nie mogą być usuwane i że kolumny odwołania nie mogą zostać usunięte lub zmienione.

Dla przykładu zmienimy widok *USACusts* za pomocą opcji *SCHEMABINDING*.

```

ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS

SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO

```

Teraz spróbujemy usunąć kolumnę *Address* z tabeli *Customers*.

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

Otrzymamy poniższy komunikat o błędzie:

```
Msg 5074, Level 16, State 1, Line 1
```

```
The object 'USACusts' is dependent on column 'address'.
```

```
Msg 4922, Level 16, State 9, Line 1
```

```
ALTER TABLE DROP COLUMN address failed because one or more objects access this column.
```

(Obiekt 'USACusts' jest zależny od kolumny 'address'. Polecenie ALTER TABLE DROP COLUMN address nie powiodło się, ponieważ jeden lub więcej obiektów uzyskuje dostęp do tej kolumny.)

Bez opcji *SCHEMABINDING* moglibyśmy dokonywać takich zmian schematu, a także usunąć całą tabelę *Customers*. Prowadziłoby to do błędów w czasie działania kodu, kiedy zapytanie do widoku napotkałoby odniesienie do obiektów czy kolumn, które nie istnieją. Jeśli utworzymy widok przy użyciu opcji *SCHEMABINDING*, możemy uniknąć tego rodzaju błędów.

Aby opcja *SCHEMABINDING* była obsługiwana, definicja obiektu musi spełniać dwa wymagania. Zapytanie nie może używać znaku gwiazdki * w klauzuli *SELECT*; zamiast tego musimy wprost wymienić nazwy kolumn. Ponadto podczas odwoływania się do obiektów musimy używać dwuczęściowych nazw z prefiksem określającym schemat. Oba wymagania są ogólnie zalecanymi rozwiązaniami praktycznymi.

Jak można się domyślać, tworzenie obiektów z opcją *SCHEMABINDING* jest zalecanym rozwiązaniem.

Opcja *CHECK OPTION*

Opcja *CHECK OPTION* służy do uniemożliwienia wprowadzania poprzez widok modyfikacji, które są w konflikcie z filtrem widoku – zakładając, że zapytanie definiujące widok zawiera filtr.

Zapytanie definiujące widok *USACusts* filtruje klientów, dla których atrybut kraj ma wartość *N'USA'*. Aktualnie widok definiowany jest bez opcji *CHECK OPTION*. Oznacza to, że poprzez widok możemy wstawiać wiersze z klientami z krajów innych niż USA i że możemy aktualizować istniejących klientów, zmieniając ich kraj na inny. Na przykład poniższy kod poprzez widok pomyślnie wstawia klienta, dla którego nazwa firmy to Customer *ABCDE* z Wielkiej Brytanii (*UK*).

```
INSERT INTO Sales.USACusts(
    companyname, contactname, contacttitle, address,
    city, region, postcode, country, phone, fax)
VALUES(
    N'Customer ABCDE', N'Contact ABCDE', N'Title ABCDE', N'Address ABCDE',
    N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

Wiersz został wstawiony poprzez widok do tabeli *Customers*. Ponieważ jednak widok filtruje tylko klientów z USA, jeśli odpytamy widok szukając nowego klienta, otrzymujemy pusty zbiór.

```
SELECT custid, companyname, country
FROM Sales.USACusts
WHERE companyname = N'Customer ABCDE';
```

Skierujmy pytanie bezpośrednio do tabeli *Customers*, by znaleźć nowego klienta.

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE companyname = N'Customer ABCDE';
```

W danych wyjściowych uzyskamy informacje o kliencie, ponieważ do tabeli *Customers* został wstawiony nowy wiersz.

custid	companyname	country
92	Customer ABCDE	UK

Podobnie, jeśli poprzez widok zaktualizujemy wiersz klienta, zmieniając atrybut kraju na inny niż USA, aktualizacja tabeli zostanie wykonana. Jednak informacje o tym kliencie nie pojawią się w widoku, ponieważ nie będą już spełnione warunki filtru zapytania widoku.

Jeśli chcemy zapobiec modyfikacjom, które nie są zgodne z filtrem widoku, dodajemy opcję *WITH CHECK OPTION* na końcu zapytania definiującego widok.

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS
SELECT
    custid, companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
WITH CHECK OPTION;
GO
```

Teraz spróbujmy wstawić wiersz, który jest w konflikcie z filtrem widoku.

```
INSERT INTO Sales.USACusts(
    companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax)
VALUES(
    N'Customer FGHIJ', N'Contact FGHIJ', N'Title FGHIJ', N'Address FGHIJ',
    N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

Pojawi się następujący komunikat o błędzie:

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies
WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or
more rows resulting from the operation did not qualify under the CHECK OPTION
constraint.
The statement has been terminated.
```

(Nie powiodła się próba wstawienia lub zaktualizowania, ponieważ widok docelowy specyfikuje opcję WITH CHECK OPTION albo obejmuje widok, który specyfikuje opcję WITH CHECK OPTION i co najmniej jeden wiersz wyniku operacji nie spełnia ograniczenia opcji WITH CHECK OPTION. Polecenie zostało zakończone.)

Po wykonaniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DELETE FROM Sales.Customers
WHERE custid > 91;
IF OBJECT_ID('Sales.USACusts') IS NOT NULL DROP VIEW Sales.USACusts;
```

Wbudowane funkcje zwracające tabele

Funkcje wbudowane (*inline*) zwracające tabele (*table-valued function* – TVF) są wyrażeniami tablicowymi, których można używać wielokrotnie i które obsługują parametry wejściowe. Poza obsługą parametrów wejściowych pod każdym względem funkcje TVF są podobne do widoków. Z tego powodu możemy myśleć o funkcjach TVF jako o sparametryzowanych widokach, chociaż formalnie nie są tak nazywane.

Dla przykładu poniższy kod tworzy funkcję TVF nazwaną *GetCustOrders* w bazie danych TSQVL4.

```
USE TSQVL4;
DROP FUNCTION IF EXIST dbo.GetCustOrders;
GO

CREATE FUNCTION dbo.GetCustOrders
(@cid AS INT) RETURNS TABLE
AS
RETURN
    SELECT orderid, custid, empid, orderdate, requireddate,
           shippeddate, shipperid, freight, shipname, shipaddress, shipcity,
           shipregion, shippostalcode, shipcountry
    FROM Sales.Orders
    WHERE custid = @cid;
GO
```

Ta funkcja TVF akceptuje parametr wejściowy nazwany @cid, reprezentujący identyfikator klienta i zwraca wszystkie zamówienia, które zostały złożone przez klienta wskazanego jako parametr wejściowy. Wbudowane funkcje TVF odpytujemy przy użyciu poleceń DML w taki sam sposób, jak odpytywane są inne tabele. Jeśli funkcja akceptuje parametry wejściowe, specyfikujemy parametry w nawiasach po nazwie funkcji. Ponadto trzeba się upewnić, że udostępniliśmy alias dla wyrażenia tablicowego. Udostępnianie wyrażenia tablicowego za pomocą aliasu nie zawsze jest wymagane, jest to jednak zalecane rozwiązanie praktyczne, ponieważ dzięki temu nasz kod jest bardziej czytelny i mniej podatny na błędy. Na przykład poniższy kod to zapytanie do funkcji żądające wszystkich zamówień, które zostały złożone przez klienta o identyfikatorze 1.

```
SELECT orderid, custid
FROM dbo.GetCustOrders(1) AS O;
```

Kod zwraca następujące wyniki:

orderid	custid
-----	-----
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1

Podobnie jak w przypadku innych tabel, możemy odwoływać się do wbudowanych funkcji TVF jako części złączenia. Poniższe zapytanie łączy funkcję TVF z tabelą *Sales.OrderDetails*, zwracając zamówienia klienta 1 z powiązanymi pozycjami zamówienia.

```
SELECT O.orderid, O.custid, OD.productid, OD.qty
FROM dbo.GetCustOrders(1) AS O
      JOIN Sales.OrderDetails AS OD
      ON O.orderid = OD.orderid;
```

Kod ten generuje następujące dane wyjściowe:

orderid	custid	productid	qty
-----	-----	-----	-----
10643	1	28	15
10643	1	39	21
10643	1	46	2
10692	1	63	20
10702	1	3	6
10702	1	76	15
10835	1	59	15
10835	1	77	2
10952	1	6	16
10952	1	28	2
11011	1	58	40
11011	1	71	20

Po wykonaniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
IF OBJECT_ID('dbo.GetCustOrders') IS NOT NULL
    DROP FUNCTION dbo.GetCustOrders;
```

Operator *APPLY*

Operator tabel *APPLY* ma duże możliwości. Podobnie do innych operatorów tabel, jest używany w klauzuli *FROM* zapytania. Obsługiwane są dwa typy operatora *APPLY* – *CROSS APPLY* i *OUTER APPLY*. Operator *CROSS APPLY* implementuje tylko jedną logiczną fazę przetwarzania zapytania, natomiast operator *OUTER APPLY* dwie fazy.

UWAGA Operator *APPLY* nie należy do standardu; jego standardowym odpowiednikiem jest operator *LATERA*; jednak operator ten w standardowej postaci nie został zaimplementowany w systemie SQL Server.



Operator *APPLY* działa na dwóch tabelach wejściowych; będziemy odnosić się do nich jako do tabeli „lewej” i „prawej”. Prawa tabela jest zazwyczaj tabelą pochodną lub funkcją TVF. Operator *CROSS APPLY* stosuje jedną fazę logicznego przetwarzania zapytania – stosuje prawe wyrażenie tablicowe do każdego wiersza z lewej tabeli i generuje tabelę wyników jako zuniifikowany zbiór wyników.

Do tej pory może wydawać się, że operator *CROSS APPLY* jest bardzo podobny do złączenia krzyżowego i w zasadzie tak jest. Na przykład poniższe dwa zapytania zwracają takie same zbiory wyników.

```
SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
      CROSS JOIN HR.Employees AS E;
```

```
SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
      CROSS APPLY HR.Employees AS E;
```

Przypomnijmy jednak, że złączenie traktuje obie tabele wejściowe jako zbiory, a tym samym nie ma pomiędzy nimi żadnego uporządkowania. Oznacza to, że nie możemy odwoływać się po jednej stronie do elementów z drugiej strony. W przypadku operatora *APPLY* lewa strona jest przetwarzana najpierw, a następnie przetwarzana jest prawa strona dla każdego wiersza dostarczonego z lewej strony. Tak więc po prawej stronie mogą występować odniesienia do elementów z lewej strony. Dla przykładu poniższy kod wykorzystuje operator *CROSS APPLY* do zwrócenia trzech najnowszych zamówień dla każdego klienta:

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
      CROSS APPLY
      (SELECT TOP (3) orderid, empid, orderdate, requireddate
       FROM Sales.Orders AS O
       WHERE O.custid = C.custid
       ORDER BY orderdate DESC, orderid DESC) AS A;
```

Możemy myśleć o wyrażeniu tablicowym *A* jak o skorelowanym podzapytaniu do tabeli. W kontekście logicznego przetwarzania zapytania prawe wyrażenie tablicowe (w tym przypadku tabela pochodna) jest stosowane do każdego wiersza w tabeli *Customers*. Zwróćmy uwagę na odwołanie do atrybutu *C.custid* z lewej tabeli w filtrze zapytania tabeli pochodnej. Tabela pochodna zwraca dla klienta z bieżącego lewego wiersza trzy najnowsze zamówienia. Ponieważ tabela pochodna jest stosowana do każdego wiersza z lewej strony, operator *CROSS APPLY* dla każdego klienta zwraca trzy najnowsze zamówienia.

Poniżej pokazano wyniki generowane przez to zapytanie (w skróconej postaci):

custid	orderid	orderdate
1	11011	2016-04-09
1	10952	2016-03-16

```

1      10835      2016-01-15
2      10926      2016-03-04
2      10759      2015-11-28
2      10625      2015-08-08
3      10856      2016-01-28
3      10682      2015-09-25
3      10677      2015-09-22
...

```

(263 row(s) affected)

Przypomnijmy, że począwszy od wersji SQL Server 2012 zamiast opcji *TOP* możemy stosować standardową opcję *OFFSET-FETCH*, jak w poniższym przykładzie:

```

SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
CROSS APPLY
    (SELECT orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC
     OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY) AS A;

```

Jeśli prawe wyrażenie tablicowe zwraca pusty zbiór, operator *CROSS APPLY* nie zwróci odpowiadającego lewego wiersza. Na przykład klienci 22 i 57 nie złożyli zamówień. W obu przypadkach tabela pochodna to pusty zbiór; z tego względu klienci ci nie są zwracani w wynikach. Jeśli chcemy zwracać wiersze z lewej tabeli, dla których prawe wyrażenie tablicowe zwraca pusty zbiór, używamy operatora *OUTER APPLY* zamiast *CROSS APPLY*. Operator *OUTER APPLY* dodaje drugą fazę logicznego przetwarzania zapytania, która identyfikuje wiersze z lewej strony, dla których prawe wyrażenie tablicowe zwraca pusty zbiór i dodaje te wiersze do tabeli wyników jako wiersze zewnętrzne – ze znacznikami *NULL* dla atrybutów prawej strony jako wypełniaczami. Tak więc faza ta jest podobna do fazy, która dodaje wiersze zewnętrzne w lewostronnym złączeniu zewnętrznym.

Dla przykładu uruchomimy poniższy kod, by dla każdego klienta zwrócić trzy najnowsze zamówienia, uwzględniając klientów, którzy nie złożyli żadnych zamówień.

```

SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
OUTER APPLY
    (SELECT TOP (3) orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS A;

```

Tym razem klienci 22 i 57, którzy nie złożyli zamówień, zostali dołączeni do danych wyjściowych (wyniki pokazane są w skróconej postaci):

```

custid      orderid      orderdate
-----
1           11011      2016-04-09

```

1	10952	2016-03-16
1	10835	2016-01-15
2	10926	2016-03-04
2	10759	2015-11-28
2	10625	2015-08-08
3	10856	2016-01-28
3	10682	2015-09-25
3	10677	2015-09-22
...		
22	NULL	NULL
...		
57	NULL	NULL
...		

(265 row(s) affected)

Niekiedy może się okazać, że wygodniej jest używać funkcji TVF zamiast tabel pochodnych. W ten sposób nasz kod będzie prostszy w analizowaniu i utrzymaniu. Na przykład poniższy kod tworzy wewnętrzną funkcję TVF nazwaną *TopOrders*, która akceptuje jako dane wejściowe identyfikator klienta (*@custid*) oraz liczbę (*@n*) i dla klienta *@custid* zwraca *@n* najnowszych zamówień.

```
IF OBJECT_ID('dbo.TopOrders') IS NOT NULL
    DROP FUNCTION dbo.TopOrders;
GO
CREATE FUNCTION dbo.TopOrders
    (@custid AS INT, @n AS INT)
    RETURNS TABLE
AS
RETURN
    SELECT TOP (@n) orderid, empid, orderdate, requireddate
    FROM Sales.Orders
    WHERE custid = @custid
    ORDER BY orderdate DESC, orderid DESC;
GO
```

Teraz możemy zastąpić użycie tabeli pochodnej z poprzednich przykładów nową funkcją.

```
SELECT
    C.custid, C.companyname,
    A.orderid, A.empid, A.orderdate, A.requireddate
FROM Sales.Customers AS C
    CROSS APPLY dbo.TopOrders(C.custid, 3) AS A;
```

Kod taki jest znacznie bardziej czytelny i łatwiej go utrzymywać. Z punktu widzenia przetwarzania fizycznego nie następują żadne zmiany, ponieważ, jak wyjaśniałem wcześniej, definicje wyrażeń tablicowych są rozwijane, a w końcowym efekcie SQL Server w każdym przypadku będzie bezpośrednio odpytywał obiekty bazowe.

Podsumowanie

Wyrażenia tablicowe upraszczają kod, ułatwiają jego utrzymywanie i opakowują logikę zapytań. Jeśli zachodzi potrzeba użycia wyrażeń tablicowych i nie planujemy ponownego użycia ich definicji, stosujemy tabele pochodne lub wyrażenia CTE. W porównaniu do tabel pochodnych wyrażenia CTE mają kilka zalet; nie zagnieżdżamy wyrażeń CTE, jak ma to miejsce w przypadku tabel pochodnych, dzięki czemu kod używający wyrażeń CTE jest bardziej modularny i łatwiejszy do utrzymania. Ponadto możemy odnosić się do wielu instancji tego samego wyrażenia CTE, co nie jest możliwe w przypadku tabel pochodnych.

Jeśli zachodzi potrzeba zdefiniowania wyrażeń tablicowych wielokrotnego użycia, stosujemy widoki lub wbudowane funkcje TVF. Jeśli nie są potrzebne parametry wejściowe, stosujemy widoki; w przeciwnym razie stosujemy funkcje TVF.

Operator *APPLY* jest używany, jeśli chcemy zastosować wyrażenie tablicowe do każdego wiersza tabeli źródłowej i scalać wszystkie zbiory wyników w jedną tabelę wyników.

Ćwiczenia

Ten podrozdział zawiera ćwiczenia, które ułatwiają lepsze przyswojenie tematyki opisanej w rozdziale. Przy wszystkich ćwiczeniach w tym rozdziale wymagane jest połączenie z bazą danych *TSQVLV4*.

Ćwiczenie 1

Poniższe zapytanie próbuje wyfiltrować zamówienia, które nie zostały złożone ostatniego dnia roku. Oczekuje się, że zwróci ono identyfikator i datę zamówienia, identyfikatory klienta i pracownika oraz odpowiednią datę ostatniego dnia roku dla każdego zamówienia:

```
SELECT orderid, orderdate, custid, empid,  
       DATEFROMPARTS(YEAR(orderdate), 12, 31) AS endofyear  
FROM Sales.Orders  
WHERE orderdate <> endofyear;
```

Przy próbie uruchomienia tego zapytania otrzymujemy błąd:

```
Msg 207, Level 16, State 1, Line 233  
Invalid column name 'endofyear'.
```

Należy wyjaśnić, na czym polega problem i zasugerować poprawne rozwiązanie.

Ćwiczenie 2-1

Napisać zapytanie, które dla każdego pracownika zwraca maksymalną wartość kolumny *orderdate* (data zamówienia).

- Wykorzystywane tabele: baza danych *TSQLV4*, tabela *Sales.Orders*
- Oczekiwane dane wyjściowe:

empid	maxorderdate
3	2016-04-30
6	2016-04-23
9	2016-04-29
7	2016-05-06
1	2016-05-06
4	2016-05-06
2	2016-05-05
5	2016-04-22
8	2016-05-06

(9 row(s) affected)

Ćwiczenie 2-2

Opakować zapytanie z ćwiczenia 2-1 do postaci tabeli pochodnej. Napisać zapytanie złączenia pomiędzy tabelą pochodną a tabelą *Orders*, by dla każdego pracownika zwrócić zamówienia o maksymalnej dacie zamówienia.

- Wykorzystywane tabele: *Sales.Orders*
- Oczekiwane dane wyjściowe:

empid	orderdate	orderid	custid
9	2016-04-29	11058	6
8	2016-05-06	11075	68
7	2016-05-06	11074	73
6	2016-04-23	11045	10
5	2016-04-22	11043	74
4	2016-05-06	11076	9
3	2016-04-30	11063	37
2	2016-05-05	11073	58
2	2016-05-05	11070	44
1	2016-05-06	11077	65

(10 row(s) affected)

Ćwiczenie 3-1

Napisać zapytanie, które dla każdego zamówienia oblicza numer wiersza w oparciu o kolejność *orderdate*, *orderid*.

- Wykorzystywane tabele: *Sales.Orders*

- Oczekiwane dane wyjściowe (w skróconej postaci):

orderid	orderdate	custid	empid	rownum
-----	-----	-----	-----	-----
10248	2014-07-04	85	5	1
10249	2014-07-05	79	6	2
10250	2014-07-08	34	4	3
10251	2014-07-08	84	3	4
10252	2014-07-09	76	4	5
10253	2014-07-10	34	3	6
10254	2014-07-11	14	5	7
10255	2014-07-12	68	9	8
10256	2014-07-15	88	3	9
10257	2014-07-16	35	4	10
...				
(830 row(s) affected)				

Ćwiczenie 3-2

Napisać zapytanie, które zwraca wiersze o numerach od 11 do 20 w oparciu o definicję numeru wiersza z ćwiczenia 3-1. Użyć wyrażenia CTE do enkapsulacji kodu z ćwiczenia 3-1.

- Wykorzystywane tabele: *Sales.Orders*

- Oczekiwane dane wyjściowe:

orderid	orderdate	custid	empid	rownum
-----	-----	-----	-----	-----
10258	2014-07-17	20	1	11
10259	2014-07-18	13	4	12
10260	2014-07-19	56	4	13
10261	2014-07-19	61	4	14
10262	2014-07-22	65	8	15
10263	2014-07-23	20	9	16
10264	2014-07-24	24	6	17
10265	2014-07-25	7	2	18
10266	2014-07-26	87	3	19
10267	2014-07-29	25	4	20
(10 row(s) affected)				

Ćwiczenie 4 (zaawansowane ćwiczenie opcjonalne)

Napisać rozwiązanie przy użyciu rekurencyjnego wyrażenia CTE, które zwraca łańcuch zależności służbowej prowadzący do pracownika Patricia Doyle (identyfikator pracownika 9).

- Wykorzystywane tabele: *HR.Employees*

- Oczekiwane dane wyjściowe:

empid	mgrid	firstname	lastname
9	5	Patricia	Doyle
5	2	Sven	Buck
2	1	Don	Funk
1	NULL	Sara	Davis

(4 row(s) affected)

Ćwiczenie 5-1

Utworzyć widok *VEmpOrders*, który zwraca łączną ilość dla każdego pracownika i roku.

- Wykorzystywane tabele: *Sales.Orders* i *Sales.OrderDetails*
- Wykonanie następującego kodu:

```
SELECT * FROM Sales.VEmpOrders ORDER BY empid, orderyear;
```

powinno dać poniższe dane wyjściowe:

empid	orderyear	qty
1	2014	1620
1	2015	3877
1	2016	2315
2	2014	1085
2	2015	2604
2	2016	2366
3	2014	940
3	2015	4436
3	2016	2476
4	2014	2212
4	2015	5273
4	2016	2313
5	2014	778
5	2015	1471
5	2016	787
6	2014	963
6	2015	1738
6	2016	826
7	2014	485
7	2015	2292
7	2016	1877
8	2014	923
8	2015	2843
8	2016	2147
9	2014	575
9	2015	955
9	2016	1140

(27 row(s) affected)

Ćwiczenie 5-2 (zaawansowane ćwiczenie opcjonalne)

Napisać zapytanie do *Sales.VEmpOrders*, które zwraca skumulowaną łączną ilość dla każdego pracownika i roku.

- Wykorzystywane tabele: widok *Sales.VEmpOrders*
- Oczekiwane dane wyjściowe:

empid	orderyear	qty	runqty
-----	-----	-----	-----
1	2014	1620	1620
1	2015	3877	5497
1	2016	2315	7812
2	2014	1085	1085
2	2015	2604	3689
2	2016	2366	6055
3	2014	940	940
3	2015	4436	5376
3	2016	2476	7852
4	2014	2212	2212
4	2015	5273	7485
4	2016	2313	9798
5	2014	778	778
5	2015	1471	2249
5	2016	787	3036
6	2014	963	963
6	2015	1738	2701
6	2016	826	3527
7	2014	485	485
7	2015	2292	2777
7	2016	1877	4654
8	2014	923	923
8	2015	2843	3766
8	2016	2147	5913
9	2014	575	575
9	2015	955	1530
9	2016	1140	2670
(27 row(s) affected)			

Ćwiczenie 6-1

Utworzyć funkcję TVF, która akceptuje jako dane wejściowe identyfikator dostawcy (*@supid AS INT*) i żadaną liczbę produktów (*@n AS INT*). Funkcja powinna zwracać liczbę *@n* produktów o najwyższych cenach jednostkowych, które zostały dostarczone przez określonego dostawcę.

- Wykorzystywane tabele: *Production.Products*
- Wykonanie następującego kodu:

```
SELECT * FROM Production.TopProducts(5, 2);
```

da następujące dane wyjściowe:

productid	productname	unitprice
12	Product OSFNS	38.00
11	Product QMVUN	21.00

(2 row(s) affected)

Ćwiczenie 6-2

Przy użyciu operatora *CROSS APPLY* i funkcji utworzonej w ćwiczeniu 6-1, dla każdego dostawcy zwrócić dwa najdroższe produkty.

- Oczekiwane dane wyjściowe (w skróconej postaci).

supplierid	companyname	productid	productname	unitprice
8	Supplier BWGYE	20	Product QHFFP	81.00
8	Supplier BWGYE	68	Product TBTBL	12.50
20	Supplier CIYNM	43	Product ZZZHR	46.00
20	Supplier CIYNM	44	Product VJIEO	19.45
23	Supplier ELCRN	49	Product FPYPN	20.00
23	Supplier ELCRN	76	Product JYGFE	18.00
5	Supplier EQPNC	12	Product OSFNS	38.00
5	Supplier EQPNC	11	Product QMVUN	21.00
...				

(55 row(s) affected)

Po ukończeniu ćwiczeń należy uruchomić poniższy kod, by wyczyścić bazę danych.

```

DROP VIEW IF EXISTS Sales.VEmpOrders;
DROP FUNCTION IF EXISTS Production.TopProducts;

```

Rozwiązania

Podrozdział ten zawiera rozwiązania ćwiczeń wraz z odpowiednimi wyjaśnieniami.

Ćwiczenie 1

Problem ten wiąże się z logiczną kolejnością przetwarzania: klauzula *SELECT* jest wykonywana po klauzuli *WHERE*. Oznacza to, że nie możemy odwoływać się w klauzuli *WHERE* do aliasów utworzonych na liście *SELECT*. Rozwiązaniem, które nie wymaga wielokrotnego powtarzania tych samych wyrażeń, jest zdefiniowanie wyrażenia tablicowego, takiego jak CTE, które definiuje alias, po czym można wielokrotnie odwoływać się do tego aliasu w zapytaniu zewnętrznym. W tym przypadku rozwiązanie może wyglądać następująco:

```

WITH C AS
(

```

```

SELECT *,
    DATEFROMPARTS(YEAR(orderdate), 12, 31) AS endofyear
FROM Sales.Orders)
SELECT orderid, orderdate, custid, empid, endofyear
FROM C
WHERE orderdate <> endofyear;

```

Ćwiczenie 2-1

Ćwiczenie to jest tylko wstępnym krokiem wymaganym do zrealizowania następnego ćwiczenia. Na tym etapie piszemy zapytanie, które dla każdego pracownika zwraca maksymalną (najnowszą) datę zamówienia.

```
USE TSQLV4;
```

```

SELECT empid, MAX(orderdate) AS maxorderdate
FROM Sales.Orders
GROUP BY empid;

```

Ćwiczenie 2-2

Ćwiczenie to wymaga użycia zapytania z poprzedniego ćwiczenia do zdefiniowania tabeli pochodnej i złączenia tej tabeli pochodnej z tabelą *Orders*, by zwrócić zamówienia o maksymalnej dacie, jak w poniższym kodzie:

```

SELECT O.empid, O.orderdate, O.orderid, O.custid
FROM Sales.Orders AS O
    JOIN (SELECT empid, MAX(orderdate) AS maxorderdate
          FROM Sales.Orders
          GROUP BY empid) AS D
    ON O.empid = D.empid
    AND O.orderdate = D.maxorderdate;

```

Ćwiczenie 3-1

Ćwiczenie to jest wstępnym krokiem do kolejnego ćwiczenia i wymaga odpytania tabeli *Orders* i obliczenia liczby wierszy w oparciu o kolejność *orderdate*, *orderid*, jak poniżej:

```

SELECT orderid, orderdate, custid, empid,
    ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
FROM Sales.Orders;

```

Ćwiczenie 3-2

W ćwiczeniu tym wymagane jest zdefiniowanie wyrażenia CTE w oparciu o zapytanie z poprzedniego ćwiczenia i odfiltrowania z wyrażenia CTE tylko tych wierszy, których numer mieści się w zakresie od 11 do 20, jak w poniższym kodzie:

```
WITH OrdersRN AS
(
    SELECT orderid, orderdate, custid, empid,
           ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
    FROM Sales.Orders
)
SELECT * FROM OrdersRN WHERE rownum BETWEEN 11 AND 20;
```

Ktoś mógłby w tym momencie zapytać, dlaczego potrzebne jest nam tu wyrażenie tablicowe. Funkcje okien (takie jak funkcja ROW_NUMBER) można stosować jedynie w klauzulach *SELECT* i *ORDER BY* zapytania, ale nie bezpośrednio w klauzuli *WHERE*. Przy użyciu wyrażenia tablicowego możemy wywołać funkcję ROW_NUMBER w klauzuli *SELECT*, przypisać alias do kolumny wynikowej i odwołać się do kolumny wynikowej w klauzuli *WHERE* zapytania zewnętrznego.

Ćwiczenie 4

Ćwiczenie to możemy traktować jako odwrotność przedstawionego w rozdziale zagadnienia zwrócenia pracownika i wszystkich jego podwładnych na wszystkich poziomach. Tutaj składnikiem zakotwiczającym jest zapytanie, które zwraca wiersz pracownika 9. Składnik rekurencyjny łączy wyrażenie CTE (nazwane C) – reprezentujące podwładnych/podrzędnych z poprzedniego poziomu – z tabelą *Employees* (nazwaną P) – reprezentującą przełożonych/nadrzędnych na następnym poziomie. W ten sposób każde wywołanie składnika rekurencyjnego zwraca przełożonego z następnego poziomu, aż na kolejnym poziomie nie zostanie znaleziony żaden przełożony (po dotarciu do naczelnego dyrektora).

Poniżej przedstawiono całe rozwiązanie zapytania:

```
WITH EmpsCTE AS
(
    SELECT empid, mgrid, firstname, lastname
    FROM HR.Employees
    WHERE empid = 9

    UNION ALL

    SELECT P.empid, P.mgrid, P.firstname, P.lastname
    FROM EmpsCTE AS C
    JOIN HR.Employees AS P
      ON C.mgrid = P.empid
)
```



```
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

Ćwiczenie 5-1

Ćwiczenie to jest krokiem wstępnym do następnego ćwiczenia. W ćwiczeniu musimy zdefiniować widok w oparciu o zapytanie, które łączy tabele *Orders* i *OrderDetails*, grupuje wiersze według identyfikatora pracownika i roku zamówienia i dla każdej grupy zwraca łączną ilość. Definicja widoku powinna mieć następującą postać:

```
USE TSQLV4;
IF OBJECT_ID('Sales.VEmpOrders') IS NOT NULL
    DROP VIEW Sales.VEmpOrders;
GO
CREATE VIEW Sales.VEmpOrders
AS
SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    SUM(qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON O.orderid = OD.orderid
GROUP BY
    empid,
    YEAR(orderdate);
GO
```

Ćwiczenie 5-2

W tym ćwiczeniu odpytujemy widok *VEmpOrders* oraz dla każdego pracownika i roku zamówienia zwracamy łączną ilość. By zrealizować to zadanie, napiszemy zapytanie do widoku *VEmpOrders* (alias *V1*), które z każdego wiersza zwraca identyfikator pracownika, rok zamówienia i ilość. Do listy *SELECT* możemy dołączyć podzapytanie do drugiej instancji widoku *VEmpOrders* (nazwijmy ją *V2*), która zwraca sumę wszystkich ilości z wierszy, dla których identyfikator pracownika jest równy jednemu z identyfikatorów w widoku *V1*, a rok zamówienia jest równy lub mniejszy niż jeden z wymienionych w widoku *V1*. Całe rozwiązanie zapytania prezentowane jest poniżej:

```
SELECT empid, orderyear, qty,
    (SELECT SUM(qty)
     FROM Sales.VEmpOrders AS V2
     WHERE V2.empid = V1.empid
     AND V2.orderyear <= V1.orderyear) AS runqty
FROM Sales.VEmpOrders AS V1
ORDER BY empid, orderyear;
```

W rozdziale 7 „Zaawansowane zagadnienia tworzenia zapytań” poznamy metody obliczania skumulowanych wartości łącznych przy użyciu funkcji okna.

Ćwiczenie 6-1

Do zrealizowania ćwiczenia wymagane jest zdefiniowanie funkcji nazwanej *TopProducts*, która akceptuje identyfikator dostawcy (*@supid*) i liczbę (*@n*) oraz zakłada się, że kod zwróci liczbę *@n* najdroższych produktów dostarczonych przez dostawcę o wprowadzonym identyfikatorze. Poniżej zaprezentowano, jak powinna wyglądać definicja funkcji:

```
USE TSQLV4;
IF OBJECT_ID('Production.TopProducts') IS NOT NULL
    DROP FUNCTION Production.TopProducts;
GO
CREATE FUNCTION Production.TopProducts
    (@supid AS INT, @n AS INT)
    RETURNS TABLE
AS
RETURN
    SELECT TOP (@n) productid, productname, unitprice
    FROM Production.Products
    WHERE supplierid = @supid
    ORDER BY unitprice DESC;
GO
```

Alternatywnie zamiast opcji *TOP* możemy stosować filtr *OFFSET-FETCH* – możemy więc zapytanie wewnętrzne funkcji zastąpić następującym kodem:

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE supplierid = @supid
ORDER BY unitprice DESC
OFFSET 0 ROWS FETCH FIRST @n ROWS ONLY;
```

Ćwiczenie 6-2

W tym ćwiczeniu trzeba napisać zapytanie do tabeli *Production.Suppliers* i użyć operatora *CROSS APPLY*, by do każdego dostawcy zastosować funkcję zdefiniowaną w poprzednim ćwiczeniu. Dla każdego dostawcy zapytanie ma zwrócić dwa najdroższe produkty. Poniżej zamieszczono całe rozwiązanie:

```
SELECT S.supplierid, S.companyname, P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
    CROSS APPLY Production.TopProducts(S.supplierid, 2) AS P;
```

ROZDZIAŁ 6

Operatory zbiorowe

Operatory zbiorowe to operatory, które jako argumentów używają dwóch zbiorów (lub wielozbiory) wyników zapytań wejściowych. Niektóre z nich usuwają duplikaty z wyników, a tym samym zwracają zbiór. Inne nie przeprowadzają takiej operacji, zatem zwracają wielozbiór. Przypomnijmy, że wielozbiór nie jest prawdziwym zbiorem, ponieważ może zawierać duplikaty. Język T-SQL obsługuje cztery operatory zbiorowe: *UNION*, *UNION ALL*, *INTERSECT* i *EXCEPT*. W tym rozdziale najpierw omówię ich postać ogólną i wymagania, a następnie szczegółowo przedstawię każdy operator.

Ogólna postać zapytania z operatorem zbiorowym jest następująca:

```
Wejściowe Zapytanie 1  
<operator_zbiorowy>  
Wejściowe Zapytanie 2  
[ORDER BY ...]
```

Operator zbiorowy porównuje całe wiersze należące do wyników dwóch zapytań wejściowych. To, czy wiersz zostanie zwrócony w wynikach działania operatora, zależy od rezultatu porównania i użytego operatora. Ponieważ zgodnie z definicją operator zbiorowy jest stosowany do dwóch zbiorów (lub wielozbiorów), a zbiór nie gwarantuje uporządkowania, żadne z używanych zapytań nie może zawierać klauzul *ORDER BY*. Pamiętamy, że zapytanie z klauzulą *ORDER BY* zapewnia kolejność prezentowania i z tego względu nie zwraca zbioru (ani wielozbioru), ale kursor. Choć zapytania wejściowe operatora nie mogą zawierać klauzul *ORDER BY*, opcjonalnie można dodać klauzulę *ORDER BY* do całego zapytania, które zostanie zastosowane do wyników działania operatora.

Pod względem logicznego przetwarzania każde zapytanie składowe może zawierać wszystkie fazy logicznego przetwarzania oprócz *ORDER BY* dla prezentacji. Operator zbiorowy działa na wynikach dwóch zapytań, a zewnętrzna klauzula *ORDER BY* (jeśli taka istnieje) jest stosowana do wyników operatora zbiorowego.

Obydwa zapytania zaangażowane w działanie operatora zbiorowego muszą generować wyniki o takiej samej liczbie kolumn, a odpowiadające sobie kolumny muszą mieć kompatybilne typy danych. Przez *kompatybilny typ danych* należy rozumieć, że dopuszczalna jest sytuacja niezgodności typu, o ile możliwe jest niejawne przekształcenie na zgodny typ danych.

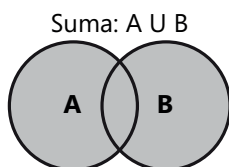
Nazwy kolumn w wynikach operatora zbiorowego są określone przez pierwsze zapytanie; z tego względu, jeśli zachodzi potrzeba przypisania aliasów do kolumn wyników, powinniśmy je określić w pierwszym zapytaniu.

Interesującym aspektem operatorów zbiorowych jest to, że podczas porównywania wierszy operator zbiorowy nie stosuje operatora równości, ale tak zwany *predykat różności* (*distinct predicate*). Traktuje on dwa znaczniki *NULL* jako wartości równe sobie. Ważność takiego działania wyjaśnię w dalszej części rozdziału.

Standard SQL zawiera dwie „odmiany” każdego operatora – *DISTINCT* (domyślnie) i *ALL*. W przypadku opcji *DISTINCT* eliminowane są duplikaty i zwracany jest zbiór. Opcja *ALL* nie próbuje usuwać duplikatów i dlatego zwraca wielozbiór. Implementacja standardu w systemie Microsoft SQL Server nie jest kompletna. W języku T-SQL wszystkie trzy operatory obsługują niejawnie wersję *DISTINCT*, natomiast tylko operator *UNION* obsługuje wersję *ALL*. Z punktu widzenia składni oznacza to, że T-SQL nie pozwala na jawne użycie klauzuli *DISTINCT*. Zamiast tego zakłada się jej użycie, jeśli nie została wprost wyspecyfikowana opcja *ALL*. W dalszej części, w podrozdziałach „Operator wielozbioru *INTERSECT ALL*” i „Operator wielozbioru *EXCEPT ALL*” przedstawione są alternatywne rozwiązania dla niedostępnych w T-SQL operatorów *INTERSECT ALL* i *EXCEPT ALL*.

Operator *UNION*

W teorii mnogości suma (*union*) dwóch zbiorów (nazwijmy je A i B) to zbiór zawierający wszystkie elementy obydwu zbiorów A i B. Inaczej mówiąc, jeśli element należy do któregośkolwiek zbioru wejściowego, należy też do zbioru wynikowego. Rysunek 6-1 przedstawia jako tzw. schemat Venna dwa zbiory wraz z graficznym przedstawieniem sumy dwóch zbiorów. Obszar zacieniowany reprezentuje wyniki działania operatora zbiorowego.



RYСУNEK 6-1 Złączenie dwóch zbiorów

W języku T-SQL operator *UNION* scala wyniki dwóch zapytań wejściowych. Jeśli wiersz występuje w którymkolwiek ze zbiorów wejściowych, wystąpi także w wynikach działania operatora *UNION*. Język T-SQL obsługuje dwie wersje operatora – *UNION ALL* i *UNION* (niejawna opcja *DISTINCT*).

Operator wielozbioru *UNION ALL*

Operator wielozbioru *UNION ALL* zwraca wszystkie wiersze, które występują w którymkolwiek z wielozbiorów wejściowych, powstałych w wyniku działania dwóch zapytań wejściowych, bez faktycznego porównywania wierszy i bez eliminowania

duplikatów. Zakładając, że *Query1* zwraca *m* wierszy, a *Query2* zwraca *n* wierszy, to wyrażenie *Query1 UNION ALL Query2* zwróci $m + n$ wierszy.

Na przykład w poniższym kodzie użyto operatora *UNION ALL* do scalenia lokalizacji pracowników z lokalizacjami klientów.

```
USE TSQL2012;
```

```
SELECT country, region, city FROM HR.Employees
UNION ALL
SELECT country, region, city FROM Sales.Customers;
```

W wynikach znalazło się 100 wierszy – 9 z tabeli *Employees* i 91 z tabeli *Customers* (wyniki pokazane w skróconej postaci):

country	region	city
-----	-----	-----
USA	WA	Seattle
USA	WA	Tacoma
USA	WA	Kirkland
USA	WA	Redmond
UK	NULL	London
UK	NULL	London
UK	NULL	London
...		
Finland	NULL	Oulu
Brazil	SP	Resende
USA	WA	Seattle
Finland	NULL	Helsinki
Poland	NULL	Warszawa

```
(100 row(s) affected)
```

Ponieważ operator *UNION ALL* nie eliminuje duplikatów, w wyniku otrzymujemy wielozbiór, a nie zbiór. W wynikach ten sam wiersz może pojawić się wielokrotnie, jak ma to miejsce w przypadku tego zapytania (*UK*, *NULL*, *London*).

Operator zbiorowy *UNION* z niejawną opcją *Distinct*

Operator zbiorowy *UNION* (ukryta opcja *DISTINCT*) łączy wyniki dwóch zapytań i eliminuje duplikaty. Zwróćmy uwagę, że jeśli wiersz pojawia się w obu zbiorach wejściowych, w wynikach wystąpi tylko raz; mówiąc inaczej, wyniki są zbiorem, a nie wielozbiorem.

Na przykład poniższy kod zwraca różne lokalizacje, które są lokalizacjami pracowników lub lokalizacjami klientów.

```
SELECT country, region, city FROM HR.Employees
UNION
SELECT country, region, city FROM Sales.Customers;
```

Różnica pomiędzy tym przykładem a poprzednim (z operatorem *UNION ALL*) polega na tym, że teraz operator usuwa duplikaty, a poprzednio duplikaty znalazły się

w wynikach. W rezultacie zapytanie zwraca 71 różnych wierszy, pokazanych poniżej w skróconej postaci:

country	region	city

Argentina	NULL	Buenos Aires
Austria	NULL	Graz
Austria	NULL	Salzburg
Belgium	NULL	Bruxelles
Belgium	NULL	Charleroi
...		
USA	WY	Lander
Venezuela	DF	Caracas
Venezuela	Lara	Barquisimeto
Venezuela	Nueva Esparta	I. de Margarita
Venezuela	Táchira	San Cristóbal

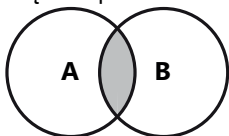
(71 row(s) affected)

Powstaje więc pytanie, kiedy używać operatora *UNION ALL*, a kiedy *UNION*? Jeśli możliwe są duplikaty w wynikach po scaleniu przez operator dwóch danych wejściowych i duplikaty te powinny być widoczne, używamy operatora *UNION ALL*. Jeśli istnieje możliwość wystąpienia duplikatów, ale w wynikach mają być tylko różne wiersze, używamy operatora *UNION*. Jeśli nie jest możliwe, by duplikaty pojawiły się po dodaniu dwóch zbiorów wejściowych, z punktu widzenia logiki działanie obu operatorów będzie takie samo. Jednakże w takiej sytuacji zaleca się stosowanie operatora *UNION ALL*, ponieważ opcja *ALL* usuwa dodatkowe operacje związane ze sprawdzaniem istnienia duplikatów przez system SQL Server.

Operator *INTERSECT*

W teorii mnogości część wspólna* (*intersection*) dwóch zbiorów (nazwijmy je A i B) jest zbiorem wszystkich elementów, które należą do zbioru A i należą do zbioru B. Rysunek 6-2 jest graficzną ilustracją części wspólnej dwóch zbiorów.

Część wspólna: $A \cap B$



RYСУNEK 6-2 Część wspólna dwóch zbiorów

W języku T-SQL operator zbiorowy *INTERSECT* zwraca część wspólną zbiorów wyników dwóch zapytań wejściowych, czyli zwraca tylko te wiersze, które występują

* Działanie to bywa również nazywane *przecięciem* albo *iloczynem* zbiorów (przyp. tłum.).

w obu zbiorach wejściowych. Po omówieniu operatora *INTERSECT* (z ukrytą opcją *DISTINCT*) przedstawione zostanie alternatywne rozwiązanie dla operatora wielozbioru *INTERSECT ALL*, który nie został zaimplementowany w języku T-SQL.

Operator *INTERSECT* (z ukrytą opcją *Distinct*)

Operator zbiorowy *INTERSECT* zwraca unikatowe wiersze, które występują w obu zbiorach. Inaczej mówiąc, wiersz jest zwracany pod warunkiem, że występuje co najmniej po razie w obydwu wielozbiorach wejściowych.

Dla przykładu poniższy kod zwraca różne lokalizacje, które są zarówno lokalizacjami pracowników, jak i lokalizacjami klientów.

```
SELECT country, region, city FROM HR.Employees
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```

Zapytanie to zwraca następujące dane wyjściowe:

country	region	city
UK	NULL	London
USA	WA	Kirkland
USA	WA	Seattle

Jak wspomniałem wcześniej, przy porównywaniu wierszy operator zbiorowy posługuje się predykatem rozróżnialności, co oznacza, że traktuje dwa znaczniki *NULL* jako równe. Zauważmy, że w wynikach mamy lokalizację (*UK, NULL, London*). Gdybyśmy zamiast operatora *INTERSECT* posłużyli się rozwiązaniem alternatywnym, takim jak złączenie wewnętrzne lub podzapytanie skorelowane, konieczne byłoby dodanie specjalnej obsługi znaczników *NULL* – na przykład, przyjmując alias *E* dla tabeli *Employees* i *C* dla *Customers*, moglibyśmy użyć predykatu *E.region = C.region OR (E.region IS NULL AND C.region IS NULL)*. Przy użyciu operatora *INTERSECT* rozwiązanie jest dużo prostsze – nie musimy jawnie porównywać odpowiadających sobie atrybutów i nie ma potrzeby dodawania specjalnej obsługi znaczników *NULL*.

Operator wielozbioru *INTERSECT ALL*

Ten podrozdział umieszczony został w książce jako materiał opcjonalny dla tych czytelników, którzy dobrze już poznali zagadnienia omawiane do tej pory. Standard SQL zawiera odmianę *ALL* operatora *INTERSECT*, jednak wersja ta nie została jeszcze zaimplementowana w języku T-SQL. Tym niemniej można utworzyć własne, alternatywne rozwiązanie w języku T-SQL.

Przypomnijmy znaczenie słowa kluczowego *ALL* dla operatora *UNION ALL*: operator zwraca wszystkie zduplikowane wiersze. Podobnie jest dla słowa kluczowego *ALL* w operatorze *INTERSECT ALL* – oznacza ono, że nie zostaną usunięte duplikaty części wspólnej. Operator *INTERSECT ALL* różni się od *UNION ALL* tym, że ten pierwszy

nie zwraca wszystkich duplikatów, a jedynie liczbę duplikatów wierszy równą mniejszej z liczb duplikatów w obu wielozbiorach. Mówiąc inaczej, można powiedzieć, że operator *INTERSECT ALL* nie tylko sprawdza istnienie wierszy po obu stronach – zajmuje się również liczbą wystąpień wierszy po każdej stronie – tak jakby dla każdego wiersza wyszukiwał zgodność w odniesieniu do jego wystąpienia. Jeśli w pierwszym wielozbiorze wejściowym istnieje x wystąpień wiersza R, a w drugim wielozbiorze jest takich wystąpień y , wiersz R w wynikach działania operatora wystąpi $\text{minimum}(x, y)$ razy, czyli tyle razy, ile wynosi mniejsza z liczb x i y . Na przykład, lokalizacja (UK, NULL, London) występuje cztery razy w tabeli *Employees* i sześć razy w tabeli *Customers*; z tego względu operator *INTERSECT ALL* pomiędzy lokalizacjami pracowników i lokalizacjami klientów powinien zwrócić cztery wystąpienia lokalizacji (UK, NULL, London), ponieważ z punktu widzenia logiki cztery wystąpienia są częścią wspólną.

Chociaż język T-SQL nie obsługuje wbudowanego operatora *INTERSECT ALL*, możemy stworzyć rozwiązanie, które daje takie same wyniki. W tym celu możemy użyć funkcji *ROW_NUMBER* do ponumerowania wystąpień każdego wiersza w każdym zapytaniu wejściowym. Aby to osiągnąć, specyfikujemy wszystkie uczestniczące atrybuty w klauzuli *PARTITION BY* i stosujemy wyrażenie (*SELECT <constant>*) w klauzuli *ORDER BY* funkcji, by wskazać, że porządek nie ma znaczenia.



Wskazówka Użycie klauzuli *ORDER BY* jest obowiązkowe w klasyfikujących funkcjach okna, takich jak *ROW_NUMBER*. Jeśli kolejność nie jest istotna, można użyć (*SELECT <constant>*) jako specyfikacji porządku. System SQL Server jest wystarczająco „inteligentny”, by zauważyć, że do wszystkich wierszy przypisana będzie ta sama stała, zatem nie jest konieczne faktycznie sortowanie danych i ponoszenie związanego z tym nakładu pracy.

Następnie stosujemy operator *INTERSECT* pomiędzy dwoma zapytaniami z funkcją *ROW_NUMBER*. Ponieważ wystąpienia każdego wiersza są policzone, wspólna część oprócz oryginalnych atrybutów określana jest na podstawie numerów wierszy. Na przykład w tabeli *Employees*, w której są cztery wystąpienia lokalizacji (UK, NULL, London), wystąpienia te zostaną ponumerowane od 1 do 4. W tabeli *Customers*, która ma sześć wystąpień lokalizacji (UK, NULL, London), wystąpienia te są ponumerowane od 1 do 6. Częścią wspólną tych dwóch będą wystąpienia od 1 do 4.

Poniższy kod ilustruje całe rozwiązanie.

```
SELECT
  ROW_NUMBER()
    OVER(PARTITION BY country, region, city
         ORDER BY (SELECT 0)) AS rownum,
  country, region, city
FROM HR.Employees
INTERSECT
SELECT
  ROW_NUMBER()
```



```

        OVER(PARTITION BY country, region, city
              ORDER BY (SELECT 0)),
    country, region, city
FROM Sales.Customers;

```

Uruchomienie kodu generuje następujące wyniki:

rownum	country	region	city

1	UK	NULL	London
1	USA	WA	Kirkland
1	USA	WA	Seattle
2	UK	NULL	London
3	UK	NULL	London
4	UK	NULL	London

Rzecz jasna, standardowy operator *INTERSECT ALL* nie został pomyślany, by zwracać numery wierszy; numery są używane jako element pomocniczy rozwiązania. Aby usunąć numery z danych wyjściowych, możemy zdefiniować wyrażenie tablicowe, w oparciu o to zapytanie i wybrać z niego jedynie te atrybuty, które mają być zwrócone. Oto pełny kod rozwiązania.

```

WITH INTERSECT_ALL
AS
(
    SELECT
        ROW_NUMBER()
        OVER(PARTITION BY country, region, city
              ORDER BY (SELECT 0)) AS rownum,
        country, region, city
    FROM HR.Employees

    INTERSECT

    SELECT
        ROW_NUMBER()
        OVER(PARTITION BY country, region, city
              ORDER BY (SELECT 0)),
        country, region, city
    FROM Sales.Customers
)
SELECT country, region, city
FROM INTERSECT_ALL;

```

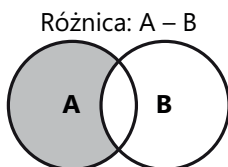
Kod ten zwraca poniższe wyniki, zgodne z oczekiwanymi dla standardowego operatora *INTERSECT ALL*.

country	region	city

UK	NULL	London
USA	WA	Kirkland
USA	WA	Seattle
UK	NULL	London
UK	NULL	London
UK	NULL	London

Operator *EXCEPT*

W teorii mnogości różnica zbiorów A i B ($A - B$) jest zbiorem tych elementów, które należą do A i nie należą do B. Różnicę zbiorów $A - B$ możemy traktować jak zbiór A bez tych elementów, które należą również do zbioru B. Na rysunku 6-3 przedstawiono graficznie różnicę zbiorów $A - B$.



RYСУNEK 6-3 Różnica zbiorów

Operator zbiorowy *EXCEPT* (z opcją *Distinct*)

W języku T-SQL różnica zbiorów została zaimplementowana za pomocą operatora zbiorowego *EXCEPT*. Operator *EXCEPT* działa na zbiorach wyników dwóch zapytań wejściowych i zwraca wiersze, które występują w pierwszym zbiorze wejściowym, ale nie występują w drugim. Mówiąc inaczej, wiersz jest zwracany, o ile występuje co najmniej raz w pierwszym wielozbiorze wyjściowym i nie występuje w drugim. Zwróćmy uwagę, że, inaczej niż w przypadku dwóch pozostałych operatorów, operator *EXCEPT* jest nieprzemienne; innymi słowy kolejność, w jakiej zostaną podane wejściowe wielozbiory operatora *EXCEPT*, ma znaczenie.

Dla przykładu poniższy kod zwraca różne lokalizacje, które są lokalizacjami pracowników, ale nie są lokalizacjami klientów.

```
SELECT country, region, city FROM HR.Employees
EXCEPT
SELECT country, region, city FROM Sales.Customers;
```

Zapytanie to zwraca dwie lokalizacje:

country	region	city
USA	WA	Redmond
USA	WA	Tacoma

Poniższe zapytanie zwraca różne lokalizacje, które są lokalizacjami klientów, ale nie są lokalizacjami pracowników.

```
SELECT country, region, city FROM Sales.Customers
EXCEPT
SELECT country, region, city FROM HR.Employees;
```

Zapytanie to zwraca 66 lokalizacji (pokazanych w skróconej postaci):

country	region	city
Argentina	NULL	Buenos Aires
Austria	NULL	Graz
Austria	NULL	Salzburg
Belgium	NULL	Bruxelles
Belgium	NULL	Charleroi
...		
USA	WY	Lander
Venezuela	DF	Caracas
Venezuela	Lara	Barquisimeto
Venezuela	Nueva Esparta	I. de Margarita
Venezuela	Táchira	San Cristóbal

(66 row(s) affected)

Oczywiście istnieją rozwiązania alternatywne dla operatora *EXCEPT*. Jednym z nich jest złączenie zewnętrzne filtrujące tylko wiersze zewnętrzne, czyli takie, które występują po jednej stronie, ale które nie występują po drugiej stronie. Innym rozwiązaniem może być użycie predykatu *NOT EXISTS*. Jeśli jednak chcemy, by dwa znaczniki *NULL* były traktowane jako równoważne, operator zbiorowy domyślnie zapewnia takie działanie i nie trzeba opracowywać dodatkowej logiki, jak w przypadku rozwiązań alternatywnych.

Operator wielozbioru *EXCEPT ALL*

Zdefiniowany w standardzie SQL operator *EXCEPT ALL* jest podobny do operatora *EXCEPT*, ale uwzględnia dodatkowo liczbę wystąpień każdego wiersza. Zakładając, że wiersz *R* występuje *x* razy w pierwszym wielozbiorze i *y* razy w drugim oraz że $x > y$, w wyrażeniu *Query1 EXCEPT ALL Query2* wiersz *R* pojawi się $x - y$ razy. Mówiąc inaczej, operator *EXCEPT ALL* zwraca tylko te wystąpienia wiersza z pierwszego wielozbioru, dla których nie ma odpowiadających im wystąpień w drugim wielozbiorze.

Język T-SQL nie zawiera wbudowanego operatora *EXCEPT ALL*, jednak możemy udostępnić rozwiązanie bardzo podobne do przedstawionego poprzednio dla operatora *INTERSECT ALL*. Mianowicie dodajemy funkcję *ROW_NUMBER* do każdego zapytania wejściowego, by ponumerować wystąpienia każdego wiersza, a następnie używamy operatora *EXCEPT* pomiędzy tymi zapytaniami wejściowymi. Zwrócone zostaną tylko te wystąpienia, dla których nie zostaną znalezione zgodne wiersze.

Poniższy przykład ilustruje, jak użyć operatora *EXCEPT ALL*, by zwrócić wystąpienia lokalizacji pracowników, które nie mają odpowiednich wystąpień w lokalizacjach klientów.

```
WITH EXCEPT_ALL
AS
(
  SELECT
    ROW_NUMBER()
```

```

        OVER(PARTITION BY country, region, city
              ORDER BY (SELECT 0)) AS rownum,
country, region, city
FROM HR.Employees

EXCEPT

SELECT
  ROW_NUMBER()
    OVER(PARTITION BY country, region, city
          ORDER BY (SELECT 0)),
country, region, city
FROM Sales.Customers
)
SELECT country, region, city
FROM EXCEPT_ALL;

```

Zapytanie to zwraca następujące dane wyjściowe:

country	region	city
-----	-----	-----
USA	WA	Redmond
USA	WA	Tacoma
USA	WA	Seattle

Pierwszeństwo

Standard SQL definiuje pierwszeństwo (kolejność wykonywania działań) dla operatorów zbiorowych. Operator *INTERSECT* poprzedza operatory *UNION* i *EXCEPT*, a operatory *UNION* i *EXCEPT* w kontekście pierwszeństwa są równoważne. W przypadku zapytania zawierającego wiele operatorów zbiorowych najpierw przetwarzane są operatory *INTERSECT*, a następnie przetwarzane są operatory o tym samym priorytecie w oparciu o kolejność występowania.

Rozważmy następujący przykład.

```

SELECT country, region, city FROM Production.Suppliers
EXCEPT
SELECT country, region, city FROM HR.Employees
INTERSECT
SELECT country, region, city FROM Sales.Customers;

```

Ponieważ operator *INTERSECT* ma pierwszeństwo przed *EXCEPT*, jest on przetwarzany najpierw, nawet jeśli występuje jako drugi. Z tego względu zapytanie oznacza „lokalizacje, które są lokalizacjami dostawców, ale nie są lokalizacjami pracowników i lokalizacjami klientów”.

Zapytanie zwraca następujące dane wyjściowe:

country	region	city
-----	-----	-----
Australia	NSW	Sydney
Australia	Victoria	Melbourne
Brazil	NULL	Sao Paulo
Canada	Québec	Montréal
Canada	Québec	Ste-Hyacinthe
Denmark	NULL	Lyngby
Finland	NULL	Lappeenranta
France	NULL	Annecy
France	NULL	Montceau
France	NULL	Paris
Germany	NULL	Berlin
Germany	NULL	Cuxhaven
Germany	NULL	Frankfurt
Italy	NULL	Ravenna
Italy	NULL	Salerno
Japan	NULL	Osaka
Japan	NULL	Tokyo
Netherlands	NULL	Zaandam
Norway	NULL	Sandvika
Singapore	NULL	Singapore
Spain	Asturias	Oviedo
Sweden	NULL	Göteborg
Sweden	NULL	Stockholm
UK	NULL	Manchester
USA	LA	New Orleans
USA	MA	Boston
USA	MI	Ann Arbor
USA	OR	Bend

(28 row(s) affected)

Aby zmienić kolejność przetwarzania operatorów zbiorowych, możemy użyć nawiasów, ponieważ nawiasy mają najwyższy priorytet. Ponadto użycie nawiasów poprawia czytelność, a tym samym zmniejsza możliwość popełnienia błędów. Jeśli na przykład chcemy zwrócić „(lokalizacje, które są lokalizacjami dostawców, ale nie są lokalizacjami pracowników) i które są również lokalizacjami klientów”, użyjemy poniższego kodu:

```
(SELECT country, region, city FROM Production.Suppliers
EXCEPT
SELECT country, region, city FROM HR.Employees)
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```

Zapytanie zwraca następujące wyniki:

country	region	city
-----	-----	-----
Canada	Québec	Montréal
France	NULL	Paris
Germany	NULL	Berlin

Omijanie nieobsługiwanych faz logicznych

Niniejszy podrozdział omawia zaawansowane zagadnienia i został udostępniony jako materiał opcjonalny. Poszczególne zapytania używane jako dane wejściowe dla operatora zbiorowego obsługują wszystkie fazy logicznego przetwarzania zapytania (takie jak operatory tabel, *WHERE*, *GROUP BY* czy *HAVING*) z wyjątkiem *ORDER BY*. Jednakże tylko faza *ORDER BY* jest dozwolona do działania na wynikach operatora.

Co zrobić, jeśli do wyników operatora trzeba zastosować inne fazy logiczne (oprócz *ORDER BY*)? Działanie takie nie jest wprost obsługiwane jako część zapytania, które wykorzystuje operator, ale przy użyciu wyrażeń tablicowych możemy dość prosto ominąć to ograniczenie. Definiujemy wyrażenie tablicowe w oparciu o zapytanie z operatorem zbiorowym i stosujemy dowolną fazę logicznego przetwarzania zapytania, która jest nam potrzebna, w zewnętrznym zapytaniu do wyrażenia tablicowego. Na przykład poniższe zapytanie zwraca liczbę różnych lokalizacji, które w każdym kraju są albo lokalizacją pracownika, albo klienta.

```
SELECT country, COUNT(*) AS numlocations
FROM (SELECT country, region, city FROM HR.Employees
      UNION
      SELECT country, region, city FROM Sales.Customers) AS U
GROUP BY country;
```

Zapytanie to zwraca następujące wyniki:

country	numlocations
Argentina	1
Austria	2
Belgium	2
Brazil	4
Canada	3
Denmark	2
Finland	2
France	9
Germany	11
Ireland	1
Italy	3
Mexico	1
Norway	1
Poland	1
Portugal	1
Spain	3
Sweden	2
Switzerland	2
UK	2
USA	14
Venezuela	4

(21 row(s) affected)

Zapytanie to pokazuje, jak można zastosować fazę logicznego przetwarzania zapytania *GROUP BY* do wyników operatora *UNION*; w podobny sposób moglibyśmy w zewnętrznym zapytaniu zastosować dowolną fazę logicznego przetwarzania.

Brak możliwości określenia klauzuli *ORDER BY* w poszczególnych zapytaniach używanych w operatorze zbiorowym może również powodować problemy logiczne. Co zrobić, jeśli trzeba ograniczyć liczbę wierszy w tych zapytaniach przy użyciu opcji *TOP* czy *OFFSET-FETCH*? Ponownie problem ten możemy rozwiązać za pomocą wyrażeń tablicowych. Pamiętajmy, że klauzula *ORDER BY* jest dopuszczona w zapytaniu z opcją *TOP* lub *OFFSET-FETCH*, gdy zapytanie jest używane do definiowania wyrażenia tablicowego. W takim przypadku klauzula *ORDER BY* służy tylko jako część specyfikacji filtru, a nie jako narzędzie prezentacji.

Jeśli więc zapytanie z opcją *TOP* lub *OFFSET-FETCH* powinno uczestniczyć w operatorze zbiorowym, po prostu definiujemy wyrażenie tablicowe i na wejściu operatora umieszczamy zapytanie do tego wyrażenia tablicowego. Na przykład poniższy kod używa opcji *TOP*, by zwrócić dwa najnowsze zamówienia pracowników, których identyfikatory to 3 lub 5.

```
SELECT empid,orderid,orderdate
FROM (SELECT TOP (2) empid,orderid,orderdate
      FROM Sales.Orders
      WHERE empid = 3
      ORDER BY orderdate DESC,orderid DESC) AS D1

UNION ALL

SELECT empid,orderid,orderdate
FROM (SELECT TOP (2) empid,orderid,orderdate
      FROM Sales.Orders
      WHERE empid = 5
      ORDER BY orderdate DESC,orderid DESC) AS D2;
```

Zapytanie to zwraca następujące dane wyjściowe:

empid	orderid	orderdate
3	11063	2016-04-30 00:00:00.000
3	11057	2016-04-29 00:00:00.000
5	11043	2016-04-22 00:00:00.000
5	10954	2016-03-17 00:00:00.000

Podsumowanie

W tym rozdziale omówiłem operatory zbiorowe *UNION*, *UNION ALL*, *INTERSECT* i *EXCEPT*. Standard SQL obsługuje również dwie dodatkowe wersje *INTERSECT ALL* oraz *EXCEPT ALL*. Są one niedostępne w języku T-SQL, ale zaproponowałem alternatywne rozwiązania dla brakujących operatorów, wykorzystujące funkcję *ROW_NUMBER* i wyrażenia tablicowe. Na koniec omówiłem pierwszeństwo przetwarzania operatorów zbiorowych i wyjaśniłem, jak przy użyciu wyrażen tablicowych omijać nieobsługiwane fazy logicznego przetwarzania zapytania.

Ćwiczenia

Ten podrozdział zawiera ćwiczenia, które ułatwią lepsze przyswojenie tematyki opisanej w rozdziale. Przy wszystkich ćwiczeniach w tym rozdziale wymagane jest połączenie z bazą danych *TSQLV4*.

Ćwiczenie 1

Wyjaśnij różnice pomiędzy operatorami *UNION ALL* i *UNION*. W jakich sytuacjach są one równoważne? Którego z nich należy użyć w takiej sytuacji?

Ćwiczenie 2

Napisać zapytanie generujące wirtualną tabelę pomocniczą 10 liczb z zakresu 1 do 10 bez użycia konstrukcji pętli. W wynikach nie trzeba zapewniać uporządkowania wierszy.

- Wykorzystywane tabele: brak
- Oczekiwane dane wyjściowe:

```
n
-----
1
2
3
4
5
6
7
8
9
10

(10 row(s) affected)
```


Ćwiczenie 3

Napisać zapytanie zwracające pary klient/pracownik, którzy wykazali się aktywnością dotyczącą zamówień w styczniu 2016, przy braku aktywności w lutym 2016.

- Wykorzystywane tabele: *Sales.Orders*
- Oczekiwane dane wyjściowe (skrócone):

custid	empid
-----	-----
1	1
3	3
5	8
5	9
6	9
7	6
9	1
12	2
16	7
17	1
20	7
24	8
25	1
26	3
...	
84	3
84	4
88	7
89	4

(50 row(s) affected)

Ćwiczenie 4

Napisać zapytanie zwracające pary klient i pracownik, którzy wykazali się aktywnością dotyczącą zamówień w styczniu i lutym 2016.

- Wykorzystywane tabele: *Sales.Orders*
- Oczekiwane dane wyjściowe:

custid	empid
-----	-----
20	3
39	9
46	5
67	1
71	4

(5 row(s) affected)

Ćwiczenie 5

Napisać zapytanie zwracające pary klient i pracownik, którzy wykazali się aktywnością dotyczącą zamówień w styczniu i lutym 2016, ale nie w roku 2015.

- Wykorzystywane tabele: *Sales.Orders*
- Oczekiwane dane wyjściowe:

custid	empid
67	1
46	5

(2 row(s) affected)

Ćwiczenie 6 (zaawansowane ćwiczenie opcjonalne)

Mamy następujące zapytanie:

```
SELECT country, region, city
FROM HR.Employees
```

```
UNION ALL
```

```
SELECT country, region, city
FROM Production.Suppliers;
```

Do zapytania trzeba dodać taką logikę, by zagwarantować, że wiersze z tabeli *Employees* będą zwracane w wynikach przed wierszami z tabeli *Suppliers*. Ponadto wewnątrz każdego segmentu wiersze powinny być posortowane według kraju, regionu i miasta.

- Wykorzystywane tabele: *HR.Employees* i *Production.Suppliers*
- Oczekiwane dane wyjściowe:

country	region	city
UK	NULL	London
UK	NULL	London
UK	NULL	London
UK	NULL	London
USA	WA	Kirkland
USA	WA	Redmond
USA	WA	Seattle
USA	WA	Seattle
USA	WA	Tacoma
Australia	NSW	Sydney
Australia	Victoria	Melbourne
Brazil	NULL	Sao Paulo
Canada	Québec	Montréal
Canada	Québec	Ste-Hyacinthe
Denmark	NULL	Lyngby
Finland	NULL	Lappeenranta

France	NULL	Annecy
France	NULL	Montceau
France	NULL	Paris
Germany	NULL	Berlin
Germany	NULL	Cuxhaven
Germany	NULL	Frankfurt
Italy	NULL	Ravenna
Italy	NULL	Salerno
Japan	NULL	Osaka
Japan	NULL	Tokyo
Netherlands	NULL	Zaandam
Norway	NULL	Sandvika
Singapore	NULL	Singapore
Spain	Asturias	Oviedo
Sweden	NULL	Göteborg
Sweden	NULL	Stockholm
UK	NULL	London
UK	NULL	Manchester
USA	LA	New Orleans
USA	MA	Boston
USA	MI	Ann Arbor
USA	OR	Bend

(38 row(s) affected)

Rozwiązania

Podrozdział ten zawiera rozwiązania ćwiczeń wraz z odpowiednimi wyjaśnieniami.

Ćwiczenie 1

Operator *UNION ALL* scala wyniki dwóch zapytań wejściowych i nie usuwa z nich duplikatów. Operator *UNION* (z niejwaną klauzulą *DISTINCT*) również scala wyniki dwóch zapytań wejściowych, ale eliminuje duplikaty wierszy.

Operatory mają odmiennie znaczenie, jeśli wyniki potencjalnie mogą zawierać duplikaty. W sytuacji, gdy duplikaty nie mogą wystąpić, czyli przy scalaniu zbiorów rozłącznych, działanie ich jest identyczne (przykładem mogą być zdarzenia sprzedaży w różnych latach).

Gdy operatory mają to samo znaczenie, zalecane jest stosowanie *UNION ALL*. Pozwoli to uniknąć dodatkowego i niepotrzebnego obciążenia zadaniem eliminowania duplikatów, gdy wiadomo, że i tak nie istnieją.

Ćwiczenie 2

Język T-SQL obsługuje polecenie *SELECT* zawierające stałe bez wskazywania żadnej klauzuli *FROM*. Takie polecenie *SELECT* zwraca tabelę w postaci pojedynczego

wiersza. Na przykład poniższe polecenie zwraca wiersz zawierający pojedynczą kolumnę nazwaną *n* z wartością 1.

```
SELECT 1 AS n;
```

Dane wyjściowe polecenia to:

```
n
-----
1
```

(1 row(s) affected)

Przy użyciu operatora *UNION ALL* możemy scalać zbiory wyników wielu instrukcji, takich jak właśnie przytoczona, każda zwracająca wiersz z inną liczbą w zakresie od 1 do 10, jak w poniższym przykładzie:

```
SELECT 1 AS n
UNION ALL SELECT 2
UNION ALL SELECT 3
UNION ALL SELECT 4
UNION ALL SELECT 5
UNION ALL SELECT 6
UNION ALL SELECT 7
UNION ALL SELECT 8
UNION ALL SELECT 9
UNION ALL SELECT 10;
```



Wskazówka SQL Server obsługuje rozszerzoną klauzulę *VALUES*, którą poznamy bliżej przy omawianiu polecenia *INSERT*. Klauzula *VALUES* może prezentować nie tylko pojedynczy wiersz – może reprezentować wiele wierszy. Ponadto stosowanie klauzuli *VALUES* nie jest ograniczone tylko do polecenia *INSERT* – klauzula ta może być używana do definiowania wyrażeń tablicowych z wierszami bazującymi na stałych. Poniższy przykład pokazuje, jak zamiast stosowania operatora *UNION ALL* użyć klauzuli *VALUES* do rozwiązania tego ćwiczenia.

```
SELECT n
FROM (VALUES(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)) AS Nums(n);
```

Szczegółowe informacje na temat klauzuli *VALUES* i konstruktorów wartości wierszy zawarłem w rozdziale 8 „Modyfikowanie danych” jako część opisu polecenia *INSERT*.

Ćwiczenie 3

Ćwiczenie to możemy rozwiązać przy użyciu operatora zbiorowego *EXCEPT*. Lewe zapytanie wejściowe zwraca pary klient i pracownik, którzy mieli zamówienia w styczniu 2016. Z prawej strony jest zapytanie zwracające pary klient i pracownik, którzy mieli zamówienia w lutym 2016. Poniżej przedstawiono całe rozwiązanie:

```

USE TSQL2012;

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160101' AND orderdate < '20160201'

EXCEPT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160201' AND orderdate < '20160301';

```

Ćwiczenie 4

W ćwiczeniu 3 potrzebne były pary klient i pracownik, którzy mieli zamówienia w jednym okresie, a nie mieli ich w innym, natomiast to ćwiczenie wyszukuje pary klient i pracownik, którzy mieli zamówienia w obu okresach. Tak więc tym razem, zamiast zastosowania operatora *EXCEPT*, trzeba użyć operatora *INTERSECT*, jak w poniższym kodzie:

```

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160101' AND orderdate < '20160201'

INTERSECT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160201' AND orderdate < '20160301';

```

Ćwiczenie 5

Rozwiązanie ćwiczenia wymaga połączenia operatorów zbiorowych. Aby zwrócić pary klient i pracownik, którzy mieli zamówienia zarówno w styczniu, jak i lutym 2016, trzeba użyć operatora *INTERSECT*, jak w ćwiczeniu 3. Aby wykluczyć z wyników pary klient i pracownik, którzy mieli zamówienia w roku 2015, trzeba użyć operatora *EXCEPT* pomiędzy wynikami a trzecim zapytaniem. Rozwiązanie zadania ma następującą postać:

```

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160101' AND orderdate < '20160201'

INTERSECT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160201' AND orderdate < '20160301'

EXCEPT

```

```
SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20150101' AND orderdate < '20160101';
```

Przypomnijmy, że operator *INTERSECT* przetwarzany jest przed operatorem *EXCEPT*. W tym przypadku domyślne pierwszeństwo jest właściwe dla postawionego zadania, tak więc nie musimy niczego korygować za pomocą nawiasów – możemy jednak dodać je dla większej przejrzystości kodu, jak poniżej:

```
(SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160101' AND orderdate < '20160201'

INTERSECT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20160201' AND orderdate < '20160301')

EXCEPT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20150101' AND orderdate < '20160101';
```

Ćwiczenie 6

Problem przedstawiony w ćwiczeniu polega na tym, że poszczególne zapytania nie mogą zawierać klauzul *ORDER BY*. Możemy rozwiązać ten problem dodając do każdego zapytania, na którym działa operator, kolumnę wyników opartą na stałej (nazwijmy ją *sortcol*). W zapytaniu do tabeli *Employees* wskażemy mniejszą stałą niż stała wyspecyfikowana w zapytaniu do tabeli *Suppliers*. Następnie zdefiniujemy wyrażenie tablicowe w oparciu o zapytanie z operatorem i w klauzuli *ORDER BY* zapytania zewnętrznego specyfikujemy *sortcol* jako pierwszą kolumnę sortowania, po której następują kolumny kraju, regionu i miasta (*country*, *region*, *city*). Poniżej przedstawiono całe rozwiązanie:

```
SELECT country, region, city
FROM (SELECT 1 AS sortcol, country, region, city
      FROM HR.Employees

      UNION ALL

      SELECT 2, country, region, city
      FROM Production.Suppliers) AS D
ORDER BY sortcol, country, region, city;
```

ROZDZIAŁ 7

Zaawansowane zagadnienia tworzenia zapytań

Rozdział ten zawiera tematykę wykraczającą poza podstawy i można go traktować jako lekturę opcjonalną. Zaczniemy od omówienia funkcji okna, które pozwalają wykonywać analizy danych w sposób elastyczny i efektywny. Następnie przejdę do omówienia metod przedstawiania i odwrotnego przedstawiania danych (*pivoting/unpivoting*). Przedstawianie danych oznacza zamianę wierszy na kolumny, natomiast odwrotne przedstawianie danych oznacza zamianę kolumn na wiersze – w ogólności przypomina to tabele przestawne znane z programu Excel. Rozdział kończy się omówieniem *zbiorów grupujących*, czyli zbiorów atrybutów, według których dane mogą być grupowane. W rozdziale przedstawiono metody stosowania wielu zbiorów grupujących w tym samym zapytaniu.

Cechą wspólną funkcjonalności omawianych w tym rozdziale, oprócz tego, że należą do bardziej zaawansowanych, jest to, że są one głównie wykorzystywane w zastosowaniach analitycznych. Jeśli ktoś potrzebuje bardziej wyczerpującego omówienia tych funkcjonalności, można je znaleźć w moich książkach *T-SQL Querying* (Microsoft Press, 2015) oraz *Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions* (Microsoft Press, 2012)*.

Funkcje okna

Funkcja okna to taka funkcja, która dla każdego wiersza wylicza skalarny wynik w oparciu o obliczenia wykonywane na podzbiórze wierszy z zapytania wyjściowego. Podzbiór wierszy nazywany jest *oknem* i bazuje na deskrytorze okna odnoszącym się do bieżącego wiersza. Składnia funkcji okna korzysta z klauzuli *OVER*, w której podawana jest specyfikacja okna.

Jeśli brzmi to zbyt technicznie i niezrozumiale, pomyślmy o przypadku przeprowadzenia obliczeń na pewnym zbiorze i zwrócenia pojedynczej wartości. Klasycznym przykładem są funkcje agregujące, takie jak *SUM*, *COUNT* czy *AVG*, ale także inne, jak na przykład funkcje klasyfikujące. Znamy już dwie metody wykonywania takich obliczeń – jedną przy użyciu grupowanych zapytań i drugą przy użyciu podzapytań. Obie

* Wydania polskie: *Zapytania w języku T-SQL* (APN Promise, 2015) oraz *Optymalizacja zapytań T-SQL przy użyciu funkcji okna* (APN Promise, 2012).

metody cechują pewne mankamenty, które można elegancko rozwiązywać za pomocą funkcji okien.

UWAGA Jeśli przedstawione wyjaśnienia dotyczące projektu funkcji okna wydają się zbyt przytłaczające, Czytelnik może przeskoczyć do kilku pierwszych przykładów, dzięki którym wszystko powinno się wyjaśnić. Po przejrzaniu tych przykładów warto ponownie przeczytać kilka kolejnych akapitów.

Grupowane zapytania zapewniają pewne spostrzeżenia dzięki agregowaniu danych, ale jednocześnie powodują utratę informacji szczegółowych. Po zgrupowaniu wierszy wszystkie obliczenia zapytania muszą być wykonywane w kontekście zdefiniowanych grup. Często jednak zachodzi potrzeba wykonywania obliczeń, które korzystają zarówno z elementów szczegółowych, jak i wyników obliczeń na zbiorach, takich jak sumowanie. Funkcje okna nie mają takich ograniczeń. Funkcja okna jest przetwarzana dla każdego szczegółowego wiersza i jest stosowana dla podzbioru wierszy wydobytego ze zbioru wyników zapytania. Wynikiem funkcji okna jest wartość skalarna, która jest dołączana jako dodatkowa kolumna do wyników zapytania. Innymi słowy, w przeciwieństwie do grupowania, funkcje okna nie powodują utraty dostępu do szczegółów. Dla przykładu przyjmijmy, że chcemy odczytać wartości zamówień i zwrócić je wraz z procentowym udziałem tego zamówienia w całkowitych zakupach danego klienta. Jeśli wykonamy grupowanie według klientów, otrzymamy tylko sumaryczną wartość dla danego klienta. Przy użyciu funkcji okna możemy zwrócić tę wartość sumaryczną obok szczegółowej wartości zamówienia, a nawet wyliczyć jego udział procentowy. Kod pozwalający uzyskać taki skutek zademonstruję w dalszej części rozdziału.

Jeśli chodzi o podzapytania, można je wykorzystać do zastosowania agregacji skalarynych wobec zbioru danych, ale ich punkt wyjścia to „świeży” widok danych, a nie zbiór wyników zewnętrznego zapytania. Załóżmy, że zapytanie zewnętrzne zawiera operatory tabel, filtry i inne elementy; nie mają one wpływu na to, co podzapytanie widzi jako swój punkt wyjścia. Jeśli chcemy, aby podzapytanie użyło zestawu wyników zewnętrznego zapytania jako punktu wyjścia, musimy powtórzyć w nim całą logikę zapytania zewnętrznego. Funkcje okna przeciwnie, stosowane są do podzbiorów wierszy wynikowych zapytania zewnętrznego. Tym samym, cokolwiek zmienimy w zewnętrznym zapytaniu, zostanie to automatycznie odzwierciedlone we wszystkich funkcjach okna użytych w zapytaniu. Co więcej, możemy jeszcze bardziej ograniczyć okno.

Inna korzyść funkcji okien to możliwość zdefiniowania kolejności (jeśli ma to zastosowanie) jako fragmentu specyfikacji obliczeń, bez sprzeczności z aspektami relacyjnymi zbioru wyników. Tak więc kolejność jest definiowana na potrzeby obliczeń i nie jest mieszana z kolejnością prezentacji. Specyfikacja uporządkowania dla funkcji okna (jeśli ma zastosowanie) jest czymś innym, niż specyfikacja kolejności dla prezentacji. Jeśli nie dołączamy klauzuli prezentacji *ORDER BY*, nie mamy żadnej pewności,

że wyniki zostaną zwrócone w pewnym konkretnym porządku. Jeśli decydujemy się wymusić pewien porządek prezentowania, wynikowa kolejność może być inna niż kolejność dla funkcji okna.

Poniższy przykład pokazuje zapytanie do widoku *Sales.EmpOrders* w bazie danych TSQLV4, która dla każdego pracownika i miesiąca stosuje agregującą funkcję okna do obliczenia kumulowanych wartości całkowitych.

```
USE TSQLV4;

SELECT empid, ordermonth, val,
       SUM(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                           AND CURRENT ROW) AS runval
FROM Sales.EmpOrders;
```

Poniżej pokazano wyniki działania tego zapytania (w skróconej postaci):

empid	ordermonth	val	runval
1	2014-07-01	1614.88	1614.88
1	2014-08-01	5555.90	7170.78
1	2014-09-01	6651.00	13821.78
1	2014-10-01	3933.18	17754.96
1	2014-11-01	9562.65	27317.61
...			
2	2014-07-01	1176.00	1176.00
2	2014-08-01	1814.00	2990.00
2	2014-09-01	2950.80	5940.80
2	2014-10-01	5164.00	11104.80
2	2014-11-01	4614.58	15719.38
...			

(192 row(s) affected)

Definicja okna w klauzuli *OVER* składa się z trzech głównych części: definiowania partycji, kolejności i ramek. Pusta klauzula *OVER()* reprezentuje cały zbiór wyników zapytania wyjściowego. Cokolwiek zostanie dodane do specyfikacji, zasadniczo będzie stanowić ograniczenie okna.

Klauzula partycjonowania okna (*PARTITION BY*) zawęży okno do podzbioru wyników zapytania bazowego złożonego z wierszy, które w kolumnach partycji mają te same wartości, co w bieżącym wierszu. W przykładzie okno zostało podzielone według kolumny *empid*. Przenalizujmy dla przykładu wiersz, w którym wartość w kolumnie *empid* to 1. Okno prezentowane funkcji dla tego wiersza będzie zawierało tylko taki podzbiór wierszy, w których wartość *empid* to 1.

Klauzula uporządkowania okna (*ORDER BY*) definiuje kolejność w oknie, ale nie należy mylić jej z kolejnością prezentowania wyników. W przypadku funkcji klasyfikujących kolejność w oknie jest tym, co nadaje sens tej klasyfikacji. W tym przykładzie kolejność okna bazuje na kolumnie *ordermonth*.

Klauzula ramki okna (*ROWS BETWEEN* <górne ograniczenie> *AND* <dolne ograniczenie>) filtruje ramkę, czyli podzbiór wierszy z partycji okna, zawężając go do podanych ograniczeń. W tym przykładzie ramka zawiera się pomiędzy początkiem partycji (*UNBOUNDED PRECEDING*) i bieżącym wierszem (*CURRENT ROW*). Oprócz definiowania ramki okna poprzez wybór wierszy (*ROWS*) istnieje jeszcze inna konstrukcja nazwana *RANGE*, jednak w języku T-SQL została ona zaimplementowana w bardzo ograniczonej postaci.

Podsumowując to wszystko, funkcja pokazana w przykładzie wylicza skumulowane wartości łączne dla każdego pracownika i miesiąca.

Ponieważ punkt początkowy funkcji okna to zbiór wyników zapytania bazowego, a zbiór wyników zapytania bazowego jest generowany, gdy osiągniemy fazę *SELECT*, funkcje okna można stosować tylko w klauzulach *SELECT* i *ORDER BY* zapytania. Jeśli do funkcji okna chcemy odwołać się we wcześniejszej fazie logicznego przetwarzania zapytania (na przykład *WHERE*), trzeba użyć wyrażenia tablicowego. Funkcję okna specyfikujemy na liście *SELECT* zapytania wewnętrznego i przypisujemy jej alias. Następnie w dowolnym miejscu zapytania zewnętrznego możemy odnieść się do tego aliasu.

Funkcje okna wymagają nieco przyzwyczajenia, ale po ich dokładnym poznaniu przekonamy się, że są faktycznie znacznie lepiej dostosowane do tego, jak zwykle myślimy o obliczeniach. Oprócz możliwości formułowania obliczeń analitycznych w sposób naturalny i intuicyjny pozwalają wykonywać wiele innych zadań w elegancji, a często również bardziej wydajny sposób.

Kolejne podrozdziały poświęcę szczegółowemu omówieniu funkcji okien dotyczące klasyfikowania, przesunięć i agregowania. Ponieważ książka ta z założenia omawia podstawy, nie będę poruszał niektórych tematów, takich jak optymalizacja funkcji okna, funkcje dystrybucji i jednostka ramki okna *RANGE*. Tematyce tej poświęcę całą książkę – jest to wspomniana wcześniej pozycja *Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions* (Microsoft Press, 2012).

Rankingowe funkcje okna

Rankingowe funkcje okna pozwalają określić pozycję każdego wiersz względem innych wierszy. Język T-SQL obsługuje cztery funkcje rankingowe: *ROW_NUMBER*, *RANK*, *DENSE_RANK* i *NTILE*. Poniższe zapytanie ilustruje użycie tych funkcji.

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(ORDER BY val) AS rownum,
       RANK()        OVER(ORDER BY val) AS rank,
       DENSE_RANK() OVER(ORDER BY val) AS dense_rank,
       NTILE(10)     OVER(ORDER BY val) AS ntile
FROM Sales.OrderValues
ORDER BY val;
```

Zapytanie to generuje następujące wyniki (pokazane w skróconej postaci):

orderid	custid	val	rownum	rank	dense_rank	ntile
10782	12	12.50	1	1	1	1
10807	27	18.40	2	2	2	1
10586	66	23.80	3	3	3	1
10767	76	28.00	4	4	4	1
10898	54	30.00	5	5	5	1
10900	88	33.75	6	6	6	1
10883	48	36.00	7	7	7	1
11051	41	36.00	8	7	7	1
10815	71	40.00	9	9	8	1
10674	38	45.00	10	10	9	1
...						
10691	63	10164.80	821	821	786	10
10540	63	10191.70	822	822	787	10
10479	65	10495.60	823	823	788	10
10897	37	10835.24	824	824	789	10
10817	39	10952.85	825	825	790	10
10417	73	11188.40	826	826	791	10
10889	65	11380.00	827	827	792	10
11030	71	12615.05	828	828	793	10
10981	34	15810.00	829	829	794	10
10865	63	16387.50	830	830	795	10

(830 row(s) affected)

Funkcja `ROW_NUMBER` przypisuje inkrementowane kolejne liczby całkowite wierszom w zbiorze wyników zapytania na podstawie kolejności wyspecyfikowanej przez klauzulę pomocniczą `ORDER BY` klauzuli `OVER`. W przykładowym zapytaniu kolejność logiczna opiera się na kolumnie `val`; z tego względu w danych wyjściowych widzimy, że numer wiersza zwiększa się wraz ze zwiększaniem wartości. Jednak nawet jeśli wartość porządkująca się nie zmienia, numer wiersza musi się powiększać. Dlatego też, jeśli lista `ORDER BY` funkcji `ROW_NUMBER` nie jest unikatowa (jak w tym przykładzie), zapytanie jest niedeterministyczne – możliwy jest więcej niż jeden prawidłowy wynik. Na przykład, widzimy, że dwa wiersze z wartością 36.00 uzyskały numery wierszy 7 i 8. Każde uporządkowanie tych numerów wierszy będzie traktowane jako prawidłowe. Jeśli chcemy, by wyznaczanie numeru wierszy było deterministyczne, do listy `ORDER BY` trzeba dodać elementy zapewniające jej unikatowość, czyli gwarantujące, że lista elementów klauzuli `ORDER BY` jednoznacznie identyfikuje wiersze. Możemy na przykład dodać kolumnę `orderid` do listy `ORDER BY` jako kryterium rozstrzygające, dzięki czemu wyznaczanie numerów wierszy będzie deterministyczne.

Jak wspomniałem, funkcja `ROW_NUMBER` zawsze generuje unikatowe wartości, nawet jeśli kolejność wartości nie jest jednoznaczna. Jeśli chcemy nadać taką samą rangę (ocenę) równym wartościom, należy zastosować funkcję `RANK` lub `DENSE_RANK`. Obie funkcje podobne są do funkcji `ROW_NUMBER`, ale generują tę samą wartość rankingu dla wszystkich wierszy, które mają tę samą wartość logicznego szeregowania. Różnica pomiędzy funkcją `RANK` a `DENSE_RANK` polega na tym, że funkcja `RANK` równa jest liczbie wierszy o mniejszej wartości sortowania plus 1 (innymi

słowy określa, ile wierszy poprzedza bieżący wiersz przy wskazanym uporządkowaniu w bieżącym oknie), natomiast funkcja *DENSE_RANK* wskazuje, ile występuje różnych mniejszych wartości. Tak więc *RANK* równe 9 dla klienta 71 wskazuje, że istnieje osiem wierszy o mniejszych wartościach. W przypadku funkcji *DENSE_RANK* ocena 8 wskazuje, że istnieje siedem mniejszych, różniących się wartości. W rezultacie wyniki funkcji *RANK* mogą zawierać luki (w przykładowym zapytaniu nie występuje wartość 8), zaś funkcja *DENSE_RANK* przybiera wartości kolejne, bez luk.

Funkcja *NTILE* przypisuje wiersze w wynikach do kolejnych grup wierszy o jednakowych rozmiarach (zawierających równe ilości wierszy). Żadaną liczbę grup podajemy jako argument funkcji, a w klauzuli *OVER* specyfikujemy logiczne porządkowanie. Przykładowe zapytanie zwraca 830 wierszy i zażądano 10 grup; z tego względu rozmiar grupy to 83 (830 podzielone przez 10). Porządkowanie logiczne bazuje na kolumnie *val*, co oznacza, że 83 wierszom o najmniejszych wartościach przypisany został numer grupy 1, następne 83 wiersze to grupa 2 itd. Jeśli liczba wierszy nie dzieli się całkowicie przez liczbę grup, do początkowych grup dodawany jest dodatkowy wiersz. Na przykład, jeśli mamy 102 wiersze i liczba grup wynosi 5, pierwsze dwie grupy będą miały po 21 wierszy zamiast 20.

Podobnie jak wszystkie funkcje okna, funkcje rankingowe obsługują klauzule partycjonowania okna. Przypomnijmy, że partycjonowanie okien zawęża okno tylko do tych wierszy, które w atrybutach partycji mają te same wartości, co wiersz bieżący. Na przykład wyrażenie *ROW_NUMBER() OVER(PARTITION BY custid ORDER BY val)* przypisuje numery wierszy niezależnie dla każdego klienta (tej samej wartości *custid*), a nie wierszom w całym zbiorze. Oto przykład użycia tego wyrażenia w zapytaniu:

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(PARTITION BY custid
                        ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

Zapytanie to generuje następujące wyniki (w skróconej postaci):

orderid	custid	val	rownum
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3
10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1
...			

(830 row(s) affected)

Jak widać w danych wyjściowych, numery wierszy są obliczane niezależnie dla każdego klienta, tak więc po zmianie klienta obliczanie jest zerowane i rozpoczyna się od początku.

Przypomnijmy, że porządkowanie okna nie ma nic wspólnego z porządkowaniem wyników dla prezentacji, a tym samym nie zmienia ich relacyjnej natury. Jeśli trzeba zagwarantować określony porządek prezentowania wyników, musimy dodać klauzulę prezentacji *ORDER BY*, jak w dwóch ostatnich zapytaniach ilustrujących funkcje rankingowe.

Funkcje okna są oceniane w logicznej fazie przetwarzania wyrażeń na liście *SELECT*, przed przetworzeniem klauzuli *DISTINCT*. Ktoś mógłby zapytać, dlaczego ta informacja ma znaczenie. Zademonstruję to na przykładzie. Aktualnie widok *OrderValues* zawiera 830 wierszy o 795 różnych wartościach w kolumnie *val*. Przeanalizujemy poniższe zapytanie i jego wyniki (pokazane w skróconej postaci):

```
SELECT DISTINCT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues;
```

val	rownum
-----	-----
12.50	1
18.40	2
23.80	3
28.00	4
30.00	5
33.75	6
36.00	7
36.00	8
40.00	9
45.00	10
...	
12615.05	828
15810.00	829
16387.50	830

(830 row(s) affected)

Funkcja *ROW_NUMBER* jest przetwarzana przed klauzulą *DISTINCT*, zatem najpierw przypisywane są unikatowe numery do 830 wierszy z widoku *OrderValues*. Dopiero potem przetwarzana jest klauzula *DISTINCT* – jednak w tym momencie nie istnieją już duplikaty wierszy, które trzeba by usuwać, zatem użycie klauzuli *DISTINCT* niczego nie zmienia. Jeśli chcemy przypisać numery wierszy do 795 unikatowych wartości, trzeba zastosować inne rozwiązanie. Na przykład, ponieważ faza *GROUP BY* jest przetwarzana przed fazą *SELECT*, moglibyśmy użyć następującego zapytania:

```
SELECT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues
GROUP BY val;
```

Zapytanie to generuje następujące wyniki (pokazane w skróconej postaci):

val	rownum
12.50	1
18.40	2
23.80	3
28.00	4
30.00	5
33.75	6
36.00	7
40.00	8
45.00	9
48.00	10
...	
12615.05	793
15810.00	794
16387.50	795

(795 row(s) affected)

W tym przypadku faza *GROUP BY* tworzy 795 grup dla 795 różnych wartości, po czym faza *SELECT* dla każdej grupy generuje wiersz z wartością *val* i numerem wiersza z sortowaniem opartym na kolumnie *val*. Alternatywne rozwiązanie pojawi się w ćwiczeniach do tego rozdziału.

Offsetowe funkcje okna

Offsetowa funkcja okna (przesunięcia) pozwala zwracać element z wiersza, który jest położony w pewnym przesunięciu (ang. *offset*) od bieżącego wiersza lub od początku bądź końca ramki okna. Język T-SQL obsługuje dwie pary funkcji offsetowych: *LAG* i *LEAD* oraz *FIRST_VALUE* i *LAST_VALUE*.

Funkcje *LAG* i *LEAD* wspierają klauzule partycjonowania i sortowania okna. Nie mają związku z definiowaniem ramki okna. Funkcje te pozwalają uzyskać element z wiersza wewnątrz partycji, który oddalony jest od bieżącego wiersza o podane przesunięcie, bazując na wskazanym porządku. Funkcja *LAG* wyszukuje przed bieżącym wierszem, a funkcja *LEAD* wyszukuje po bieżącym wierszu. Pierwszym (obowiązkowym) argumentem tych funkcji jest element, który ma zostać zwrócony; drugim argumentem (opcjonalnym) jest offset (przyjmujący wartość 1, jeśli nie zostanie wyspecyfikowany); trzecim argumentem (również opcjonalnym) jest wartość domyślna, która ma zostać zwrócona, jeśli żaden wiersz nie znajduje się w miejscu wskazanym przez offset (przy pominięciu tego argumentu zwracany jest znacznik *NULL*).

Poniższe zapytanie przykładowe zwraca informacje o zamówieniu z widoku *Order-Values*. Dla każdego zamówienia klienta użyta została funkcja *LAG*, by zwrócić wartość poprzedniego zamówienia klienta, oraz funkcja *LEAD*, by zwrócić wartość następnego zamówienia.

```

SELECT custid, orderid, val,
       LAG(val) OVER(PARTITION BY custid
                     ORDER BY orderdate, orderid) AS prevval,
       LEAD(val) OVER(PARTITION BY custid
                      ORDER BY orderdate, orderid) AS nextval
FROM Sales.OrderValues;

```

Poniżej, w skróconej postaci, pokazane są wyniki tego zapytania:

custid	orderid	val	prevval	nextval
1	10643	814.50	NULL	878.00
1	10692	878.00	814.50	330.00
1	10702	330.00	878.00	845.80
1	10835	845.80	330.00	471.20
1	10952	471.20	845.80	933.50
1	11011	933.50	471.20	NULL
2	10308	88.80	NULL	479.75
2	10625	479.75	88.80	320.00
2	10759	320.00	479.75	514.40
2	10926	514.40	320.00	NULL
3	10365	403.20	NULL	749.06
3	10507	749.06	403.20	1940.85
3	10535	1940.85	749.06	2082.00
3	10573	2082.00	1940.85	813.37
3	10677	813.37	2082.00	375.50
3	10682	375.50	813.37	660.00
3	10856	660.00	375.50	NULL
...				

(830 row(s) affected)

Ponieważ offset nie został wskazany, funkcje przyjmują wartość domyślną, czyli 1; inaczej mówiąc, funkcja *LAG* uzyskuje wartość poprzedniego zamówienia klienta, a funkcja *LEAD* – wartość następnego. Ponadto, ponieważ nie został wyspecyfikowany trzeci argument, użyty zostanie domyślny znacznik *NULL*, jeśli poprzednie lub następne zamówienie nie istnieje. Wyrażenie *LAG(val, 3, 0)* uzyskałoby wartość położoną o trzy wiersze wstecz i zwróciłoby wartość 0, jeśli wiersz nie zostanie znaleziony.

W tym przykładzie po prostu odczytałem wartości z poprzednich i następnych zamówień, ale zwykle na podstawie zwróconych wartości przeprowadzane są pewne obliczenia. Na przykład moglibyśmy obliczyć różnicę pomiędzy wartością bieżącego zamówienia klienta a wartością zamówienia poprzedniego: *val – LAG(val) OVER(...)* lub z wartością następnego zamówienia: *val – LEAD(val) OVER(...)*.

Funkcje *FIRST_VALUE* i *LAST_VALUE* zwracają element odpowiednio z pierwszego i ostatniego wiersza ramki okna. Z tego względu funkcje te obsługują klauzule definiowania partycji, porządku i ramek. Aby uzyskać element z pierwszego wiersza w partycji okna, stosujemy funkcję *FIRST_VALUE* wraz z określeniem zakresu ramki *ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW*. Do uzyskania elementu z ostatniego wiersza partycji okna użyjemy funkcji *LAST_VALUE* wraz z określeniem

rozmiaru ramki okna *ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING*. Zwróćmy uwagę, że jeśli wyspecyfikujemy klauzulę *ORDER BY* bez jednostki ramki okna (takiej jak *ROWS*), dolnym ograniczeniem będzie domyślnie *CURRENT ROW* (wiersz bieżący) i zdecydowanie to nie jest to, co chcieliśmy uzyskać za pomocą funkcji *LAST_VALUE*. Ponadto z powodów wydajnościowych, których uzasadnianie wykracza poza zakres tej książki, powinniśmy zawsze jednoznacznie określać rozmiar ramki okna, także dla funkcji *FIRST_VALUE*.

Poniższe zapytanie przykładowe używa funkcji *FIRST_VALUE*, by zwrócić wartość pierwszego zamówienia klienta, a funkcja *LAST_VALUE* używana jest do zwrócenia ostatniego zamówienia klienta.

```
SELECT custid,orderid, val,
       FIRST_VALUE(val) OVER(PARTITION BY custid
                             ORDER BY orderdate,orderid
                             ROWS BETWEEN UNBOUNDED PRECEDING
                                     AND CURRENT ROW) AS firstval,
       LAST_VALUE(val) OVER(PARTITION BY custid
                             ORDER BY orderdate,orderid
                             ROWS BETWEEN CURRENT ROW
                                     AND UNBOUNDED FOLLOWING) AS lastval
FROM Sales.OrderValues
ORDER BY custid,orderdate,orderid;
```

Zapytanie to generuje następujące dane wyjściowe (pokazane w skróconej postaci):

custid	orderid	val	firstval	lastval
1	10643	814.50	814.50	933.50
1	10692	878.00	814.50	933.50
1	10702	330.00	814.50	933.50
1	10835	845.80	814.50	933.50
1	10952	471.20	814.50	933.50
1	11011	933.50	814.50	933.50
2	10308	88.80	88.80	514.40
2	10625	479.75	88.80	514.40
2	10759	320.00	88.80	514.40
2	10926	514.40	88.80	514.40
3	10365	403.20	403.20	660.00
3	10507	749.06	403.20	660.00
3	10535	1940.85	403.20	660.00
3	10573	2082.00	403.20	660.00
3	10677	813.37	403.20	660.00
3	10682	375.50	403.20	660.00
3	10856	660.00	403.20	660.00
...				

(830 row(s) affected)

Podobnie jak w przypadku funkcji *LAG* i *LEAD*, zwrócone wartości zwykle wykorzystywane są do wykonania pewnych obliczeń. Na przykład moglibyśmy obliczyć różnicę pomiędzy wartością bieżącego zamówienia klienta a wartością pierwszego zamówienia:

val – *FIRST_VALUE(val) OVER(...)* lub różnicę pomiędzy bieżącym a ostatnim zamówieniem: *val* – *LAST_VALUE(val) OVER(...)*.

Agregujące funkcje okna

Agregujące funkcje okna umożliwiają agregowanie (sumowanie lub zliczanie) wierszy w zdefiniowanym oknie. Funkcje te obsługują klauzule partycjonowania, definiowania kolejności i ramek.

Omawianie rozpoczniemy od przykładu, który nie obejmuje porządku ani tworzenia ramki. Przypomnijmy, że użycie pustej klauzuli *OVER* powoduje utworzenie okna zawierającego wszystkie wiersze zbioru wyników zapytania bazowego. Tak więc funkcja *SUM(val) OVER()* zwraca całkowitą sumę wszystkich wartości. Jeśli dodamy klauzulę partycji okna, funkcja będzie działać na ograniczonym oknie, zawierającym tylko te wiersze zbioru wyników zapytania bazowego, które w elementach partycjonujących mają te same wartości, co wiersz bieżący. Dla przykładu funkcja *SUM(val) OVER(PARTITION BY custid)* zwraca sumę wartości dla bieżącego klienta.

Poniżej pokazano zapytanie do *OrderValue*, które dla każdego zamówienia zwraca sumę łączną wszystkich zamówień, a także sumę łączną zamówień klienta.

```
SELECT orderid, custid, val,
       SUM(val) OVER() AS totalvalue,
       SUM(val) OVER(PARTITION BY custid) AS custtotalvalue
FROM Sales.OrderValues;
```

Wyniki tego zapytania są następujące (pokazane w skróconej postaci):

orderid	custid	val	totalvalue	custtotalvalue
-----	-----	-----	-----	-----
10643	1	814.50	1265793.22	4273.00
10692	1	878.00	1265793.22	4273.00
10702	1	330.00	1265793.22	4273.00
10835	1	845.80	1265793.22	4273.00
10952	1	471.20	1265793.22	4273.00
11011	1	933.50	1265793.22	4273.00
10926	2	514.40	1265793.22	1402.95
10759	2	320.00	1265793.22	1402.95
10625	2	479.75	1265793.22	1402.95
10308	2	88.80	1265793.22	1402.95
10365	3	403.20	1265793.22	7023.98
...				

(830 row(s) affected)

Kolumna *totalvalue* w każdym wierszu pokazuje łączną wartość obliczoną dla wszystkich wierszy. Kolumna *custtotalvalue* zawiera sumę wartości zamówień bieżącego klienta.



WAŻNE Jak wspomniałem na początku rozdziału, jedną z większych zalet funkcji okien jest możliwość zwracania w tym samym wierszu elementów szczegółowych i zagregowanych, a także konstruowanie wyrażeń, które łączą elementy obu rodzajów. Kolejny przykład demonstruje takie działanie.

Jako przykład łączenia w jednym zbiorze wynikowych danych szczegółowych i zagregowanych, poniższe zapytanie dla każdego wiersza oblicza procentowy udział wartości bieżącego zamówienia względem wartości całkowitej wszystkich zamówień, a także procent, jaki stanowi bieżąca wartość w odniesieniu do wartości łącznej zamówień danego klienta.

```
SELECT orderid, custid, val,
       100. * val / SUM(val) OVER() AS pctall,
       100. * val / SUM(val) OVER(PARTITION BY custid) AS pctcust
FROM Sales.OrderValues;
```

Zapytanie to zwraca następujące wyniki (pokazane w skróconej postaci):

orderid	custid	val	pctall	pctcust
10643	1	814.50	0.0643470029014691672941	19.0615492628130119354083
10692	1	878.00	0.0693636201505830925528	20.5476246197051252047741
10702	1	330.00	0.0260706089103558320528	7.7229113035338169904048
10835	1	845.80	0.0668197606556938265161	19.7940556985724315469225
10952	1	471.20	0.0372256694501808123130	11.0273812309852562602387
11011	1	933.50	0.0737482224782338461253	21.8464778843903580622513
10926	2	514.40	0.0406385491620819394181	36.6655974910011048148544
10759	2	320.00	0.0252805904585268674452	22.8090808653195053280587
10625	2	479.75	0.0379011352264945770526	34.1958017035532271285505
10308	2	88.80	0.0070153638522412057160	6.3295199401261627285362
10365	3	403.20	0.0318535439777438529809	5.7403352515240647040566
...				

(830 row(s) affected)

Funkcje agregujące obsługują również definiowanie kolejności i ramek. Dzięki temu możliwe są bardziej skomplikowane obliczenia, takie jak sumy ruchome, obliczenia typu „od początku roku” (YTD – Year-To-Date) i inne. Przeanalizujemy ponownie zapytanie pokazane na początku tego podrozdziału.

```
SELECT empid, ordermonth, val,
       SUM(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                              AND CURRENT ROW) AS runval
FROM Sales.EmpOrders;
```

Zapytanie to generuje następujące wyniki (w skróconej postaci).

empid	ordermonth	val	runval
1	2014-07-01	1614.88	1614.88
1	2014-08-01	5555.90	7170.78
1	2014-09-01	6651.00	13821.78
1	2014-10-01	3933.18	17754.96
1	2014-11-01	9562.65	27317.61
...			
2	2014-07-01	1176.00	1176.00
2	2014-08-01	1814.00	2990.00
2	2014-09-01	2950.80	5940.80
2	2014-10-01	5164.00	11104.80
2	2014-11-01	4614.58	15719.38
...			

(192 row(s) affected)

Każdy wiersz widoku *EmpOrders* dla każdego pracownika i miesiąca przechowuje informacje o aktywności pracownika. Dla każdego pracownika i miesiąca zapytanie zwraca łączną wartość miesięczną i skumulowane wartości od początku aktywności pracownika do bieżącego miesiąca. Aby wykonać obliczenia niezależnie do każdego pracownika, tworzymy partycję okna według kolumny *empid*. Następnie definiujemy kolejność w oparciu o kolumnę *ordermonth*, nadając rozmiar ramki okna: *ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW*. Ramka taka oznacza: „cała aktywność od początku partycji do bieżącego miesiąca”.

Język T-SQL obsługuje również inne ograniczniki dla jednostki ramki okna *ROWS*. Możemy wskazać przesunięcie (offset) w tył od bieżącego wiersza, a także możemy wskazać przesunięcie w przód. Na przykład, aby przechwycić wszystkie wiersze od dwóch wierszy przed wierszem bieżącym do jednego wiersza po wierszu bieżącym, użyjemy składni *ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING*. Ponadto, jeśli nie chcemy mieć ograniczenia od góry, używamy wyrażenia *UNBOUNDED FOLLOWING*.

Ponieważ funkcje okna są tak potężne i mają różnorodne praktyczne zastosowania, gorąco namawiam, by zainwestować czas i wysiłki na ich dogłębne poznanie – zacząć można od wspomnianych wcześniej książek, które wyczerpująco omawiają te zagadnienia.

Przestawianie danych

Przestawianie danych (pivoting) to zamiana wierszy na kolumny, przy czym zazwyczaj w trakcie tych operacji wartości są odpowiednio agregowane. W wielu przypadkach przestawianie danych jest obsługiwane przez warstwę prezentacji, na przykład przez narzędzia do generowania raportów. W tym podrozdziale pokażę, jak obsługiwać przestawianie danych za pomocą języka T-SQL w tych sytuacjach, kiedy chcemy operacje te wykonać w bazie danych.

W pozostałej części rozdziału będziemy używać tabeli *dbo.Orders*, którą utworzymy w bazie danych TSQLV4 i wypełnimy danymi, uruchamiając kod prezentowany w listingu 7-1.

LISTING 7-1 Kod do utworzenia i wypełnienia danymi tabeli *dbo.Orders*

```
USE TSQLV4;

DROP TABLE IF EXISTS dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid INT NOT NULL,
    orderdate DATE NOT NULL,
    empid INT NOT NULL,
    custid VARCHAR(5) NOT NULL,
    qty INT NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
VALUES
    (30001, '20140802', 3, 'A', 10),
    (10001, '20141224', 2, 'A', 12),
    (10005, '20141224', 1, 'B', 20),
    (40001, '20150109', 2, 'A', 40),
    (10006, '20150118', 1, 'C', 14),
    (20001, '20150212', 2, 'B', 12),
    (40005, '20160212', 3, 'A', 10),
    (20002, '20160216', 1, 'C', 20),
    (30003, '20160418', 2, 'B', 15),
    (30004, '20140418', 3, 'C', 22),
    (30007, '20160907', 3, 'D', 30);

SELECT * FROM dbo.Orders;
```

Zapytanie na końcu kodu z listingu 7-1 generuje następujące wyniki, które pokazują zawartość tabeli *dbo.Orders*.

orderid	orderdate	empid	custid	qty
10001	2014-12-24	2	A	12
10005	2014-12-24	1	B	20
10006	2015-01-18	1	C	14
20001	2015-02-12	2	B	12
20002	2016-02-16	1	C	20
30001	2014-08-02	3	A	10
30003	2016-04-18	2	B	15
30004	2014-04-18	3	C	22
30007	2016-09-07	3	D	30
40001	2015-01-09	2	A	40
40005	2016-02-12	3	A	10

Żałujemy, że dla każdego pracownika i klienta chcemy uzyskać łączną liczbę zamówionych towarów. Zadanie to można zrealizować za pomocą prostego zapytania grupującego:

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid;
```

Poniżej pokazano wyniki zapytania:

empid	custid	sumqty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30

Żałujemy jednak, że chcemy wygenerować dane wyjściowe w postaci pokazanej w tabeli 7-1.

TABELA 1-1 Przestawiony widok danych łącznej ilości dla kombinacji pracowników (wiersze) i klientów (kolumny)

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

Tym, co pokazuje tabela 7-1, jest zagregowany i przestawiony widok danych tabeli *dbo.Orders*; metoda generowania takiego widoku danych nazwana jest *przestawianiem* (*pivoting*).

Każde żądanie przestawienia danych obejmuje trzy fazy logicznego przetwarzania, a z każdą fazą powiązane są odpowiednie elementy:

1. Faza grupowania z powiązaniem elementem grupowania, czyli elementem w wierszach.
2. Faza rozpraszania z powiązaniem elementem rozpraszania, czyli elementem w kolumnach.
3. Faza agregowania z powiązaniem elementem agregującym lub z funkcją agregującą.

W tym przykładzie potrzeba wygenerować po jednym wierszu dla każdego unikatowego identyfikatora pracownika. Oznacza to, że wiersze z tabeli *dbo.Orders* trzeba pogrupować według atrybutu *empid* – zatem w tym przypadku elementem grupowania jest atrybut *empid*.

Tabela *dbo.Orders* zawiera pojedynczą kolumnę, w której przechowuje wartości identyfikatorów wszystkich klientów i pojedynczą kolumnę z ilościami towarów zamówionych przez klientów. Proces przedstawiania danych zakłada wygenerowanie dla każdego unikatowego identyfikatora klienta oddzielnej kolumny wyników, przy czym każda z nich ma zawierać posumowane ilości towaru zamówionego przez tego klienta. Proces ten możemy traktować jako „rozpraszanie” ilości na poszczególne identyfikatory klientów. W tym przypadku element rozpraszania to atrybut *custid*.

Na koniec, ponieważ przedstawianie danych stosuje grupowanie, trzeba zagregować dane, by wygenerować wartości wyników, które znajdą się na „przecięciu” elementów grupowania (wierszy) i rozpraszania (kolumn). Trzeba wybrać funkcję agregującą (w tym przypadku *SUM*) i element agregowany (w tym przypadku atrybut *qty*).

Podsumowując, przedstawianie danych obejmuje trzy fazy: grupowanie, rozpraszanie i agregowanie. W tym przykładzie grupujemy według atrybutu *empid*, rozpraszamy według atrybutu *custid* i agregujemy przy użyciu funkcji *SUM(qty)*. Po zidentyfikowaniu elementów, które używane są w procesie przedstawiania danych, pozostaje tylko umieścić te elementy na właściwych miejscach w ogólnym szablonie zapytania przedstawiania danych.

W tym rozdziale zaprezentowano dwie metody przedstawiania danych – rozwiązanie wynikające ze standardu SQL oraz rozwiązanie wykorzystujące operator tablicowy *PIVOT*, specyficzny dla języka T-SQL.

Przedstawianie danych przy użyciu zapytania grupującego

Rozwiązanie używające zapytania grupującego obsługuje wszystkie trzy fazy w jawny i bardzo prosty sposób.

Faza grupowania jest realizowana przy użyciu klauzuli *GROUP BY*; w tym przypadku: *GROUP BY empid*.

Faza rozpraszania jest realizowana za pomocą klauzuli *SELECT* z wyrażeniem *CASE* dla każdej kolumny docelowej. Musimy wcześniej znać możliwe wartości elementu rozpraszania i dla każdej zbudować oddzielne wyrażenie. Ponieważ w tym przypadku trzeba rozproszyć ilości dla czterech klientów (A, B, C i D), mamy cztery wyrażenia *CASE*. Dla przykładu poniżej pokazano wyrażenie *CASE* dla klienta A:

```
CASE WHEN custid = 'A' THEN qty END
```

Wyrażenie to zwraca ilość tylko wtedy, gdy wiersz bieżący reprezentuje zamówienie dla klienta A; w przeciwnym razie wyrażenie zwraca wartość *NULL*. Przypomnijmy, że jeśli w wyrażeniu *CASE* nie została wyspecyfikowana klauzula *ELSE*, domyślnie użyte zostanie wyrażenie *ELSE NULL*. Oznacza to, że w kolumnie docelowej dla klienta A tylko ilości powiązane z klientem A pojawią się jako wartości kolumny, a we wszystkich innych sytuacjach wartościami kolumny będzie znacznik *NULL*.

Jeśli nie znamy wcześniej wartości, które trzeba rozproszyć (w tym przypadku różne identyfikatory klientów) i na podstawie danych chcemy uzyskać odpowiedź, trzeba odczytać je z danych, samodzielnie skonstruować ciąg zapytania i wykonać go. Ten sposób realizacji tego zadania przedstawię w rozdziale 11 „Obiekty programowalne”.

Na koniec faza agregowania jest realizowana poprzez zastosowanie odpowiedniej funkcji agregującej (w tym przypadku *SUM*) do wyników każdego wyrażenia *CASE*. Dla przykładu poniższe wyrażenie generuje kolumnę wyników dla klienta A.

```
SUM(CASE WHEN custid = 'A' THEN qty END) AS A
```

Oczywiście, w zależności od żądania, może zachodzić potrzeba zastosowania innej funkcji agregującej (na przykład *MAX*, *MIN* lub *COUNT*).

Poniżej zaprezentowano pełne rozwiązanie, które przedstawia dane zamówienia, zwracając łączną ilość dla każdego pracownika (w wierszach) i klienta (w kolumnach).

```
SELECT empid,  
       SUM(CASE WHEN custid = 'A' THEN qty END) AS A,  
       SUM(CASE WHEN custid = 'B' THEN qty END) AS B,  
       SUM(CASE WHEN custid = 'C' THEN qty END) AS C,  
       SUM(CASE WHEN custid = 'D' THEN qty END) AS D  
FROM dbo.Orders  
GROUP BY empid;
```

Zapytanie to generuje wyniki przedstawione wcześniej w tabeli 7-1. SQL Server wygeneruje dodatkowo następujące ostrzeżenie w panelu Messages programu SSMS:

Warning: Null value is eliminated by an aggregate or other SET operation.
(Ostrzeżenie: Znacznik *NULL* jest eliminowany przez agregacje lub inne operacje na zbiorach)

Ostrzeżenie informuje, że znaczniki *NULL* są ignorowane przez funkcje agregujące.

Przestawianie danych przy użyciu operatora *PIVOT*

Rozwiązanie oparte na grupowaniu danych jest zgodne ze standardem SQL. Język T-SQL obsługuje operator tablicowy nazwany *PIVOT*, który jest specyficzny dla tego dialektu, umożliwiający realizację przestawiania w bardziej spójny sposób. Operator *PIVOT* działa w kontekście klauzuli *FROM* zapytania, podobnie jak inne operatory tablicowe (na przykład *JOIN*). Wejściem dla operatora jest tabela źródłowa lub

wyrażenie tablicowe. Operator przestawia dane i zwraca tabelę wyników. Operator *PIVOT* wykonuje te same fazy logicznego przetwarzania, które zostały opisane wcześniej (grupowanie, rozpraszanie i agregacja), ale wymaga znacznie mniej kodu, niż poprzednie rozwiązanie.

Ogólna postać zapytania z operatorem *PIVOT* jest następująca:

```
SELECT ...
FROM <tabela_źródłowa_lub_wyrażenie_tablicowe>
    PIVOT(<funkcja_agregująca>(<element_agregowany>)
        FOR <element_rozpraszania>
            IN (<lista_kolumn_docelowych>)) AS <alias_tabeli_wynikowej>
...;
```

W nawiasach operatora *PIVOT* podajemy funkcję agregowania (na przykład *SUM*), element agregacji (*qty*), element rozpraszania (*custid*) i listę nazw kolumn docelowych (A, B, C, D). Po nawiasach operatora *PIVOT* specyfikujemy alias tabeli wynikowej.

Warto zwrócić uwagę na to, że za pomocą operatora *PIVOT* nie musimy wprost specyfikować elementów grupowania, eliminując w ten sposób klauzulę *GROUP BY*. Operator *PIVOT* określa w sposób pośredni elementy grupowania jako wszystkie atrybuty z tabeli źródłowej (lub wyrażenia tablicowego), które nie zostały wyspecyfikowane ani jako element rozpraszania, ani jako element sumowania. Musimy zatem zapewnić, że tabela źródłowa dla operatora *PIVOT* nie ma żadnych innych atrybutów oprócz grupowania, rozpraszania i agregacji, tak by po wyspecyfikowaniu elementów rozpraszania i agregacji pozostały tylko atrybuty, które mają być elementami grupowania. Zadanie to można zrealizować poprzez niestosowanie operatora *PIVOT* bezpośrednio do pierwotnej tabeli (w tym przypadku *Orders*), a do wyrażenia tablicowego, zawierającego tylko te atrybuty, które reprezentują elementy przestawiania danych i nie zawierające żadnych innych atrybutów. Poniższy przykład realizuje oryginalne żądanie przestawienia danych, tym razem przy użyciu operatora *PIVOT*.

```
SELECT empid, A, B, C, D
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
    PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;
```

Zamiast działania bezpośrednio na tabeli *dbo.Orders*, operator *PIVOT* działa na tabeli pochodnej nazwanej D, która zawiera tylko elementy przestawiania danych (*empid*, *custid* i *qty*). Po uwzględnieniu elementu rozpraszania, którym jest *custid*, i elementu agregacji, którym jest *qty*, pozostaje element *empid*, który będzie traktowany jak element grupowania.

Zapytanie to zwraca wyniki pokazane wcześniej w tabeli 7-1.

Aby zrozumieć, dlaczego w tym miejscu konieczne jest zastosowanie wyrażenia tablicowego, przeanalizujmy następujące zapytanie stosujące operator *PIVOT* bezpośrednio do tabeli *dbo.Orders*.


```
SELECT empid, A, B, C, D
FROM dbo.Orders
PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;
```

Tabela *dbo.Orders* zawiera atrybuty *orderid*, *orderdate*, *empid*, *custid* i *qty*. Ponieważ zapytanie wskazuje *custid* jako element rozpraszania, a *qty* jako element agregacji, wszystkie pozostałe atrybuty (*orderid*, *orderdate* i *empid*) będą uważane za elementy grupowania. Z tego względu zapytanie zwraca następujące wyniki:

empid	A	B	C	D
2	12	NULL	NULL	NULL
1	NULL	20	NULL	NULL
1	NULL	NULL	14	NULL
2	NULL	12	NULL	NULL
1	NULL	NULL	20	NULL
3	10	NULL	NULL	NULL
2	NULL	15	NULL	NULL
3	NULL	NULL	22	NULL
3	NULL	NULL	NULL	30
2	40	NULL	NULL	NULL
3	10	NULL	NULL	NULL

(11 row(s) affected)

Ponieważ *orderid* jest częścią elementów grupowania, uzyskujemy wiersz dla każdego zamówienia, a nie dla każdego pracownika. Logicznym ekwiwalentem tego zapytania korzystającym ze standardowej składni dla przestawiania danych byłoby wymienienie atrybutów *orderid*, *orderdate* i *empid* w klauzuli *GROUP BY*, jak w poniższym kodzie:

```
SELECT empid,
SUM(CASE WHEN custid = 'A' THEN qty END) AS A,
SUM(CASE WHEN custid = 'B' THEN qty END) AS B,
SUM(CASE WHEN custid = 'C' THEN qty END) AS C,
SUM(CASE WHEN custid = 'D' THEN qty END) AS D
FROM dbo.Orders
GROUP BY orderid, orderdate, empid;
```

Najlepsze praktyki zalecają, by nigdy nie działać bezpośrednio na tabeli bazowej. Nawet jeśli tabela obecnie zawiera tylko kolumny używane jako elementy przestawiania danych, nigdy nie wiadomo, czy w przyszłości do tabeli nie zostanie dodana nowa kolumna, co spowoduje generowanie nieprawidłowych wyników. Ważne jest także jawne wyliczenie potrzebnych kolumn (unikanie gwiazdki) zarówno w zapytaniu wewnętrznym, jak i zewnętrznym wyrażenia tablicowego.

Jako inny przykład zadania przestawiania danych, załóżmy, że zamiast zwracania pracowników w wierszach i klientów w kolumnach chcemy inaczej poukładać te informacje: elementem grupowania jest *custid*, elementem rozpraszania jest *empid*, a elementem i funkcją agregacji pozostaje *SUM(qty)*. Po poznaniu „szablonu” przestawiania danych (opartego na standardzie lub własnego SQL Server) pozostaje kwestia

umieszczenia tych elementów na właściwych miejscach. Poniższe zapytanie korzysta z operatora *PIVOT* do wygenerowania wyników.

```
SELECT custid, [1], [2], [3]
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
      PIVOT(SUM(qty) FOR empid IN([1], [2], [3])) AS P;
```

Identyfikatory pracowników 1, 2 i 3 to wartości w kolumnie *empid* w tabeli źródłowej, jednak w kontekście wyników wartości te staną się nazwami kolumn docelowych. Z tego względu w klauzuli *PIVOT IN* musimy odwoływać się do nich jak do identyfikatorów. Jeśli identyfikatorami są wyrażenia nieregularne (na przykład rozpoczynające się od cyfry), trzeba je ograniczyć delimiterami – w tym celu użyte są nawiasy kwadratowe.

Zapytanie to zwraca następujące dane wyjściowe:

custid	1	2	3
A	NULL	52	20
B	20	27	NULL
C	34	NULL	22
D	NULL	NULL	30

Odwrotne przestawianie danych

Odwrotne przestawianie danych (unpivoting) to metoda zamiany kolumn na wiersze. Zazwyczaj operacja ta wiąże się z odpytaniem przestawionych danych i generuje z każdego wiersza źródłowego wiele wierszy wyników, każdy z inną wartością kolumny źródłowej. Typowe zastosowanie tej operacji to przestawienie danych zaimportowanych z arkusza kalkulacyjnego w celu łatwiejszego przetwarzania w bazie danych.

Poniższy kod utworzy w bazie danych TSQLV4 tabelę *EmpCustOrders* i wypełni ją danymi.

```
USE TSQLV4;

DROP TABLE IF EXISTS dbo.EmpCustOrders;

CREATE TABLE dbo.EmpCustOrders
(
    empid INT NOT NULL
    CONSTRAINT PK_EmpCustOrders PRIMARY KEY,
    A VARCHAR(5) NULL,
    B VARCHAR(5) NULL,
    C VARCHAR(5) NULL,
    D VARCHAR(5) NULL
);

INSERT INTO dbo.EmpCustOrders(empid, A, B, C, D)
SELECT empid, A, B, C, D
```

```
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
      PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;
SELECT * FROM dbo.EmpCustOrders;
```

Poniżej pokazano wyniki zapytania do tabeli *EmpCustOrders*.

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

Tabela zawiera wiersz dla każdego pracownika, kolumnę dla każdego z czterech klientów A, B, C i D oraz sumę ilości zamówień dla każdego pracownika i klienta w miejscu przecięcia pracownik-klient. Zwróćmy uwagę, że przecięcia niemające znaczenia (połączenia pracownik-klient, którzy nie mają wspólnej aktywności dotyczącej zamówień) są reprezentowane przez znaczniki *NULL*. Załóżmy, że trzeba zrealizować żądanie odwrotnego przestawiania danych, tak by dla każdej istotnej kombinacji pracownika i klienta zwrócić wiersz z ilością zamówienia. Wyniki powinny mieć następującą postać:

empid	custid	qty
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

W kolejnych podrozdziałach opiszę dwie metody rozwiązania tego problemu – metodę zgodną ze standardem SQL i metodę, która korzysta z operatora *UNPIVOT* specyficznego dla języka T-SQL.

Odwrotne przestawianie danych przy użyciu operatora **APPLY**

Rozwiązanie standardowe odwrotnego przestawiania danych wykorzystuje trzy fazy logicznego przetwarzania: tworzenie kopii, wydobycie elementów i eliminacja nieistotnych wierszy.

Pierwszy krok rozwiązania generuje wiele kopii każdego wiersza źródłowego – po jednym dla każdej kolumny, która ma być poddana odwrotnemu przestawianiu danych. W tym przypadku trzeba utworzyć kopie dla każdej z kolumn A, B, C i D, które reprezentują identyfikatory klientów. W algebrze relacyjnej i języku SQL operacją używaną do utworzenia wielu kopii każdego wiersza jest iloczyn kartezjański

(złączenie krzyżowe). Musimy zastosować złączenie krzyżowe pomiędzy tabelą *EmpCustOrders* a tabelą, która zawiera po jednym wierszu dla każdego klienta.

Jeśli nasza baza danych zawiera już tabelę klientów, można jej użyć. Jeśli takiej tabeli nie ma, możemy zbudować tabelę wirtualną przy użyciu klauzuli *VALUES*. Zapytanie implementujące pierwszy krok rozwiązania może wyglądać podobnie do pokazanego poniżej:

```
SELECT *
FROM dbo.EmpCustOrders
CROSS JOIN (VALUES('A'),('B'),('C'),('D')) AS C(custid);
```

Klauzula *VALUES* zwraca zbiór o czterech wierszach. Kod definiuje tabelę pochodną jako C i określa nazwę jedynej kolumny tej tabeli jako *custid*. Następnie wykonuje złączenie krzyżowe tej tabeli z *EmpCustOrders*.



DODATKOWE INFORMACJE Szczegółowy opis klauzuli *VALUES* zawiera rozdział 8 „Modyfikowanie danych”.

W tym przykładzie zapytanie realizujące pierwszy krok rozwiązania zwraca następujące wyniki:

empid	A	B	C	D	custid
1	NULL	20	34	NULL	A
1	NULL	20	34	NULL	B
1	NULL	20	34	NULL	C
1	NULL	20	34	NULL	D
2	52	27	NULL	NULL	A
2	52	27	NULL	NULL	B
2	52	27	NULL	NULL	C
2	52	27	NULL	NULL	D
3	20	NULL	22	30	A
3	20	NULL	22	30	B
3	20	NULL	22	30	C
3	20	NULL	22	30	D

Jak można zauważyć, utworzone zostały cztery kopie każdego wiersza źródłowego – po jednej dla klientów A, B, C i D.

Drugim etapem rozwiązania jest utworzenie kolumny (w tym przypadku nazwiemy ją *qty*), która będzie zawierać wartości z kolumny odpowiadającej klientowi reprezentowanemu przez bieżący wiersz. Mówiąc precyzyjniej, jeśli bieżąca wartość *custid* to A, kolumna *qty* powinna reprezentować wartość z kolumny A, jeśli *custid* to B, kolumna *qty* powinna zwracać wartość z kolumny B itd.

Aby zrealizować ten krok, mogłoby się wydawać, że wystarczy dodać kolumnę *qty* w każdym wierszu konstruktora tabeli (klauzuli *VALUES*), jak poniżej:

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
CROSS JOIN (VALUES('A', A),('B', B),('C', C),('D', D)) AS C(custid, qty);
```

Jednak przypomnijmy, że złączenie traktuje obydwa obiekty wejściowe jako zbiory, a tym samym nie istnieje w nich żaden określony porządek. Nie możemy się odwoływać do elementów żadnej z tabel wejściowych przy konstruowaniu drugiej. W tym przypadku konstruktor o wartościach tablicowej po prawej stronie złączenia zawiera odwołania do kolumn A, B, C i D z lewej strony złączenia (*EmpCustOrders*). W rezultacie próba uruchomienia tego kodu zwróci następujące komunikaty o błędach:

```
Msg 207, Level 16, State 1, Line 222
Invalid column name 'A'.
Msg 207, Level 16, State 1, Line 222
Invalid column name 'B'.
Msg 207, Level 16, State 1, Line 222
Invalid column name 'C'.
Msg 207, Level 16, State 1, Line 222
Invalid column name 'D'.
```

Rozwiązaniem jest użycie operatora *CROSS APPLY* zamiast *CROSS JOIN*. Są one podobne do siebie, ale ten pierwszy przetwarza najpierw wyrażenie po lewej stronie, po czym stosuje wyrażenie z prawej strony do każdego wiersza z lewej strony, dzięki czemu elementy z lewej strony są dostępne. Poniższy kod implementuje to rozwiązanie:

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
CROSS APPLY (VALUES('A', A),('B', B),('C', C),('D', D)) AS C(custid, qty);
```

Zapytanie to zostanie wykonane z powodzeniem, zwracając następujące wyniki:

empid	custid	qty
1	A	NULL
1	B	20
1	C	34
1	D	NULL
2	A	52
2	B	27
2	C	NULL
2	D	NULL
3	A	20
3	B	NULL
3	C	22
3	D	30

Przypomnijmy, że w oryginalnej tabeli znaczniki *NULL* reprezentują nieistotne przecięcia (pary klientów i pracowników, dla których nie istnieją wspólne zamówienia). Z tego względu zazwyczaj nie ma powodu przechowywania wierszy, w których *qty* to *NULL*. Przyjemną rzeczą w tym przypadku jest to, że operator *CROSS APPLY*, który utworzył kolumnę *qty*, jest przetwarzany w klauzuli *FROM*, która jest wykonywana przed

klauzulą *WHERE*. Oznacza to, że kolumna *qty* jest dostępna dla wyrażeń zawartych w klauzuli *WHERE*. Aby usunąć niepotrzebne wiersze, wystarczy dodać do klauzuli *WHERE* filtr odrzucający wiersze zawierające *NULL* w kolumnie *qty*, jak poniżej:

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
  CROSS APPLY (VALUES('A', A),('B', B),('C', C),('D', D)) AS C(custid, qty)
WHERE qty IS NOT NULL;
```

Zapytanie to zwraca następujące wyniki:

empid	custid	qty
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

Odwrotne przestawianie danych za pomocą operatora *UNPIVOT*

Odwrotne przestawianie danych tworzy dwie kolumny wyników z dowolnej liczby kolumn źródłowych – jedna przechowuje oryginalne nazwy kolumn jako ciągi znaków, a druga zawiera wartości kolumn źródłowych. W naszym przykładzie trzeba poddać tej operacji kolumny źródłowe A, B, C i D, tworząc dwie kolumny wyników *custid* i *qty*. Język T-SQL udostępnia elegancki operator tablicowy *UNPIVOT*. Postać ogólna zapytania z użyciem operatora *UNPIVOT* jest następująca:

```
SELECT ...
FROM <tabela_źródłowa_lub_wyrażenie_tablicowe>
  UNPIVOT(<kolumna_docelowa_przechowywania_wartości_kolumn_źródłowych>
    FOR <kolumna_docelowa_przechowywania_nazw_kolumn_źródłowych>
      IN(<lista_kolumn_źródłowych>))
AS <alias_tabeli_wynikowej>
WHERE ...;
```

Podobnie jak w przypadku operatora *PIVOT*, operator *UNPIVOT* został zaimplementowany jako operator tablicowy w kontekście klauzuli *FROM*. Operator ten działa na tabeli źródłowej lub wyrażeniu tablicowym (w tym przykładzie *EmpCustOrders*). Wewnątrz nawiasów operatora *UNPIVOT* specyfikujemy nazwę, która ma być przypisana do kolumny przechowywania wartości kolumn źródłowych (w tym przypadku *qty*), nazwę kolumny przechowującej nazwy kolumn źródłowych (*custid*) oraz listę nazw kolumn źródłowych (A, B, C i D). Po nawiasach specyfikujemy alias tabeli utworzonej przez operator tablicowy.

Poniżej zaprezentowano pełne rozwiązanie zapytania realizującego odwrotne przedstawianie danych w naszym przykładzie z użyciem operatora *UNPIVOT*.

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
UNPIVOT(qty FOR custid IN(A, B, C, D)) AS U;
```

Warto zauważyć, że operator *UNPIVOT* implementuje te same, opisane wcześniej fazy logicznego przetwarzania – generowanie kopii, wydobywanie elementów i eliminowanie przecięć *NULL*. Ostatnia faza nie jest fazą opcjonalną, tak jak ma to miejsce w przypadku rozwiązania bazującego na operatorze *APPLY*.

Po wykonaniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXISTS dbo.EmpCustOrders;
```

Zbiory grupujące

Podrozdział ten wyjaśnia pojęcie zbiorów grupujących i jakie funkcje języka T-SQL obsługują te konstrukcje.

Zbiór grupujący (*grouping set*) to po prostu zestaw atrybutów używanych do grupowania. Powodem użycia terminu „zbiór” jest fakt, że kolejność, w jakiej występują wyrażenia w klauzuli *GROUP BY*, nie ma znaczenia. Tradycyjnie w języku SQL pojedyncze zapytanie grupujące definiuje jeden zbiór grupujący. Na przykład każde z poniższych zapytań definiuje pojedynczy zbiór grupujący:

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid;
```

```
SELECT empid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid;
```

```
SELECT custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY custid;
```

```
SELECT SUM(qty) AS sumqty
FROM dbo.Orders;
```

Pierwsze zapytanie definiuje zbiór grupujący (*empid, custid*); drugie (*empid*), trzecie (*custid*); ostatnie zapytanie definiuje konstrukcję, która nazywana jest pustym zbiorem grupującym – *()*. Kod ten zwraca cztery zbiory wyników – po jednym dla każdego z czterech zapytań.

Załóżmy, że na potrzeby tworzenia raportu zamiast czterech oddzielnych zbiorów wyników chcemy uzyskać pojedynczy zunifikowany zbiór wyników. Zadanie takie można zrealizować za pomocą operacji zbiorowej *UNION ALL*, scalając zbiory wyników wszystkich czterech zapytań. Ponieważ operacje zbiorowe wymagają, by wszystkie

zbiory wyników miały zgodne schematy o tej samej liczbie i typach kolumn, trzeba dostosować zapytania poprzez dodanie wypełniaczy (na przykład znaczników *NULL*) dla kolumn brakujących w pewnych zapytaniach, które występują w innych. Poniżej przedstawiono przykład takiego kodu.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid

UNION ALL

SELECT empid, NULL, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid

UNION ALL

SELECT NULL, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY custid

UNION ALL

SELECT NULL, NULL, SUM(qty) AS sumqty
FROM dbo.Orders;
```

Kod ten generuje pojedynczy zestaw wyników zagregowanych dla wszystkich czterech zbiorów grupujących.

empid	custid	sumqty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30
1	NULL	54
2	NULL	79
3	NULL	72
NULL	A	72
NULL	B	47
NULL	C	56
NULL	D	30
NULL	NULL	205

(15 row(s) affected)

Chociaż zrealizowaliśmy zaplanowane zadanie, rozwiązanie to ma dwie wady – długość kodu i wydajność. Rozwiązanie to wymaga wypisania całego zapytania dla każdego zbioru grupującego. W przypadku dużej liczby zbiorów grupujących kod staje się bardzo długi. Ponadto w tym rozwiązaniu system SQL Server będzie skanował tabelę źródłową oddzielnie dla każdego cząstkowego zapytania, co zdecydowanie nie jest efektywnym rozwiązaniem.

Język T-SQL udostępnia kilka funkcji, które są zgodne ze standardem SQL i pozwalają na zdefiniowanie w jednym zapytaniu wielu zbiorów grupujących. Są to klauzule pomocnicze *GROUPING SETS*, *CUBE* i *ROLLUP* klauzuli *GROUP BY* oraz funkcje *GROUPING* i *GROUPING_ID*. Ich główne zastosowanie to raportowanie i analiza danych. Funkcje te zazwyczaj wymagają od warstwy prezentacji stosowanie bardziej wyrafinowanych kontrolerek interfejsu użytkownika do wyświetlania danych, niż typowa siatka z kolumnami i wierszami. Jednak w tej książce skupiam się na kodzie T-SQL, a nie na warstwie prezentacji, zatem pominąłem ten element.

Klauzula pomocnicza *GROUPING SETS*

Klauzula pomocnicza *GROUPING SETS* to potężne rozszerzenie klauzuli *GROUP BY*. Pozwala ona w jednym zapytaniu zdefiniować wiele zestawów grupujących. Po prostu wymieniamy zestawy grupujące, które chcemy zdefiniować, oddzielając je przecinkami wewnątrz nawiasów podklauzuli *GROUPING SETS* i dla każdego zbioru grupującego wymieniamy jego elementy składowe oddzielane przecinkami wewnątrz nawiasów. Na przykład poniższe zapytanie definiuje cztery zbiory grupujące: (*empid*, *custid*), (*empid*), (*custid*) i ().

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY
    GROUPING SETS
    (
        (empid, custid),
        (empid),
        (custid),
        ()
    );
```

Zapytanie to jest ekwiwalentem logicznym poprzedniego rozwiązania, które scala zbiory wyników czterech zapytań i zwraca taki sam wynik. Zapytanie to w porównaniu do poprzedniego rozwiązania ma dwie istotne zalety – jej kod jest znacznie krótszy, a SQL Server będzie mógł zoptymalizować liczbę przeszukiwań tabeli źródłowej i nie będzie musiał skanować tabeli oddzielnie dla każdego zbioru grupującego.

Klauzula pomocnicza *CUBE*

Klauzula pomocnicza *CUBE* klauzuli *GROUP BY* udostępnia skróconą metodę definiowania wielu zbiorów grupujących. W nawiasach klauzuli pomocniczej *CUBE* wprowadzamy listę elementów zbioru, oddzielane przecinkami i na podstawie tych elementów wejściowych klauzula definiuje wszystkie możliwe zbiory grupujące. Na przykład, klauzula *CUBE*(*a*, *b*, *c*) jest równoważna klauzuli *GROUPING SETS*((*a*, *b*, *c*), (*a*, *b*), (*a*, *c*), (*b*, *c*), (*a*), (*b*), (*c*), ()). W teorii mnogości zbiór wszystkich podzbiorów, które mogą być utworzone dla danego zbioru, nazywany jest *zbiorem potęgowym* (ang. *power*

set). Klauzulę pomocniczą *CUBE* możemy więc traktować jako utworzenie zbioru potęgowego zbiorów grupujących, które można utworzyć na podstawie danego zbioru elementów.

Zamiast używania w poprzednim zapytaniu klauzuli pomocniczej *GROUPING SETS* do zdefiniowania czterech zbiorów grupujących (*empid*, *custid*), (*empid*), (*custid*) i *()*, możemy użyć prostej klauzuli *CUBE(empid, custid)*, co ilustruje poniższy przykład kodu:

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Klauzula pomocnicza *ROLLUP*

Klauzula pomocnicza *ROLLUP* klauzuli *GROUP BY* również służy do skrótowego definiowania wielu zbiorów grupujących. Inaczej niż w przypadku klauzuli *CUBE*, klauzula *ROLLUP* nie tworzy wszystkich możliwych zbiorów grupujących, które można zdefiniować na podstawie elementów wejściowych – klauzula ta tworzy tylko pewien podzbiór tych zbiorów grupujących. Klauzula *ROLLUP* zakłada istnienie hierarchii pomiędzy elementami wejściowymi i tworzy wszystkie zbiory grupujące, które mają sens, jeśli uwzględnimy tę hierarchię. Dla przykładu, na podstawie trzech elementów wejściowych klauzula *CUBE(a, b, c)* tworzy wszystkie osiem możliwych zbiorów grupujących, natomiast klauzula *ROLLUP(a, b, c)* utworzy tylko cztery zbiory grupujące, zakładając, że istnieje hierarchia $a > b > c$, czyli klauzula ta jest ekwiwalentem wyspecyfikowania wyrażenia *GROUPING SETS*(*(a, b, c), (a, b), (a), ()*).

Załóżmy na przykład, że chcemy zwrócić łączną ilość dla wszystkich zbiorów grupujących, które można zdefiniować na podstawie hierarchii związanej z czasem – rok > miesiąc > dzień zamówienia. W takim przypadku możemy użyć klauzuli pomocniczej *GROUPING SETS* i wprost wymienić wszystkie cztery możliwe zbiory grupujące.

```
GROUPING SETS(
    (YEAR(orderdate), MONTH(orderdate), DAY(orderdate)),
    (YEAR(orderdate), MONTH(orderdate)),
    (YEAR(orderdate)),
    () )
```

Ekwiwalentem logicznym używającym klauzuli *ROLLUP* jest znacznie bardziej „oszczędny” kod:

```
ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))
```

Oto całe zapytanie:

```
SELECT
    YEAR(orderdate) AS orderyear,
    MONTH(orderdate) AS ordermonth,
    DAY(orderdate) AS orderday,
    SUM(qty) AS sumqty
```

```
FROM dbo.Orders
GROUP BY ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

Zapytanie generuje następujące dane wyjściowe:

orderyear	ordermonth	orderday	sumqty
-----	-----	-----	-----
2015	4	18	22
2015	4	NULL	22
2015	8	2	10
2015	8	NULL	10
2015	12	24	32
2015	12	NULL	32
2015	NULL	NULL	64
2016	1	9	40
2016	1	18	14
2016	1	NULL	54
2016	2	12	12
2016	2	NULL	12
2016	NULL	NULL	66
2009	2	12	10
2009	2	16	20
2009	2	NULL	30
2009	4	18	15
2009	4	NULL	15
2009	9	7	30
2009	9	NULL	30
2009	NULL	NULL	75
NULL	NULL	NULL	205

Funkcje *GROUPING* i *GROUPING_ID*

Jeśli mamy pojedyncze zapytanie, które definiuje wiele zbiorów grupujących, może zachodzić potrzeba powiązania wierszy wyników z tymi zbiorami grupującymi – inaczej mówiąc, dla każdego wiersza wyników chcemy zidentyfikować powiązany z nim zbiór grupujący. O ile wszystkie elementy grupujące zostały zdefiniowane jako *NOT NULL*, zadanie jest proste. Przeanalizujmy dla przykładu następujące zapytanie:

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Zapytanie to generuje następujące wyniki:

empid	custid	sumqty
-----	-----	-----
2	A	52
3	A	20
NULL	A	72
1	B	20
2	B	27
NULL	B	47

1	C	34
3	C	22
NULL	C	56
3	D	30
NULL	D	30
NULL	NULL	205
1	NULL	54
2	NULL	79
3	NULL	72

Ponieważ zarówno kolumna *empid*, jak i *custid* zostały zdefiniowane w tabeli *dbo.Orders* jako *NOT NULL*, znacznik *NULL* w tych kolumnach może reprezentować tylko wypełniacz wskazujący, że kolumna nie uczestniczy w bieżącym zbiorze grupującym. Tak więc w tym przykładzie wszystkie wiersze, gdzie *empid* i *custid* nie mają wartości *NULL*, są powiązane ze zbiorem grupującym (*empid*, *custid*). Wszystkie wiersze, w których *empid* nie ma wartości *NULL*, a *custid* ma wartość *NULL*, są powiązane ze zbiorem grupującym (*empid*) itd.

Jeśli jednak kolumny grupujące w tabeli dopuszczają znaczniki *NULL*, nie możemy rozstrzygnąć, czy znaczniki *NULL* w zbiorze wyników to dane, czy wypełniacze dla elementu nienależącego do zbioru grupującego. Sposobem określenia powiązań zbioru grupującego w sposób jednoznaczny, nawet jeśli kolumny grupujące dopuszczają stosowanie znaczników *NULL*, jest użycie funkcji *GROUPING*. Funkcja ta akceptuje nazwę kolumny i zwraca 0, jeśli element należy do bieżącego zbioru grupującego; w przeciwnym razie funkcja zwraca 1.



UWAGA To, że funkcja *GROUPING* zwraca 1, jeśli element nie należy do zbioru grupującego i 0, jeśli jest jego częścią, wydaje się nieintuicyjne. W mojej opinii bardziej sensowne byłoby zwracanie 1 (oznaczenie prawdy), jeśli element należy do zestawu grupującego, i 0 w przeciwnej sytuacji. Faktyczne podejście jest jednak takie, że 1 oznacza element agregowany, a 0 – grupujący. Po prostu trzeba o tym pamiętać.

Dla przykładu poniższe zapytanie wywołuje funkcję *GROUPING* dla każdego elementu grupującego.

```
SELECT
    GROUPING(empid) AS grpemp,
    GROUPING(custid) AS grpcust,
    empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Zapytanie to zwraca następujące wyniki:

grpemp	grpcust	empid	custid	sumqty

0	0	2	A	52
0	0	3	A	20

1	0	NULL	A	72
0	0	1	B	20
0	0	2	B	27
1	0	NULL	B	47
0	0	1	C	34
0	0	3	C	22
1	0	NULL	C	56
0	0	3	D	30
1	0	NULL	D	30
1	1	NULL	NULL	205
0	1	1	NULL	54
0	1	2	NULL	79
0	1	3	NULL	72

(15 row(s) affected)

Teraz nie musimy już polegać na znacznikach *NULL*, by określić powiązanie pomiędzy wierszami wyników a zbiorami grupującymi. Na przykład wszystkie wiersze, gdzie *grpemp* i *grpcust* mają wartość 0, są powiązane ze zbiorem grupującym (*empid*, *custid*). Wszystkie wiersze, gdzie *grpemp* to 0, a *grpcust* to 1, są powiązane ze zbiorem grupującym (*empid*) itd.

Język T-SQL obsługuje także inną funkcję nazwaną *GROUPING_ID*, która jeszcze bardziej upraszcza proces wyszukiwania powiązań pomiędzy wierszami wyników a zbiorami grupującymi. Na wejściu funkcji dostarczamy wszystkie elementy, które należą do jakiegokolwiek zbioru grupującego – na przykład *GROUPING_ID(a, b, c, d)* – a funkcja zwraca mapę bitową (liczbę całkowitą), w której każdy bit reprezentuje inny element wejściowy – skrajny element z prawej strony reprezentowany jest przez skrajny prawy bit. Również tu przynależność do zbioru grupującego jest oznaczana przez 0, a 1 oznacza, że dany element nie należy do zbioru. Na przykład zbiór grupujący (*a, b, c, d*) jest reprezentowany przez liczbę całkowitą 0 ($0 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1$). Zbiorowi grupującemu (*a, c*) odpowiada liczba 5 ($0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$) itd.

Zamiast wywoływać funkcję *GROUPING* dla każdego elementu grupującego, jak w poprzednim zapytaniu, możemy tylko raz wywołać funkcję *GROUPING_ID* i dostarczyć jej na wejściu wszystkie elementy grupujące, jak w poniższym przykładzie kodu:

```
SELECT
    GROUPING_ID(empid, custid) AS groupingset,
    empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Zapytanie to generuje następując wyniki:

groupingset	empid	custid	sumqty
0	2	A	52
0	3	A	20
2	NULL	A	72
0	1	B	20
0	2	B	27

2	NULL	B	47
0	1	C	34
0	3	C	22
2	NULL	C	56
0	3	D	30
2	NULL	D	30
3	NULL	NULL	205
1	1	NULL	54
1	2	NULL	79
1	3	NULL	72

Obecnie łatwo możemy określić, z którym zbiorem grupującym powiązany jest każdy wiersz. Liczba całkowita 0 (binarnie 00) reprezentuje zbiór grupujący (*empid*, *custid*); liczba 1 (binarnie 01) reprezentuje (*empid*); liczba 2 (binarnie 10) reprezentuje (*custid*); a liczba 3 (binarnie 11) reprezentuje zbiór pusty ().

Podsumowanie

W rozdziale tym omówiłem funkcje okna, przestawianie i odwrotne przestawianie danych oraz zbiory grupujące.

Funkcje okna, w porównaniu do alternatywnych metod, pozwalają wykonywać analizę danych bardziej elastycznie i wydajnie. Funkcje okna mają wiele praktycznych zastosowań, warto więc poświęcić czas i dobrze poznać ich działanie.

Przedstawiłem także zarówno standardowe, jak i niestandardowe metody przestawiania danych i odwrotnego przestawiania danych. Niestandardowe metody korzystają z operatorów specyficznych dla języka T-SQL – *PIVOT* i *UNPIVOT*; ich główną zaletą w porównaniu do metod standardowych jest skrócenie kodu.

Język T-SQL udostępnia kilka funkcji pozwalających na elastyczne i efektywne obsługiwanie zbiorów grupujących: klauzule pomocnicze *GROUPING SETS*, *CUBE* i *ROLLUP* oraz funkcje *GROUPING* i *GROUPING_ID*.

Ćwiczenia

Podrozdział ten zawiera ćwiczenia, które ułatwią lepsze przyswojenie tematyki opisanej w rozdziale. We wszystkich ćwiczeniach w tym rozdziale odpyttywana jest tabela *dbo.Orders*, utworzona w bazie danych TSQV4 poprzez uruchomienie kodu z listingu 7-1.

Ćwiczenie 1

Napisać zapytanie tabeli *dbo.Orders*, które dla każdego zamówienia klienta wyliczy funkcje rankingowe *RANK* i *DENSE_RANK*, z partycjonowaniem według *custid* i uporządkowaniem według *qty*.

- Wykorzystywane tabele: baza danych TSQV4 i tabela *dbo.Orders*

- Oczekiwane dane wyjściowe:

custid	orderid	qty	rnk	drnk
-----	-----	-----	----	-----
A	30001	10	1	1
A	40005	10	1	1
A	10001	12	3	2
A	40001	40	4	3
B	20001	12	1	1
B	30003	15	2	2
B	10005	20	3	3
C	10006	14	1	1
C	20002	20	2	2
C	30004	22	3	3
D	30007	30	1	1

Ćwiczenie 2

W podrozdziale „Klasyfikujące funkcje okna” zamieściłem poniższe zapytanie do widoku *Sales.OrderValues*, zwracające różne wartości i odpowiadające im numery wierszy:

```
SELECT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues
GROUP BY val;
```

Jak inaczej można zrealizować to zadanie?

- Wykorzystywane tabele: baza danych TSQVLV4 i widok *Sales.OrderValues*
- Oczekiwane dane wyjściowe:

val	rownum
-----	-----
12.50	1
18.40	2
23.80	3
28.00	4
30.00	5
33.75	6
36.00	7
40.00	8
45.00	9
48.00	10
...	
12615.05	793
15810.00	794
16387.50	795

(795 row(s) affected)

Ćwiczenie 3

Napisać zapytanie tabeli *dbo.Orders*, które dla każdego zamówienia klienta oblicza różnicę pomiędzy ilością bieżącego zamówienia a ilością poprzedniego zamówienia klienta oraz różnicę pomiędzy ilością bieżącego zamówienia i ilością następnego zamówienia klienta.

- Wykorzystywane tabele: baza danych TSQVLV4 i tabela *dbo.Orders*
- Oczekiwane dane wyjściowe:

custid	orderid	qty	diffprev	diffnext
A	30001	10	NULL	-2
A	10001	12	2	-28
A	40001	40	28	30
A	40005	10	-30	NULL
B	10005	20	NULL	8
B	20001	12	-8	-3
B	30003	15	3	NULL
C	30004	22	NULL	8
C	10006	14	-8	-6
C	20002	20	6	NULL
D	30007	30	NULL	NULL

Ćwiczenie 4

Napisać zapytanie tabeli *dbo.Orders*, która zwraca wiersz dla każdego pracownika, kolumnę dla każdego roku zamówienia i liczbę zamówień dla każdego pracownika i roku zamówienia.

- Wykorzystywane tabele: baza danych TSQVLV4 i tabela *dbo.Orders*
- Oczekiwane dane wyjściowe:

empid	cnt2015	cnt2016	cnt2009
1	1	1	1
2	1	2	1
3	2	0	2

Ćwiczenie 5

Uruchomić poniższy kod, by utworzyć tabelę *EmpYearOrders* i wypełnić ją danymi.

```
USE TSQVLV4;
```

```
IF OBJECT_ID('dbo.EmpYearOrders', 'U') IS NOT NULL DROP TABLE dbo.EmpYearOrders;
```

```
CREATE TABLE dbo.EmpYearOrders
(
    empid INT NOT NULL
```



```

        CONSTRAINT PK_EmpYearOrders PRIMARY KEY,
        cnt2015 INT NULL,
        cnt2016 INT NULL,
        cnt2009 INT NULL
    );

INSERT INTO dbo.EmpYearOrders(empid, cnt2015, cnt2016, cnt2009)
    SELECT empid, [2015] AS cnt2015, [2016] AS cnt2016, [2009] AS cnt2009
    FROM (SELECT empid, YEAR(orderdate) AS orderyear
        FROM dbo.Orders) AS D
        PIVOT(COUNT(orderyear)
            FOR orderyear IN([2015], [2016], [2009])) AS P;

SELECT * FROM dbo.EmpYearOrders;

```

Dane wyjściowe zapytania są następujące:

```

empid cnt2015 cnt2016 cnt2009
-----
1 1 1 1
2 1 2 1
3 2 0 2

```

Napisać zapytanie tabeli *EmpYearOrders*, które przeprowadza odwrotne przestawienie danych, zwraca wiersz dla każdego pracownika i roku zamówienia wraz z liczbą zamówień. Należy wykluczyć wiersze, w których liczba zamówień wynosi 0 (w tym przykładzie pracownik 3 w roku 2016).

■ Oczekiwane dane wyjściowe:

empid	orderyear	numorders
1	2015	1
1	2016	1
1	2009	1
2	2015	1
2	2016	2
2	2009	1
3	2015	2
3	2009	2

Ćwiczenie 6

Napisać zapytanie tabeli *dbo.Orders*, które zwraca łączną ilość dla każdego zbioru grupującego: (pracownik, klient i rok zamówienia), (pracownik i rok zamówienia) oraz (klient i rok zamówienia). W wynikach dołączyć kolumnę, która jednoznacznie identyfikuje zbiór grupujący, z którym powiązany jest bieżący wiersz.

- Wykorzystywane tabele: baza danych TSQVLV4 i tabela *dbo.Orders*
- Oczekiwane dane wyjściowe:

groupingset	empid	custid	orderyear	sumqty
0	2	A	2015	12
0	3	A	2015	10
4	NULL	A	2015	22
0	2	A	2016	40
4	NULL	A	2016	40
0	3	A	2009	10
4	NULL	A	2009	10
0	1	B	2015	20
4	NULL	B	2015	20
0	2	B	2016	12
4	NULL	B	2016	12
0	2	B	2009	15
4	NULL	B	2009	15
0	3	C	2015	22
4	NULL	C	2015	22
0	1	C	2016	14
4	NULL	C	2016	14
0	1	C	2009	20
4	NULL	C	2009	20
0	3	D	2009	30
4	NULL	D	2009	30
2	1	NULL	2015	20
2	2	NULL	2015	12
2	3	NULL	2015	32
2	1	NULL	2016	14
2	2	NULL	2016	52
2	1	NULL	2009	20
2	2	NULL	2009	15
2	3	NULL	2009	40

(29 row(s) affected)

Po wykonaniu ćwiczeń z tego rozdziału należy uruchomić poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXIST dbo.Orders;
```

Rozwiązania

W tym podrozdziale udostępniono rozwiązania ćwiczeń wraz z odpowiednimi wyjaśnieniami.

Ćwiczenie 1

Ćwiczenie to ma bardzo techniczny charakter. Kwestią jest znajomość składników funkcji okien dotyczących rankingu. Poniżej pokazano zapytanie, która zwraca dla każdego zamówienia zarówno wartość funkcji *RANK*, jak *DENSE RANK*, przy partycjonowaniu według atrybutu *custid* i uporządkowaniu według *qty*.

```
SELECT custid, orderid, qty,
       RANK() OVER(PARTITION BY custid ORDER BY qty) AS rnk,
       DENSE_RANK() OVER(PARTITION BY custid ORDER BY qty) AS drnk
FROM dbo.Orders;
```

Ćwiczenie 2

Inną metodą rozwiązania problemu jest napisanie zapytania zwracającego różne wartości bez obliczania numerów wierszy, utworzenie na jego podstawie wyrażenia tablicowego, po czym wyliczenie numerów wierszy w zapytaniu zewnętrznym do tego wyrażenia tablicowego. Oto rozwiązanie:

```
WITH C AS
(
    SELECT DISTINCT val
    FROM Sales.OrderValues
)
SELECT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM C;
```

Ćwiczenie 3

Funkcje okna przesunięcia – *LAG* i *LEAD* – pozwalają zwrócić element odpowiednio z poprzedniego lub następnego wiersza na podstawie wskazanego partycjonowania i uporządkowania. W ćwiczeniu tym trzeba wykonać obliczenia dla zamówień poszczególnych klientów, zatem partycjonowanie oparte jest na atrybucie *custid*. Podobnie jest w przypadku definiowania kolejności – używamy atrybutu *orderdate* jako pierwszej kolumny sortowania i *orderid* jako kryterium rozstrzygania. Poniżej zaprezentowane zostało całe rozwiązanie:

```
SELECT custid, orderid, qty,
       qty - LAG(qty) OVER(PARTITION BY custid
                          ORDER BY orderdate, orderid) AS diffprev,
       qty - LEAD(qty) OVER(PARTITION BY custid
```

```
ORDER BY orderdate, orderid) AS diffnext
FROM dbo.Orders;
```

Zapytanie to jest dobrym przykładem ilustrującym możliwość łączenia w tym samym wyrażeniu elementów szczegółowych wiersza z funkcjami okien.

Ćwiczenie 4

Rozwiązywanie problemów przedstawiania danych to przede wszystkim zidentyfikowanie elementów zaangażowanych w ten proces: element grupowania, element rozpraszania, element agregowania i funkcja agregująca. Po zidentyfikowaniu używanych elementów trzeba je po prostu umieścić w „szablonie” zapytania przedstawiania danych – niezależnie od tego, czy jest to rozwiązanie standardowe, czy rozwiązanie z użyciem niestandardowego operatora *PIVOT*.

W ćwiczeniu tym elementem grupującym jest pracownik (*empid*), elementem rozpraszającym jest rok zamówienia (*YEAR(orderdate)*), a funkcją agregującą *COUNT*; zidentyfikowanie elementu agregowania nie jest jednak takie proste. Funkcji agregującej *COUNT* chcemy użyć do policzenia zgodnych wierszy i zamówień – tak naprawdę nie interesuje nas, który atrybut jest zliczany. Inaczej mówiąc, możemy użyć dowolnego atrybutu, o ile atrybut ten nie zezwala na użycie znaczników *NULL*, ponieważ funkcje agregujące ignorują znaczniki *NULL*, tak więc w przypadku zliczania atrybutu, który dopuszcza znaczniki *NULL*, otrzymana liczba zamówień będzie nieprawidłowa.

Jeśli nie ma znaczenia, który atrybut będzie użyty jako wejście funkcji agregującej *COUNT*, dlaczego nie użyć tego samego atrybutu, który został już użyty jako element rozpraszania? W tym przypadku możemy użyć roku zamówienia zarówno jako elementu rozpraszania, jak i agregowania.

Teraz, po zdefiniowaniu wszystkich elementów przedstawiania danych, możemy już napisać całe rozwiązanie. Poniższe zapytanie nie korzysta z operatora *PIVOT*.

```
USE TSQLV4;

SELECT empid,
       COUNT(CASE WHEN orderyear = 2014 THEN orderyear END) AS cnt2015,
       COUNT(CASE WHEN orderyear = 2015 THEN orderyear END) AS cnt2016,
       COUNT(CASE WHEN orderyear = 2016 THEN orderyear END) AS cnt2009
FROM (SELECT empid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
GROUP BY empid;
```

Pamiętamy, że jeśli wyrażenie *CASE* nie zawiera klauzuli *ELSE*, zakłada się użycie wyrażenia *ELSE NULL*. Tak więc wyrażenie *CASE* generuje wartości inne niż *NULL* tylko dla zgodnych zamówień (zamówień złożonych przez bieżącego pracownika w bieżącym roku) i tylko te zgodne zamówienia uwzględniane są przez funkcję agregującą *COUNT*.

Zwróćmy uwagę, że chociaż rozwiązanie standardowe nie wymaga stosowania wyrażenia tablicowego, wyrażenie takie zostało jednak użyte w przykładzie, by utworzyć

alias wyrażenia *YEAR(orderdate)*, czyli *orderyear* i uniknąć wielokrotnego powtarzania wyrażenia *YEAR(orderdate)* w zapytaniu zewnętrznym.

A oto rozwiązanie, które wykorzystuje niestandardowy operator *PIVOT*.

```
SELECT empid, [2014] AS cnt2014, [2015] AS cnt2015, [2016] AS cnt2016
FROM (SELECT empid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
PIVOT(COUNT(orderyear)
      FOR orderyear IN([2014], [2015], [2016])) AS P;
```

Jak widzimy, kwestia polega jedynie na umieszczeniu elementów przedstawiania danych we właściwych miejscach.

Jeśli wolimy używać własnych nazw kolumn docelowych, a nie nazw opartych na rzeczywistych danych, możemy rzecz jasna wprowadzić własne aliasy na liście *SELECT*. W tym zapytaniu utworzone zostały aliasy kolumn wyników *[2014]*, *[2015]* i *[2016]* odpowiednio jako *cnt2014*, *cnt2015* i *cnt2016*.

Ćwiczenie 5

Ćwiczenie to wiąże się ze zrealizowaniem żądania odwrotnego przestawienia danych, czyli zamianą kolumn źródłowych *cnt2014*, *cnt2015* i *cnt2016* na dwie kolumny docelowe – *orderyear*, w której przechowywany będzie rok reprezentowany przez nazwę kolumny źródłowej i *numorders* do przechowywania wartości kolumny źródłowej. Możemy użyć rozwiązania zaprezentowanego w rozdziale jako podstawy rozwiązania tego ćwiczenia i wprowadzić w nim niewielkie zmiany.

W przykładach użytych w rozdziale znaczniki *NULL* w tabeli reprezentowały nieistotne wartości kolumn. Prezentowane rozwiązania odwrotnego przestawiania danych odrzucały wiersze ze znacznikami *NULL*. Tabela *EmpYearOrders* nie ma znaczników *NULL*, ale w niektórych przypadkach ma wartość zero, a żądanie określa, by odfiltrować wiersze, dla których liczba zamówień wynosi 0. W przypadku rozwiązania standardowego po prostu używamy predykatu *numorders <> 0* zamiast wyrażenia *IS NOT NULL*. Poniżej wersja rozwiązania, która używa klauzuli *VALUES*.

```
SELECT *
FROM (SELECT empid, orderyear,
      CASE orderyear
        WHEN 2015 THEN cnt2014
        WHEN 2016 THEN cnt2015
        WHEN 2009 THEN cnt2016
      END AS numorders
      FROM dbo.EmpYearOrders
      CROSS JOIN (VALUES(2014),(2015),(2016)) AS Years (orderyear)) AS D
WHERE numorders <> 0;
```

Podobnie jak w przypadku rozwiązania stosującego niestandardowy operator *UNPIVOT*, pamiętajmy, że jako integralna część jego logiki eliminowane są znaczniki

NULL. Nie są jednak eliminowane zera – musimy pamiętać, by samemu je odrzucić, dodając klauzulę *WHERE*, jak w poniższym kodzie:

```
SELECT empid, CAST(RIGHT(orderyear, 4) AS INT) AS orderyear, numorders
FROM dbo.EmpYearOrders
UNPIVOT(numorders FOR orderyear IN(cnt2014, cnt2015, cnt2016)) AS U
WHERE numorders <> 0;
```

Zwróćmy uwagę na wyrażenie użyte na liście *SELECT* do wygenerowania kolumny wyników *orderyear*: *CAST(RIGHT(orderyear, 4) AS INT)*. Nazwy oryginalnych kolumn, które poddawane są odwrotnemu przestawianiu danych, to: *cnt2014*, *cnt2015* i *cnt2016*. Nazwy tych kolumn stają się odpowiednio wartościami *'cnt2014'*, *'cnt2015'* i *'cnt2016'* w kolumnie *orderyear* w wynikach operatora *UNPIVOT*. Funkcją tego wyrażenia jest uzyskanie czterech znaków skrajnych z prawej strony, reprezentujących rok zamówienia i przekształcenie tej wartości na liczbę całkowitą. Operacja ta nie jest wymagana w przypadku rozwiązania standardowego, ponieważ stałe użyte do konstruowania wyrażenia tablicowego *Years* były wyspecyfikowane tak, by zaczynały się od liczby całkowitej roku zamówienia.

Ćwiczenie 6

Stosujemy klauzulę pomocniczą *GROUPING SETS* do wymienienia listy żądanych zbiorów grupujących i funkcję *GROUPING_ID*, by wygenerować unikatowy identyfikator dla zbioru grupującego, z którym powiązany jest każdy wiersz. Poniżej przedstawiono całe rozwiązanie:

```
SELECT
    GROUPING_ID(empid, custid, YEAR(Orderdate)) AS groupingset,
    empid, custid, YEAR(Orderdate) AS orderyear, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY
    GROUPING SETS
    (
        (empid, custid, YEAR(orderdate)),
        (empid, YEAR(orderdate)),
        (custid, YEAR(orderdate))
    );
```

Żądane zbiory grupujące nie są ani zbiorem potęgowym, ani nie jest zakładane istnienie hierarchii pewnego zbioru atrybutów. Z tego względu nie możemy użyć klauzul pomocniczych *CUBE* lub *ROLLUP*, by jeszcze bardziej skrócić kod.

ROZDZIAŁ 8

Modyfikowanie danych

Język SQL zawiera podzbiór instrukcji określanych mianem DML (Data Manipulation Language), pozwalających na modyfikowanie danych. Dość często można spotkać pogląd, że kategoria DML zawiera tylko te instrukcje, które modyfikują dane, ale w rzeczywistości uwzględnia ona także polecenia odczytywania danych. Do kategorii DML należą instrukcje *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *TRUNCATE* i *MERGE* oraz związane z nimi klauzule i operatory. Jak dotąd, skupiałem się na instrukcji *SELECT*. W tym rozdziale zajmiemy się instrukcjami modyfikowania danych. Oprócz omówienia zawartych w standardzie aspektów modyfikowania danych opiszę także cechy specyficzne dla języka T-SQL.

Aby zapobiec zmodyfikowaniu istniejących danych w naszej przykładowej bazie danych, większość przykładów pokazanych w tym rozdziale tworzy, wypełnia i modyfikuje tabele w schemacie *dbo* bazy danych TSQV4.

Wstawianie danych

Język T-SQL udostępnia kilka konstrukcji umożliwiających wstawianie danych do tabel: *INSERT VALUES*, *INSERT SELECT*, *INSERT EXEC*, *SELECT INTO* i *BULK INSERT*. Najpierw przedstawię te instrukcje, a następnie przejdę do narzędzi umożliwiających automatyczne generowanie kluczy, takich jak właściwość kolumny *identity* i obiekt sekwencji.

Wyrażenie *INSERT VALUES*

Instrukcja *INSERT VALUES* służy do wstawiania do tabeli wierszy zawierających określone wartości. Do zademonstrowania działania tej (i innych) instrukcji będziemy wykorzystywać tabelę nazwaną *Orders* w schemacie *dbo* bazy danych TSQV4. Poniższy kod tworzy tabelę *Orders*.

```
USE TSQV4;

DROP TABLE IF EXIST dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid INT NOT NULL
    CONSTRAINT PK_Orders PRIMARY KEY,
    orderdate DATE NOT NULL
```

```

        CONSTRAINT DFT_orderdate DEFAULT(SYSDATETIME()),
empid      INT          NOT NULL,
custid     VARCHAR(10) NOT NULL
)

```

Pokazany poniżej przykład kodu pokazuje użycie instrukcji *INSERT VALUES* do wstawienia pojedynczego wiersza w tabeli *Orders*.

```

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
VALUES(10001, '20160212', 3, 'A');

```

Wskazanie nazw kolumn docelowych zaraz po nazwie tabeli jest opcjonalne, jednak umożliwia kontrolowanie powiązania wartości z kolumnami, bez polegania na kolejności, w jakiej kolumny występowały podczas definiowania tabeli (lub w ostatniej zmianie struktury tabeli). W dialekcie T-SQL użycie klauzuli *INTO* jest opcjonalne.

Jeśli wartość dla pewnej kolumny nie zostanie określona, SQL Server sprawdzi, czy dla kolumny została zdefiniowana wartość domyślna i jeśli tak, użyje tej wartości. Jeśli wartość domyślna nie została zdefiniowana, a kolumna dopuszcza znaczniki *NULL*, użyty zostanie *NULL*. Jeśli dla kolumny nie została zdefiniowana wartość domyślna i kolumna ta nie dopuszcza stosowania znaczników *NULL* (czyli i nie wiadomo, skąd automatycznie uzyskać potrzebną wartość), instrukcja *INSERT* zakończy się niepowodzeniem. Poniższa instrukcja wstawia do tabeli *Orders* wiersz bez specyfikowania wartości dla kolumny *orderdate*; ponieważ jednak kolumna ta ma zdefiniowane wyrażenie domyślne (*SYSDATETIME*), użyta zostanie ta wartość domyślna.

```

INSERT INTO dbo.Orders(orderid, empid, custid)
VALUES(10002, 5, 'B');

```

Język T-SQL rozszerza działanie klauzuli *VALUES*, pozwalając na wyspecyfikowanie wartości dla wielu wierszy, rozdzielanych przecinkami. Na przykład poniższa instrukcja wstawia cztery wiersze do tabeli *Orders*.

```

INSERT INTO dbo.Orders
(orderid, orderdate, empid, custid)
VALUES
(10003, '20160213', 4, 'B'),
(10004, '20160214', 1, 'A'),
(10005, '20160213', 1, 'C'),
(10006, '20160215', 3, 'C');

```

Instrukcja ta jest przetwarzana jako transakcja, czyli jako operacja niepodzielna, co oznacza, że jeśli nie powiedzie się wstawienie choć jednego wiersza, żaden z wierszy nie zostanie wprowadzony do tabeli.

Rozszerzonej klauzuli *VALUES* można użyć również jako konstruktora tabeli pochodnej (wirtualnej), jak w poniższym przykładzie:

```

SELECT *
FROM ( VALUES
      (10003, '20160213', 4, 'B'),

```



```
(10004, '20160214', 1, 'A'),
(10005, '20160213', 1, 'C'),
(10006, '20160215', 3, 'C') )
AS O(orderid, orderdate, empid, custid);
```

Po zawartym w nawiasach konstruktorze tabeli przypisujemy alias tabeli (w tym przypadku *O*), uzupełniony o umieszczone w nawiasach nazwy (aliasy) kolumn wyników. Zapytanie to generuje następujące wyniki:

orderid	orderdate	empid	custid
10003	20160213	4	B
10004	20160214	1	A
10005	20160213	1	C
10006	20160215	3	C

Instrukcja *INSERT SELECT*

Instrukcja *INSERT SELECT* wstawia do tabeli docelowej zbiór wierszy zwrócony przez zapytanie *SELECT*. Składnia jest bardzo podobna do składni instrukcji *INSERT VALUES*, ale zamiast klauzuli *VALUES* używamy zapytania *SELECT*. Dla przykładu poniższy kod wstawia do tabeli *dbo.Orders* wyniki zapytania do tabeli *Sales.Orders*, zwracającego zamówienia wysłane do Wielkiej Brytanii.

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE shipcountry = 'UK';
```

Instrukcja *INSERT SELECT* również pozwala określać nazwy kolumn docelowych i zalecenia dotyczące specyfikowania tych nazw przedstawione wcześniej się nie zmieniają. Również zasady dotyczące brakujących wartości pozostają takie same jak dla instrukcji *INSERT VALUES* – stosowane są wartości domyślne, o ile są zdefiniowane, albo znaczniki *NULL*, jeśli są dozwolone. Instrukcja *INSERT SELECT* jest wykonywana jako transakcja, tak więc również w tym przypadku, jeśli nie powiedzie się wstawienie jednego z wierszy, do tabeli docelowej nie zostanie wstawiony żaden wiersz.

UWAGA Jeśli w zapytaniu wstawiającym dane zostanie użyta funkcja systemowa, taka jak *SYSDATETIME*, funkcja ta zostanie wywołana tylko jeden raz dla całego zapytania, a nie dla każdego ze wstawianych wierszy. Wyjątkiem od tej reguły jest przypadek generowania globalnie unikatowych identyfikatorów (GUID) przy użyciu funkcji *NEWID*, która jest wywoływana dla każdego wstawianego wiersza.

Instrukcja *INSERT EXEC*

Instrukcja *INSERT EXEC* służy do wstawiania do tabeli docelowej zbioru wyników zwróconego przez procedurę składowaną lub dynamiczny wsad SQL. Informacje na temat procedur składowanych, operacji wsadowych i dynamicznego kodu SQL zawiera rozdział 11 „Obiekty programowalne”. Instrukcja *INSERT EXEC* ma bardzo podobną składnię i działanie do instrukcji *INSERT SELECT*, ale zamiast zapytania wykorzystującego *SELECT* zawiera polecenie *EXEC*.

Poniższy kod tworzy procedurę składowaną *Sales.usp_getorders*, zwracającą zamówienia wysłane do określonego kraju (wskazywanego w parametrze *@country*).

```

DROP PROC IF EXISTS Sales.GetOrders;
GO

CREATE PROC Sales.usp_getorders
    @country AS NVARCHAR(40)
AS

SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE shipcountry = @country;
GO

```

W celu przetestowania procedury składowanej wykonamy ją dla Francji.

```
EXEC Sales.usp_getorders @country = 'France';
```

Uzyskamy następujące wyniki:

orderid	orderdate	empid	custid
10248	2014-07-04 00:00:00.000	5	85
10251	2014-07-08 00:00:00.000	3	84
10265	2014-07-25 00:00:00.000	2	7
10274	2014-08-06 00:00:00.000	6	85
10295	2014-09-02 00:00:00.000	2	85
10297	2014-09-04 00:00:00.000	5	7
10311	2014-09-20 00:00:00.000	1	18
10331	2014-10-16 00:00:00.000	9	9
10334	2014-10-21 00:00:00.000	8	84
10340	2014-10-29 00:00:00.000	1	9
...			

(77 row(s) affected)

Przy użyciu instrukcji *INSERT EXEC* możemy wstawić do tabeli *dbo.Orders* zbiór wyników zwracany przez procedurę.

```

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
EXEC Sales.usp_getorders @country = 'France';

```

Instrukcja *SELECT INTO*

Instrukcja *SELECT INTO* jest niestandardową instrukcją języka T-SQL, która tworzy tabelę docelową i wypełnia ją zbiorem wyników kwerendy. Poprzez „niestandardowe” rozumie się, że nie należy do standardów ISO ani ANSI SQL. Nie można używać tej instrukcji do wstawiania danych do już istniejącej tabeli. Składnia jest prosta – dodajemy wyrażenie *INTO <nazwa_tabeli_docelowej>* tuż przed klauzulą *FROM* zapytania *SELECT* używanego do utworzenia zbioru wyników. Na przykład poniższy kod tworzy tabelę nazwaną *dbo.Orders* i wypełnia ją wszystkimi wierszami tabeli *Sales.Orders*.

```
DROP TABLE IF EXIST dbo.Orders;  
  
SELECT orderid, orderdate, empid, custid  
INTO dbo.Orders  
FROM Sales.Orders;
```

Struktura i dane tabeli docelowej opierają się na tabeli źródłowej. Instrukcja *SELECT INTO* kopiuje ze źródła strukturę bazową (nazwy kolumn, typy, możliwość stosowania znaczników *NULL* i właściwość *identity*) oraz dane. Instrukcja nie kopiuje ze źródła ograniczeń, indeksów, wyzwalaczy, takich właściwości kolumn, jak *SPARSE* czy *FILESTREAM* ani uprawnień. Jeśli są one potrzebne w tabeli docelowej, trzeba je samemu utworzyć.

Jedną z zalet stosowania instrukcji *SELECT INTO* jest jej wydajność. O ile model odzyskiwania (*Recovery Model*) bazy danych nie jest ustawiony jako *FULL*, operacje *SELECT INTO* są wykonywane przy minimalnym trybie rejestrowania. Dzięki temu operacje te są bardzo szybkie (w porównaniu do operacji przy pełnym zakresie rejestrowania). Można zauważyć, że instrukcją *INSERT SELECT* również korzysta z minimalnego rejestrowania, jednak dłuższa jest lista wymagań, które muszą być spełnione, aby zakończyła się powodzeniem. Szczegółowe informacje na ten temat znaleźć można w dokumencie „Prerequisites for Minimal Logging in Bulk Import” w dokumentacji SQL Server Books Online pod adresem <http://msdn.microsoft.com/en-us/library/ms190422.aspx>.

Jeśli chcemy użyć instrukcji *SELECT INTO* z operacjami na zbiorach, klauzulę *INTO* należy umieścić bezpośrednio przed klauzulą *FROM* pierwszego zapytania. Na przykład poniższa instrukcja *SELECT INTO* tworzy tabelę nazwaną *Locations* i wypełnia ją wynikami operacji zbiorowej *EXCEPT*, zwracającej lokalizacje, w których istnieją klienci, ale nie pracownicy.

```
DROP TABLE IF EXIST dbo.Locations;  
  
SELECT country, region, city  
INTO dbo.Locations  
FROM Sales.Customers  
  
EXCEPT  
  
SELECT country, region, city  
FROM HR.Employees;
```

Instrukcja **BULK INSERT**

Instrukcja **BULK INSERT** pozwala wstawić do istniejącej tabeli dane pochodzące z pliku. W instrukcji należy wskazać tabelę docelową, plik źródłowy oraz opcje. Dostępnych jest wiele opcji, takich jak typ pliku danych (*datafiletype*, na przykład *char* lub *native*), znak zakończenia pola (*fieldterminator*), zakończenia wiersza (*rowterminator*) i inne – wszystkie są w pełni udokumentowane.

Dla przykładu poniższy kod wstawia zawartość pliku *c:\temp\orders.txt* do tabeli *dbo.Orders*, określając, że typem pliku jest *char*, zakończeniem pola jest przecinek, a końcem wiersza znak nowego wiersza.

```
BULK INSERT dbo.Orders FROM 'c:\temp\orders.txt'
WITH
(
    DATAFILETYPE      = 'char',
    FIELDTERMINATOR    = ',',
    ROWTERMINATOR      = '\n'
);
```

Warto zauważyć, że aby móc rzeczywiście uruchomić tę instrukcję, trzeba w folderze *c:\temp* umieścić plik *orders.txt* dołączony do kodu źródłowego dla tej książki.

Instrukcję **BULK INSERT** możemy uruchamiać w szybkim trybie minimalnego rejestrowania, o ile spełnione są pewne wymagania. Informacje szczegółowe na ten temat znaleźć można w artykule „Prerequisites for Minimal Logging in Bulk Import” w dokumentacji SQL Server Books Online.

Właściwość **Identity** i obiekt sekwencji

SQL Server obsługuje dwa wbudowane mechanizmy automatycznego generowania kluczy: właściwość kolumny *identity* (tożsamość) i obiekt sekwencji. Właściwość *identity* jest obsługiwana od dawna i dobrze nadaje się w niektórych sytuacjach, ma jednak wiele ograniczeń. Obiekt sekwencji rozwiązuje wiele ograniczeń właściwości *identity*. Omówienie rozpoczne od właściwości *identity*.

Identity

Właściwość *identity* (tożsamość) jest standardową cechą kolumny tabeli (opisaną w standardzie ISO/ANSI). Można ją zdefiniować dla dowolnego typu numerycznego bez części ułamkowej. Przy definiowaniu można określić opcjonalne parametry *seed* (pierwsza wartość) i *increment* (przyrost). Jeśli parametry te nie zostaną podane, ich domyślnymi wartościami jest liczba 1. Zazwyczaj właściwość ta jest używana do generowania kluczy zastępczych (*surrogate key*), czyli kluczy generowanych przez system, a nie wynikających z danych aplikacji.

Dla przykładu poniższy kod tworzy tabelę nazwaną *dbo.T1*.

```

DROP TABLE IF EXIST dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL IDENTITY(1, 1)
    CONSTRAINT PK_T1 PRIMARY KEY,
    datacol VARCHAR(10) NOT NULL
    CONSTRAINT CHK_T1_datacol CHECK(datacol LIKE '[A-Za-z]%')
);

```

Tabela zawiera kolumnę *keycol*, dla której została zdefiniowana właściwość *identity* przy użyciu parametrów *seed* i *increment* o wartości 1. Tabela zawiera także kolumnę znakową *datacol*, której dane ograniczone są za pomocą *CHECK* do ciągów rozpoczynających się literą.

W instrukcjach *INSERT* musimy całkowicie ignorować kolumnę *identity*, tak jakby jej nie było w tabeli. Poniższy kod wstawia trzy wiersze do tabeli, specyfikując wartości tylko dla kolumny *datacol*.

```

INSERT INTO dbo.T1(datacol) VALUES('AAAAA');
INSERT INTO dbo.T1(datacol) VALUES('CCCCC');
INSERT INTO dbo.T1(datacol) VALUES('BBBBB');

```

SQL Server automatycznie generuje wartości dla kolumny *keycol*. Aby zobaczyć wartości utworzone przez SQL Server, odpytajmy tabelę:

```
SELECT * FROM dbo.T1;
```

Uzyskamy następujące dane wyjściowe:

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB

Podczas odpytywania tabeli możemy oczywiście odwoływać się do kolumny *identity* przy użyciu jej nazwy (w tym przypadku *keycol*). System SQL Server udostępnia także metodę odnoszenia się do kolumny *identity* przy użyciu ogólnej postaci *\$identity*.

Na przykład poniższe zapytanie używa postaci ogólnej do wybrania kolumny *identity* z tabeli *T1*.

```
SELECT $identity FROM dbo.T1;
```

Zapytanie to zwraca następujące wyniki:

keycol
1
2
3

Przy wstawianiu nowego wiersza do tabeli SQL Server generuje nową wartość *identity* w oparciu o bieżącą wartość *identity* i przyrost (*increment*). Jeśli chcemy uzyskać nowo wygenerowaną wartość *identity* – na przykład, by wstawić powiązane wiersze do tabeli referencyjnej – możemy użyć jednej z dwóch funkcji *@@identity* i *SCOPE_IDENTITY*.

@@identity zwraca ostatnią wartość *identity* wygenerowaną przez sesję, niezależnie od zakresu (na przykład procedura wysyłająca polecenie *INSERT* i wyzwalacz uruchomiony przez tę instrukcję należą do różnych zakresów). *SCOPE_IDENTITY* zwraca ostatnią wartość *identity* wygenerowaną przez sesję w bieżącym zakresie (na przykład w tej samej procedurze). Za wyjątkiem bardzo rzadkich, specjalnych przypadków, kiedy zakres naprawdę nas nie interesuje, należy używać funkcji *SCOPE_IDENTITY*.

Dla przykładu, poniższy kod wstawia wiersz do tabeli *T1*, ustawia wartość zmiennej poprzez użycie funkcji *SCOPE_IDENTITY* i odpytuje tę zmienną.

```
DECLARE @new_key AS INT;
INSERT INTO dbo.T1(datacol) VALUES('AAAAA');
SET @new_key = SCOPE_IDENTITY();
SELECT @new_key AS new_key
```

Jeśli wszystkie poprzednie przykłady kodu zamieszczone w tym podrozdziale zostały wykonane, kod ten zwróci następujące wyniki:

```
new_key
-----
4
```

Przypomnijmy, że obie funkcje *@@identity* i *SCOPE_IDENTITY* zwracają ostatnią wartość *identity* wygenerowaną przez bieżącą sesję. Na żadną z tych wartości nie wpływają wstawienia wykonywane przez inne sesje. Jeśli chcemy poznać bieżącą wartość *identity* w tabeli (ostatnią wygenerowaną wartość) niezależnie od sesji, trzeba użyć funkcji *IDENT_CURRENT*, podając jako wejście nazwę tabeli. Dla przykładu można uruchomić poniższy kod w nowej sesji (innej niż ta, w których były wywoływane poprzednie instrukcje *INSERT*).

```
SELECT
    SCOPE_IDENTITY() AS [SCOPE_IDENTITY],
    @@identity AS [@@identity],
    IDENT_CURRENT('dbo.T1') AS [IDENT_CURRENT];
```

Otrzymamy następujące wyniki:

```
SCOPE_IDENTITY  @@identity  IDENT_CURRENT
-----
NULL            NULL        4
```

Obie funkcje *@@identity* i *SCOPE_IDENTITY* zwróciły znaczniki *NULL*, ponieważ żadne wartości *identity* nie zostały jeszcze utworzone w sesji, w której działa to zapytanie.

Funkcja *IDENT_CURRENT* zwróciła 4, ponieważ funkcja ta zwraca bieżącą wartość *identity* dla tabeli niezależnie od tego, w której sesji wartość ta powstała.

Istnieje szczególna cecha projektu właściwości *identity*, która może wydać się zaskakująca. Zmiana bieżącej wartości *identity* w tabeli nie jest cofana nawet wtedy, gdy wykonanie instrukcji *INSERT* generującej zmianę się nie powiedzie lub zostanie wycofana transakcja, w ramach której polecenie zostało uruchomione. Dla przykładu uruchomimy następującą instrukcję *INSERT*, która pozostaje w sprzeczności z ograniczeniem *CHECK* zdefiniowanym w tabeli.

```
INSERT INTO dbo.T1(datacol) VALUES('12345');
```

Wykonanie instrukcji wstawienia nie udaje się i pojawia się następujący komunikat o błędzie:

Msg 547, Level 16, State 0, Line 1

The INSERT statement conflicted with the CHECK constraint "CHK_T1_datacol". The conflict occurred in database "TSQLV4", table "dbo.T1", column 'datacol'.

The statement has been terminated.

(Instrukcja *INSERT* jest w sprzeczności z ograniczeniem *CHECK* – "CHK_T1_datacol". Konflikt miał miejsce w bazie danych "TSQLV4", w tabeli "dbo.T1", kolumnie 'datacol'. Instrukcja została przerwana.)

Pomimo że wstawianie się nie powiodło, bieżąca wartość *identity* dla tabeli zmieniła się z 4 na 5 i zmiana ta nie została cofnięta ze względu na niepowodzenie wykonania instrukcji. Oznacza to, że kolejna instrukcja wstawiania wygeneruje wartość 6.

```
INSERT INTO dbo.T1(datacol) VALUES('EEEEEE');
```

Odpytajmy tabelę:

```
SELECT * FROM dbo.T1;
```

Zwróćmy uwagę na przerwę pomiędzy wartościami 4 i 6.

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB
4	AAAAA
6	EEEEEE

Dodatkowo SQL Server wykorzystuje funkcję buforowania dla właściwości *identity* ze względów wydajnościowych, co również może prowadzić do powstawania luk pomiędzy wartościami kluczy, jeśli nastąpi nagle zakończenie procesu SQL Server – na przykład z powodu awarii zasilania. Oznacza to oczywiście, że z właściwości *identity* powinniśmy korzystać do automatycznego generowania wartości tylko wtedy, gdy możemy dopuścić występowanie takich luk. W innych sytuacjach powinniśmy stosować własne mechanizmy.

Inne ograniczenie właściwości *identity* polega na tym, że nie można jej dodać do istniejącej kolumny lub z niej usunąć; właściwość tę można jedynie definiować wraz z kolumną, jako część instrukcji *CREATE TABLE* lub *ALTER TABLE*, które dodaje nową kolumnę.

System SQL Server pozwala jednak wprost wyspecyfikować własne wartości kolumny *identity* w instrukcjach *INSERT*, o ile dla używanej tabeli ustawimy opcję sesji *IDENTITY_INSERT*. Jednakże żadna opcja nie umożliwia aktualizowania kolumny *identity*.

Dla przykładu poniższy kod pokazuje, jak wstawić wiersz do tabeli *T1* z wyspecyfikowaną wprost wartością 5 w kolumnie *keycol*.

```
SET IDENTITY_INSERT dbo.T1 ON;
INSERT INTO dbo.T1(keycol, datacol) VALUES(5, 'FFFFF');
SET IDENTITY_INSERT dbo.T1 OFF;
```

Interesujące jest to, że jeśli włączymy opcję *IDENTITY_INSERT*, SQL Server zmienia bieżącą wartość *identity* dla tabeli tylko wtedy, gdy dostarczona wprost wartość dla kolumny *identity* jest większa niż bieżąca wartość *identity* w tabeli. Ponieważ bieżąca wartość *identity* w tabeli przed uruchomieniem poprzedniego kodu wynosiła 6, a instrukcja *INSERT* używa niższej (podawanej wprost) wartości 5, bieżąca wartość *identity* nie zostaje zmieniona. Tak więc, jeśli w tym momencie, po uruchomieniu poprzedniego kodu odpytamy tę tabelę przy użyciu funkcji *IDENT_CURRENT*, uzyskamy wynik 6, a nie 5. Tak więc kolejna instrukcja *INSERT* dla tej tabeli wygeneruje wartość 7.

```
INSERT INTO dbo.T1(datacol) VALUES('GGGGG');
```

Odpytajmy bieżącą zawartość tabeli *T1*.

```
SELECT * FROM dbo.T1;
```

Uzyskamy następujące wyniki:

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB
4	AAAAA
5	FFFFF
6	EEEEE
7	GGGGG

Trzeba także zdawać sobie sprawę, że samo ustawienie właściwości *identity* nie wymusza unikatowości w kolumnie. Jak pokazałem, możemy dostarczyć wprost własne wartości po ustawieniu opcji *IDENTITY_INSERT* na *ON* i te wartości mogą już występować w wierszach tabeli. Ponadto można ponownie określić parametr *seed* dla bieżącej wartości *identity* tabeli poprzez użycie instrukcji *DBCC CHECKIDENT*. Szczegółowe informacje na temat składni instrukcji *DBCC CHECKIDENT* znaleźć można w artykule

„*DBCC CHECKIDENT* (Transact-SQL)” w dokumentacji SQL Server Books Online. Krótko mówiąc, właściwość *identity* nie wymusza unikatowości. Jeśli trzeba zagwarantować unikatowość w kolumnie *identity*, musimy także zdefiniować dla tej kolumny klucz główny lub ograniczenie unikatowości.

Sekwencje

Język T-SQL obsługuje obiekt sekwencji jako alternatywę dla mechanizmu generowania klucza właściwości *identity*. Jest to funkcja zdefiniowana w standardzie SQL. Pod wieloma względami obiekt sekwencji jest bardziej elastyczny niż właściwość *identity*, co sprawia, że w wielu sytuacjach lepiej będzie stosować ten mechanizm.

Jedną z zalet obiektu sekwencji jest to, że przeciwieństwie do właściwości *identity* nie jest powiązany z konkretną kolumną w konkretnej tabeli; jest to raczej niezależny obiekt bazy danych. Ilekroć trzeba wygenerować nową wartość, wywołujemy funkcję w odniesieniu do obiektu, a zwróconej wartości używamy w dowolnym miejscu. Oznacza to, że możemy stosować jeden obiekt sekwencji, ułatwiający utrzymywanie kluczy, które nie powodują konfliktów w wielu tabelach.

Do utworzenia obiektu sekwencji służy instrukcja *CREATE SEQUENCE*. Wymagane jest jedynie podanie nazwy sekwencji, jednak trzeba pamiętać, że wartości domyślne nie muszą stanowić konfiguracji odpowiedniej dla konkretnego zastosowania. Jeśli nie wskażemy typu, SQL Server domyślnie użyje *BIGINT*. Jeśli potrzebny jest inny typ danych, trzeba go wskazać w wyrażeniu *AS <typ>*. Typem obiektu sekwencji może być dowolny typ numeryczny bez części ułamkowych. Jeśli na przykład typem sekwencji ma być *INT*, wskazujemy *AS INT*.

W odróżnieniu od właściwości *identity*, obiekt sekwencji obsługuje określenie wartości minimalnej (*MINVALUE <val>*) i wartości maksymalnej (*MAXVALUE <val>*) wewnątrz typu. Jeśli wartości te nie zostaną wskazane, obiekt sekwencji przyjmie wartość minimalną i maksymalną obsługiwaną przez dany typ. Na przykład dla typu *INT* wartości te to odpowiednio -2 147 483 648 i 2 147 483 647.

Ponadto, inaczej niż w przypadku właściwości *identity*, obiekt sekwencji obsługuje tworzenie wartości cyklicznych. Domyślnie ustawienie tej opcji to *NO CYCLE* (bez cykli). Jeśli chcemy, aby wartości obiektu sekwencji powtarzały się cyklicznie, trzeba jawnie wyspecyfikować opcję *CYCLE*.

Podobnie jak dla właściwości *identity*, obiekt sekwencji pozwala określić wartość początkową (*START WITH <val>*) i przyrost (*INCREMENT BY <val>*). Jeśli nie wyspecyfikujemy wartości początkowej, domyślnie użyta zostanie wartość minimalna (*MINVALUE*). Przy braku specyfikacji dla przyrostu wartość domyślnie będzie inkrementowana o 1.

Żałóżmy dla przykładu, że chcemy utworzyć sekwencję ułatwiającą generowanie identyfikatorów zamówień. Chcemy, by identyfikator był typu *INT*, miał minimalną wartość równą 1 i maksymalną wartość, która jest wartością maksymalną obsługiwaną

przez typ, rozpoczynał się od wartości 1, był inkrementowany o 1 i zezwalał na tworzenie wartości cyklicznych. Poniższa instrukcja *CREATE SEQUENCE* tworzy taką sekwencję.

```
CREATE SEQUENCE dbo.SeqOrderIDs AS INT
    MINVALUE 1
    CYCLE;
```

W tym przypadku musimy jawnie wyspecyfikować typ, minimalną wartość i opcję dotyczącą cykliczności, ponieważ różnią się od wartości domyślnych. Nie musimy wskazywać wartości maksymalnej, początkowej i inkrementowania, ponieważ pasują do wartości domyślnych.

Obiekt sekwencji obsługuje także opcję buforowania (*CACHE <val> | NO CACHE*), która informuje SQL Server, jak często wartości mają być zapisywane na dysku. Jeśli na przykład określimy rozmiar bufora jako 10 000, SQL Server będzie dokonywał zapisu na dysku co 10 000 żądań, a pomiędzy zapisami będzie przechowywał bieżącą wartość oraz liczbę pozostałych wartości w pamięci. Rzadsze zapisy na dysku zapewniają lepszą wydajność generowania wartości (przeciętnie), ale ryzykujemy utratę większej liczby wartości sekwencji w razie nieoczekiwanego zamknięcia procesu SQL Server, na przykład z powodu awarii zasilania. SQL Server domyślnie używa bufora o wielkości 50, choć liczba ta nie jest udokumentowana oficjalnie, gdyż firma Microsoft chce zachować możliwość jej zmiany w razie potrzeby.

Dowolną cechę obiektu sekwencji oprócz typu można zmienić przy użyciu instrukcji *ALTER SEQUENCE* (*MINVAL <val>*, *MAXVAL <val>*, *RESTART WITH <val>*, *INCREMENT BY <val>*, *CYCLE | NO CYCLE* lub *CACHE <val> | NO CACHE*). Załóżmy na przykład, że chcemy wykluczyć tworzenie wartości cyklicznych dla identyfikatorów *dbo.SeqOrderIDs*. W tym celu możemy zmienić aktualną definicję sekwencji poniższą instrukcją *ALTER SEQUENCE*:

```
ALTER SEQUENCE dbo.SeqOrderIDs
    NO CYCLE;
```

W celu wygenerowania nowej wartości sekwencji trzeba wywołać standardową funkcję *NEXT VALUE FOR <nazwa_sekwencji>*. Oto przykład wywołania tej funkcji:

```
SELECT NEXT VALUE FOR dbo.SeqOrderIDs;
```

Kod ten generuje następujący wynik:

```
-----
1
```

Zwróćmy uwagę, że inaczej niż w przypadku właściwości *identity*, nie musieliśmy wstawić wiersza do tabeli, by wygenerować nową wartość. Niektóre aplikacje wymagają wygenerowania nowej wartości przed jej zastosowaniem. Przy korzystaniu z obiektu sekwencji możemy przechowywać wyniki funkcji w zmiennej, a następnie użyć jej

w dowolnym miejscu. Aby to zademonstrować, najpierw utworzymy tabelę nazwaną *T1* za pomocą poniższego kodu.

```
DROP TABLE IF EXIST dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL
        CONSTRAINT PK_T1 PRIMARY KEY,
    datacol VARCHAR(10) NOT NULL
);
```

Poniższy kod generuje nową wartość sekwencji, przechowuje ją w zmiennej, a następnie korzysta z tej zmiennej w instrukcji *INSERT*, by wstawić wiersz do tabeli.

```
DECLARE @neworderid AS INT = NEXT VALUE FOR dbo.SeqOrderIDs;
INSERT INTO dbo.T1(keycol, datacol) VALUES(@neworderid, 'a');

SELECT * FROM dbo.T1;
```

Kod ten zwraca następujące wyniki:

keycol	datacol
2	a

Jeśli zachodzi potrzeba użycia tego klucza w powiązanych wierszach wstawianych w innej tabeli, w tych instrukcjach *INSERT* również możemy użyć tej samej zmiennej.

Jeśli nie trzeba generować nowej wartości sekwencji przed jej zastosowaniem, możemy bezpośrednio użyć funkcji *NEXT VALUE FOR* w ramach instrukcji *INSERT*, jak w poniższym kodzie:

```
INSERT INTO dbo.T1(keycol, datacol)
VALUES(NEXT VALUE FOR dbo.SeqOrderIDs, 'b');

SELECT * FROM dbo.T1;
```

Kod ten zwraca następujące wyniki:

keycol	datacol
2	a
3	b

Inaczej niż w przypadku właściwości *identity*, możemy generować nowe wartości sekwencji w instrukcji *UPDATE*, jak w przykładzie poniżej:

```
UPDATE dbo.T1
SET keycol = NEXT VALUE FOR dbo.SeqOrderIDs;

SELECT * FROM dbo.T1;
```

Kod ten zwraca następujące wyniki:

keycol	datacol
4	a
5	b

W celu uzyskania informacji o sekwencjach odpytujemy widok nazwany *sys.sequences*. Na przykład, aby wyszukać aktualną wartość sekwencji w obiekcie *SeqOrderIDs*, użyjemy następującego kodu:

```
SELECT current_value
FROM sys.sequences
WHERE OBJECT_ID = OBJECT_ID('dbo.SeqOrderIDs');
```

Kod ten generuje następujące wyniki:

current_value
5

W systemie SQL Server rozszerzono obsługę sekwencji o możliwości wykraczające poza to, co aktualnie zapewniają inne systemy *RDBMS* i zdefiniowane w standardzie. Jedno z rozszerzeń umożliwia nam kontrolowanie kolejności przypisanych wartości sekwencji podczas wstawiania wielu wierszy przy użyciu klauzuli *OVER*. Poniższy przykład demonstrowa zastosowanie tej opcji:

```
INSERT INTO dbo.T1(keycol, datacol)
SELECT
    NEXT VALUE FOR dbo.SeqOrderIDs OVER(ORDER BY hiredate),
    LEFT(firstname, 1) + LEFT(lastname, 1)
FROM HR.Employees;

SELECT * FROM dbo.T1;
```

Kod ten zwraca następujące wyniki:

keycol	datacol
4	a
5	b
6	JL
7	SD
8	DF
9	YP
10	SB
11	PS
12	RK
13	MC
14	ZD

Inne rozszerzenie obsługi sekwencji pozwala na użycie funkcji *NEXT VALUE FOR* w ograniczeniu domyślnym, jak w poniższym przykładzie:

```
ALTER TABLE dbo.T1
  ADD CONSTRAINT DFT_T1_keycol
  DEFAULT (NEXT VALUE FOR dbo.SeqOrderIDs)
  FOR keycol;
```

Teraz przy wstawianiu wierszy do tabeli, nie musimy wskazywać wartości dla kolumny *keycol*.

```
INSERT INTO dbo.T1(datacol) VALUES('c');
SELECT * FROM dbo.T1;
```

Kod ten zwraca następujące wyniki:

keycol	datacol
4	a
5	b
6	JL
7	SD
8	DF
9	YP
10	SB
11	PS
12	RK
13	MC
14	ZD
15	C

W przeciwieństwie do właściwości *identity*, której nie można dodać do istniejącej kolumny ani jej usunąć, ograniczenia domyślne można zarówno dodawać do istniejących kolumn tabeli, jak i je usuwać. Wcześniejszy przykład demonstruje dodanie ograniczenia do istniejącej kolumny. Aby usunąć ograniczenie, należy posłużyć się poleceniem `ALTER TABLE <nazwa_tabeli> DROP CONSTRAINT <nazwa_ograniczenia>`.

Istnieje jeszcze inne rozszerzenie, które pozwala przydzielać cały zakres wartości sekwencji w jednym kroku przy użyciu systemowej procedury składowanej nazwanej `sp_sequence_get_range`. Pomysł polega na tym, że jeśli aplikacja potrzebuje przypisać zakres wartości sekwencji, najłatwiej będzie tylko raz aktualizować sekwencję, inkrementując ją o rozmiar zakresu. Wywołujemy procedurę, wskazujemy rozmiar zakresu i pobieramy pierwszą wartość zakresu, a także inne informacje przy użyciu parametrów wyjściowych. Poniższy przykład demonstruje wywołanie procedury i żądanie zakresu 1000 wartości sekwencji.

```
DECLARE @first AS SQL_VARIANT;
EXEC sys.sp_sequence_get_range
  @sequence_name      = N'dbo.SeqOrderIDs',
  @range_size         = 1000,
  @range_first_value  = @first OUTPUT ;
SELECT @first;
```

Jeśli dwukrotnie uruchomimy kod, okaże się, że pierwsza wartość zwrócona w drugim wywołaniu jest większa od pierwszej o 1000.

Podobnie jak w przypadku właściwości *identity*, obiekt sekwencji nie gwarantuje tego, że nie powstaną żadne przerwy. Jeśli nowa wartość sekwencji zostanie wygenerowana przez transakcję, która się nie powiedzie, zmiana sekwencji nie jest cofana.

Po ukończeniu ćwiczeń należy wywołać poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXISTS dbo.T1;
DROP SEQUENCE IF EXISTS dbo.SeqOrderIDs;
```

Usuwanie danych

Język T-SQL udostępnia dwie instrukcje usuwania wierszy z tabeli – *DELETE* i *TRUNCATE*. Użyte w tym podrozdziale przykłady odnoszą się do kopii tabel *Customers* i *Orders* ze schematu *Sales*, utworzonych w schemacie *dbo*. W celu utworzenia i wypełnienia tych tabel uruchamiamy poniższy kod:

```
DROP TABLE IF EXIST dbo.Orders;
DROP TABLE IF EXIST dbo.Customers;

CREATE TABLE dbo.Customers
(
    custid          INT          NOT NULL,
    companyname     NVARCHAR(40) NOT NULL,
    contactname     NVARCHAR(30) NOT NULL,
    contacttitle    NVARCHAR(30) NOT NULL,
    address         NVARCHAR(60) NOT NULL,
    city            NVARCHAR(15) NOT NULL,
    region          NVARCHAR(15) NULL,
    postalcode      NVARCHAR(10) NULL,
    country         NVARCHAR(15) NOT NULL,
    phone           NVARCHAR(24) NOT NULL,
    fax             NVARCHAR(24) NULL,
    CONSTRAINT PK_Customers PRIMARY KEY(custid)
);

CREATE TABLE dbo.Orders
(
    orderid         INT          NOT NULL,
    custid          INT          NULL,
    empid           INT          NOT NULL,
    orderdate       DATETIME     NOT NULL,
    requireddate    DATETIME     NOT NULL,
    shippeddate      DATETIME     NULL,
    shipperid       INT          NOT NULL,
    freight         MONEY        NOT NULL,
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname        NVARCHAR(40) NOT NULL,
    shipaddress     NVARCHAR(60) NOT NULL,
    shipcity        NVARCHAR(15) NOT NULL,
```

```
shipregion      NVARCHAR(15) NULL,  
shippostalcode NVARCHAR(10) NULL,  
shipcountry     NVARCHAR(15) NOT NULL,  
CONSTRAINT PK_Orders PRIMARY KEY(orderid),  
CONSTRAINT FK_Orders_Customers FOREIGN KEY(custid)  
REFERENCES dbo.Customers(custid)  
);  
GO  
  
INSERT INTO dbo.Customers SELECT * FROM Sales.Customers;  
INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
```

Instrukcja *DELETE*

Instrukcja *DELETE* jest należącym do standardu poleceniem używanym do usuwania danych z tabeli w oparciu o predykat. Instrukcja zawiera tylko dwie klauzule – klauzulę *FROM*, w której wskazujemy nazwę tabeli docelowej, i klauzulę *WHERE*, w której specyfikujemy predykat. Usunięty zostaje jedynie podzbiór wierszy, dla których predykat przyjmuje wartość *TRUE*.

Dla przykładu, poniższa instrukcja usuwa z tabeli *dbo.Orders* wszystkie zamówienia, które zostały złożone przed rokiem 2015.

```
DELETE FROM dbo.Orders  
WHERE orderdate < '20150101';
```

Po uruchomieniu tej instrukcji system SQL Server zgłosi usunięcie 152 wierszy.

```
(152 row(s) affected)
```

Warto zauważyć, że można wyłączyć wyświetlenie komunikatu z liczbą uwzględnionych wierszy, włączając opcję sesji *NOCOUNT*. Jeśli opcja ta ma wartość *ON*, SQL Server Management Studio poinformuje jedynie, że wykonanie polecenia zakończyło się pomyślnie. Opcja ta domyślnie ma wartość *OFF*.

Instrukcja *DELETE* jest zawsze w pełni rejestrowana bez względu na ustawiony tryb rejestrowania w bazie danych – z tego względu nie powinno dziwić, że usuwanie dużej liczby wierszy może zająć sporo czasu.

Instrukcja *TRUNCATE*

Instrukcja *TRUNCATE* usuwa wszystkie wiersze tabeli. W przeciwieństwie do instrukcji *DELETE*, *TRUNCATE* nie zawiera filtru. Na przykład, aby usunąć wszystkie wiersze tabeli *dbo.T1*, uruchamiamy poniższy kod:

```
TRUNCATE TABLE dbo.T1;
```

Przewaga instrukcji *TRUNCATE* w porównaniu do instrukcji *DELETE* polega na tym, że ta pierwsza wykonywana jest przy minimalnym rejestrowaniu, natomiast druga przy pełnym, co powoduje znaczne różnice w wydajności. Jeśli na przykład użyjemy

instrukcji *TRUNCATE* do usunięcia wszystkich wierszy z tabeli zawierającej milion wierszy, czas trwania operacji wyniesie pojedyncze sekundy. W przypadku użycia instrukcji *DELETE* może to potrwać wiele minut, a nawet godzin.

Wspomniałem, że polecenie *TRUNCATE* wykonywane jest przy *minimalnym* poziomie rejestrowania, co nie oznacza, że w ogóle nic nie jest rejestrowane – instrukcja ta jest w pełni transakcyjna (wbrew powszechnemu przekonaniu). SQL Server rejestruje, jakie bloki danych są zwalniane przez operację, zatem możliwe jest ich odzyskanie, jeśli transakcja musi zostać wycofana. Zarówno *DELETE*, jak i *TRUNCATE* są działaniami transakcyjnymi.

Instrukcje *TRUNCATE* i *DELETE* różnią się także pod względem funkcjonalnym, jeśli tabela zawiera kolumnę *identity*. Polecenie *TRUNCATE* resetuje wartość *identity* przywracając pierwotną wartość parametru *seed*, natomiast polecenie *DELETE* tego nie robi, nawet jeśli zostanie użyta bez wskazania filtra (czyli gdy usuwane są wszystkie wiersze tabeli). Co ciekawe, standard definiuje opcję *identity column restart* dla instrukcji *TRUNCATE*, która pozwala określić, czy wartość *identity* ma zostać zresetowana, czy pozostawiona bez zmian, jednak nieszczęśliwie język T-SQL nie obsługuje tej opcji.

Instrukcja *TRUNCATE* nie jest dopuszczalna, kiedy do tabeli odwołuje się ograniczenie klucza obcego, nawet jeśli odwołująca się tabela jest pusta i jeśli klucz obcy jest wyłączony. W takiej sytuacji jedyną metodą dopuszczenia instrukcji *TRUNCATE* jest wcześniejsze usunięcie wszystkich kluczy obcych odwołujących się do tabeli.

Niekiedy zdarzają się wypadki polegające na usunięciu lub wyczyszczeniu niewłaściwej tabeli. Załóżmy dla przykładu, że mamy otwarte połączenia zarówno ze środowiskiem produkcyjnym, jak projektowym i wpisaliśmy kod w niewłaściwe połączenie. Instrukcje *TRUNCATE* i *DROP* są tak szybkie, że zanim zdaliśmy sobie sprawę z pomyłki, transakcja została już wykonana. Aby uchronić się przed takimi przypadkami, możemy ochronić tabelę produkcyjną po prostu tworząc pustą tabelę z kluczem obcym wskazującym tabelę produkcyjną. Następnie można nawet wyłączyć klucz obcy, by nie miał żadnego wpływu na wydajność. Jak wspomniałem wcześniej, nawet jeśli klucz obcy jest wyłączony, będzie uniemożliwiał usunięcie lub wyczyszczenie tabeli, do której się odwołuje.

Jeśli baza danych zawiera partycjonowane tabele, polecenie *TRUNCATE* może zostać zastosowane wobec indywidualnych partycji. Rozszerzenie to zostało wprowadzone w wersji SQL Server 2016. Można określić listę partycji i ich zakresów (ze słowem kluczowym *TO* pomiędzy ograniczeniami zakresów). Dla przykładu założymy, że mamy spartycjonowaną tabelę *T1* i chcemy wyczyścić partycje 1, 3, 5 oraz od 7 do 10. Poniższy kod pozwoli zrealizować to zadanie:

```
TRUNCATE TABLE dbo.T1 WITH ( PARTITIONS(1, 3, 5, 7 TO 10) );
```

UWAGA Partycjonowanie tabel polega na podziale tabeli na wiele jednostek nazywanych partycjami, głównie w celu usprawnienia utrzymywania wielkich ilości danych. Pozwala to na wydajniejsze wykonywanie takich procesów, jak importowanie danych i usuwanie danych historycznych. Szczegółowe omówienie tego zagadnienia zawiera dokumentacja SQL Server Books Online pod następującym adresem: <https://msdn.microsoft.com/en-us/library/ms190787.aspx>.

DELETE oparte na złączeniu

Język T-SQL udostępnia niestandardową składnię instrukcji *DELETE* bazujące na złączeniu. Samo złączenie służy jako filtr oparty na predykacie (klauszula *ON*). Złączenie udostępnia także atrybuty powiązanych wierszy z innej tabeli, do których odwołujemy się w klauzuli *WHERE*. Oznacza to, że możemy usuwać wiersze z jednej tabeli na podstawie filtru dotyczącego atrybutów w powiązanych wierszach z innej tabeli. Na przykład poniższa instrukcja usuwa zamówienia złożone przez klientów z USA.

```
DELETE FROM O
FROM dbo.Orders AS O
      JOIN dbo.Customers AS C
        ON O.custid = C.custid
WHERE C.country = N'USA';
```

Podobnie jak w przypadku instrukcji *SELECT*, logicznie pierwszą klauzulą przetwarzaną w instrukcji *DELETE* jest klauzula *FROM* (w poleceniu występuje jako druga). Następnie przetwarzana jest klauzula *WHERE*, a na koniec klauzula *DELETE*. Wyrażenie to można zinterpretować następująco: „zapytanie łączy tabelę *Orders* (alias *O*) z tabelą *Customers* (alias *C*) na podstawie zgodności pomiędzy identyfikatorem klienta dla wierszy zamówień i identyfikatorem klienta dla wierszy klientów. Następnie filtruje tylko te zamówienia, które zostały złożone przez klientów z USA. Na koniec usuwa wszystkie zakwalifikowane wiersze z tabeli *O* (alias tabeli *Orders*)”.

Dwie klauzule *FROM* w instrukcji *DELETE* opartej na złączeniu mogą wydawać się niejednoznaczne. Jednak przy tworzeniu kodu projektujemy go tak, jakby była to instrukcja *SELECT* ze złączeniem. Innymi słowy, rozpoczynamy od klauzuli *FROM* ze złączeniem, przechodzimy do klauzuli *WHERE* i na koniec, zamiast specyfikowania klauzuli *SELECT*, używamy *DELETE*, wskazując w jej klauzuli *FROM* alias tej strony złączenia, które ma być docelową tabelą operacji usuwania. Warto zauważyć, że pierwsza klauzula *FROM* jest opcjonalna – w powyższym przykładzie moglibyśmy napisać po prostu *DELETE O* w pierwszym wierszu wyrażenia.

Jak wspomniałem, instrukcja *DELETE* oparta na złączeniu nie jest częścią standardu SQL. Aby zachować zgodność ze standardem, należałoby użyć podzapytań, a nie złączeń. Na przykład poniższa instrukcja *DELETE* stosuje podzapytanie, by wykonać to samo zadanie.

```
DELETE FROM dbo.Orders
WHERE EXISTS
(SELECT *
 FROM dbo.Customers AS C
 WHERE Orders.Custid = C.Custid
 AND C.Country = 'USA');
```

Kod ten usuwa z tabeli *Orders* wszystkie wiersze, dla których w tabeli *Customers* istnieje powiązany klient z USA.

SQL Server będzie przetwarzał te dwa wyrażenia w ten sam sposób (utworzy identyczne plany wykonania); z tego względu nie powinniśmy oczekiwać różnic w wydajności tych wyrażeń. Zazwyczaj zalecam stosowanie składni zgodnej ze standardem, aby zapewnić większą przenośność kodu. Wyjątkiem jest sytuacja, gdy istnieją uzasadnione powody korzystania z rozwiązań niestandardowych – na przykład duże różnice w wydajności.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXIST dbo.Orders, dbo.Customers;
```

Aktualizowanie danych

Język T-SQL obsługuje standardową instrukcję *UPDATE*, która pozwala aktualizować wiersze tabeli. Dodatkowo udostępnia niestandardowe postaci polecenia *UPDATE* wykorzystującego złączenia i zmienne. W tym podrozdziale opisano różne sposoby stosowania instrukcji *UPDATE*.

Przykłady przytoczone w tym podrozdziale odnoszą się do kopii tabel *Orders* i *OrderDetails* ze schematu *Sales*, utworzonych w schemacie *dbo*. W celu utworzenia i wypełnienia tych tabel uruchamiamy następujący kod:

```
DROP TABLE IF EXIST dbo.OrderDetails, dbo.Orders;
GO
```

```
CREATE TABLE dbo.Orders
(
   orderid          INT          NOT NULL,
    custid          INT          NULL,
    empid          INT          NOT NULL,
    orderdate       DATETIME     NOT NULL,
    requireddate    DATETIME     NOT NULL,
    shippeddate      DATETIME     NULL,
    shipperid       INT          NOT NULL,
    freight         MONEY        NOT NULL,
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname        NVARCHAR(40) NOT NULL,
    shipaddress     NVARCHAR(60) NOT NULL,
    shipcity        NVARCHAR(15) NOT NULL,
    shipregion      NVARCHAR(15) NULL,
    shippostalcode NVARCHAR(10) NULL,
```

```
    shipcountry    NVARCHAR(15) NOT NULL,  
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)  
);  
  
CREATE TABLE dbo.OrderDetails  
(  
    orderid INT NOT NULL,  
    productid INT NOT NULL,  
    unitprice MONEY NOT NULL  
        CONSTRAINT DFT_OrderDetails_unitprice DEFAULT(0),  
    qty SMALLINT NOT NULL  
        CONSTRAINT DFT_OrderDetails_qty DEFAULT(1),  
    discount NUMERIC(4, 3) NOT NULL  
        CONSTRAINT DFT_OrderDetails_discount DEFAULT(0),  
    CONSTRAINT PK_OrderDetails PRIMARY KEY(orderid, productid),  
    CONSTRAINT FK_OrderDetails_Orders FOREIGN KEY(orderid)  
        REFERENCES dbo.Orders(orderid),  
    CONSTRAINT CHK_discount CHECK (discount BETWEEN 0 AND 1),  
    CONSTRAINT CHK_qty CHECK (qty > 0),  
    CONSTRAINT CHK_unitprice CHECK (unitprice >= 0)  
);  
GO  
  
INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;  
INSERT INTO dbo.OrderDetails SELECT * FROM Sales.OrderDetails;
```

Instrukcja *UPDATE*

Instrukcja *UPDATE* jest standardowym poleceniem, które pozwala aktualizować podzbiór wierszy tabeli. Do zidentyfikowania podzbioru wierszy do aktualizowania służy predykat zawarty w klauzuli *WHERE*. Przypisanie wartości do kolumn specyfikujemy w klauzuli *SET*, oddzielając je przecinkami.

Na przykład poniższa instrukcja *UPDATE* zwiększa rabat pozycji zamówienia dla produktu 51 o 5%.

```
UPDATE dbo.OrderDetails  
    SET discount = discount + 0.05  
WHERE productid = 51;
```

Oczywiście możemy wykonać instrukcję *SELECT* z tym samym filtrem przed i po aktualizacji, by przekonać się, jaka nastąpiła zmiana. W dalszej części rozdziału pokażę inną metodę sprawdzania zmian, używającą klauzuli *OUTPUT* dodawanej do instrukcji modyfikacji.

Język T-SQL obsługuje złożone operatory przypisywania: *+=* (czytane jako *plus równość*), *-=* (*minus równość*), **=* (*mnożenie równość*), */=* (*dzielenie równość*) oraz *%=* (*modulo równość*), które pozwalają skracać wyrażenia przypisywania, takie jak użyte w poprzednim zapytaniu. Zamiast wyrażenia *discount = discount + 0.05*, możemy użyć skróconej postaci: *discount += 0.05*. Pełna instrukcja *UPDATE* będzie wyglądała teraz następująco:

```
UPDATE dbo.OrderDetails
  SET discount += 0.05
 WHERE productid = 51;
```

Operacje jednoczesne (*all-at-once*) są ważnym aspektem języka SQL, o którym warto pamiętać pisząc instrukcje *UPDATE*. Operacje te zostały opisane w rozdziale 2 „Zapytania do pojedynczej tabeli” w kontekście instrukcji *SELECT*, ale zasady te stosują się również do instrukcji *UPDATE*. Przypomnę, że założenie funkcjonalne jest takie, że wszystkie wyrażenia tej samej fazy logicznej są przetwarzane w tym samym momencie. Aby lepiej zrozumieć znaczenie tej koncepcji, przeanalizujmy poniższą instrukcję *UPDATE*.

```
UPDATE dbo.T1
  SET col1 = col1 + 10, col2 = col1 + 10;
```

Załóżmy, że przed aktualizacją jeden wiersz tabeli ma wartość 100 w kolumnie *col1* i 200 w kolumnie *col2*. Czy jesteśmy w stanie określić wartości tych kolumn po modyfikacji?

Jeśli nie uwzględnimy koncepcji operacji jednoczesnych, można by sądzić, że kolumna *col1* będzie miała wartość 110, a kolumna *col2* będzie miała wartość 120, przy założeniu, że przypisywanie wykonywane jest od lewej do prawej. Jednakże z logicznego punktu widzenia obydwie przypisania mają miejsce w tym samym momencie, co oznacza, że w obu użyta będzie ta sama wartość kolumny *col1* – wartość sprzed aktualizacji. Wynikiem tej aktualizacji będzie ta sama wartość dla obu kolumn, czyli 110.

Mając na uwadze działanie operacji jednoczesnych, jak powinna wyglądać instrukcja *UPDATE*, która zamienia miejscami wartości kolumn *col1* i *col2*? W większości języków programowania, gdzie wyrażenia i przypisania są przetwarzane w pewnej kolejności (zazwyczaj od lewej do prawej), potrzebna byłaby zmienna tymczasowa. Ponieważ jednak w języku SQL wszystkie przypisania wykonywane są tak, jakby następowały w tym samym momencie, rozwiązanie jest bardzo proste.

```
UPDATE dbo.T1
  SET col1 = col2, col2 = col1;
```

W obu przypisaniach użyte wartości kolumn źródłowych są wartościami sprzed aktualizacji, tak więc nie będzie potrzebna zmienna tymczasowa.

UPDATE oparte na złączeniu

Podobnie jak w przypadku instrukcji *DELETE*, język T-SQL również obsługuje niestandardową składnię instrukcji *UPDATE* wykorzystującą złączenia i tak jak dla instrukcji *DELETE* złączenie służy do filtrowania.

Składnia jest bardzo podobna do instrukcji *SELECT* opartej na złączeniu; to znaczy klauzule *FROM* i *WHERE* są takie same, ale zamiast klauzuli *SELECT* używamy klauzuli *UPDATE*. Słowo kluczowe *UPDATE* znajduje się po aliasie tabeli docelowej

dla operacji aktualizowania (w jednym poleceniu nie można aktualizować więcej niż jednej tabeli), a po niej umieszczana jest klauzula *SET* z przypisaniami kolumn.

Na przykład instrukcja *UPDATE* z listingu 8-1 zwiększa o 5% rabat dla wszystkich pozycji zamówienia złożonego przez klienta 1.

LISTING 8-1 Instrukcja *UPDATE* w oparciu o złączenie

```
UPDATE OD
  SET discount += 0.05
FROM dbo.OrderDetails AS OD
  JOIN dbo.Orders AS O
    ON OD.orderid = O.orderid
WHERE O.custid = 1;
```

Interpretację tego wyrażenia rozpoczynamy od klauzuli *FROM*, następnie przechodzimy do klauzuli *WHERE*, a na koniec do klauzuli *UPDATE*. Zapytanie łączy tabelę *OrderDetails* (alias *OD*) z tabelą *Orders* (alias *O*) na podstawie zgodności pomiędzy identyfikatorem zamówienia w tabeli pozycji zamówień a identyfikatorem w tabeli zamówień. Następnie zapytanie filtruje tylko te wiersze, w których identyfikator klienta w zamówieniu ma wartość 1. Na koniec zapytanie wykonuje klauzulę *UPDATE* dla *OD* (alias tabeli *OrderDetails*) jako docelowej tabeli aktualizacji, zwiększając rabat o 5%.

Aby wykonać to samo zadanie przy użyciu składni zgodnej ze standardem, trzeba użyć podzapytania, a nie złączenia, jak poniżej:

```
UPDATE dbo.OrderDetails
  SET discount += 0.05
WHERE EXISTS
  (SELECT * FROM dbo.Orders AS O
    WHERE O.orderid = OrderDetails.orderid
    AND O.custid = 1);
```

Klauzula *WHERE* zapytania filtruje jedynie pozycje zamówień powiązane z zamówieniami złożonymi przez klienta 1. W przypadku tego konkretnego zadania system SQL Server zinterpretuje obie wersje w ten sam sposób i z tego względu nie należy oczekiwać różnic w wydajności pomiędzy nimi. Jak już wspomniałem wcześniej przy okazji instrukcji *DELETE*, jeśli nie ma istotnych powodów innego postępowania, zalecane jest trzymanie się kodu standardowego. W przypadku tego zadania brak jest takich powodów.

Istnieją jednak sytuacje, w których wersja wykorzystująca złączenia będzie wydajniejsza w porównaniu z wersją używającą podzapytań. Złączenie oprócz filtrowania pozwala uzyskać dostęp do atrybutów z innych tabel, których można użyć w przypisaniach kolumn w klauzuli *SET*. Ten sam dostęp do innej tabeli pozwala nam zarówno filtrować, jak i uzyskiwać wartości atrybutów z innej tabeli dla przypisań. Metoda wykorzystująca podzapytania wymaga użycia oddzielnych podzapytań do celu filtrowania i do pobrania wartości atrybutów – w istocie potrzebne jest osobne podzapytanie

dla każdego przypisania, co wręcz lawinowo zwiększa liczbę niezbędnych dostępów do innej tabeli..

Przeanalizujemy dla przykładu następującą niestandardową instrukcję *UPDATE* opartą na złączeniu.

```
UPDATE T1
  SET col1 = T2.col1,
      col2 = T2.col2,
      col3 = T2.col3
FROM dbo.T1 JOIN dbo.T2
  ON T2.keycol = T1.keycol
WHERE T2.col4 = 'ABC';
```

Instrukcja ta łączy tabele *T1* i *T2* na podstawie porównania *T1.keycol* i *T2.keycol*. Klauzula *WHERE* filtruje tylko te wiersze, dla których *T2.col4* jest równe 'ABC'. Instrukcja *UPDATE* oznacza tabelę *T1* jako docelową tabelę polecenia *UPDATE*, a klauzula *SET* ustawia wartości kolumn *col1*, *col2* i *col3* w tabeli *T1* przy użyciu wartości odpowiadających im kolumn w tabeli *T2*.

Próba sformułowania tego zadania przy użyciu kodu standardowego z wykorzystaniem podzapytań wymaga znacznie bardziej „rozwlekłego” zapytania.

```
UPDATE dbo.T1
  SET col1 = (SELECT col1
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol),

      col2 = (SELECT col2
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol),

      col3 = (SELECT col3
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol)
WHERE EXISTS
  (SELECT *
   FROM dbo.T2
   WHERE T2.keycol = T1.keycol
        AND T2.col4 = 'ABC');
```

W odróżnieniu od wersji korzystającej ze złączenia, oprócz braku zwięzłości ta wersja powoduje, że każde podzapytanie oddzielnie uzyskuje dostęp do tabeli *T2* – bez głębszej analizy można stwierdzić, że będzie mniej wydajna niż wersja ze złączeniem.

Standard SQL obsługuje mechanizm *konstruktora wiersza* (nazywanego również *wyrażeniem wektorowym*), który jest tylko częściowo zaimplementowany w języku T-SQL. Do wersji SQL Server 2016 włącznie wiele aspektów konstruktorów wierszy nie jest jeszcze zaimplementowanych, w tym możliwość stosowania ich w klauzuli *SET* instrukcji *UPDATE*, jak w poniższym kodzie (przykład ten wykorzystuje niezaimplementowaną funkcjonalność i nie będzie działał w SQL Server).

```
UPDATE dbo.T1
    SET (col1, col2, col3) =
        (SELECT col1, col2, col3
         FROM dbo.T2
         WHERE T2.keycol = T1.keycol)

WHERE EXISTS
    (SELECT *
     FROM dbo.T2
     WHERE T2.keycol = T1.keycol
     AND T2.col4 = 'ABC');
```

Jednak, jak widać na tym przykładzie, wersja ta nadal jest bardziej skomplikowana, niż wersja z użyciem złączenia, ponieważ wymaga oddzielnych podzapytań do filtrowania i do uzyskiwania atrybutów z innej tabeli do przypisań.

UPDATE z przypisaniem

Język T-SQL obsługuje własną, niestandardową składnię instrukcji *UPDATE*, która jednocześnie aktualizuje dane tabeli i przypisuje wartości do zmiennych. Składnia ta pozwala uniknąć konieczności stosowania oddzielnych instrukcji *UPDATE* i *SELECT*, by zrealizować to samo zadanie.

Jednym z typowych zadań, w których wykorzystywana jest ta składnia, jest utrzymywanie niestandardowego mechanizmu sekwencji/automatycznego numerowania, gdy właściwość *identity* kolumny i obiekt sekwencji nie działają tak, jakbyśmy chcieli. Przykładem może być sytuacja, kiedy potrzebny jest mechanizm gwarantujący, że w tworzonej numeracji nie powstaną przerwy. Pomysł polega na utrzymywaniu ostatnio użytej wartości w tabeli i zastosowaniu specjalnej składni instrukcji *UPDATE* do inkrementowania wartości w tabeli i przypisania nowej wartości do zmiennej.

Na początek uruchomimy poniższy kod, by utworzyć tabelę *Sequence* z kolumną *val*, a następnie wstawimy do niej pojedynczy wiersz z wartością 0 – o jeden mniej niż pierwsza wartość, której chcemy użyć.

```
DROP TABLE IF EXIST dbo.Sequences;

CREATE TABLE dbo.Sequences
(
    id VARCHAR(10) NOT NULL
    CONSTRAINT PK_Sequences PRIMARY KEY(id),
    val INT NOT NULL
);
INSERT INTO dbo.Sequences VALUES('SEQ1', 0);
```

Teraz, ilekroć trzeba uzyskać nową wartość sekwencji, wykonamy następujący kod:

```
DECLARE @nextval AS INT;
```

```

UPDATE dbo.Sequences
    SET @nextval = val += 1
WHERE id = 'SEQ1';

SELECT @nextval;

```

Kod deklaruje zmienną lokalną nazwaną *@nextval*, a następnie wykorzystuje specjalną składnię instrukcji *UPDATE* do inkrementowania wartości kolumny o 1 i przypisania zaktualizowanej wartości kolumny do zmiennej, po czym zwraca wartość zmiennej. Przypisania w klauzuli *SET* wykonywane są od prawej do lewej – tak więc, pierwsza wartość *val* to *val + 1*, a następnie wynik (*val + 1*) jest przypisywany do zmiennej *@nextval*.

Specjalizowana składnia instrukcji *UPDATE* jest wykonywana jako transakcja i jest bardziej wydajna, niż używanie oddzielnych instrukcji *UPDATE* i *SELECT*, ponieważ dostęp do danych uzyskiwany jest tylko raz.

Po ukończeniu ćwiczeń uruchamiamy następujący kod, by wyczyścić bazę danych:

```
DROP TABLE IF EXIST dbo.Sequences;
```

Scalanie danych

Język T-SQL obsługuje instrukcję *MERGE*, która umożliwia modyfikowanie danych poprzez stosowanie różnych działań (*INSERT*, *UPDATE* i *DELETE*) opartych na logice warunkowej. Instrukcja *MERGE* jest częścią standardu SQL, natomiast w języku T-SQL dodano do niej kilka niestandardowych rozszerzeń.

Zadania realizowane przy użyciu pojedynczej instrukcji *MERGE* zazwyczaj można przetłumaczyć na połączenie kilku innych instrukcji DML (*INSERT*, *UPDATE* i *DELETE*) bez użycia instrukcji *MERGE*. Użycie instrukcji *MERGE* sprawia, że kod jest bardziej zwężły i działa efektywniej, ponieważ wymaga mniejszej liczbyostępów do wykorzystywanych tabel.

W celu zademonstrowania działania instrukcji *MERGE* użyjemy tabel nazwanych *dbo.Customers* i *dbo.CustomersStage*. W celu utworzenia tych tabel i wypełnienia ich danymi uruchamiamy kod z listingu 8-2.

LISTING 8-2 Kod tworzący i wypełniający danymi tabele *Customers* i *CustomersStage*

```

DROP TABLE IF EXIST dbo.Customers;
GO

CREATE TABLE dbo.Customers
(
    custid INT NOT NULL,
    companyname VARCHAR(25) NOT NULL,
    phone VARCHAR(20) NOT NULL,
    address VARCHAR(50) NOT NULL,
    CONSTRAINT PK_Customers PRIMARY KEY(custid)
);

```



```

INSERT INTO dbo.Customers(custid, companyname, phone, address)
VALUES
  (1, 'cust 1', '(111) 111-1111', 'address 1'),
  (2, 'cust 2', '(222) 222-2222', 'address 2'),
  (3, 'cust 3', '(333) 333-3333', 'address 3'),
  (4, 'cust 4', '(444) 444-4444', 'address 4'),
  (5, 'cust 5', '(555) 555-5555', 'address 5');

DROP TABLE IF EXIST dbo.CustomersStage;
GO

CREATE TABLE dbo.CustomersStage
(
  custid INT NOT NULL,
  companyname VARCHAR(25) NOT NULL,
  phone VARCHAR(20) NOT NULL,
  address VARCHAR(50) NOT NULL,
  CONSTRAINT PK_CustomersStage PRIMARY KEY(custid)
);

INSERT INTO dbo.CustomersStage(custid, companyname, phone, address)
VALUES
  (2, 'AAAAA', '(222) 222-2222', 'address 2'),
  (3, 'cust 3', '(333) 333-3333', 'address 3'),
  (5, 'BBBBB', 'CCCCC', 'DDDDD'),
  (6, 'cust 6 (new)', '(666) 666-6666', 'address 6'),
  (7, 'cust 7 (new)', '(777) 777-7777', 'address 7');

```

Poniższe zapytanie pokazuje zawartość tabeli *Customers*:

```
SELECT * FROM dbo.Customers;
```

Zapytanie to generuje następujące wyniki:

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	cust 2	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	cust 5	(555) 555-5555	address 5

Zawartość tabeli *CustomersStage* pokazuje poniższe zapytanie:

```
SELECT * FROM dbo.CustomersStage;
```

Zapytanie to generuje następujące wyniki:

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Załóżmy, że chcemy scalić zawartość tabeli *CustomersStage* (tabela źródłowa) z zawartością tabeli *Customers* (tabela docelowa). Mówiąc dokładniej, chcemy dodać klientów, którzy jeszcze nie istnieją i zaktualizować atrybuty tych klientów, którzy już istnieją.

Jeśli rozumiemy mechanizmy usuwania i aktualizowania oparte na złączeniach, powinniśmy równie swobodnie posługiwać się instrukcją *MERGE*, która opiera się na składni złączenia. Nazwę tabeli docelowej podajemy w klauzuli *MERGE*, zaś nazwę tabeli źródłowej w klauzuli pomocniczej *USING*. Warunki scalania definiujemy w predykanie klauzuli *ON*. Warunek scalania określa, które wiersze w tabeli źródłowej pasują do wierszy tabeli docelowej, a które nie są zgodne. W klauzuli *WHEN MATCHED THEN* definiujemy działanie w sytuacji, kiedy dopasowanie zostaje znalezione, natomiast w klauzuli *WHEN NOT MATCHED THEN* określamy działanie przy braku zgodności.

Oto pierwszy przykład użycia instrukcji *MERGE*: dodanie nieistniejących klientów i aktualizowanie istniejących.

```

MERGE INTO dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
  ON TGT.custid = SRC.custid
WHEN MATCHED THEN
  UPDATE SET
    TGT.companyname = SRC.companyname,
    TGT.phone = SRC.phone,
    TGT.address = SRC.address
WHEN NOT MATCHED THEN
  INSERT (custid, companyname, phone, address)
  VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address);

```



UWAGA Zakończenie średnikiem instrukcji *MERGE* jest obowiązkowe, chociaż w przypadku większości instrukcji języka T-SQL działanie takie jest opcjonalne. Jak wspominałem wielokrotnie, zalecane jest kończenie wszystkich instrukcji znakiem średnika – jego użycie w miejscach niewymaganych nie wprowadza problemów, a w przyszłych wersjach SQL Server postępowanie takie zapewne będzie obowiązkowe.

Powyższa instrukcja *MERGE* definiuje tabelę *Customers* jako tabelę docelową (w klauzuli *MERGE*), a tabelę *CustomersStage* jako tabelę źródłową (w klauzuli *USING*). Zwróćmy uwagę, że dla zachowania zwięzłości możemy przypisywać aliasy do tabel źródłowych i docelowych (w tym przypadku *TGT* i *SRC*). Predykat *TGT.custid = SRC.custid* definiuje, co zostanie, a co nie zostanie uznane za zgodność. W tym przypadku dopasowanie mamy w sytuacji, kiedy identyfikator klienta istniejący w tabeli źródłowej istnieje również w tabeli docelowej. W przeciwnym razie nie mamy zgodności.

W przypadku znalezienia dopasowania instrukcja *MERGE* definiuje działanie *UPDATE*, ustawiające wartości *companyname*, *phone* i *address* jako odpowiednie wartości z wiersza tabeli źródłowej. Zwróćmy uwagę, że składnia działania *UPDATE* jest podobna do zwykłej instrukcji *UPDATE* z wyjątkiem tego, że nie musimy dostarczać

nazwy tabeli docelowej aktualizacji, ponieważ została ona już zdefiniowana w klauzuli *MERGE*.

Dla przypadku braku dopasowania instrukcja *MERGE* definiuje działanie *INSERT*, wstawiając wiersz z tabeli źródłowej. I ponownie, składnia działania *INSERT* jest podobna do zwykłej instrukcji *INSERT* z wyjątkiem tego, że nie musimy określać nazwy tabeli docelowej działania, ponieważ jest ona już zdefiniowana w klauzuli *MERGE*.

Nasza instrukcja *MERGE* zgłasza, że zmodyfikowanych zostało 5 wierszy.

(5 row(s) affected)

Liczba ta obejmuje 3 wiersze zaktualizowane (klienci 2, 3 i 5) oraz dwa wiersze wstawione (klienci 6 i 7). Odpytamy tabelę *Customers*, by uzyskać nową jej zawartość.

```
SELECT * FROM dbo.Customers;
```

Zapytanie to zwraca następujące wyniki:

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Klauzula *WHEN MATCHED* definiuje, jakie działanie zostanie podjęte, kiedy wiersz źródłowy ma pasujący wiersz w tabeli docelowej, natomiast klauzula *WHEN NOT MATCHED* definiuje działanie w sytuacji braku takiego dopasowania. Język T-SQL obsługuje także trzecią klauzulę, która definiuje działanie podejmowane w sytuacji, kiedy pewien wiersz docelowy nie pasuje do żadnego wiersza źródłowego; klauzula ta nazwana jest *WHEN NOT MATCHED BY SOURCE*. Załóżmy, że do naszego przykładu instrukcji *MERGE* chcemy dodać logikę powodującą usuwanie wierszy z tabeli docelowej, jeśli dany wiersz docelowy nie pasuje do żadnego wiersza źródłowego. W takiej sytuacji wystarczy dodać klauzulę *WHEN NOT MATCHED BY SOURCE* z działaniem *DELETE*, jak poniżej:

```
MERGE dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
  ON TGT.custid = SRC.custid
  WHEN MATCHED THEN
    UPDATE SET
      TGT.companyname = SRC.companyname,
      TGT.phone = SRC.phone,
      TGT.address = SRC.address
  WHEN NOT MATCHED THEN
    INSERT (custid, companyname, phone, address)
    VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address)
```

```
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Odpytajmy tabelę Customers, by sprawdzić działanie tej instrukcji *MERGE*.

```
SELECT * FROM dbo.Customers;
```

Zapytanie to zwraca następujące wyniki, pokazując, że klienci 1 i 4 zostali usunięci.

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Jeśli wrócimy do pierwszego przykładu instrukcji *MERGE*, która aktualizuje istniejących klientów i dodaje nieistniejących, można zauważyć, że przed zastąpieniem atrybutów istniejących klientów instrukcja nie sprawdza, czy wartości kolumn zostaną rzeczywiście zmienione. Oznacza to, że wiersz klienta jest modyfikowany także w sytuacji, kiedy wiersz źródłowy i docelowy są identyczne, co oczywiście jest mało efektywnym działaniem. Byłoby znacznie lepiej, gdyby modyfikacja wiersza następowała tylko wtedy, gdy przynajmniej jeden atrybut się zmienił. Istnieje sposób uzyskania takiego działania.

Instrukcja *MERGE* obsługuje dodawanie predykatów do różnych klauzul działania przy użyciu opcji *AND*; oprócz spełnienia pierwotnego warunku, działanie będzie wykonywane tylko wtedy, gdy spełniony jest dodatkowy predykat (przyjmuje wartość *TRUE*). W tym przypadku trzeba dodać predykat pod klauzulą *WHEN MATCHED AND*, który sprawdza, czy zmieniony zostanie co najmniej jeden z atrybutów, co uzasadni działanie *UPDATE*. Cała instrukcja *MERGE* uzyska następującą postać:

```
MERGE dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
    ON TGT.custid = SRC.custid
WHEN MATCHED AND
    ( TGT.companyname <> SRC.companyname
    OR TGT.phone <> SRC.phone
    OR TGT.address <> SRC.address) THEN
    UPDATE SET
        TGT.companyname = SRC.companyname,
        TGT.phone = SRC.phone,
        TGT.address = SRC.address
WHEN NOT MATCHED THEN
    INSERT (custid, companyname, phone, address)
    VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address);
```

Jak widzimy, instrukcja *MERGE* ma bardzo duże możliwości, pozwalając na umieszczenie złożonej logiki modyfikowania w pojedynczym wyrażeniu.

Modyfikowanie danych przy użyciu wyrażeń tablicowych

Język T-SQL nie ogranicza działań podejmowanych względem wyrażeń tablicowych jedynie do instrukcji *SELECT*, ale pozwala również stosować do tych wyrażeń inne instrukcje DML (*INSERT*, *UPDATE*, *DELETE* i *MERGE*). Przypomnijmy, że jak wiemy z rozdziału 5, wyrażenie tablicowe tak naprawdę nie zawiera danych – jest odzwierciedleniem danych zawartych w tabelach bazowych. Mając to na uwadze, modyfikowanie wyrażeń tablicowych jest w istocie modyfikowaniem danych w tabelach bazowych poprzez wyrażenie tablicowe. Tak jak w przypadku instrukcji *SELECT* do wyrażenia tablicowego, przy użyciu instrukcji modyfikujących definicja wyrażenia tablicowego zostaje rozwinięta, tak więc faktyczne działanie jest wykonywane wobec tabel bazowych.

Modyfikowanie danych przy użyciu wyrażeń tablicowych podlega kilku ograniczeniom logicznym. Na przykład:

- Jeśli zapytanie definiujące wyrażenie tablicowe złącza tabele, w tej samej instrukcji modyfikowania możemy zmieniać tylko jedną ze stron złączenia, ale nie obie.
- Nie można aktualizować kolumny, która jest wynikiem obliczeń; SQL Server nie będzie próbował odtwarzać źródłowych danych.
- Instrukcje *INSERT* muszą specyfikować wartości dla dowolnych kolumn w tabeli bazowej, których nie można uzyskać niejawnie. Kolumna może uzyskać wartość niejawnie, jeśli dopuszcza znaczniki *NULL*, ma ustawioną wartość domyślną, ma właściwość *identity* lub jest typu *ROWVERSION*.

Dodatkowe wymagania znaleźć można w dokumentacji SQL Server Books Online.

Jednym z powodów modyfikowania danych poprzez wyrażenie tablicowe jest łatwiejsze debugowanie i rozwiązywanie problemów. Na przykład pokazany wcześniej listing 8-1 zawiera następującą instrukcję *UPDATE*.

```
UPDATE OD
  SET discount += 0.05
FROM dbo.OrderDetails AS OD
  JOIN dbo.Orders AS O
    ON OD.orderid = O.orderid
WHERE O.custid = 1;
```

Żałujemy, że w celu rozwiązywania problemów chcemy najpierw sprawdzić, które wiersze zostaną zmodyfikowane przez tę instrukcję, bez rzeczywistego modyfikowania danych. Jedną z możliwości jest przerobienie kodu tak, by używał instrukcji *SELECT*, a po zakończeniu poprawek powrót do stosowania instrukcji *UPDATE*. Jednak zamiast ciągłego wykonywania takich przeróbek możemy po prostu użyć wyrażenia tablicowego – możemy zdefiniować wyrażenie tablicowe oparte na *SELECT* ze złączeniem

i wykonać instrukcję *UPDATE* względem wyrażenia tablicowego. W poniższym przykładzie użyłem wyrażenia CTE.

```
WITH C AS
(
    SELECT custid, OD.orderid,
           productid, discount, discount + 0.05 AS newdiscount
    FROM dbo.OrderDetails AS OD
    JOIN dbo.Orders AS O
      ON OD.orderid = O.orderid
    WHERE O.custid = 1
)
UPDATE C
    SET discount = newdiscount;
```

Poniższy kod używa tabeli pochodnej.

```
UPDATE D
    SET discount = newdiscount
FROM ( SELECT custid, OD.orderid,
           productid, discount, discount + 0.05 AS newdiscount
    FROM dbo.OrderDetails AS OD
    JOIN dbo.Orders AS O
      ON OD.orderid = O.orderid
    WHERE O.custid = 1 ) AS D;
```

Dzięki wyrażeniu tablicowemu prostsze jest rozwiązywanie problemów, ponieważ zawsze możemy wydzielić instrukcję *SELECT*, która definiuje wyrażenie tablicowe i uruchomić ją bez wykonywania jakichkolwiek zmian danych. W tym przypadku wyrażenie tablicowe zostało zastosowane dla wygody, jednakże w niektórych sytuacjach użycie wyrażenia tablicowego jest jedyną możliwością. Aby zilustrować taki przypadek, posłużymy się tabelą nazwaną *T1*, którą utworzymy i wypełnimy danymi, uruchamiając poniższy kod.

```
DROP TABLE IF EXIST dbo.T1;
CREATE TABLE dbo.T1(col1 INT, col2 INT);
GO

INSERT INTO dbo.T1(col1) VALUES(10),(20),(30);

SELECT * FROM dbo.T1;
```

Instrukcja *SELECT* zwraca następujące wyniki, pokazując aktualną zawartość tabeli *T1*.

col1	col2
10	NULL
20	NULL
30	NULL

Żałujemy, że chcemy zaktualizować tabelę, ustawiając w kolumnie *col2* numery wierszy uzyskane przy użyciu funkcji *ROW_NUMBER*. Problem polega na tym, że nie jest

dozwolone stosowanie funkcji *ROW_NUMBER* (podobnie jak innych funkcji okna) w klauzuli *SET* instrukcji *UPDATE*. Spróbujmy uruchomić poniższy kod:

```
UPDATE dbo.T1
  SET col2 = ROW_NUMBER() OVER(ORDER BY col1);
```

Otrzymamy następujący komunikat o błędzie:

```
Msg 4108, Level 15, State 1, Line 2
Windowed functions can only appear in the SELECT or ORDER BY clauses.
```

(Funkcje okna mogą występować jedynie w klauzulach *SELECT* lub *ORDER BY*.)

Aby ominąć ten problem, definiujemy wyrażenie tablicowe, które zwraca zarówno kolumnę, która ma być aktualizowana (*col2*), jak i kolumnę wyników uzyskanych z funkcji *ROW_NUMBER* (nazwaną *rownum*). Zewnętrzną instrukcją do wyrażenia tablicowego może teraz być *UPDATE*, ustawiająca kolumnę *col2* przy użyciu wartości kolumny *rownum*. Poniższy przykład używa wyrażenia CTE.

```
WITH C AS
(
  SELECT col1, col2, ROW_NUMBER() OVER(ORDER BY col1) AS rownum
  FROM dbo.T1
)
UPDATE C
  SET col2 = rownum;
```

Odpytamy tabelę, by sprawdzić wyniki aktualizacji.

```
SELECT * FROM dbo.T1;
```

Zapytanie to zwraca następujące wyniki:

col1	col2
10	1
20	2
30	3

Modyfikacje przy użyciu opcji *TOP* i *OFFSET-FETCH*

Język T-SQL pozwala na bezpośrednie użycie opcji *TOP* w instrukcjach *INSERT*, *UPDATE*, *DELETE* i *MERGE*. Jeśli użyjemy opcji *TOP* w takich wyrażeniach, system SQL Server przerwie wykonywanie instrukcji modyfikowania, kiedy przetworzona zostanie określona liczba wierszy. Niestety, inaczej niż w przypadku instrukcji *SELECT*, nie możemy wyspecyfikować klauzuli *ORDER BY* dla opcji *TOP* użytej w instrukcji modyfikacji. Zasadniczo, zostaną zmodyfikowane te wiersze, do których SQL Server najpierw uzyska dostęp, jest to więc działanie niedeterministyczne.

Filtr *OFFSET-FETCH* nie może być bezpośrednio stosowany w modyfikacjach, gdyż wymaga on klauzuli *ORDER BY*, która nie jest obsługiwana przez instrukcje modyfikujące.

Typowym przykładem użycia modyfikacji z opcją *TOP* jest sytuacja, kiedy przeprowadzana jest bardzo wielka modyfikacja, na przykład duże operacje usuwania, i chcemy rozdzielić ją na kilka mniejszych części.

Zademonstrujemy działanie modyfikacji wraz z opcją *TOP* przy użyciu tabeli *dbo.Orders*, którą tworzymy i wypełniamy danymi, uruchamiając poniższy kod.

```
DROP TABLE IF EXIST dbo.OrderDetails, dbo.Orders;
```

```
CREATE TABLE dbo.Orders
(
    orderid          INT          NOT NULL,
    custid           INT          NULL,
    empid            INT          NOT NULL,
    orderdate        DATETIME     NOT NULL,
    requireddate     DATETIME     NOT NULL,
    shippeddate       DATETIME     NULL,
    shipperid        INT          NOT NULL,
    freight          MONEY        NOT NULL,
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname         NVARCHAR(40) NOT NULL,
    shipaddress      NVARCHAR(60) NOT NULL,
    shipcity         NVARCHAR(15) NOT NULL,
    shipregion       NVARCHAR(15) NULL,
    shippostalcode  NVARCHAR(10) NULL,
    shipcountry      NVARCHAR(15) NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
GO
```

```
INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
```

Poniższy przykład ilustruje użycie instrukcji *DELETE* wraz z opcją *TOP* do usunięcia 50 wierszy z tabeli *Orders*.

```
DELETE TOP(50) FROM dbo.Orders;
```

Ponieważ w instrukcji modyfikowania dla opcji *TOP* nie można wyspecyfikować logicznej instrukcji *ORDER BY*, zapytanie to jest problematyczne w tym sensie, że nie kontrolujemy, które 50 wierszy zostanie usuniętych. To będzie 50 pierwszych wierszy tabeli, do których SQL Server najpierw uzyska dostęp. Kolejność będzie wynikać z fizycznego rozmieszczenia danych i wyborów dokonanych przez optymalizator.

Analogicznie możemy zastosować opcję *TOP* w instrukcjach *UPDATE* i *INSERT*, jednak i tu nie jest dopuszczona instrukcja *ORDER BY*. Przykład użycia instrukcji *UPDATE* wraz z opcją *TOP* prezentuje poniższy kod, który aktualizuje 50 wierszy tabeli *Orders*, zwiększając wartości *freight* o 10.


```
UPDATE TOP(50) dbo.Orders  
SET freight += 10.00;
```

I ponownie, nie możemy kontrolować, które wiersze zostaną zaktualizowane; będzie to 50 pierwszych wierszy, do których SQL Server uzyska dostęp.

W praktyce zazwyczaj będziemy chcieli wiedzieć, które wiersze są modyfikowane i nie będziemy zezwalać na przypadkowy wybór. Aby rozwiązać ten problem, możemy skorzystać z możliwości modyfikowania danych za pośrednictwem wyrażeń tablicowych. Wyrażenie tablicowe definiujemy jako zapytanie *SELECT* z opcją *TOP*, bazując na logicznej klauzuli *ORDER BY*, która określa pierwszeństwo wierszy. Następnie możemy wykonać instrukcję modyfikowania wobec wyrażenia tablicowego.

Na przykład poniższy kod powoduje usunięcie 50 zamówień o najmniejszych wartościach identyfikatora zamówienia.

```
WITH C AS  
(  
    SELECT TOP(50) *  
    FROM dbo.Orders  
    ORDER BY orderid  
)  
DELETE FROM C;
```

Analogicznie poniższy kod aktualizuje 50 zamówień o najwyższych wartościach identyfikatora zamówienia, zwiększając wartości *freight* o 10.

```
WITH C AS  
(  
    SELECT TOP(50) *  
    FROM dbo.Orders  
    ORDER BY orderid DESC  
)  
UPDATE C  
SET freight += 10.00;
```

Zamiast opcji *TOP* w wewnętrznych zapytaniach *SELECT* możemy użyć opcji *OFFSET-FETCH*, jak poniżej.

```
WITH C AS  
(  
    SELECT *  
    FROM dbo.Orders  
    ORDER BY orderid  
    OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY  
)  
DELETE FROM C;
```

A oto poprawiony przykład dla instrukcji *UPDATE*.

```
WITH C AS  
(  
    SELECT *  
    FROM dbo.Orders
```

```

ORDER BY orderid DESC
OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY
)
UPDATE C
SET freight += 10.00;

```

Klauzula **OUTPUT**

Zazwyczaj instrukcja modyfikowania po prostu zmienia dane i nie zwraca żadnych informacji poza liczbą przetworzonych wierszy. Niekiedy jednak chcielibyśmy uzyskać dane ze zmodyfikowanych wierszy, na przykład dla celów rozwiązywania problemów, inspekcji lub archiwizowania. Język T-SQL obsługuje taką funkcjonalność poprzez klauzulę *OUTPUT* dołączaną do instrukcji modyfikowania. W tej klauzuli specyfikujemy atrybuty i wyrażenia, które mają być zwracane z modyfikowanych wierszy.

Klauzula *OUTPUT* jest zaprojektowana podobnie do klauzuli *SELECT* z tym, że nazwy atrybutów trzeba poprzedzić prefiksem *inserted* (wstawione) lub *deleted* (usunięte). W przypadku instrukcji *INSERT* korzystamy ze słowa kluczowego *inserted*; w instrukcji *DELETE* ze słowa kluczowego *deleted*; w przypadku instrukcji *UPDATE* ze słowa kluczowego *deleted*, jeśli interesuje nas stan wiersza przed wykonaniem zmiany, oraz *inserted*, jeśli interesuje nas wiersz po zmianie.

Klauzula *OUTPUT* zwróci żądane atrybuty ze zmodyfikowanych wierszy jako zbiór wyników, podobnie jak instrukcja *SELECT*. Jeśli chcemy skierować zbiór wyników do tabeli, dodajemy klauzulę *INTO* wraz z nazwą tabeli docelowej. Jeśli chcemy, by zmodyfikowane wiersze zostały zwrócone z powrotem do mechanizmu wywołującego, a także przekierowane do tabeli, trzeba użyć dwóch klauzul *OUTPUT* – jednej z klauzulą *INTO* i drugiej bez tej klauzuli.

W kolejnych podpunktach prezentuję przykłady użycia klauzuli *OUTPUT* z różnymi instrukcjami modyfikowania.

INSERT z klauzulą OUTPUT

Przykładem sytuacji, w której przydatna może być klauzula *OUTPUT* dla instrukcji *INSERT*, może być wstawienie zbioru wierszy do tabeli z kolumną *identity*, gdy chcemy z powrotem zwrócić wszystkie wygenerowane wartości identyfikacyjne. Funkcja *SCOPE_IDENTITY* zwraca tylko ostatnią wartość wygenerowaną w bieżącej sesji, a nie wszystkie wartości wygenerowane poprzez wstawienie zbioru wierszy. Klauzula *OUTPUT* bardzo upraszcza realizację tego zadania. W celu zilustrowania metody najpierw utworzymy tabelę *T1* z kolumną *identity* nazwaną *keycol* i drugą kolumną nazwaną *datacol*.

```

DROP TABLE IF EXIST dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL IDENTITY(1, 1) CONSTRAINT PK_T1 PRIMARY KEY,
    datacol NVARCHAR(40) NOT NULL
);

```

Założmy, że chcemy wstawić do tabeli *T1* wynik zapytania do tabeli *HR.Employees*. Aby zwrócić wszystkie wartości identyfikacyjne wygenerowane przez instrukcję *INSERT*, po prostu dodajemy klauzulę *OUTPUT* i specyfikujemy atrybuty, które chcemy zwrócić.

```

INSERT INTO dbo.T1(datacol)
    OUTPUT inserted.keycol, inserted.datacol
    SELECT lastname
    FROM HR.Employees
    WHERE country = N'USA';

```

Instrukcja ta zwraca następujący zbiór wyników:

keycol	datacol
1	Davis
2	Funk
3	Lew
4	Peled
5	Cameron

(5 row(s) affected)

Podobną metodę można zastosować do zwrócenia wartości sekwencji wygenerowanej dla instrukcji *INSERT* przez funkcję *NEXT VALUE FOR* (albo bezpośrednio, albo w ograniczeniu domyślnym).

Jak już wspomniałem, możemy także skierować zbiór wyników do tabeli. Może ona być rzeczywistą tabelą, tabelą tymczasową lub zmienną tablicową. Kiedy zbiór wyników przechowywany jest w tabeli docelowej, możemy operować na danych wykonując zapytania do tej tabeli. Dla przykładu poniższy kod deklaruje zmienną tablicową nazwaną *@NewRows*, wstawia inny zbiór wyników do tabeli *T1* i kieruje zbiór wyników zwrócony przez klauzulę *OUTPUT* do zmiennej tablicowej. Następnie kod odpytuje zmienną tablicową – tylko po to, by pokazać, jakie dane są w niej przechowywane.

```

DECLARE @NewRows TABLE(keycol INT, datacol NVARCHAR(40));

INSERT INTO dbo.T1(datacol)
    OUTPUT inserted.keycol, inserted.datacol
    INTO @NewRows
    SELECT lastname
    FROM HR.Employees
    WHERE country = N'UK';

SELECT * FROM @NewRows;

```

Kod ten zwraca następujące wyniki, pokazując zawartość zmiennej tablicowej.

keycol	datacol
6	Buck
7	Suurs
8	King
9	Dolgopyatova

(4 row(s) affected)

DELETE z klauzulą OUTPUT

Kolejny przykład ilustruje użycie klauzuli *OUTPUT* z instrukcją *DELETE*. Najpierw uruchomimy poniższy kod, by utworzyć kopię tabeli *Orders* ze schematu *Sales* w schemacie *dbo*.

```

DROP TABLE IF EXIST dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid          INT          NOT NULL,
    custid           INT          NULL,
    empid            INT          NOT NULL,
    orderdate        DATETIME     NOT NULL,
    requireddate     DATETIME     NOT NULL,
    shippeddate       DATETIME     NULL,
    shipperid        INT          NOT NULL,
    freight           MONEY        NOT NULL,
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname          NVARCHAR(40) NOT NULL,
    shipaddress       NVARCHAR(60) NOT NULL,
    shipcity          NVARCHAR(15) NOT NULL,
    shipregion        NVARCHAR(15) NULL,
    shippostalcode   NVARCHAR(10) NULL,
    shipcountry       NVARCHAR(15) NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
GO

INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;

```

Przedstawiony poniżej kod usuwa wszystkie zamówienia, które złożone zostały przed 2016 i zwraca atrybuty z usuniętych wierszy przy użyciu klauzuli *OUTPUT*.

```

DELETE FROM dbo.Orders
OUTPUT
    deleted.orderid,
    deleted.orderdate,
    deleted.empid,
    deleted.custid
WHERE orderdate < '20160101';

```

Ta instrukcja *DELETE* zwraca następujący zbiór wyników:

orderid	orderdate	empid	custid

10248	2014-07-04	5	85
10249	2014-07-05	6	79
10250	2014-07-08	4	34
10251	2014-07-08	3	84
...			
10803	2015-12-30	4	88
10804	2015-12-30	6	72
10805	2015-12-30	2	77
10806	2015-12-31	3	84
10807	2015-12-31	4	27

(560 row(s) affected)

Jeśli chcemy zarchiwizować usuwane wiersze, po prostu dodajemy klauzulę *INTO* i wskazujemy nazwę tabeli archiwum jako tabeli docelowej.

UPDATE z klauzulą OUTPUT

Przy użyciu klauzuli *OUTPUT* w instrukcji *UPDATE* możemy odwoływać się zarówno do postaci zmodyfikowanych wierszy przed zmianą (umieszczając przed nazwami atrybutów prefiks *deleted*), jak i do obrazu wierszy po zmianie (umieszczając przed nazwami atrybutów prefiks *inserted*). W ten sposób możemy uzyskiwać zarówno stare, jak i nowe stany aktualizowanych atrybutów.

Przed zademonstrowaniem użycia klauzuli *OUTPUT* w instrukcji *UPDATE*, uruchomimy poniższy kod, by utworzyć kopię tabeli *Sales.OrderDetails* ze schematu *Sales* w schemacie *dbo*.

```

DROP TABLE IF EXIST dbo.OrderDetails;

CREATE TABLE dbo.OrderDetails
(
    orderid INT NOT NULL,
    productid INT NOT NULL,
    unitprice MONEY NOT NULL
    CONSTRAINT DFT_OrderDetails_unitprice DEFAULT(0),
    qty SMALLINT NOT NULL
    CONSTRAINT DFT_OrderDetails_qty DEFAULT(1),
    discount NUMERIC(4, 3) NOT NULL
    CONSTRAINT DFT_OrderDetails_discount DEFAULT(0),
    CONSTRAINT PK_OrderDetails PRIMARY KEY(orderid, productid),
    CONSTRAINT CHK_discount CHECK (discount BETWEEN 0 AND 1),
    CONSTRAINT CHK_qty CHECK (qty > 0),
    CONSTRAINT CHK_unitprice CHECK (unitprice >= 0)
);
GO

INSERT INTO dbo.OrderDetails SELECT * FROM Sales.OrderDetails;
```

Poniższa instrukcja *UPDATE* zwiększa rabat wszystkich pozycji zamówienia dla produktu 51 o 5% oraz stosuje klauzulę *OUTPUT*, by ze zmodyfikowanych wierszy zwrócić identyfikator produktu, stary rabat i nowy rabat.

```
UPDATE dbo.OrderDetails
    SET discount += 0.05
OUTPUT
    inserted.productid,
    deleted.discount AS olddiscount,
    inserted.discount AS newdiscount
WHERE productid = 51;
```

Instrukcja ta zwraca następujące wyniki:

productid	olddiscount	newdiscount
51	0.000	0.050
51	0.150	0.200
51	0.100	0.150
51	0.200	0.250
51	0.000	0.050
51	0.150	0.200
51	0.000	0.050
51	0.000	0.050
51	0.000	0.050
51	0.000	0.050
...		

(39 row(s) affected)

MERGE z klauzulą *OUTPUT*

Klauzuli *OUTPUT* można również używać w instrukcji *MERGE*. Trzeba jednak pamiętać, że pojedyncza instrukcja *MERGE* może wywoływać wiele różnych działań DML. Oznacza to, że klauzula *OUTPUT* instrukcji *MERGE* może zwrócić wiersze wygenerowane przez różne działania DML. W celu zidentyfikowania, które działania DML utworzyły wiersz wyjściowy, możemy w klauzuli *OUTPUT* wywołać funkcję nazwaną *\$action*, która zwraca ciąg reprezentujący działanie (*INSERT*, *UPDATE* lub *DELETE*). Aby zaprezentować użycie klauzuli *OUTPUT* z instrukcją *MERGE*, użyjemy jednego z przykładów z wcześniejszego podrozdziału „Scalanie danych”. Aby uruchomić ten przykład, trzeba ponownie uruchomić kod z listingu 8-2, by odtworzyć tabele *dbo.Customers* i *dbo.CustomersStage*.

Poniższy kod scala zawartość tabeli *CustomersStage* z tabelą *Customers*, aktualizując klientów, którzy już istnieją w tabeli docelowej i dodając klientów, których ta tabela nie zawiera.

```
MERGE INTO dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
    ON TGT.custid = SRC.custid
WHEN MATCHED THEN
```

```

UPDATE SET
    TGT.companyname = SRC.companyname,
    TGT.phone = SRC.phone,
    TGT.address = SRC.address
WHEN NOT MATCHED THEN
    INSERT (custid, companyname, phone, address)
    VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address)
OUTPUT $action AS theaction, inserted.custid,
    deleted.companyname AS oldcompanyname,
    inserted.companyname AS newcompanyname,
    deleted.phone AS oldphone,
    inserted.phone AS newphone,
    deleted.address AS oldaddress,
    inserted.address AS newaddress;

```

Ta instrukcja *MERGE* korzysta z klauzuli *OUTPUT*, by zwrócić stare i nowe wartości zmodyfikowanych wierszy. Oczywiście w przypadku działań *INSERT* nie istnieją żadne stare wartości, tak więc wszystkie odwołania do usuniętych atrybutów zwracają znaczniki *NULL*. Funkcja *\$action* informuje nas, czy wiersz wyjściowy utworzyło działanie *UPDATE* czy *INSERT*. Poniżej przedstawiono wyniki tej instrukcji *MERGE* (przeformatowane dla poprawy czytelności):

theaction	custid	oldcompanyname	newcompanyname
UPDATE	2	cust 2	AAAAA
UPDATE	3	cust 3	cust 3
UPDATE	5	cust 5	BBBBB
INSERT	6	NULL	cust 6 (new)
INSERT	7	NULL	cust 7 (new)

theaction	custid	oldphone	newphone	oldaddress	newaddress
UPDATE	2	(222) 222-2222	(222) 222-2222	address 2	address 2
UPDATE	3	(333) 333-3333	(333) 333-3333	address 3	address 3
UPDATE	5	(555) 555-5555	CCCCC	address 5	DDDDD
INSERT	6	NULL	(666) 666-6666	NULL	address 6
INSERT	7	NULL	(777) 777-7777	NULL	address 7

(5 row(s) affected)

Zagnieżdżone wyrażenia DML

Klauzula *OUTPUT* zwraca wiersz wyjściowy dla każdego zmodyfikowanego wiersza. Co jednak, jeśli chcielibyśmy skierować do tabeli tylko pewien podzbiór zmodyfikowanych wierszy, na przykład na potrzeby inspekcji? T-SQL obsługuje funkcjonalność nazywaną *zagnieżdżanymi wyrażeniami DML*, która pozwala na wstawianie do tabeli docelowej tylko potrzebnych wierszy, będących podzbiorem całego zbioru zmodyfikowanych wierszy.

W celu zademonstrowania tych możliwości najpierw utworzymy kopię tabeli *Products* ze schematu *Production* w schemacie *dbo*, a także tabelę *dbo.ProductsAudit*, uruchamiając poniższy kod.

```

DROP TABLE IF EXISTS dbo.ProductsAudit, dbo.Products;

CREATE TABLE dbo.Products
(
    productid      INT          NOT NULL,
    productname    NVARCHAR(40) NOT NULL,
    supplierid     INT          NOT NULL,
    categoryid     INT          NOT NULL,
    unitprice      MONEY        NOT NULL
        CONSTRAINT DFT_Products_unitprice DEFAULT(0),
    discontinued   BIT          NOT NULL
        CONSTRAINT DFT_Products_discontinued DEFAULT(0),
    CONSTRAINT PK_Products PRIMARY KEY(productid),
    CONSTRAINT CHK_Products_unitprice CHECK(unitprice >= 0)
);

INSERT INTO dbo.Products SELECT * FROM Production.Products;

CREATE TABLE dbo.ProductsAudit
(
    LSN INT NOT NULL IDENTITY PRIMARY KEY,
    TS DATETIME NOT NULL DEFAULT(CURRENT_TIMESTAMP),
    productid INT NOT NULL,
    colname SYSNAME NOT NULL,
    oldval SQL_VARIANT NOT NULL,
    newval SQL_VARIANT NOT NULL
);

```

Założmy, że zachodzi potrzeba zaktualizowania wszystkich produktów dostarczanych przez dostawcę 1 (supplier 1) poprzez zwiększenie cen o 15%. Trzeba także przejrzeć stare i nowe wartości zaktualizowanych produktów, ale tylko tych, których stara cena była mniejsza niż 20, a nowa cena jest większa lub równa 20.

Zadanie to możemy zrealizować przy użyciu zagnieżdżonych wyrażeń DML. Napiszemy instrukcję *UPDATE* z klauzulą *OUTPUT* i zdefiniujemy tabelę pochodną w oparciu o instrukcję *UPDATE*. Użyjemy instrukcji *INSERT SELECT* do odpytania tabeli pochodnej, filtrując tylko ten podzbiór wierszy, który jest nam potrzebny. Poniżej przedstawiono całe rozwiązanie:

```

INSERT INTO dbo.ProductsAudit(productid, colname, oldval, newval)
SELECT productid, N'unitprice', oldval, newval
FROM (UPDATE dbo.Products
      SET unitprice *= 1.15
      OUTPUT
        inserted.productid,
        deleted.unitprice AS oldval,
        inserted.unitprice AS newval
      WHERE supplierid = 1) AS D
WHERE oldval < 20.0 AND newval >= 20.0;

```


Przypomnijmy sobie wcześniejszą dyskusję na temat logicznego przetwarzania zapytań i wyrażeń tablicowych – wyniki jednego zapytania mogą być użyte jako dane wejściowe kolejnego. W tym przykładzie wyniki klauzuli *OUTPUT* są wejściem dla instrukcji *SELECT*, a następnie dane wyjściowe instrukcji *SELECT* są wstawiane do tabeli.

Poniższe zapytanie odpytuje tabelę *ProductsAudit*.

```
SELECT * FROM dbo.ProductsAudit;
```

Uzyskujemy następujące dane wyjściowe:

LSN	TS	ProductID	ColName	OldVal	NewVal
1	2008-08-05 18:56:04.793	1	unitprice	18.00	20.70
2	2008-08-05 18:56:04.793	2	unitprice	19.00	21.85

Trzy produkty zostały zaktualizowane, ale tylko dwa zostały odfiltrowane przez zapytanie zewnętrzne – dlatego w wynikach prezentowane są tylko te dwa wiersze.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXISTS dbo.OrderDetails, dbo.ProductsAudit, dbo.Products, dbo.Orders,
dbo.Customers, dbo.T1, dbo.MySequences, dbo.CustomersStage;
```

Podsumowanie

W tym rozdziale omówiłem różne aspekty modyfikowania danych – wstawianie, aktualizowanie, usuwanie i scalanie danych. Ponadto opisałem metody modyfikowania danych za pośrednictwem wyrażeń tablicowych, użycie opcji *TOP* (i pośrednio opcji *OFFSET-FETCH*) oraz sposoby zwracania zmodyfikowanych wierszy przy użyciu klauzuli *OUTPUT*.

Ćwiczenia

W tym podrozdziale zawarte są ćwiczenia, które ułatwią lepsze przyswojenie tematyki opisanej w rozdziale. W ćwiczeniach wykorzystywana jest baza danych *TSQV4*.

Ćwiczenie 1

Ćwiczenie to składa się z trzech części. Przed przystąpieniem do nich należy uruchomić poniższy kod, by w bazie danych *TSQV4* utworzyć tabelę *dbo.Customers*.

```
USE TSQV4;
```

```
DROP TABLE IF EXIST dbo.Customers;
```

```
CREATE TABLE dbo.Customers
(
```

```

    custid      INT          NOT NULL PRIMARY KEY,
    companyname NVARCHAR(40) NOT NULL,
    country     NVARCHAR(15) NOT NULL,
    region      NVARCHAR(15) NULL,
    city        NVARCHAR(15) NOT NULL
);

```

Ćwiczenie 1-1

Wstawić do tabeli *dbo.Customers* wiersz, który zawiera:

- *custid*: 100
- *companyname*: Coho Winery
- *country*: USA
- *region*: WA
- *city*: Redmond

Ćwiczenie 1-2

Wstawić do tabeli *dbo.Customers* wszystkich klientów z tabeli *Sales.Customers*, którzy złożyli zamówienia.

Ćwiczenie 1-3

Użyć instrukcji *SELECT INTO* do utworzenia tabeli *dbo.Orders* i wypełnienia jej zamówieniami z tabeli *Sales.Orders*, które zostały złożone w latach 2014 do 2016.

Ćwiczenie 2

Z tabeli *dbo.Orders* usunąć zamówienia złożone przed sierpniem 2014. Użyć klauzuli *OUTPUT*, by zwrócić atrybuty *orderid* i *orderdate* usuniętych zamówień.

- Oczekiwane dane wyjściowe (skrótowe):

orderid	orderdate
10248	2014-07-04
10249	2014-07-05
10250	2014-07-08
10251	2014-07-08
10252	2014-07-09
10253	2014-07-10
10254	2014-07-11
10255	2014-07-12
10256	2014-07-15
10257	2014-07-16
10258	2014-07-17
10259	2014-07-18

```

10260      2014-07-19
10261      2014-07-19
10262      2014-07-22
10263      2014-07-23
...

```

```
(22 row(s) affected)
```

Ćwiczenie 3

Z tabeli *dbo.Orders* usunąć zamówienia złożone przez klientów z Brazylii.

Ćwiczenie 4

Uruchomić poniższe zapytanie w odniesieniu do tabeli *dbo.Customers*; zwrócić uwagę, że w niektórych wierszach, w kolumnie *region* znajdują się znaczniki *NULL*.

```
SELECT * FROM dbo.Customers;
```

Dane wyjściowe tego zapytania są następujące:

custid	companyname	country	region	city
1	Customer NRZBB	Germany	NULL	Berlin
2	Customer MLTDN	Mexico	NULL	México D.F.
3	Customer KBUDE	Mexico	NULL	México D.F.
4	Customer HFBZG	UK	NULL	London
5	Customer HGVLZ	Sweden	NULL	Luleå
6	Customer XHXJV	Germany	NULL	Mannheim
7	Customer QXVLA	France	NULL	Strasbourg
8	Customer QUHWH	Spain	NULL	Madrid
9	Customer RTXGC	France	NULL	Marseille
10	Customer EEALV	Canada	BC	Tsawassen
...				

```
(90 row(s) affected)
```

Zaktualizować tabelę *dbo.Customers* i zmienić wszystkie wartości regionu określone przez znacznik *NULL* na wartość *<None>*. Użyć klauzuli *OUTPUT* do wyświetlenia atrybutów *custid*, *oldregion* i *newregion*.

- Oczekiwane dane wyjściowe (skrócone):

custid	oldregion	newregion
1	NULL	<None>
2	NULL	<None>
3	NULL	<None>
4	NULL	<None>
5	NULL	<None>
6	NULL	<None>
7	NULL	<None>

8	NULL	<None>
11	NULL	<None>
12	NULL	<None>
13	NULL	<None>
14	NULL	<None>
...		

(58 row(s) affected)

Ćwiczenie 5

Zaktualizować wszystkie zamówienia w tabeli *dbo.Orders* złożone przez klientów z Wielkiej Brytanii oraz ustawić wartości *shipcountry*, *shipregion* i *shipcity* za pomocą wartości *country*, *region* i *city* odpowiednich klientów.

Ćwiczenie 6

Uruchomić poniższy kod, aby utworzyć tabele *Orders* i *OrderDetails* i wypełnić je danymi:

```
USE TSQLV4;
```

```
DROP TABLE IF EXISTS dbo.OrderDetails, dbo.Orders;
```

```
CREATE TABLE dbo.Orders
```

```
(
 orderid      INT          NOT NULL,
  custid      INT          NULL,
  empid       INT          NOT NULL,
  orderdate   DATE         NOT NULL,
  requireddate DATE       NOT NULL,
  shippeddate  DATE         NULL,
  shipperid   INT          NOT NULL,
  freight     MONEY        NOT NULL
  CONSTRAINT DFT_Orders_freight DEFAULT(0),
  shipname    NVARCHAR(40) NOT NULL,
  shipaddress NVARCHAR(60) NOT NULL,
  shipcity    NVARCHAR(15) NOT NULL,
  shipregion  NVARCHAR(15) NULL,
  shippostalcode NVARCHAR(10) NULL,
  shipcountry NVARCHAR(15) NOT NULL,
  CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
```

```
CREATE TABLE dbo.OrderDetails
```

```
(
  orderid INT          NOT NULL,
  productid INT        NOT NULL,
  unitprice MONEY      NOT NULL
  CONSTRAINT DFT_OrderDetails_unitprice DEFAULT(0),
  qty      SMALLINT    NOT NULL
);
```

```

    CONSTRAINT DFT_OrderDetails_qty DEFAULT(1),
    discount NUMERIC(4, 3) NOT NULL
    CONSTRAINT DFT_OrderDetails_discount DEFAULT(0),
    CONSTRAINT PK_OrderDetails PRIMARY KEY(orderid, productid),
    CONSTRAINT FK_OrderDetails_Orders FOREIGN KEY(orderid)
    REFERENCES dbo.Orders(orderid),
    CONSTRAINT CHK_discount CHECK (discount BETWEEN 0 AND 1),
    CONSTRAINT CHK_qty CHECK (qty > 0),
    CONSTRAINT CHK_unitprice CHECK (unitprice >= 0)
);
GO

```

```

INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
INSERT INTO dbo.OrderDetails SELECT * FROM Sales.OrderDetails;

```

Napisać i przetestować kod T-SQL wymagany do wyczyszczenia obu tabel i upewnić się, że kod ten działa poprawnie.

Po ukończeniu ćwiczeń uruchomić poniższy kod, by wyczyścić bazę danych.

```

DROP TABLE IF EXISTS dbo.OrderDetails, dbo.Orders, dbo.Customers;

```

Rozwiązania

Podrozdział ten zawiera rozwiązania ćwiczeń wraz z odpowiednimi wyjaśnieniami.

Ćwiczenie 1

Ćwiczenie to obejmuje trzy części, których rozwiązania są przedstawione w kolejnych podpunktach.

Ćwiczenie 1-1

Upewniamy się, że jesteśmy połączeni z bazą danych *TSQLV4*.

```

USE TSQLV4;

```

Poniższej instrukcji *INSERT VALUES* używamy do wstawienia wiersza do tabeli *Customers* z wartościami udostępnionymi w ćwiczeniu.

```

INSERT INTO dbo.Customers(custid, companyname, country, region, city)
VALUES(100, N'Coho Winery', N'USA', N'WA', N'Redmond');

```

Ćwiczenie 1-2

Jednym ze sposobów zidentyfikowania klientów, którzy złożyli zamówienia, jest użycie predykatu *EXISTS*, jak w poniższym zapytaniu:

```

SELECT custid, companyname, country, region, city
FROM Sales.Customers AS C

```

```
WHERE EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

Aby do tabeli *dbo.Customers* wstawić wiersze zwrócone przez to zapytanie, stosujemy instrukcję *INSERT SELECT*:

```
INSERT INTO dbo.Customers(custid, companyname, country, region, city)
  SELECT custid, companyname, country, region, city
  FROM Sales.Customers AS C
  WHERE EXISTS
    (SELECT * FROM Sales.Orders AS O
     WHERE O.custid = C.custid);
```

Ćwiczenie 1-3

Poniższy kod najpierw zapewnia połączenie z bazą danych TSQV4, następnie usuwa tabelę *dbo.Orders*, jeśli taka tabela już istnieje, oraz stosuje instrukcję *SELECT INTO* do utworzenia nowej tabeli *dbo.Orders* i wypełnienia jej zamówieniami z tabeli *Sales.Orders*, które zostały złożone w latach 2014 do 2016.

```
USE TSQV4;

DROP TABLE IF EXIST dbo.Orders;

SELECT *
INTO dbo.Orders
FROM Sales.Orders
WHERE orderdate >= '20140101'
   AND orderdate < '20160101';
```

Ćwiczenie 2

Aby usunąć zamówienia złożone przed sierpniem 2014, potrzebna jest instrukcja *DELETE* z filtrem opartym na predykacie *orderdate < '20140801'*. Zgodnie z żądaniem, używamy klauzuli *OUTPUT*, by zwrócić atrybuty usuniętych wierszy.

```
DELETE FROM dbo.Orders
  OUTPUT deleted.orderid, deleted.orderdate
WHERE orderdate < '20140801';
```

Ćwiczenie 3

Aby wykonać to ćwiczenie, trzeba napisać instrukcję *DELETE*, która usuwa wiersze z jednej tabeli (*dbo.Orders*) pod warunkiem, że istnieje zgodny wiersz w innej tabeli (*dbo.Customers*). Jednym ze sposobów rozwiązania problemu jest użycie standardowej instrukcji *DELETE* z predykatem *EXISTS* w klauzuli *WHERE*, jak w poniższym kodzie:

```
DELETE FROM dbo.Orders
WHERE EXISTS
  (SELECT *
```

```
FROM dbo.Customers AS C
WHERE Orders.custid = C.custid
      AND C.country = N'Brazil');
```

Ta instrukcja *DELETE* usuwa wiersze z tabeli *dbo.Orders*, dla których istnieją odpowiednie wiersze w tabeli *dbo.Customers* o tym samym identyfikatorze klienta co identyfikator klienta dla zamówienia i dla których krajem klienta jest Brazylia.

Innym sposobem rozwiązywania tego zadania jest użycie specyficznej dla języka T-SQL składni instrukcji *DELETE* opartej o złączenie, jak w poniższym kodzie:

```
DELETE FROM O
FROM dbo.Orders AS O
      JOIN dbo.Customers AS C
        ON O.custid = C.custid
WHERE country = N'Brazil';
```

Zadaniem złączenia tabel *dbo.Orders* i *dbo.Customers* jest filtrowanie. Złączenie porównuje każde zamówienie z klientem, który złożył zamówienie. Klauzula *WHERE* filtruje tylko te wiersze, dla których krajem klienta jest Brazylia. Klauzula *DELETE FROM* odnosi się do aliasu *O* reprezentującego tabelę *Orders*, wskazując, że tabela *Orders* jest tabelą docelową operacji *DELETE*.

Alternatywnie można zastosować instrukcję *MERGE*. Choć zazwyczaj instrukcja *MERGE* jest stosowana, gdy trzeba wykonać różne działania w oparciu o logikę warunkową, możemy jej również użyć do wykonania jednego działania, gdy pewien predykat przyjmuje wartość *TRUE*. Inaczej mówiąc, możemy użyć instrukcji *MERGE* tylko z samą klauzulą *WHEN MATCHED*; nie musimy mieć klauzuli *WHEN NOT MATCHED*. Pokazana poniżej instrukcja *MERGE* obsługuje żądanie określone w ćwiczeniu.

```
MERGE INTO dbo.Orders AS O
USING dbo.Customers AS C
      ON O.custid = C.custid
      AND country = N'Brazil'
WHEN MATCHED THEN DELETE;
```

Ta instrukcja *MERGE* definiuje tabelę *dbo.Orders* jako tabelę docelową, a tabelę *dbo.Customers* jako źródło. Zamówienie jest usuwane z tabeli docelowej (*dbo.Orders*), jeśli znaleziony zostaje pasujący wiersz w tabeli źródłowej (*dbo.Customers*) – ten sam identyfikator klienta i kraj to Brazylia.

Ćwiczenie 4

Ćwiczenie to wymaga napisania instrukcji *UPDATE*, filtrującej tylko te wiersze, dla których atrybut *region* ma wartość *NULL*. Pamiętajmy, by podczas wyszukiwania znaczników *NULL* użyć predykatu *IS NULL*, a nie operatora równości. Do zwrócenia żądanych informacji użyjemy klauzuli *OUTPUT*. Poniżej przedstawiono całą instrukcję *UPDATE*:

```

UPDATE dbo.Customers
    SET region = '<None>'
OUTPUT
    deleted.custid,
    deleted.region AS oldregion,
    inserted.region AS newregion
WHERE region IS NULL;

```

Ćwiczenie 5

Jeden ze sposobów rozwiązania tego zadania polega na użyciu specyficznej dla języka T-SQL składni instrukcji *UPDATE* opartej na złączeniu. Łączymy tabele *dbo.Orders* i *dbo.Customers* według zgodności identyfikatorów klienta. W klauzuli *WHERE* filtrujemy tylko te wiersze, dla których krajem klienta jest Wielka Brytania. W klauzuli *UPDATE* specyfikujemy alias przypisany do tabeli *dbo.Orders*, by wskazać, że jest to tabela docelowa modyfikacji. W klauzuli *SET* przypisujemy wartości atrybutów lokalizacji wysyłki zamówienia do atrybutów lokalizacji odpowiednich klientów. Poniżej przedstawiono całą instrukcję *UPDATE*:

```

UPDATE O
    SET shipcountry = C.country,
        shipregion = C.region,
        shipcity = C.city
FROM dbo.Orders AS O
    JOIN dbo.Customers AS C
        ON O.custid = C.custid
WHERE C.country = 'UK';

```

Inne rozwiązanie tego ćwiczenia polega na użyciu wyrażeń CTE. Definiujemy wyrażenie CTE, które łączy tabele *dbo.Orders* i *dbo.Customers* oraz zwraca zarówno atrybuty lokalizacji z tabeli *dbo.Orders*, jak i atrybuty lokalizacji źródła z tabeli *dbo.Customers*. Zapytaniem zewnętrznym będzie instrukcja *UPDATE* modyfikująca atrybuty docelowe za pomocą wartości atrybutów źródłowych. Poniżej przedstawiono całą instrukcję:

```

WITH CTE_UPD AS
(
    SELECT
        O.shipcountry AS ocountry, C.country AS ccountry,
        O.shipregion AS oregion, C.region AS cregion,
        O.shipcity AS ocity, C.city AS ccity
    FROM dbo.Orders AS O
        JOIN dbo.Customers AS C
            ON O.custid = C.custid
    WHERE C.country = 'UK'
)
UPDATE CTE_UPD
    SET ocountry = ccountry, oregion = cregion, ocity = ccity;

```


Do wykonania tego zadania możemy również użyć instrukcji *MERGE*. Przy użyciu klauzuli *WHEN MATCHED* z działaniem *UPDATE* możemy napisać rozwiązanie, które jest logicznym ekwiwalentem dwóch poprzednich, co ilustruje poniższy kod:

```
MERGE INTO dbo.Orders AS O
USING dbo.Customers AS C
  ON O.custid = C.custid
 AND C.country = 'UK'
WHEN MATCHED THEN
  UPDATE SET shipcountry = C.country,
             shipregion = C.region,
             shipcity = C.city;
```

Ćwiczenie 6

Pomiędzy tabelami *OrderDetails* i *Orders* istnieje ograniczenie klucza obcego. W takim przypadku możliwe jest wyczyszczenie tabeli odwołującej się, ale nie tabeli odwołania, nawet jeśli w tabeli odwołującej się nie istnieje żaden wiersz powiązany z tabelą odwołania. Konieczne jest usunięcie ograniczenia klucza obcego, wyczyszczenie obu tabel, po czym przywrócenie ograniczenia, jak poniżej:

```
ALTER TABLE dbo.OrderDetails DROP CONSTRAINT FK_OrderDetails_Orders;

TRUNCATE TABLE dbo.OrderDetails;
TRUNCATE TABLE dbo.Orders;

ALTER TABLE dbo.OrderDetails ADD CONSTRAINT FK_OrderDetails_Orders
  FOREIGN KEY(orderid) REFERENCES dbo.Orders(orderid);
```

Po zakończeniu ćwiczeń należy wykonać poniższy kod, aby wyczyścić bazę danych:

```
DROP TABLE IF EXISTS dbo.OrderDetails, dbo.Orders, dbo.Customers;
```


ROZDZIAŁ 9

Tabele temporalne

Gdy modyfikujemy dane w tabelach, w normalnych okolicznościach tracimy jakiekolwiek informacje o wcześniejszym stanie wierszy. Możemy uzyskiwać dostęp tylko do bieżącego stanu. Co jednak, jeśli potrzebujemy mieć możliwość dostępu do historycznych wartości danych? Potrzeba taka może wynikać ze względu na inspekcję, prowadzenie analizy dla różnych momentów czasowych, porównania bieżącego stanu z wcześniejszym, a także w przypadku tak zwanych wolno zmieniających się wymiarów (szczegółowe omówienie tego pojęcia zawiera artykuł w Wikipedii https://en.wikipedia.org/wiki/Slowly_changing_dimension). Można też rozważać potrzebę przywrócenia wcześniejszego stanu wierszy w razie przypadkowego usunięcia lub aktualizacji. Rozwiązanie takie można stworzyć samemu, wykorzystując wyzwalacze. Jednak poczynawszy od wersji Microsoft SQL Server 2016 można użyć lepszego wyjścia – wbudowanej funkcjonalności nazywanej *wersjonowanymi systemowo tabelami temporalnymi* (*system-versioned temporal tables*). Ta nowa funkcjonalność zapewnia rozwiązanie, które jest zarówno prostsze, jak i bardziej wydajne, niż najlepiej zaprojektowane własne rozwiązanie.

Wersjonowana systemowo tabela temporalna zawiera dwie kolumny reprezentujące okres ważności oraz połączoną tabelę historii z lustrzanym schematem, przechowującą starsze stany modyfikowanych wierszy. Gdy chcemy zmodyfikować dane, odwołujemy się do bieżącej tabeli, posługując się zwykłymi instrukcjami modyfikowania danych. SQL Server automatycznie aktualizuje kolumny okresu ważności i przenosi starsze wersje wierszy do tabeli historii. Podczas odpytywania danych, jeśli interesuje nas stan bieżący, odwołujemy się do tabeli jak zwykle. Jeśli chcemy uzyskać dostęp do starszych stanów, nadal odpytujemy bieżącą tabelę, ale dołączamy dodatkową klauzulę wskazującą, że chodzi nam o wcześniejszy stan lub przedział czasu. SQL Server w tle odpyta zarówno tabelę bieżącą, jak i tabelę historii.

Standard SQL obsługuje trzy typy tabel temporalnych:

- Wersjonowane systemowo tabele temporalne, polegające na systemowym czasie transakcji do definiowania okresu ważności wiersza.
- Tabele oparte na czasie aplikacji, które polegają na zdefiniowanym w aplikacji okresie ważności wiersza. Oznacza to, że możliwe jest zdefiniowanie okresu ważności, który dopiero stanie się skuteczny (w przyszłości).
- Tabele bitemporalne łączą oba wymienione typy (czas transakcji i czas zdefiniowany w aplikacji).

SQL Server 2016 obsługuje tylko wersjonowane systemowo tabele temporalne. Mam nadzieję, że firma Microsoft doda obsługę czasu aplikacji w przyszłych wydaniach SQL Server.

W tym rozdziale zajmę się wersjonowanymi systemowo tabelami temporalnymi: ich tworzeniem, modyfikowaniem danych i odpytaniem danych.

Tworzenie tabel

Przy tworzeniu wersjonowanej systemowo tabeli systemowej musimy zapewnić, że definicja tabeli będzie zawierała wszystkie z poniższych elementów:

- Klucz główny.
- Dwie kolumny zdefiniowane jako typ *DATETIME2* o dowolnej precyzji, nie dopuszczające znaczników NULL, które będą reprezentować początek i koniec okresu wartości wiersza w strefie czasowej UTC.
- Kolumnę startową, która musi być oznakowana opcją *GENERATED ALWAYS AS ROW START*.
- Kolumnę końcową, która musi być oznakowana opcją *GENERATED ALWAYS AS ROW END*.
- Oznaczenie kolumn okresu opcją *PERIOD FOR SYSTEM_TIME (<startcol>, <endcol>)*.
- Opcję tabeli *SYSTEM_VERSIONING* ustawioną jako ON (włączoną).
- Połączoną tabelę historii (którą SQL Server może dla nas utworzyć), która będzie przechowywać wcześniejsze stany zmodyfikowanych wierszy.

Opcjonalnie można oznaczyć kolumny okresu jako ukryte, przez co nie będą zwracane przy zapytaniach typu *SELECT ** i będą ignorowane przy wstawianiu danych.

Poniższy kod tworzy wersjonowaną systemowo tabelę temporalną o nazwie *Employees* oraz połączoną z nią tabelę historii *EmployeesHistory*:

```
USE TSQLV4;
```

```
-- Create Employees table
CREATE TABLE dbo.Employees
(
    empid          INT                                NOT NULL
        CONSTRAINT PK_Employees PRIMARY KEY NONCLUSTERED,
    empname        VARCHAR(25)                        NOT NULL,
    department     VARCHAR(50)                        NOT NULL,
    salary         NUMERIC(10, 2)                     NOT NULL,
    sysstart       DATETIME2(0)
        GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
    sysend         DATETIME2(0)
        GENERATED ALWAYS AS ROW END   HIDDEN NOT NULL,
    PERIOD FOR SYSTEM_TIME (sysstart, sysend),
```

```
INDEX ix_Employees CLUSTERED(empid, sysstart, sysend)
)
WITH ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.EmployeesHistory ) );
```

Warto przejrzeć listę wymaganych elementów, aby się upewnić, że potrafimy zidentyfikować je w przykładowym kodzie.

Zakładając, że tabela historii nie istnieje, gdy uruchomimy ten kod, SQL Server ją utworzy. Jeśli nie podamy jawnie nazwy tej tabeli, SQL Server sam przypisze nazwę, używając formy *MSSQL_TemporalHistoryFor_<object_id>*, gdzie *object_id* jest systemowym identyfikatorem bieżącej tabeli (w tym przykładzie *Employees*). Tworzona przez SQL Server tabela historii ma schemat odzwierciedlający oryginalną tabelę, ale z następującymi różnicami:

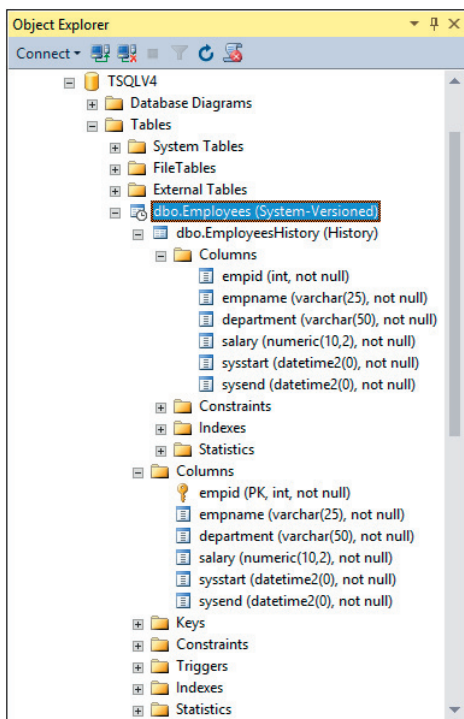
- Brak klucza głównego.
- Indeks klastrowy dla klucza (<endcol>, <startcol>), z kompresją stron, o ile jest możliwa.
- Kolumny okresu nie są oznaczone żadnymi specjalnymi opcjami, jak *GENERATED ALWAYS AS ROW START/END* lub *HIDDEN*.
- Kolumny okresu nie są oznaczone opcją *PERIOD FOR SYSTEM_TIME*.
- Tabela historii nie jest oznaczona opcją *SYSTEM_VERSIONING*.

Jeśli tabela historii już istnieje przy tworzeniu tabeli bieżącej, SQL Server weryfikuje zarówno jej zgodność ze schematem (zgodnie z powyższym opisem), jak i dane (nie może być nakładających się okresów). Jeśli tabela historii nie przejdzie tego testu, SQL Server wygeneruje komunikat o błędzie i nie utworzy tabeli bieżącej. Opcjonalnie możliwe jest wyłączenie przez SQL Server wykonania testu spójności danych.

Jeśli następnie przejrzymy drzewo obiektów w panelu Object Explorer narzędzia SQL Server Management Studio (SSMS), zauważymy, że tabela *Employees* jest oznaczona jako (System-Versioned), a poniżej niej znajduje się połączona tabela *EmployeesHistory* oznaczona jako (History), co widać na rysunku 9-1.

Możliwe jest również przekształcenie istniejącej zwykłej tabeli, już zawierającej dane, w tabelę temporalną. Dla przykładu założymy, że mamy tabelę *Employees* w naszej bazie danych i chcemy przekształcić ją na tabelę temporalną. Musimy najpierw zmodyfikować tę tabelę, dodając kolumny okresu i oznaczając je odpowiednio, wykonując poniższy kod (nie należy uruchamiać poniższego przykładu, gdyż nasza tabela *Employees* już jest temporalna):

```
ALTER TABLE dbo.Employees ADD
  sysstart DATETIME2(0) GENERATED ALWAYS AS ROW START HIDDEN NOT NULL
  CONSTRAINT DFT_Employees_sysstart DEFAULT('19000101'),
  sysend DATETIME2(0) GENERATED ALWAYS AS ROW END HIDDEN NOT NULL
  CONSTRAINT DFT_Employees_sysend DEFAULT('99991231 23:59:59'),
  PERIOD FOR SYSTEM_TIME (sysstart, sysend);
```



RYСУNEK 9-1 Tabela temporalna i powiązana z nią tabela historii w SSMS

Zwróćmy uwagę na domyślne wartości ustawiające przedział ważności dla istniejących wierszy. Możemy wybrać dowolny czas startowy okresu ważności, o ile tylko nie jest to punkt z przyszłości. Wartość końcowa powinna być maksymalną wartością obsługiwaną przez typ danych.

Następnie modyfikujemy tabelę, aby włączyć wersjonowanie i połączyć ją z tabelą historii (ponownie nie należy uruchamiać tego kodu, jeśli tabela *Employees* została już utworzona jako wersjonowana):

```
ALTER TABLE dbo.Employees
SET ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.EmployeesHistory ) );
```

Trzeba pamiętać, że jeśli oznaczymy kolumny okresu jako ukryte, zapytanie takie jak *SELECT ** ich nie zwróci. Możemy to sprawdzić na naszej tabeli *Employees*:

```
SELECT *
FROM dbo.Employees;
```

Otrzymamy następujący wynik:

```
empid empname department salary
-----
```

Jeśli chcemy otrzymać w wynikach kolumny okresu, musimy je jawnie wymienić na liście *SELECT*, jak poniżej:

```
SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.Employees;
```

Tym razem wynik będzie następujący:

empid	empname	department	salary	sysstart	sysend
-----	-----	-----	-----	-----	-----

SQL Server obsługuje dokonywanie zmian w strukturze tabeli temporalnej bez konieczności wcześniejszego wyłączenia wersjonowania. Zmianę wykonujemy tylko wobec tabeli bieżącej, zaś SQL Server zastosuje ją do obydwu tabel – bieżącej i historii. Oczywiście, jeśli zechcemy dodać kolumnę nie dopuszczającą znaczników *NULL*, musimy dołączyć dla niej wartość domyślną. Dla przykładu założmy, że chcemy dodać kolumnę *hiredate* (data zatrudnienia) do tabeli *Employees* jako *NOT NULL* – musimy podać jakąś datę domyślną, na przykład 1 stycznia 1900. Można to zrobić wykonując poniższy kod:

```
ALTER TABLE dbo.Employees
ADD hiredate DATE NOT NULL
CONSTRAINT DFT_Employees_hiredate DEFAULT('19000101');
```

Następnie można zaktualizować daty zatrudnienia istniejących pracowników.

Odpytajmy tabelę *Employees* po dodaniu kolumny *hiredate*:

```
SELECT *
FROM dbo.Employees;
```

Otrzymamy poniższy wynik:

empid	empname	department	salary	hiredate
-----	-----	-----	-----	-----

A teraz odpytajmy tabelę *EmployeesHistory*:

```
SELECT *
FROM dbo.EmployeesHistory;
```

Można zauważyć, że również w tej tabeli pojawiła się kolumna *hiredate*:

empid	empname	department	salary	sysstart	sysend	hiredate
-----	-----	-----	-----	-----	-----	-----

SQL Server dodał kolumnę *hiredate* do obu tabel, ale ograniczenie wartości domyślnej zostało dołączone tylko do tabeli bieżącej. Tym niemniej, gdy w tabeli historii były już jakieś wiersze, SQL Server przypisałby wartość domyślną kolumnie *hiredate* dla tych wierszy.

Założmy, że chcemy usunąć kolumnę *hiredate* z obu tabel. Najpierw należy usunąć ograniczenie wartości domyślnej z tabeli bieżącej, wykonując poniższy kod:

```
ALTER TABLE dbo.Employees
    DROP CONSTRAINT DFT_Employees_hiredate;
```

Następnie można usunąć kolumnę z tabeli bieżącej, wykonując poniższy kod:

```
ALTER TABLE dbo.Employees
    DROP COLUMN hiredate;
```

SQL Server usunie kolumnę z obydwu tabel.

Modyfikowanie danych

Modyfikacje tabel temporalnych wykonujemy podobnie, jak zwykłych tabel. Zmiany wykonujemy tylko w tabeli bieżącej, używając instrukcji *INSERT*, *UPDATE*, *DELETE* lub *MERGE* (instrukcja *TRUNCATE* nie jest obsługiwana dla tabel temporalnych w SQL Server 2016). W tle SQL Server zaktualizuje kolumny okresu i przeniesie starsze wersje wierszy do tabeli historii. Przypomnijmy, że kolumny okresu oznaczają przedział ważności wiersza wyrażony w czasie *UTC*.

Jeśli kolumny okresu zostały zdefiniowane jako ukryte, jak w naszym przykładzie, po prostu pomijamy je w wyrażeniach *INSERT*. Jeśli nie zostały oznaczone jako ukryte, to możemy nadal je pominąć, o ile będziemy stosować się do zalecanych praktyk i jawnie wymieniać nazwy kolumn docelowych. Jeśli nie są ukryte i nie zostaną jawnie wymienione nazwy kolumn docelowych, konieczne jest użycie słowa kluczowego *DEFAULT* jako wartości dla tych kolumn.

W poniższych przykładach zademonstruję różne modyfikacje tabeli *Employees*, za każdym razem podając czas *UTC*, w którym wykonałem te zmiany. Oczywiście czasy modyfikacji będą inne, gdy Czytelnik będzie uruchamiał te przykłady, zatem niezłym pomysłem jest zanotowanie czasu ich wykonania. Do uzyskania tej informacji można wykorzystać funkcję *SYSUTCDATETIME*.

Poniższy kod dodaje kilka wierszy do tabeli *Employees* (mój czas wykonania to 2016-02-16 17:08:41):

```
INSERT INTO dbo.Employees(empid, empname, department, salary)
VALUES(1, 'Sara', 'IT', 50000.00),
      (2, 'Don', 'HR', 45000.00),
      (3, 'Judy', 'Sales', 55000.00),
      (4, 'Yael', 'Marketing', 55000.00),
      (5, 'Sven', 'IT', 45000.00),
      (6, 'Paul', 'Sales', 40000.00);
```

Teraz odpytamy zarówno tabelę bieżącą, jak i historii, aby zobaczyć, co SQL Server zrobił w tle:

```
SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.Employees;

SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.EmployeesHistory;
```


Tabela bieżąca ma sześć nowych wierszy, przy czym kolumna *sysstart* odzwierciedla czas modyfikacji, zaś kolumna *sysend* zawiera maksymalną możliwą wartość dla tego typu danych z wybraną dokładnością:

empid	empname	department	salary	sysstart	sysend
1	Sara	IT	50000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	IT	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
6	Paul	Sales	40000.00	2016-02-16 17:08:41	9999-12-31 23:59:59

Okres ważności wskazuje, że wiersze są uważane za bieżące (ważne) od chwili ich wstawienia i bez daty końcowej.

Tabela historii w tym momencie jest jeszcze pusta:

empid	empname	department	salary	sysstart	sysend
-----	-----	-----	-----	-----	-----

Poniższy kod usuwa wiersz dla pracownika o identyfikatorze równym 6 (mój czas wykonania to 2016-02-16 17:15:26):

```
DELETE FROM dbo.Employees
WHERE empid = 6;
```

SQL Server przenosi usunięty wiersz do tabeli historii, ustawiając wartość *sysend* na czas usunięcia. Zawartość tabeli bieżącej wygląda teraz następująco:

empid	empname	department	salary	sysstart	sysend
-----	-----	-----	-----	-----	-----
1	Sara	IT	50000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	IT	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59

A oto zawartość tabeli historii:

empid	empname	department	salary	sysstart	sysend
-----	-----	-----	-----	-----	-----
6	Paul	Sales	40000.00	2016-02-16 17:08:41	2016-02-16 17:15:26

Aktualizacja wiersza jest traktowana jako usunięcie i wstawienie. SQL Server przenosi starą wersję wiersza do tabeli historii z czasem transakcji jako czasem końca okresu, a zmienioną wersję wiersza utrzymuje w tabeli bieżącej, ustawiając czas transakcji jako czas startu. Dla przykładu wykonajmy poniższą aktualizację, aby podnieść pensję (*salary*) wszystkich pracowników działu *IT* o 5 procent (mój czas wykonania to 2016-02-16 17:20:02):

```
UPDATE dbo.Employees
SET salary *= 1.05
WHERE department = 'IT';
```

Poniżej pokazana jest zawartość tabeli bieżącej po aktualizacji:

empid	empname	department	salary	sysstart	sysend
1	Sara	IT	52500.00	2016-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	IT	47250.00	2016-02-16 17:20:02	9999-12-31 23:59:59

Zwróćmy uwagę na wartości w kolumnach *salary* i *sysstart* dla pracowników *IT*.

A oto zawartość tabeli historii:

empid	empname	department	salary	sysstart	sysend
6	Paul	Sales	40000.00	2016-02-16 17:08:41	2016-02-16 17:15:26
1	Sara	IT	50000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
5	Sven	IT	45000.00	2016-02-16 17:08:41	2016-02-16 17:20:02

Czas modyfikacji, który SQL Server rejestruje w kolumnach okresu, odzwierciedlają czas rozpoczęcia transakcji. Jeśli mamy długo wykonywaną transakcję, która zaczęła się w chwili *T1* i zakończyła w *T2*, SQL Server zarejestruje *T1* jako czas modyfikacji dla wszystkich wyrażeń zawartych w tej transakcji. Dla przykładu poniższy kod jawnie otwiera transakcję i wykonuje zmianę działu pracownika 5 na *Sales* (mój czas wykonania to 2016-02-16 17:28:10):

```
BEGIN TRAN;

UPDATE dbo.Employees
SET department = 'Sales'
WHERE empid = 5;
```

Odczekajmy kilka chwil, po czym wykonajmy poniższy kod, aby zmienić dział pracownika 3 na *IT* (mój czas wykonania to 2016-02-16 17:29:22):

```
UPDATE dbo.Employees
SET department = 'IT'
WHERE empid = 3;

COMMIT TRAN;
```

Poniżej pokazana jest zawartość tabeli bieżącej po wykonaniu tej transakcji:

empid	empname	department	salary	sysstart	sysend
1	Sara	IT	52500.00	2016-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	IT	55000.00	2016-02-16 17:28:10	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	Sales	47250.00	2016-02-16 17:28:10	9999-12-31 23:59:59

A oto zawartość tabeli historii:

empid	empname	department	salary	sysstart	sysend
6	Paul	Sales	40000.00	2016-02-16 17:08:41	2016-02-16 17:15:26
1	Sara	IT	50000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
5	Sven	IT	45000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
3	Judy	Sales	55000.00	2016-02-16 17:08:41	2016-02-16 17:28:10
5	Sven	IT	47250.00	2016-02-16 17:20:02	2016-02-16 17:28:10

Można zauważyć, że we wszystkich zmodyfikowanych wierszach czas zmiany (*sysstart* dla wierszy w tabeli bieżącej i *sysend* dla wierszy historycznych) odpowiada czasowi rozpoczęcia transakcji.

Odpytywanie danych

Wydobywanie danych z tabel temporalnych jest proste i eleganckie. Jeśli chcemy odczytać bieżący stan danych, wykonujemy zwykle zapytanie do tabeli bieżącej, tak samo jak do każdej innej tabeli. Jeśli potrzebujemy stanu z jakiejś chwili w przeszłości, nadal kierujemy zapytanie do tabeli bieżącej, ale dodajemy klauzulę *FOR SYSTEM_TIME* oraz podklauzulę wskazującą pewien punkt w czasie lub okres, który nas interesuje.

Zanim zaczniemy badać szczególne cechy odpytywania tabel temporalnych, należy wykonać poniższy kod, aby ponownie utworzyć tabele *Employees* i *EmployeesHistory* i wypełnić je tymi samymi danymi, jak w moim środowisku, włącznie z wartościami zawartymi w kolumnach okresu:

```
USE TSQLV4;

-- Usunięcie tabel, jeśli istnieją
IF OBJECT_ID(N'dbo.Employees', N'U') IS NOT NULL
BEGIN
    IF OBJECTPROPERTY(OBJECT_ID(N'dbo.Employees', N'U'), N'TableTemporalType') =
    2
        ALTER TABLE dbo.Employees SET ( SYSTEM_VERSIONING = OFF );
    DROP TABLE IF EXISTS dbo.EmployeesHistory, dbo.Employees;
END;
GO

-- Utworzenie i wypełnienie tabeli Employees
CREATE TABLE dbo.Employees
(
    empid          INT          NOT NULL
        CONSTRAINT PK_Employees PRIMARY KEY NONCLUSTERED,
    empname        VARCHAR(25)  NOT NULL,
    department     VARCHAR(50)  NOT NULL,
    salary         NUMERIC(10, 2) NOT NULL,
    sysstart       DATETIME2(0) NOT NULL,
```

```

    sysend    DATETIME2(0)    NOT NULL,
    INDEX ix_Employees CLUSTERED(empid, sysstart, sysend)
);

INSERT INTO dbo.Employees(empid, empname, department, salary, sysstart, sysend)
VALUES
    (1 , 'Sara', 'IT'          , 52500.00, '2016-02-16 17:20:02', '9999-12-31
23:59:59'),
    (2 , 'Don' , 'HR'          , 45000.00, '2016-02-16 17:08:41', '9999-12-31
23:59:59'),
    (3 , 'Judy', 'IT'          , 55000.00, '2016-02-16 17:28:10', '9999-12-31
23:59:59'),
    (4 , 'Yael', 'Marketing', 55000.00, '2016-02-16 17:08:41', '9999-12-31
23:59:59'),
    (5 , 'Sven', 'Sales'       , 47250.00, '2016-02-16 17:28:10', '9999-12-31
23:59:59');

-- Utworzenie i wypełnienie tabeli EmployeesHistory

CREATE TABLE dbo.EmployeesHistory
(
    empid      INT          NOT NULL,
    empname    VARCHAR(25)  NOT NULL,
    department VARCHAR(50)  NOT NULL,
    salary     NUMERIC(10, 2) NOT NULL,
    sysstart   DATETIME2(0) NOT NULL,
    sysend     DATETIME2(0) NOT NULL,
    INDEX ix_EmployeesHistory CLUSTERED(sysend, sysstart)
    WITH (DATA_COMPRESSION = PAGE)
);

INSERT INTO dbo.EmployeesHistory(empid, empname, department, salary, sysstart,
sysend) VALUES
    (6 , 'Paul', 'Sales' , 40000.00, '2016-02-16 17:08:41', '2016-02-16
17:15:26'),
    (1 , 'Sara', 'IT'     , 50000.00, '2016-02-16 17:08:41', '2016-02-16
17:20:02'),
    (5 , 'Sven', 'IT'     , 45000.00, '2016-02-16 17:08:41', '2016-02-16
17:20:02'),
    (3 , 'Judy', 'Sales' , 55000.00, '2016-02-16 17:08:41', '2016-02-16
17:28:10'),
    (5 , 'Sven', 'IT'     , 47250.00, '2016-02-16 17:20:02', '2016-02-16
17:28:10');

-- Włączenie wersjonowania

ALTER TABLE dbo.Employees ADD PERIOD FOR SYSTEM_TIME (sysstart, sysend);

ALTER TABLE dbo.Employees ALTER COLUMN sysstart ADD HIDDEN;
ALTER TABLE dbo.Employees ALTER COLUMN sysend ADD HIDDEN;

ALTER TABLE dbo.Employees
    SET ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.EmployeesHistory ) );

```

W ten sposób wyniki zapytań w środowisku Czytelnika będą takie same jak w książce. Trzeba tylko pamiętać, że jeśli zapytanie nie zawiera klauzuli *ORDER BY*, żadna

kolejność prezentowania wyników nie jest gwarantowana. Możliwe jest zatem, że wiersze w wynikach Czytelnika będą podawane w innej kolejności, niż pokazana w książce.

Jak wspomniałem, aby odczytać bieżący stan wierszy, wykonujemy zwykle zapytanie do tabeli bieżącej:

```
SELECT *  
FROM dbo.Employees;
```

To zapytanie generuje poniższy wynik:

empid	empname	department	salary
1	Sara	IT	52500.00
2	Don	HR	45000.00
3	Judy	IT	55000.00
4	Yael	Marketing	55000.00
5	Sven	Sales	47250.00

Przypomnijmy, że ponieważ kolumny okresu zostały zdefiniowane jako ukryte, zapytanie *SELECT ** ich nie zwraca. W tym przypadku użyłem *SELECT ** w celu zilustrowania tego faktu, ale przypominam, że najlepsze praktyki zalecają podawanie jawnej listy kolumn w kodzie produkcyjnym. To samo dotyczy wyrażeń *INSERT*. Jeśli będziemy stosować się do tych zaleceń, nie będzie miało znaczenia czy kolumny okresu są ukryte, czy nie.

Spróbujmy teraz sprawdzić stan danych w przeszłości. Aby sprawdzić stan, jaki miały dane w tabeli w określonym punkcie czasowym, odpytujemy bieżącą tabelę, uzupełniając zapytanie klauzulą *FOR SYSTEM_TIME* z pomocniczą klauzulą określającą szczegóły. SQL Server pobierze dane z tabeli bieżącej i historii, zgodnie z potrzebami. Co wygodne, można użyć klauzuli *FOR SYSTEM_TIME* również przy odpytywaniu widoków, a klauzula jest propagowana do obiektów leżących w tle.

Składnia klauzuli *FOR SYSTEM_TIME* wygląda następująco:

```
SELECT ... FROM <tabela_lub_widok> FOR SYSTEM_TIME <podklauzula> AS <alias>;
```

Spośród pięciu podklauzul obsługiwanych przez *SYSTEM_TIME* najczęściej używana jest klauzula pomocnicza *AS OF*. Służy ona do wskazania danych, które były właściwe w określonym punkcie w czasie. Składnia tej podklauzuli to *FOR SYSTEM_TIME AS OF <wartość typu datetime2>*. Wartość wejściowa może być stałą, zmienną lub parametrem. Załóżmy, że wejściem jest zmienna o nazwie *@datetime*. Otrzymamy wiersze, dla których zmienna *@datetime* wypada po *sysstart* (jest większa lub równa), ale przed *sysend*. Innymi słowy, zwrócone zostaną dane, których okres ważności zaczyna się wcześniej (lub w tej samej chwili) co wartość *@datetime* i kończy później. Kwalifikujące się wiersze spełniają więc następujący predykat:

```
sysstart <= @datetime AND sysend > @datetime
```

Poniższy przykład zwraca wiersze pracowników, które obowiązywały w punkcie w czasie 2016-02-16 17:00:00:

```
SELECT *
FROM dbo.Employees FOR SYSTEM_TIME AS OF '2016-02-16 17:00:00';
```

Otrzymamy pusty zbiór wyników, gdyż pierwsze wstawienie danych do tabeli nastąpiło w chwili 2016-02-16 17:08:41:

```
empid empname department salary
-----
```

Ponownie odpytajmy tabelę, ale tym razem dla punktu w czasie 2016-02-16 17:10:00:

```
SELECT *
FROM dbo.Employees FOR SYSTEM_TIME AS OF '2016-02-16 17:10:00';
```

Uzyskamy następujące wyniki:

```
empid empname department salary
-----
```

2	Don	HR	45000.00
4	Yael	Marketing	55000.00
6	Paul	Sales	40000.00
1	Sara	IT	50000.00
5	Sven	IT	45000.00
3	Judy	Sales	55000.00

Możliwe jest również odpytanie wielu wystąpień tej samej tabeli z porównaniem różnych stanów danych w różnych punktach w czasie. Dla przykładu poniższe zapytanie zwraca procentowy wzrost wynagrodzeń dla pracowników, których płace wzrosły pomiędzy podanymi punktami w czasie:

```
SELECT T2.empid, T2.empname,
       CAST( (T2.salary / T1.salary - 1.0) * 100.0 AS NUMERIC(10, 2) ) AS pct
FROM dbo.Employees FOR SYSTEM_TIME AS OF '2016-02-16 17:10:00' AS T1
     INNER JOIN dbo.Employees FOR SYSTEM_TIME AS OF '2016-02-16 17:25:00' AS T2
       ON T1.empid = T2.empid
      AND T2.salary > T1.salary;
```

Kod ten generuje następujący wynik:

```
empid empname pct
-----
```

1	Sara	5.00
5	Sven	5.00

Podklauzula *FROM @start TO @end* zwraca wiersze, które spełniają predykat *sysstart < @end AND sysend > @start*. Innymi słowy, zwrócone zostaną wiersze, dla których okres ważności i przedział czasowy wskazany na wejściu mają niepustą część wspólną. Poniższe zapytanie demonstuje użycie tej podklauzuli:

```
SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.Employees
FOR SYSTEM_TIME FROM '2016-02-16 17:15:26' TO '2016-02-16 17:20:02';
```

Zapytanie to generuje następujące wyniki:

empid	empname	department	salary	sysstart	sysend
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
1	Sara	IT	50000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
5	Sven	IT	45000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
3	Judy	Sales	55000.00	2016-02-16 17:08:41	2016-02-16 17:28:10

Zauważmy, że wiersze, w których wartość *sysstart* to 2016-02-16 17:20:02, nie zostały dołączone do wyników. Jeśli chcemy, aby wartość wejściowa *@end* była uwzględniana, należy użyć klauzuli pomocniczej *BETWEEN* zamiast *FROM TO*. Składnia klauzuli *BETWEEN* to *BETWEEN @start AND @end*; zwraca ona wiersze spełniające predykat *sysstart <= @end AND sysend > @start*. Inaczej mówiąc, zwrócone zostaną wiersze, których okres ważności zaczyna się w chwili wskazywanej przez koniec przedziału wejściowego lub wcześniej, a kończy później, niż początek tego przedziału. Poniższe zapytanie demonstruje użycie tej podklauzuli dla tych samych wartości wejściowych, co w poprzednim przykładzie:

```
SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.Employees
FOR SYSTEM_TIME BETWEEN '2016-02-16 17:15:26' AND '2016-02-16 17:20:02';
```

Uzyskamy następujące wyniki, tym razem zawierające wiersze, w których *sysstart* ma wartość 2016-02-16 17:20:02:

empid	empname	department	salary	sysstart	sysend
1	Sara	IT	52500.00	2016-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
1	Sara	IT	50000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
5	Sven	IT	45000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
3	Judy	Sales	55000.00	2016-02-16 17:08:41	2016-02-16 17:28:10
5	Sven	IT	47250.00	2016-02-16 17:20:02	2016-02-16 17:28:10

Klauzula pomocnicza *FOR SYSTEM_TIME CONTAINED IN(@start, @end)* zwraca wiersze spełniające predykat *sysstart >= @start AND sysend <= @end*. Inaczej mówiąc, zwrócone zostaną wiersze, których okres ważności zawiera się w przedziale podanym jako dane wejściowe.

Poniższy przykład demonstruje użycie tej klauzuli:

```
SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.Employees
FOR SYSTEM_TIME CONTAINED IN('2016-02-16 17:00:00', '2016-02-16 18:00:00');
```

Zapytanie to generuje następujące wyniki:

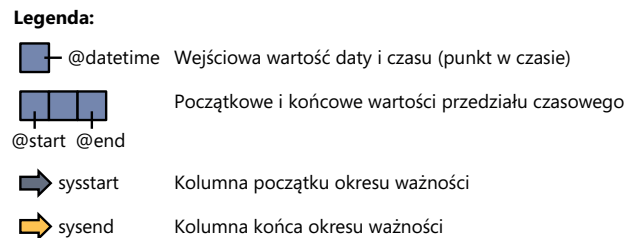
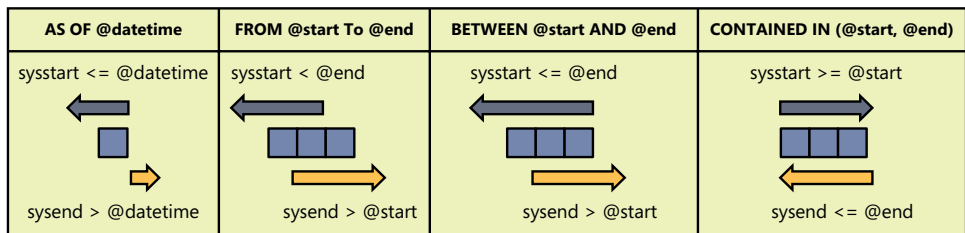
empid	empname	department	salary	sysstart	sysend
6	Paul	Sales	40000.00	2016-02-16 17:08:41	2016-02-16 17:15:26
1	Sara	IT	50000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
5	Sven	IT	45000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
3	Judy	Sales	55000.00	2016-02-16 17:08:41	2016-02-16 17:28:10
5	Sven	IT	47250.00	2016-02-16 17:20:02	2016-02-16 17:28:10

Tabela 9-1 zawiera podsumowanie przedstawionych podklauzul i odpowiadających im predykatów kwalifikujących wiersze wyników.

TABELA 9-1 Klauzule pomocnicze *FOR SYSTEM_TIME*

Podklauzula	Warunek spełniany przez kwalifikujące się wiersze
AS OF @datetime	sysstart <= @datetime AND sysend > @datetime
FROM @start TO @end	sysstart < @end AND sysend > @start
BETWEEN @start AND @end	sysstart <= @end AND sysend > @start
CONTAINED IN(@start, @end)	sysstart >= @start AND sysend <= @end

Rysunek 9-2 pokazuje to samo zestawienie klauzul pomocniczych z graficznym przedstawieniem predykatów kwalifikujących wiersze.



RYСУNEK 9-2 Działanie klauzul pomocniczych *FOR SYSTEM_TIME*

Język T-SQL obsługuje również podklauzulę *ALL*, która po prostu zwraca wszystkie wiersze z obydwu tabel. Poniższe zapytanie demonstruje użycie tej klauzuli:

```
SELECT empid, empname, department, salary, sysstart, sysend
FROM dbo.Employees FOR SYSTEM_TIME ALL;
```


Zapytanie to generuje następujące wyniki:

empid	empname	department	salary	sysstart	sysend
1	Sara	IT	52500.00	2016-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	IT	55000.00	2016-02-16 17:28:10	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2016-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	Sales	47250.00	2016-02-16 17:28:10	9999-12-31 23:59:59
6	Paul	Sales	40000.00	2016-02-16 17:08:41	2016-02-16 17:15:26
1	Sara	IT	50000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
5	Sven	IT	45000.00	2016-02-16 17:08:41	2016-02-16 17:20:02
3	Judy	Sales	55000.00	2016-02-16 17:08:41	2016-02-16 17:28:10
5	Sven	IT	47250.00	2016-02-16 17:20:02	2016-02-16 17:28:10

Przypomnijmy, że kolumny okresu wyznaczają przedział ważności dla wiersza jako wartości typu *datetime2* dla strefy czasowej *UTC*. Jeśli potrzebujemy zwrócić te wartości dla innej, określonej strefy czasowej jako typ *datetimeoffset*, możemy wykorzystać funkcję *AT TIME ZONE*. Warto zauważyć, że funkcji trzeba będzie użyć dwukrotnie: pierwszy raz, aby przekształcić dane wejściowe z typu *datetime2* na typ *datetimeoffset* ze wskazaniem strefy *UTC*, a drugi raz w celu przekształcenia tej wartości na docelową strefę czasową – na przykład *sysstart AT TIME ZONE 'UTC' AT TIME ZONE 'Pacific Standard Time'*. Gdybyśmy użyli tylko pojedynczej konwersji ze wskazaniem docelowej strefy, SQL Server założy, że wartość źródłowa jest już zlokalizowana w docelowej strefie i nie wykona odpowiedniego przestawienia (wykona jedynie zmianę typu danych).

Inną rzeczą, którą trzeba rozważyć, jest kolumna *sysend*. Jeśli przechowywana w niej wartość jest maksymalną dopuszczalną dla typu danych (dla bieżącego okresu ważności), możemy ją traktować tak, jakby używała czasu *UTC*. W innych sytuacjach możemy chcieć ją przekształcić do docelowej strefy czasowej, podobnie jak kolumnę *sysstart*. Do realizacji takiej logiki można wykorzystać wyrażenie *CASE*.

Dla przykładu poniższe zapytanie zwraca wszystkie wiersze z obu tabel i prezentuje kolumny okresu z wartościami przeliczonymi na strefę czasową *Pacific Standard Time*:

```
SELECT empid, empname, department, salary,
       sysstart AT TIME ZONE 'UTC' AT TIME ZONE 'Pacific Standard Time' AS sysstart,
       CASE
         WHEN sysend = '9999-12-31 23:59:59'
         THEN sysend AT TIME ZONE 'UTC'
         ELSE sysend AT TIME ZONE 'UTC' AT TIME ZONE 'Pacific Standard Time'
       END AS sysend
FROM dbo.Employees FOR SYSTEM_TIME ALL;
```

Zapytanie to generuje poniższe wyniki:

empid	empname	department	salary	sysstart	sysend
1	Sara	IT	52500.00	2016-02-16 09:20:02 -08:00	9999-12-31 23:59:59 +00:00
2	Don	HR	45000.00	2016-02-16 09:08:41 -08:00	9999-12-31 23:59:59 +00:00
3	Judy	IT	55000.00	2016-02-16 09:28:10 -08:00	9999-12-31 23:59:59 +00:00

4	Yael	Marketing	55000.00	2016-02-16 09:08:41	-08:00	9999-12-31 23:59:59	+00:00
5	Sven	Sales	47250.00	2016-02-16 09:28:10	-08:00	9999-12-31 23:59:59	+00:00
6	Paul	Sales	40000.00	2016-02-16 09:08:41	-08:00	2016-02-16 09:15:26	-08:00
1	Sara	IT	50000.00	2016-02-16 09:08:41	-08:00	2016-02-16 09:20:02	-08:00
5	Sven	IT	45000.00	2016-02-16 09:08:41	-08:00	2016-02-16 09:20:02	-08:00
3	Judy	Sales	55000.00	2016-02-16 09:08:41	-08:00	2016-02-16 09:28:10	-08:00
5	Sven	IT	47250.00	2016-02-16 09:20:02	-08:00	2016-02-16 09:28:10	-08:00

Po zakończeniu eksperymentów z danymi należy wykonać poniższy kod, aby wyczyścić bazę danych:

```
IF OBJECT_ID(N'dbo.Employees', N'U') IS NOT NULL
BEGIN
    IF OBJECTPROPERTY(OBJECT_ID(N'dbo.Employees', N'U'), N'TableTemporalType') =
    2
        ALTER TABLE dbo.Employees SET ( SYSTEM_VERSIONING = OFF );
    DROP TABLE IF EXISTS dbo.EmployeesHistory, dbo.Employees;
END;
```

Podsumowanie

Począwszy od wersji SQL Server 2016 dostępna jest funkcjonalność wersjonowanych systemowo tabel temporalnych. W przeszłości konieczne było implementowanie własnych, niestandardowych rozwiązań realizujących wersjonowanie, opartych na wyzwalaczach i podobnych mechanizmach. Dzięki wbudowanej funkcjonalności rozwiązania takie stają się znacznie prostsze, a zarazem wydajniejsze. W tym rozdziale pokazałem, jak tworzyć, modyfikować i odpytywać tabele temporalne. Warto pamiętać, że w przypadku wersjonowanych systemowo tabel okresy ważności wierszy są ustalane na podstawie czasu transakcji, a mówiąc ściślej – początku transakcji, w której została wykonana modyfikacja danych. Mam nadzieję, że w przyszłości zobaczymy w SQL Server również obsługę dla tabel temporalnych opartych na czasie definiowanym przez aplikację, włącznie z możliwością ustawienia okresu ważności w przyszłości, a także tabel podwójnego działania, łączących obydwa typy.

Ćwiczenia

Podrozdział ten zawiera ćwiczenia pomagające w lepszym opanowaniu tematyki omówionej w tym rozdziale.

Ćwiczenie 1

W tym ćwiczeniu utworzymy wersjonowaną systemowo tabelę temporalną i odszukamy ją w narzędziu SSMS.

Ćwiczenie 1-1

Utworzyć wersjonowaną systemowo tabelę o nazwie *Departments* z powiązaną tabelą historii *DepartmentsHistory* w bazie danych *TSQLV4*. Tabela powinna zawierać następujące kolumny: *deptid* INT, *deptname* VARCHAR(25) oraz *mgrid* INT, wszystkie bez możliwości używania *NULL*. Dodatkowo należy dołączyć kolumny o nazwach *validfrom* i *validto*, definiujące okres ważności dla wiersza danych. Kolumny te należy zdefiniować z poziomem precyzji zero (1 sekunda), jako ukryte.

Ćwiczenie 1-2

Przejrzeć drzewo obiektów w panelu Object Explorer w SSMS i zlokalizować tabelę *Departments* oraz powiązaną z nią tabelę historii.

Ćwiczenie 2

W tym ćwiczeniu zmodyfikujemy dane w tabeli *Departments*. Należy zanotować chwilę w czasie, gdy będą wysyłane poszczególne wyrażenia, oznaczając je jako P1, P2 i tak dalej. Można to zrealizować, wywołując funkcję *SYSUTCDATETIME* w tym samym wsadzie, w którym jest wysyłana instrukcja modyfikacji danych.

Ćwiczenie 2-1

Wstawić cztery wiersze do tabeli *Departments*, zawierające następujące dane, i zanotować czas wykonania tego wstawienia (oznaczony jako P1):

- *deptid*: 1, *deptname*: HR, *mgrid*: 7
- *deptid*: 2, *deptname*: IT, *mgrid*: 5
- *deptid*: 3, *deptname*: Sales, *mgrid*: 11
- *deptid*: 4, *deptname*: Marketing, *mgrid*: 13

Ćwiczenie 2-2

W tej samej transakcji zmienić nazwę departamentu 3 na *Sales and Marketing* oraz usunąć departament 4. Zanotować czas rozpoczęcia transakcji i oznaczyć go jako P2.

Ćwiczenie 2-3

Zaktualizować identyfikator menedżera departamentu 3 na 13. Zanotować czas wykonania tej zmiany jako P3.

Ćwiczenie 3

W tym ćwiczeniu będziemy odczytywać dane z tabeli *Departments*.

Ćwiczenie 3-1

Odpytać bieżący stan tabeli *Departments*:

- Oczekiwane dane wyjściowe:

deptid	deptname	mgrid
1	HR	7
2	IT	5
3	Sales and Marketing	13

Ćwiczenie 3-2

Napisać zapytanie do tabeli *Departments* dla punktu w czasie wypadającego po P2, ale przed P3:

- Oczekiwane dane wyjściowe:

deptid	deptname	mgrid
1	HR	7
2	IT	5
3	Sales and Marketing	11

Ćwiczenie 3-3

Odczytać stan tabeli *Departments* dla przedziału czasowego pomiędzy P2 a P3. Należy jawnie podać nazwy kolumn w liście *SELECT*, uwzględniając kolumny *validfrom* i *validto*:

- Oczekiwane dane wyjściowe (z dokładnością do wartości w kolumnach *validfrom* i *validto*):

deptid	deptname	mgrid	validfrom	validto
1	HR	7	2016-02-18 10:26:07	9999-12-31 23:59:59
2	IT	5	2016-02-18 10:26:07	9999-12-31 23:59:59
3	Sales and Marketing	13	2016-02-18 10:30:40	9999-12-31 23:59:59
3	Sales and Marketing	11	2016-02-18 10:28:27	2016-02-18 10:30:40

Ćwiczenie 4

Usunąć tabelę *Departments* i powiązaną z nią tabelę historii.

Rozwiązania

W tym podrozdziale zawarte są rozwiązania ćwiczeń wraz z wyjaśnieniami.

Ćwiczenie 1

To ćwiczenie podzielone jest na dwie części.

Ćwiczenie 1-1

Poniższy kod tworzy tabelę *Departments* jako wersjonowaną systemowo tabelę temporalną, łącznie z powiązaną tabelą historii o nazwie *DepartmentsHistory*:

```
USE TSQLV4;
```

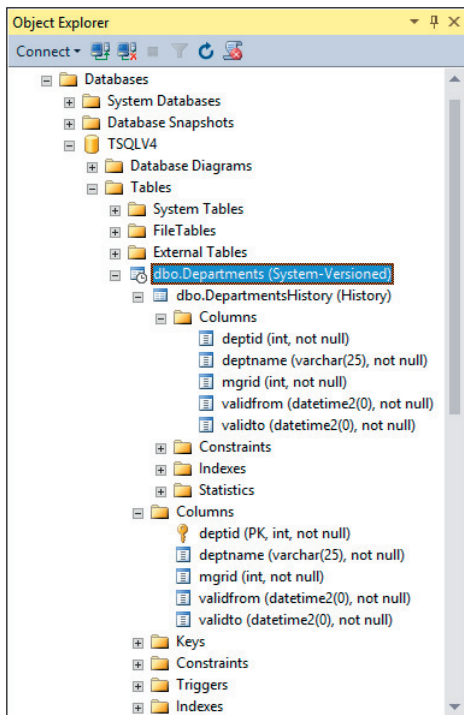
```
CREATE TABLE dbo.Departments
(
    deptid INT NOT NULL
        CONSTRAINT PK_Departments PRIMARY KEY,
    deptname VARCHAR(25) NOT NULL,
    mgrid INT NOT NULL,
    validfrom DATETIME2(0)
        GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
    validto DATETIME2(0)
        GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,
    PERIOD FOR SYSTEM_TIME (validfrom, validto)
)
WITH ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.DepartmentsHistory ) );
```

Poniższa lista pokazuje wymagania niezbędne do utworzenia tabeli temporalnej i ich zastosowanie w tabeli *Departments*:

- Klucz główny zdefiniowany dla kolumny *deptid* column.
- Opcja tabeli *SYSTEM_VERSIONING* ustawiona na *ON* (włączona).
- Dwie kolumny typu *DATETIME2* nie dopuszczające znaczników *NULL*, o dowolnej precyzji (w naszym przypadku 0), reprezentujące początek i koniec okresu ważności wiersza; w naszej tabeli kolumny te nazywają się *validfrom* i *validto*.
- Kolumna początku (*validfrom*) oznaczona jest opcją *GENERATED ALWAYS AS ROW START*.
- Kolumna końca (*validto*) oznaczona jest opcją *GENERATED ALWAYS AS ROW END*.
- Oznaczenie jako kolumny okresu: *PERIOD FOR SYSTEM_TIME (validfrom, validto)*.
- Połączona tabela historii o nazwie *DepartmentsHistory* (którą utworzył SQL Server przy tworzeniu tabeli bieżącej) do przechowywania dawnych stanów modyfikowanych wierszy.

Ćwiczenie 1-2

W panelu Object Explorer przejdź do węzła Databases, następnie do bazy danych TSQLV4 i następnie do Tables. Wewnątrz węzła Tables znajdziesz tabelę *Departments* oznaczoną jako *System-Versioned*, a poniżej jej (jako węzeł podrzędny) tabelę *DepartmentsHistory* oznaczoną jako *History*, co pokazuje rysunek 9-3.



RYСУNEK 9-3 Tabela temporalna *Departments* i powiązana z nią tabela historii w narzędziu SSMS.

Ćwiczenie 2

To ćwiczenie podzielone jest na trzy części, których rozwiązania zostały pokazane w kolejnych podpunktach.

Ćwiczenie 2-1

Poniższy kod pozwala odczytać bieżący czas jako P1 i dodać cztery wymagane wiersze:

```
SELECT CAST(SYSUTCDATETIME() AS DATETIME2(0)) AS P1;
```

```
INSERT INTO dbo.Departments(deptid, deptname, mgrid)
VALUES(1, 'HR', 7),
```

```
(2, 'IT' , 5 ),
(3, 'Sales' , 11),
(4, 'marketing', 13);
```

Wpo wykonaniu tego kodu uzyskałem poniższy wynik:

P1

```
-----
2016-02-18 10:26:07
```

Należy zanotować wartość P1 – niewątpliwie będzie inna, niż w moim przypadku.

Ćwiczenie 2-2

Poniższy kod identyfikuje bieżący punkt w czasie jako P2 i wykonuje dwie żądane modyfikacje w ramach jednej transakcji:

```
SELECT CAST(SYSUTCDATETIME() AS DATETIME2(0)) AS P2;

BEGIN TRAN;

UPDATE dbo.Departments
    SET deptname = 'Sales and Marketing'
WHERE deptid = 3;

DELETE FROM dbo.Departments
WHERE deptid = 4;

COMMIT TRAN;
```

Wykonanie tego kodu w moim systemie zwróciło następujący wynik:

P2

```
-----
2016-02-18 10:28:27
```

Ponownie należy zanotować wartość P2 uzyskaną w ćwiczeniu.

Ćwiczenie 2-3

Poniższy kod identyfikuje bieżący punkt w czasie i wykonuje żądaną aktualizację:

```
SELECT CAST(SYSUTCDATETIME() AS DATETIME2(0)) AS P3;

UPDATE dbo.Departments
    SET mgrid = 13
WHERE deptid = 3;
```

Należy zanotować otrzymaną wartość P3.

W tym momencie zalecam odpytanie obydwu tabel z jawnym odwołaniem się do kolumn *validfrom* i *validto*. Upewnij się, że rozumiesz, dlaczego wartości w tych kolumnach są właśnie takie, jakie są.

Ćwiczenie 3

To ćwiczenie jest podzielone na trzy części, których rozwiązania zostały pokazane w kolejnych podpunktach.

Ćwiczenie 3-1

Poniższy kod powoduje odpytanie bieżącego stanu tabeli *Departments* bez jawnego odwoływania się do nazw kolumn (wykorzystanie *):

```
SELECT *
FROM dbo.Departments;
```

Otrzymamy poniższy wynik:

deptid	deptname	mgrid
1	HR	7
2	IT	5
3	Sales and Marketing	13

Ćwiczenie 3-2

Należy wykonać poniższe zapytanie, aby odczytać stan tabeli *Departments* w punkcie w czasie znajdującym się pomiędzy P2 a P3 (wartość w przykładowym kodzie należy zastąpić taką, która wypada pomiędzy zanotowanymi wartościami P2 i P3):

```
SELECT *
FROM dbo.Departments
    FOR SYSTEM_TIME AS OF '2016-02-18 10:29:00'; -- replace this with your time
```

Otrzymamy poniższy wynik:

deptid	deptname	mgrid
1	HR	7
2	IT	5
3	Sales and Marketing	11

Można zauważyć, że nazwa departamentu 3 jest już nowa (*Sales and Marketing*), zmieniona w chwili P2. Jednak identyfikator menedżera departamentu 3 nadal jest równy 11, gdyż zmiana na 13 nastąpiła w chwili P3.

Ćwiczenie 3-3

Należy wykonać poniższe zapytanie, aby odczytać stan tabeli *Departments* dla okresu pomiędzy P2 i P3 (trzeba zastąpić wartości w przykładowym kodzie zanotowanymi wartościami P2 i P3):

```
SELECT deptid, deptname, mgrid, validfrom, validto
FROM dbo.Departments
```



```
FOR SYSTEM_TIME BETWEEN '2016-02-18 10:28:27' -- zastąpić własną wartością P2
AND '2016-02-18 10:30:40'; -- zastąpić własną wartością P3
```

Otrzymamy poniższe wyniki (z wartościami *validfrom* i *validto* odpowiadającymi czasem modyfikacji uzyskanym w ćwiczeniu):

deptid	deptname	mgrid	validfrom	validto
1	HR	7	2016-02-18 10:26:07	9999-12-31 23:59:59
2	IT	5	2016-02-18 10:26:07	9999-12-31 23:59:59
3	Sales and Marketing	13	2016-02-18 10:30:40	9999-12-31 23:59:59
3	Sales and Marketing	11	2016-02-18 10:28:27	2016-02-18 10:30:40

Wynik ten pokazuje wiersze, których początek ważności wypada w punkcie P3 lub wcześniej, zaś koniec ważności następuje po P2.

Ćwiczenie 4

Nie można usunąć tabel, które stanowią części powiązania wersjonowania systemowego. Konieczne jest wcześniejsze wyłączenie wersjonowania. Poniższy kod realizuje to dla tabel wykorzystanych w ćwiczeniach:

```
ALTER TABLE dbo.Departments SET ( SYSTEM_VERSIONING = OFF );
DROP TABLE dbo.DepartmentsHistory, dbo.Departments;
```


ROZDZIAŁ 10

Transakcje i współbieżność

W tym rozdziale zajmuję się pojęciem transakcji i ich właściwościami oraz przedstawię, jak Microsoft SQL Server obsługuje sytuację, gdy użytkownicy jednocześnie próbują uzyskać dostęp do tych samych danych. Wyjaśnię stosowanie blokad do izolowania niespójnych danych, jak rozwiązywać problemy związane z blokadami oraz jak można kontrolować spójność danych przy użyciu różnych poziomów izolacji. Na koniec omówię również zakleszczenia i sposoby ich unikania.

Ponieważ jest to książka o podstawach, skupiłem się na aspektach współbieżności typowych dla tradycyjnego przechowywania danych w tabelach umieszczonych na dyskach. SQL Server obsługuje również zoptymalizowany pamięciowo silnik bazodanowy o nazwie *In-Memory OLTP*, który przechowuje dane w tabelach umieszczonych w pamięci. Obsługa współbieżności dla tabel zoptymalizowanych pamięciowo różni się bardzo od sytuacji, gdy tabele znajdują się na dyskach. Ponieważ ta funkcjonalność jest zaawansowanym, skoncentrowanym na wydajności mechanizmem, wykracza poza zakres tematyczny tej książki. Czytelnicy, którzy są już gotowi na poznanie bardziej wyrafinowanych aspektów T-SQL ze szczególnym zwróceniem uwagi na wydajność, powinni zainteresować się książką *T-SQL Querying* (Microsoft Press, 2015)*. Omówienie funkcjonalności *In-Memory OLTP* można również znaleźć w dokumentacji SQL Server Books Online pod następującym adresem: <https://msdn.microsoft.com/en-us/library/dn133186.aspx>.

Transakcje

Transakcja jest niepodzielną jednostką pracy, która może obejmować wiele działań odpytujących i modyfikujących dane, a nawet zmiany definicji danych.

Granice transakcji można definiować wprost lub niejawnie. Jawny początek transakcji definiujemy za pomocą instrukcji *BEGIN TRAN* (lub *BEGIN TRANSACTION*), natomiast koniec transakcji definiujemy jawne za pomocą instrukcji *COMMIT TRAN*, jeśli chcemy ją zatwierdzić, albo przy użyciu instrukcji *ROLLBACK TRAN* (lub *ROLLBACK TRANSACTION*), jeśli chcemy wycofać zmiany. Poniżej zamieszczono przykład oznaczania granic transakcji obejmującej dwie instrukcje *INSERT*.

```
BEGIN TRAN;  
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(4, 101, 'C');
```

* Wydanie polskie: *Zapytania w języku T-SQL*, APN Promise 2015, ISBN 978-83-7541-158-4.

```
INSERT INTO dbo.T2(keycol, col1, col2) VALUES(4, 201, 'X');
COMMIT TRAN;
```

Jeśli jawnie nie oznaczymy granic transakcji, domyślnie SQL Server traktuje każdą instrukcję jako transakcję; inaczej mówiąc, domyślnie system SQL Server automatycznie zatwierdza transakcję na koniec każdej pojedynczej instrukcji. Obsługę niejawnych transakcji przez SQL Server można zmienić przy użyciu opcji sesji o nazwie *IMPLICIT_TRANSACTIONS*. Opcja ta domyślnie jest wyłączona. Po włączeniu tej opcji nie trzeba specyfikować instrukcji *BEGIN TRAN*, by oznaczyć początek transakcji, ale musimy oznaczać koniec transakcji za pomocą instrukcji *COMMIT TRAN* lub *ROLLBACK TRAN*.

Gdy jedna transakcja zostanie zatwierdzona lub wycofana, kolejne wywołane wyrażenie niejawnie otworzy nową transakcję, chyba że otworzymy taką transakcję jawnie.

Transakcje charakteryzują się czterema szczególnymi właściwościami. Są to *atomowość* (*atomicity*), *spójność* (*consistency*), *izolacja* (*isolation*) i *trwałość* (*durability*) – określane często akronimem *ACID* od angielskich nazw.

- **Atomowość** Transakcja jest atomową (niepodzielną) jednostką pracy, co oznacza, że wchodzące w jej skład operacje przeprowadzane zostaną albo w całości, albo wcale. Jeśli przed zakończeniem transakcji nastąpi awaria systemu (zanim instrukcja zatwierdzenia zostanie zapisana w dzienniku transakcji), po ponownym uruchomieniu system SQL Server wycofa zmiany, które zostały wykonane w bazie danych w ramach tej transakcji. Również jeśli w trakcie transakcji wystąpią błędy i są one dostatecznie poważne, na przykład gdy docelowa grupa plików jest przepełniona, SQL Server automatycznie wycofa transakcję. Niektóre błędy, takie jak naruszenie klucza podstawowego lub przekroczenie limitu czasu blokady (temat omawiany w dalszej części w podrozdziale „Rozwiązywanie problemów związanych z blokowaniem”), nie są uważane za wystarczające, by uzasadnić automatyczne wycofanie transakcji. Jeśli chcemy, aby każdy błąd powodował przerwanie wykonywania i wycofanie dowolnych otwartych transakcji, możemy włączyć opcję sesji *XACT_ABORT*, powodującą właśnie takie działanie systemu. Można również napisać własny kod obsługi błędów do przechwytywania takich błędów i stosowania odpowiednich działań (na przykład zarejestrowanie błędu i cofnięcie transakcji). Rozdział 11 „Obiekty programowalne” zawiera ogólny opis technik obsługi błędów.



Wskazówka W dowolnym miejscu kodu możemy programowo sprawdzić, czy znajdujemy się wewnątrz otwartej transakcji, wykonując funkcję *@@TRANCOUNT*. Funkcja ta zwraca 0, jeśli nie jesteśmy wewnątrz otwartej transakcji, w przeciwnym razie zwraca wartość większą od zera.

- **Spójność** Pojęcie spójności odnosi się do stanu danych udostępnianych przez RDBMS w trakcie modyfikowania i odpytywania danych przez bieżące transakcje. Jak można się domyśleć, spójność jest pojęciem subiektywnym, zależnym od wymagań aplikacji. W dalszej części w podrozdziale „Poziomy izolacji” omówię poziomy spójności udostępniane domyślnie przez SQL Server oraz sposoby ich kontrolowania, jeśli działanie domyślne nie jest odpowiednie dla naszej aplikacji. Spójność dotyczy także tego, że baza danych musi stosować się do wszystkich reguł integralności, które zdefiniowaliśmy w niej poprzez ograniczenia (takie jak klucze podstawowe, ograniczenia unikalności czy klucze obce). Transakcja przenosi bazę danych z jednego spójnego stanu w inny, natomiast w czasie jej wykonywania spójność nie musi być zapewniona.
- **Izolacja** Izolacja jest mechanizmem zapewniającym, że transakcje uzyskują dostęp do spójnych danych. Kontrolę tego, co rozumiemy przez „spójne dane”, zapewnia mechanizm *poziomów izolacji*. W przypadku tabel opartych na dyskach (tradycyjnych) SQL Server obsługuje dwa różne modele obsługi izolacji: jeden oparty wyłącznie na blokadach, a drugi na kombinacji blokad i tworzeniu wersji wierszy. Dla uproszczenia ten drugi będę nazywać *wersjonowaniem wierszy*. Model korzystający z blokad jest modelem domyślnym w wersji pudełkowej oprogramowania SQL Server. W tym modelu komponenty odczytujące muszą stosować blokady współużytkowane (*shared*). Jeśli bieżący stan danych nie jest spójny, odczytujący są blokowani, aż dane znowu będą spójne. Model wersjonowania wierszy jest modelem domyślnym dla Azure SQL Database. W modelu tym odczytujący nie stosują blokad współużytkowanych i nie muszą czekać. Jeśli bieżący stan danych nie jest spójny, odczytujący uzyskuje starszy spójny stan danych.
- **Trwałość** Zmiany danych są zawsze zapisywane na dysku w dzienniku transakcji bazy danych, zanim zostaną zapisane na dysku w odpowiednim fragmencie bazy danych. Po zarejestrowaniu instrukcji zatwierdzenia w dzienniku transakcji na dysku transakcja uważana jest za utrwaloną, nawet jeśli zmiany nie zostały dokonane we fragmencie danych na dysku. Po uruchomieniu systemu (zarówno przy zwykłym uruchomieniu, jak i po awarii) SQL Server przeprowadza inspekcję dziennika transakcji każdej bazy danych i uruchamia proces przywracania złożony z dwóch faz – ponawiania (*redo*) i wycofywania (*undo*). W fazie ponawiania odtwarzane są wszystkie zmiany każdej transakcji, dla której zapisana została w dzienniku instrukcja zatwierdzenia, natomiast nie zostały jeszcze zapisane same zmiany danych. W fazie wycofywania odwracane są zmiany dla każdej transakcji, dla której w dzienniku nie zarejestrowano instrukcji zatwierdzenia.

Dla przykładu, poniższy kod definiuje transakcję, która rejestruje informacje o nowym zamówieniu w bazie danych *TSQLV4*.

```
USE TSQLV4;
```

```

-- Początek nowej transakcji
BEGIN TRAN;

-- Deklaracja zmiennej
DECLARE @neworderid AS INT;

-- Wstawienie nowego zamówienia do tabeli Sales.Orders
INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
VALUES
    (85, 5, '20150212', '20150301', '20150216',
     3, 32.38, N'Ship to 85-B', N'6789 rue de l'Abbaye', N'Reims',
     N'10345', N'France');

-- Zapisanie identyfikatora nowego zamówienia w zmiennej
SET @neworderid = SCOPE_IDENTITY();

-- Zwrócenie identyfikatora nowego zamówienia
SELECT @neworderid AS neworderid;

-- Wstawienie pozycji zamówienia dla nowego zamówienia do tabeli Sales.
OrderDetails
INSERT INTO Sales.OrderDetails
    (orderid, productid, unitprice, qty, discount)
VALUES(@neworderid, 11, 14.00, 12, 0.000),
    (@neworderid, 42, 9.80, 10, 0.000),
    (@neworderid, 72, 34.80, 5, 0.000);

-- Zatwierdzenie transakcji
COMMIT TRAN;

```

Kod transakcji wstawia wiersz z informacjami nagłówka zamówienia do tabeli *Sales.Orders* i kilka wierszy z informacjami o pozycjach zamówienia do tabeli *Sales.OrderDetails*. Identyfikator nowego zamówienia generowany jest automatycznie przez system SQL Server, ponieważ kolumna *orderid* ma właściwość *identity*. Zaraz po tym, jak kod wstawi nowy wiersz do tabeli *Sales.Orders*, zapisuje nowo wygenerowany identyfikator zamówienia w zmiennej lokalnej, a następnie używa tej zmiennej podczas wstawiania wierszy do tabeli *Sales.OrderDetails*. Dla celów testowych dodana została instrukcja *SELECT*, która zwraca identyfikator zamówienia nowo wygenerowanego zamówienia. Poniżej pokazano dane wyjściowe instrukcji *SELECT* po uruchomieniu tego kodu.

```

neworderid
-----
11078

```

Zwróćmy uwagę, że w tym przykładzie brak jest obsługi błędów i nie ma żadnej instrukcji *ROLLBACK* na wypadek powstania błędu. Aby obsługiwać błędy, obejmujemy transakcję konstrukcją *TRY/CATCH*. Omówienie obsługi błędów znajduje się w rozdziale 11.

Po wykonaniu ćwiczenia uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DELETE FROM Sales.OrderDetails
WHERE orderid > 11077;

DELETE FROM Sales.Orders
WHERE orderid > 11077;
```

Blokowanie

System SQL Server stosuje blokady do wymuszania izolacji transakcji. W kolejnych podrozdziałach omówię szczegółowo blokowanie i sposoby rozwiązywania problemów związanych z blokowaniem spowodowanym konfliktowymi żądaniami blokad.

Jak wspomniałem, Azure SQL Database domyślnie używa modelu wersjonowania wierszy. Aby móc wykonać testy zawarte w tym rozdziale w Azure SQL Database, trzeba wyłączyć właściwość bazy danych `READ_COMMITTED_SNAPSHOT`, aby przełączyć się do modelu blokad jako domyślnego. W tym celu należy wywołać poniższą instrukcję:

```
ALTER DATABASE TSQLV4 SET READ_COMMITTED_SNAPSHOT OFF;
```

Jeśli już jesteśmy połączeni z bazą danych `TSQLV4`, można alternatywnie użyć słowa kluczowego `CURRENT` zamiast nazwy bazy danych. Ponadto domyślnie połączenia z Azure SQL Database mają krótki czas wygaśnięcia (*time out*). Jeśli uruchamiane demonstracje nie działają zgodnie z oczekiwaniem, może to być spowodowane tym, że połączenie używane w danym teście już wygasło.

Blokady

Blokady to zasoby kontrolne uzyskiwane przez transakcje do ochrony danych przed konfliktowymi lub niespójnymi dostęпами do danych przez inne transakcje. Najpierw omówię istotne tryby blokowania obsługiwane przez system SQL Server, a następnie typy zasobów, które mogą być blokowane.

Tryby i zgodność blokad

Na początku poznawania tematyki transakcji i współbieżności trzeba się zapoznać z dwoma głównymi trybami blokad – *wyłączna* (*exclusive*) i *współdzielona* (*shared*).

Podczas próby zmodyfikowania danych transakcja żąda blokady wyłącznej dla zasobów danych, niezależnie od poziomu izolacji (poziomy izolacji omówię dokładniej w dalszej części rozdziału). Jeśli blokada wyłączna zostaje przydzielona, jest ona utrzymywana do końca transakcji. W przypadku transakcji dla pojedynczej instrukcji oznacza to, że blokada jest utrzymywana, aż instrukcja zostanie zakończona. W przypadku transakcji z wieloma instrukcjami oznacza to, że blokada jest utrzymywana,

aż wszystkie instrukcje zostaną wykonane, a transakcja zakończona poleceniem *COMMIT TRAN* lub *ROLLBACK TRAN*.

Blokady wyłączne są nazywane „wyłącznymi”, ponieważ nie możemy nałożyć blokady wyłącznej na zasób, jeśli inna transakcja nałożyła na ten zasób dowolny tryb blokady, a także nie można nałożyć żadnego trybu blokady na zasób, dla którego inna transakcja utrzymuje blokadę wyłączną. Jest to domyślny sposób działania modyfikacji i sposób ten nie może być zmieniony – zarówno pod względem trybu blokowania wymaganego do modyfikowania zasobów danych (wyłączny), jak i okresu trwania blokady (do zakończenia transakcji). W praktyce oznacza to, że jeśli jedna transakcja modyfikuje wiersze, do momentu zakończenia tej transakcji żadna inna transakcja nie może modyfikować tych samych wierszy. Jednak to, czy inna transakcja może, czy nie może odczytywać tych samych wierszy, zależy od poziomu izolacji.

Podobnie jak w przypadku odczytywania danych, ustawienia domyślne są inne dla pudełkowej wersji SQL Server, niż dla Azure SQL Database. W przypadku instalacji SQL Server w siedzibie domyślny poziom izolacji nazywany jest *READ COMMITTED* (odczyt zatwierdzonych). Na tym poziomie podczas odczytu danych transakcja domyślnie nakłada na zasób danych współdzieloną blokadę i zwalnia ją zaraz po wykonaniu instrukcji odczytu. Ten tryb blokady nazywany jest „współdzielonym”, ponieważ wiele transakcji na ten sam zasób danych może jednocześnie nakładać blokady współdzielone. Chociaż nie można zmieniać trybu blokady i czasu jej trwania wymaganego podczas modyfikowania danych, przy odczytywaniu danych możemy kontrolować sposób obsługi blokowania poprzez zmianę poziomu izolacji. Jak już wspomniałem, temat ten zostanie omówiony w dalszej części rozdziału.

W Azure SQL Database domyślny poziom izolacji nazywany jest *READ COMMITTED SNAPSHOT*. Ten tryb izolacji nie polega wyłącznie na blokadach, lecz opiera się na kombinacji blokad i tworzenia wersji wierszy. Na tym poziomie izolacji komponenty odczytujące nie używają współdzielonych blokad i z tego względu nigdy nie muszą oczekiwać; odczytujący polegają na wersjonowaniu wierszy, by zapewnić spodziewany poziom izolacji. W praktyce oznacza to, że w przypadku trybu izolacji *READ COMMITTED*, jeśli transakcja modyfikuje wiersze, do momentu zakończenia transakcji żadna inna transakcja nie może odczytywać tych samych wierszy. Takie podejście do współbieżności nazywane jest *współbieżnością pesymistyczną*. Dla poziomu izolacji *READ COMMITTED SNAPSHOT*, jeśli transakcja modyfikuje wiersze, inna transakcja próbująca odczytać dane uzyska ostatni zatwierdzony stan wierszy, który był dostępny podczas uruchomienia instrukcji. To podejście do współbieżności nazywane jest *optymistycznym*.

Interakcja blokad pomiędzy transakcjami nazywana jest *zgodnością blokad*. W tabeli 10-1 pokazano zgodność dla blokad wyłącznych i współdzielonych (podczas pracy w trybie izolacji, który generuje te blokady). Kolumny reprezentują przydzielone tryby blokad, a wiersze reprezentują żądane tryby blokad.

TABELA 10-1 Zgodność blokad wyłącznych i współdzielonych

Żądany tryb	Nałożony tryb wyłączny (X)	Nałożony tryb współdzielony (S)
Wyłączny	Nie	Nie
Współdzielony	Nie	Tak

„Nie” na przecięciu oznacza, że blokada jest niezgodna i żądany tryb jest odrzucony; żądający musi czekać. „Tak” na przecięciu oznacza, że blokady są zgodne i żądany tryb jest akceptowany.

Upraszczając: dane, które zostały zmodyfikowane przez jedną transakcję, nie mogą być modyfikowane ani odczytywane (przynajmniej w przypadku ustawień domyślnych w instalacji SQL Server w siedzibie) przez inną transakcję, dopóki pierwsza nie zakończy działania. Podczas odczytywania danych przez jedną transakcję, dane nie mogą być modyfikowane przez inną transakcję (przynajmniej w przypadku ustawień domyślnych w instalacji SQL Server w siedzibie).

Typy blokowanych zasobów

System SQL Server może blokować różne typy zasobów. Typami zasobów, które mogą być blokowane, są wiersze (identyfikatory *RID* w przypadku sterty lub klucze w indeksach), strony danych, obiekty (na przykład tabele), bazy danych i inne zasoby. Wiersze znajdują się wewnątrz stron, a strony to fizyczne bloki danych zawierające tabelę lub dane indeksu. Warto najpierw zapoznać się z tymi typami zasobów, a następnie przyrzeć się bardziej zaawansowanym rodzajom, które mogą być blokowane, takim jak ekstenty, jednostki alokacji, sterty lub B-drzewa.

Aby nałożyć blokadę na pewien typ zasobu, nasza transakcja musi najpierw uzyskać blokady intencyjne tego samego trybu nałożone na wyższych poziomach hierarchii obiektów. Na przykład, aby nałożyć blokadę wyłączną dla wiersza, transakcja musi najpierw uzyskać intencyjną blokadę wyłączną dla strony, w której ten wiersz się znajduje, oraz intencyjną blokadę wyłączną dla obiektu, który jest właścicielem strony. Podobnie, aby uzyskać blokadę współdzieloną na pewnym poziomie szczegółowości, transakcja musi najpierw uzyskać intencyjne blokady współdzielone na wyższych poziomach szczegółowości. Zadaniem blokad intencyjnych jest sprawne wykrywanie niezgodnej blokady na wyższych poziomach szczegółowości i zapobieganie ich nakładaniu. Jeśli na przykład jedna transakcja utrzymuje blokadę wiersza, a inna żąda niezgodnej blokady całej strony lub tabeli (w której znajduje się wiersz), dzięki blokadom intencyjnym nałożonym przez pierwszą transakcję na stronę lub tabelę system SQL Server łatwo wykryje konflikt. Blokady intencyjne nie przeszkadzają żądaniom blokad na niższych poziomach szczegółowości. Na przykład blokada intencyjna nałożona na stronę nie uniemożliwia innym transakcjom uzyskania niezgodnych trybów blokad dla wierszy wewnątrz tej strony. Tabela 10-2 stanowi rozszerzenie poprzedniej

(10-1) tabeli zgodności blokad, poprzez dodanie intencyjnych blokad wyłącznych i współdzielonych.

TABELA 10-2 Zgodność blokad z uwzględnieniem blokad intencyjnych

Żądany tryb	Nałożony tryb wyłączny (X)	Nałożony tryb współdzielony (S)	Nałożony tryb intencyjny wyłączny (IX)	Nałożony intencyjny współdzielony (IS)
Wyłączny	Nie	Nie	Nie	Nie
Współdzielony	Nie	Tak	Nie	Tak
Intencyjny wyłączny	Nie	Nie	Tak	Tak
Intencyjny współdzielony	Nie	Tak	Tak	Tak

SQL Server dynamicznie ustala, które typy zasobów mają być blokowane. Oczywiście w celu zapewnienia idealnej współbieżności najlepiej byłoby blokować tylko to, co jest konieczne – mówiąc wprost, jedynie używane wiersze. Blokady jednak wymagają zasobów pamięci i nakładu pracy związanego z wewnętrzną obsługą. Z tego względu, podczas wyboru typu blokowanych zasobów, system SQL Server uwzględnia zarówno zasoby systemu, jak i zagadnienia dotyczące współbieżności.

SQL Server może najpierw nakładać blokady szczegółowe (takie jak blokada wiersza czy strony), po czym, w pewnych warunkach, może próbować rozszerzyć te szczegółowe blokady na bardziej ogólne (takie jak blokada tabeli). Działanie takie nosi nazwę eskalacji blokad. Na przykład eskalacja blokady jest wywoływana, jeśli pojedyncza instrukcja uzyskuje co najmniej 5000 blokad dotyczących elementów (np. wierszy) tego samego obiektu, a następnie co każde kolejne 1250 nowych blokad, jeśli poprzednie próby rozszerzenia blokady się nie powiodły.

Działanie eskalacji blokad można kontrolować poprzez opcję `LOCK_ESCALATION` na poziomie tabeli (przy użyciu instrukcji `ALTER TABLE`). Przy użyciu tej opcji można wyłączyć mechanizm eskalacji blokad lub określić, czy rozszerzenie wykonywane będzie na poziomie tabeli (ustawienie domyślne), czy na poziomie partycji (tabela może być fizycznie zorganizowana w postaci wielu mniejszych jednostek nazywanych *partycjami*).

Rozwiązywanie problemów związanych z blokadami

Jeśli transakcja nałoży blokadę na zasób danych, a inna transakcja żąda niezgodnej blokady dla tych samych zasobów, żądanie jest blokowane, a żądający wchodzi w stan oczekiwania. Domyślnie zablokowane żądanie oczekuje, aż blokujący zwolni sporną blokadę. W dalszej części tego podrozdziału pokażę, jak definiować limity czasu wygasania blokad na poziomie sesji, jeśli chcemy ograniczyć maksymalny czas oczekiwania blokowanych żądań.

Blokowanie jest normalnym zjawiskiem w systemie bazodanowym, o ile żądania są realizowane w rozsądnym czasie. Jeśli jednak pewne żądania oczekują zbyt długo, trzeba zająć się takimi sytuacjami blokowania i sprawdzić, czy możemy zrobić coś, aby zapobiegać tak długim opóźnieniom. Na przykład długo wykonywane transakcje skutkują długimi okresami blokad. Możemy próbować skracać takie transakcje, przenosząc na zewnątrz transakcji działania, które nie muszą być częścią danej jednostki pracy. Błąd w aplikacji może powodować powstanie transakcji, które w pewnych warunkach pozostają otwarte. Jeśli znajdziemy taki błąd, można go poprawić i zapewnić, że transakcja będzie zamykana w każdej sytuacji.

W kolejnym przykładzie zademonstruję sytuacje blokowania i sposoby ich rozwiązywania. W programie SQL Server Management Studio otwieramy trzy oddzielne okna zapytań (dla potrzeb tego przykładu będę nazywać je Connection 1, Connection 2 i Connection 3). Należy się upewnić, że w każdym oknie połączeni jesteśmy z testową bazą danych *TSQLV4*:

```
USE TSQLV4;
```

W oknie Connection 1 uruchamiamy poniższy kod, by zaktualizować wiersz w tabeli *Production.Products*, dodając wartość 1.00 do bieżącej ceny jednostkowej 19.00 dla produktu 2.

```
BEGIN TRAN;  
  UPDATE Production.Products  
    SET unitprice += 1.00  
  WHERE productid = 2;
```

Aby zaktualizować wiersz, nasza sesja musi uzyskać blokadę wyłączną i jeśli aktualizacja powiodła się, system SQL Server przydzielił blokadę naszej sesji. Pamiętamy, że blokady wyłączne są utrzymywane do zakończenia transakcji, a ponieważ transakcja pozostaje otwarta, blokada jest nadal utrzymywana.

W oknie Connection 2 uruchamiamy poniższy kod, by spróbować wykonać zapytanie do tego samego wiersza:

```
SELECT productid, unitprice  
FROM Production.Products  
WHERE productid = 2;
```

Ta sesja potrzebuje blokady współdzielonej, by odczytać dane, ponieważ jednak na wiersz nałożona jest blokada wyłączna przez inną sesję, a blokada współdzielona nie jest zgodna z blokadą wyłączną, sesja zostaje zablokowana i musi czekać.

Zakładając, że taka sytuacja blokowania ma miejsce w naszym systemie i jest powodem długiego czasu oczekiwania sesji, zapewne będziemy próbować naprawić tę sytuację. W pozostałej części tego podrozdziału pokażę zapytania do dynamicznych widoków i funkcji zarządzania (obiektów zapewniających dynamiczne, aktualizowane w czasie rzeczywistym informacje o różnych aspektach systemu bazodanowego), które będziemy uruchamiać w oknie Connection 3.

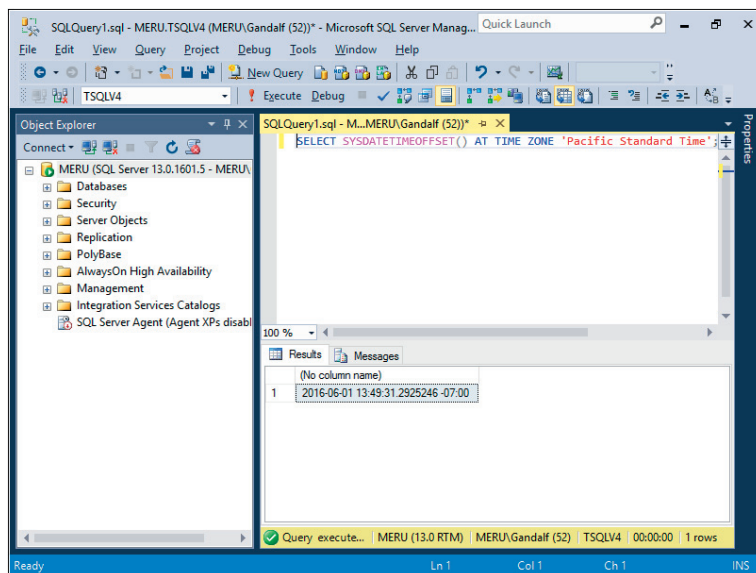
Aby uzyskać informacje o blokadach, a w tym o blokadach aktualnie przydzielonych sesjom i blokadom, na które sesje oczekują, w oknie Connection 3 wykonamy zapytanie do dynamicznego widoku zarządzania (DMV) *sys.dm_tran_locks*.

```
SELECT      -- można użyć *, by poznać inne dostępne atrybuty
  request_session_id      AS spid,
  resource_type            AS restype,
  resource_database_id    AS dbid,
  DB_NAME(resource_database_id) AS dbname,
  resource_description    AS res,
  resource_associated_entity_id AS resid,
  request_mode            AS mode,
  request_status          AS status
FROM sys.dm_tran_locks;
```

Po uruchomieniu tego kodu w mojej instalacji (żadne inne okna zapytań nie były otwarte), uzyskałem następujące wyniki:

spid	restype	dbid	dbname	res	resid	mode	status
53	DATABASE	8	TSQV4		0	S	GRANT
52	DATABASE	8	TSQV4		0	S	GRANT
51	DATABASE	8	TSQV4		0	S	GRANT
54	DATABASE	8	TSQV4		0	S	GRANT
53	PAGE	8	TSQV4	1:127	72057594038845440	IS	GRANT
52	PAGE	8	TSQV4	1:127	72057594038845440	IX	GRANT
53	OBJECT	8	TSQV4		133575514	IS	GRANT
52	OBJECT	8	TSQV4		133575514	IX	GRANT
52	KEY	8	TSQV4	(020068e8b274)	72057594038845440	X	GRANT
53	KEY	8	TSQV4	(020068e8b274)	72057594038845440	S	WAIT

Każda sesja jest identyfikowana przez unikatowy identyfikator *SPID*. Identyfikator *SPID* naszej sesji możemy poznać, odpytując funkcję @@SPID. Jeśli korzystamy z narzędzia SQL Server Management Studio, identyfikator *SPID* sesji umieszczony jest w nawiasach z prawej strony nazwy logowania w pasku stanu na dole ekranu, a także w opisie połączonego okna zapytań. Na rysunku 10-1 przedstawiono ekran programu SQL Server Management Studio, gdzie identyfikator *SPID* 52 wyświetlany jest z prawej strony nazwy logowania MERU\Gandalf.



RYSUNEK 10-1 Identyfikator SSID pokazywany w programie SQL Server Management Studio

W wynikach zapytania do widoku `sys.dm_tran_locks` można zauważyć, że cztery sesje (51, 52, 53 i 54) aktualnie utrzymują blokady. Widoczne są następujące informacje:

- Typ zablokowanego zasobu (na przykład `KEY` dla wiersza indeksu).
- Identyfikator bazy danych, w której jest zablokowany, a który można przetłumaczyć na nazwę bazy danych przy użyciu funkcji `DB_NAME`.
- Zasób i identyfikator zasobu (kolumny `res` i `resid`).
- Tryb blokady.
- Informacja, czy blokada jest przydzielona (`GRANT`), czy też sesja oczekuje na spełnienie żądania blokady (`WAIT`).

Warto zauważyć, że jest to tylko podzbiór atrybutów udostępnianych przez ten widok; warto przeanalizować pozostałe atrybuty, by dowiedzieć się, jakie inne informacje są udostępniane na temat blokad.

W wynikach naszego zapytania możemy zauważyć, że proces 53 oczekuje na nałożenie blokady współdzielonej na wiersz w testowej bazie danych `TSQV4` (nazwa bazy danych jest uzyskiwana za pomocą funkcji `DB_NAME`). Zwróćmy uwagę, że na ten sam wiersz proces 52 utrzymuje blokadę wyłączną. Możemy to stwierdzić zauważając, że oba procesy odwołują się do zasobu o tych samych wartościach `res` i `resid`. Która tabela jest wykorzystywana, możemy stwierdzić, przechodząc w górę w hierarchii blokad dla jednego z procesów, 52 lub 53 i analizując blokady intencyjne dla strony i obiektu (tabeli), w których wiersz się znajduje. Do przetłumaczenia identyfikatora obiektu (w tym przykładzie 133575514) na nazwę możemy użyć funkcji


```
SELECT session_id, text
FROM sys.dm_exec_connections
CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(52, 53);
```

Po uruchomieniu tego zapytania uzyskałem pokazane poniżej wyniki, które prezentują ostatnią porcję kodu wywoływanego przez każde połączenie zaangażowane w łańcuch blokowania.

```
session_id  text
-----
52          BEGIN TRAN;

            UPDATE Production.Products
               SET unitprice += 1.00
            WHERE productid = 2;

53          (@! tinyint)
            SELECT [productid],[unitprice]
            FROM [Production].[Products]
            WHERE [productid]=@1
```

Zablokowany proces – 53 – pokazuje zapytanie, które oczekuje, ponieważ jest to ostatnia rzecz, którą proces uruchomił. W przypadku procesu blokującego w tym przykładzie widzimy instrukcję, która jest przyczyną problemu, ale trzeba pamiętać, że proces blokujący może kontynuować działanie i że ostatnia rzecz widoczna w kodzie niekoniecznie musi być tą instrukcją, która powoduje problemy.

W wydaniu SQL Server 2016 możemy użyć funkcji *sys.dm_exec_input_buffer* zamiast *sys.dm_exec_sql_text* do uzyskania ostatniego kodu przesłanego przez badaną sesję. Funkcja ta akceptuje jako parametry identyfikator sesji oraz identyfikator żądania (dostępny w widoku *sys.dm_exec_requests*, który omówię nieco dalej), albo *NULL* zamiast identyfikatora żądania, jeśli nie jest on istotny. Poniższy kod zastępuje poprzedni przykład przy wykorzystaniu nowej funkcji:

```
SELECT session_id, event_info
FROM sys.dm_exec_connections
CROSS APPLY sys.dm_exec_input_buffer(session_id, NULL) AS IB
WHERE session_id IN(52, 53);
```

Wiele przydatnych informacji na temat sesji zaangażowanych w tę sytuację blokowania możemy znaleźć w dynamicznym widoku *sys.dm_exec_sessions*. Poniższe zapytanie na temat tych sesji zwraca tylko mały podzbiór atrybutów.

```
SELECT -- użyj *, by poznać inne atrybuty
       session_id AS spid,
       login_time,
       host_name,
       program_name,
       login_name,
       nt_user_name,
       last_request_start_time,
```

```

    last_request_end_time
FROM sys.dm_exec_sessions
WHERE session_id IN(52, 53);

```

W tym przykładzie zapytanie to zwraca następujące wyniki, podzielone na kilka części dla większej czytelności.

sid	login_time	host_name
52	2016-06-25 15:20:03.407	K2
53	2016-06-25 15:20:07.303	K2

sid	program_name	login_name
52	Microsoft SQL Server Management Studio - Query	K2\Gandalf
53	Microsoft SQL Server Management Studio - Query	K2\Gandalf

sid	nt_user_name	last_request_start_time	last_request_end_time
52	Gandalf	2016-06-25 15:20:15.703	2016-06-25 15:20:15.750
53	Gandalf	2016-06-25 15:20:20.693	2016-06-25 15:20:07.320

Te dane wyjściowe zawierają informacje o czasie zalogowania sesji, nazwie hosta, nazwie programu, nazwie logowania, nazwie użytkownika systemu Windows, czasie, w którym uruchomione zostało ostatnie żądanie i czasie, w którym ostatnie żądanie zostało zakończone. Informacje tego rodzaju dają dobry obraz działania sesji.

Innym widokiem, który prawdopodobnie okaże się bardzo przydatny podczas rozwiązywania problemów związanych z sytuacjami blokowania, jest widok *sys.dm_exec_requests*. Widok ten zawiera wiersz dla każdego aktywnego żądania, w tym dla żądań zablokowanych. Tak naprawdę łatwo możemy wyizolować zablokowane żądania, ponieważ atrybut *blocking_session_id* jest większy niż zero. Na przykład poniższe zapytanie filtruje tylko zablokowane żądania.

```

SELECT -- użyj *, by poznać inne atrybuty
    session_id AS spid,
    blocking_session_id,
    command,
    sql_handle,
    database_id,
    wait_type,
    wait_time,
    wait_resource
FROM sys.dm_exec_requests
WHERE blocking_session_id > 0;

```

Zapytanie to zwraca następujące wyniki, rozdzielone na kilka wierszy.

spid	blocking_session_id	command
53	52	SELECT

spid	sql_handle	database_id
53	0x0200000063FC7D052E09844778CDD615CFE7A2D1FB411802	8

spid	wait_type	wait_time	wait_resource
53	LCK_M_S	1383760	KEY: 8:72057594038845440 (020068e8b274)

Łatwo możemy zidentyfikować sesje uczestniczące w łańcuchu blokowania, sporne zasoby, czas oczekiwania sesji (w milisekundach) i inne informacje.

Alternatywnie możemy odpytać widok `sys.dm_os_waiting_tasks`, który pokazuje tylko zadania aktualnie oczekujące. On również zawiera atrybut o nazwie `blocking_session_id`, a podczas rozwiązywania problemów możemy wyfiltrować tylko te oczekujące zadania, w których atrybut ten jest większy od zera. Część informacji z tego widoku pokrywa się z tymi udostępnianymi przez `sys.dm_exec_requests`, ale są tu obecne atrybuty unikatowe dla tego widoku, takie jak opis zasobu zaangażowanego w konflikt.

Jeśli zdecydujemy się na zakończenie blokowania – na przykład, gdy zdaliśmy sobie sprawę, że w wyniku błędu aplikacji transakcja pozostaje otwarta i żaden mechanizm nie potrafi jej zamknąć – możemy zakończyć blokujący proces przy użyciu polecenia `KILL <spid>` (ale jeszcze nie wykonujemy tego polecenia).

Jak wspomniałem wcześniej, domyślnie sesja nie ma ustawionego limitu czasu trwania blokady. Jeśli chcemy ograniczyć czas oczekiwania sesji na blokady, możemy ustawić opcję sesji nazwaną `LOCK_TIMEOUT`. Wartość okresu specyfikowana jest w milisekundach – na przykład 5000 dla 5 sekund, 0 dla natychmiastowego limitu i -1 dla braku limitów czasu (opcja domyślna). Aby zilustrować działanie tej opcji, najpierw zatrzymamy zapytanie w oknie Connection 2, wybierając polecenie Cancel Executing Query z menu Query (lub naciskając klawisze Alt+Break). Następnie uruchamiamy poniższy kod, by ustawić limit czasu blokady na 5 sekund i ponownie uruchamiamy zapytanie.

```
SET LOCK_TIMEOUT 5000;

SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

Zapytanie jest nadal zablokowane, ponieważ w oknie Connection 1 nie zakończyła się transakcja aktualizacji, jeśli jednak po 5 sekundach żądanie nałożenia blokady nie zostanie zrealizowane, SQL Server zakończy zapytanie generując następujący komunikat o błędzie:

```
Msg 1222, Level 16, State 51, Line 3
Lock request time out period exceeded.
(Przekroczony został limit czasu żądania blokady)
```

Trzeba zauważyć, że przekroczenie limitów czasu blokad nie powoduje wycofania transakcji. Takie działanie musimy obsłużyć samodzielnie, czym zajmiemy się w rozdziale 11.

Teraz przywrócimy domyślną wartość limitu czasu blokady (nieograniczoną) i wykonamy ponownie zapytanie, uruchamiając poniższy kod w oknie Connection 2.

```
SET LOCK_TIMEOUT -1;

SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

Aby zakończyć transakcję aktualizacji w oknie Connection 1, uruchamiamy poniższy kod w oknie Connection 3.

```
KILL 52;
```

Instrukcja ta powoduje nie tylko zamknięcie sesji, ale również wycofanie transakcji w oknie Connection 1, co oznacza, że zmiana ceny produktu 2 z 19.00 na 20.00 zostanie cofnięta i zwolnienie blokady wyłącznej. Przechodzimy do okna Connection 2. Zauważmy, że odebrane dane są wartościami sprzed zmiany ceny.

productid	unitprice
2	19.00



UWAGA Próba zamknięcia okna zapytania, w którym jest otwarta (aktywna) transakcja, spowoduje wyświetlenie komunikatu ostrzegawczego, w którym można wybrać pomiędzy zatwierdzeniem a odwołaniem transakcji.

Poziomy izolacji

Poziom izolacji określa poziom spójności danych, który uzyskamy podczas interakcji z danymi. W domyślnym poziomie izolacji dla produktu w siedzibie proces odczytujący (*reader*) używa blokad współdzielonych, zaś zapisujący (*writer*) blokad wyłącznych. Nie można kontrolować sposobu zachowania zapisujących, jeśli chodzi o stosowanie i czas trwania blokad, ale mamy pewną kontrolę nad zachowaniem odczytujących. Dodatkowo, jako skutek kontrolowania zachowań procesów odczytujących, mamy pośredni wpływ na działanie procesów zapisujących. Uzyskujemy to poprzez ustawienie poziomu izolacji, albo na poziomie sesji (poprzez opcję sesji), albo pojedynczego zapytania (za pomocą wskazówki tabeli – *table hint*).

System SQL Server obsługuje cztery poziomy izolacji oparte wyłącznie na blokadach (pesymistyczna kontrola współbieżności): *READ UNCOMMITTED*, *READ COMMITTED* (domyślny poziom dla instancji SQL Server w siedzibie), *REPEATABLE READ* i *SERIALIZABLE*. SQL Server obsługuje także dwa poziomy izolacji oparte na kombinacji blokad i wersjonowaniu wierszy: *SNAPSHOT* i *READ COMMITTED SNAPSHOT* (domyślny poziom dla Azure SQL Database). Poziomy *SNAPSHOT*

i *READ COMMITTED SNAPSHOT* są odpowiednikami poziomów *READ COMMITTED* i *SERIALIZABLE*.

Niektóre teksty odnoszące się do poziomów *READ COMMITTED* i *READ COMMITTED SNAPSHOT* traktują je jak jeden poziom izolacji różnicowany semantycznie.

Poziom izolacji dla całej sesji możemy określić przy użyciu poniższego polecenia:

```
SET TRANSACTION ISOLATION LEVEL <isolation name>;
```

Do ustawienia poziomu izolacji dla zapytania możemy posłużyć się wskazówką tabeli.

```
SELECT ... FROM <table> WITH (<isolationname>);
```

UWAGA Nie można jawnie ustawić poziomu izolacji *READ COMMITTED SNAPSHOT* jako opcji sesji lub zapytania. Aby móc używać tego poziomu izolacji, konieczne jest ustawienie odpowiedniej flagi bazy danych. Wyjaśnię to szczegółowo w dalszej części rozdziału w podpunkcie „Poziomy izolacji oparte na wersjonowaniu wierszy”.



Warto zauważyć, że przy ustawianiu opcji sesji wpisujemy spację pomiędzy słowami nazwy poziomu izolacji (*isolation name*), jeśli nazwa składa się z więcej niż jednego słowa, na przykład *REPEATABLE READ*. W przypadku wskazówki tabeli nie wpisujemy spacji pomiędzy słowami – na przykład *WITH (REPEATABLE READ)*. Ponadto niektóre nazwy poziomów izolacji używane jako wskazówki tabel mają jednowyrazowe synonimy. Na przykład *NOLOCK* jest ekwiwalentem nazwy *READUNCOMMITTED*, a *HOLDLOCK* to zamiennik nazwy *SERIALIZABLE*.

Domyślny poziom izolacji w przypadku instancji SQL Server dla siedziby to *READ COMMITTED* (oparty na blokadach). Domyślny poziom dla Azure SQL Database to *READ COMMITTED SNAPSHOT* (oparty na blokadach i wersjach wierszy). Zmiana poziomu izolacji ma wpływ zarówno na współbieżne działania użytkowników bazy danych, jak i spójność uzyskiwanych przez nich danych.

W przypadku czterech pierwszych poziomów izolacji – im wyższy poziom izolacji, tym mocniejsze blokady żądane przez odczytujących i dłuższy czas trwania blokad; z tego względu też, im wyższy poziom izolacji, tym wyższy poziom spójności, a niższy współbieżności. Oczywiście prawdziwe jest również twierdzenie odwrotne.

Jeśli chodzi o dwa poziomy izolacji oparte na wersjonowaniu wierszy, SQL Server jest w stanie przechowywać poprzednio zatwierdzone wersje wierszy w bazie *tempdb*. Odczytujący nie muszą żądać blokad współdzielonych; zamiast tego, jeśli bieżąca wersja wierszy nie jest tym, co odczytujący powinni zobaczyć (innymi słowy, trwa ich modyfikacja i stan danych nie jest spójny), SQL Server udostępni im starszą wersję.

Kolejne podrozdziały opisują każdy z sześciu obsługiwanych poziomów izolacji i demonstrowują ich zachowanie.

Poziom izolacji *READ UNCOMMITTED*

READ UNCOMMITTED (odczyty niezatwierdzonych danych) jest najniższym dostępnym poziomem izolacji. Na tym poziomie odczytujący nie żąda blokady współdzielonej. Odczytujący, który nigdy nie żąda blokady współdzielonej, nigdy nie znajduje się w sytuacji konfliktowej z zapisującym, który nałożył blokadę wyłączną. Oznacza to, że odczytujący potrafi odczytać niezatwierdzone zmiany (nazywane również *brudnymi odczytami*). Oznacza to również, że odczytujący nie będzie przeszkadzał zapisującemu, który żąda blokady wyłącznej. Inaczej mówiąc, zapisujący może zmieniać dane w trakcie ich odczytywania przez odczytującego, który działa na poziomie izolacji *READ UNCOMMITTED*.

Aby zilustrować działanie niezatwierdzonych odczytów (brudne odczyty), otwieramy dwa okna zapytania (nazwane Connection 1 i Connection 2) upewniając się, że wszystkie połączenia z bazą danych dotyczą bazy testowej *TSQVLV4*. Dla uniknięcia nieporozumień trzeba też zapewnić, aby były to jedyne połączenia z tą bazą danych.

W oknie Connection 1 uruchamiamy poniższy kod, by otworzyć transakcję, zaktualizować cenę jednostkową produktu 2, dodając 1.00 do ceny bieżącej (19.00), a następnie odpytać wiersz produktu.

```
BEGIN TRAN;
```

```
UPDATE Production.Products
    SET unitprice += 1.00
WHERE productid = 2;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Zwróćmy uwagę, że transakcja pozostaje otwarta, co oznacza, że wiersz produktu jest zablokowany na wyłączność przez zapytanie w oknie Connection 1. Kod w oknie Connection 1 zwraca następujące wyniki, które pokazują nową cenę produktu.

productid	unitprice
2	20.00

W oknie Connection 2 uruchamiamy poniższy kod, by ustawić poziom izolacji *READ UNCOMMITTED* i odpytać wiersz dotyczący produktu 2.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Ponieważ na tym poziomie nie żądamy blokady współdzielonej, nie jesteśmy w konflikcie z inną transakcją. Zapytanie to zwraca stan wiersza po zmianie ceny, pomimo tego, że zmiana nie została zatwierdzona.

productid	unitprice
2	20.00

Pamiętajmy, że w oknie Connection 1 mogą być wprowadzane dalsze zmiany wiersza lub cała transakcja może zostać wycofana w pewnym momencie. Dla przykładu uruchomimy w oknie Connection 1 poniższy kod, by wycofać transakcję.

```
ROLLBACK TRAN;
```

Operacja ta anuluje aktualizację produktu 2, przywracając poprzednią jego cenę 19.00. Wartość 20.00, którą uzyskał odczytujący, nigdy nie została zatwierdzona. Jest to przykład brudnego odczytu.

Poziom izolacji *READ COMMITTED*

Jeśli chcemy zapobiec sytuacjom, gdy odczytujący odbierają niezatwierdzone zmiany, trzeba użyć silniejszego poziomu izolacji. Najniższym poziomem izolacji, który uniemożliwia brudne odczyty, jest poziom *READ COMMITTED* (odczyty zatwierdzonych danych). Jest to domyślny poziom izolacji w przypadku instalacji SQL Server dla siedziby. Jak wskazuje nazwa poziomu, pozwala on odczytującemu na odczyt jedynie zatwierdzonych zmian. Ten poziom uniemożliwia odczyty niezatwierdzone poprzez wymaganie, by odczytujący nakładał blokadę współdzieloną. Oznacza to, że jeśli zapisujący nałożył blokadę wyłączną, żądanie blokady współdzielonej odczytującego będzie w konflikcie z zapisującym i odczytujący musi czekać. Zaraz po zatwierdzeniu transakcji przez zapisującego, odczytujący może uzyskać blokadę współdzieloną, a dane, które odczyta, muszą być zatwierdzonymi zmianami.

Poniższy przykład pokazuje, że w przypadku tego poziomu izolacji odczytujący może odczytywać jedynie zatwierdzone zmiany.

W oknie Connection 1 uruchamiamy poniższy kod, by otworzyć transakcję, zaktualizować produkt 2 i odpytać wiersz w celu pokazania nowej ceny produktu.

```
BEGIN TRAN;
```

```
UPDATE Production.Products
SET unitprice += 1.00
WHERE productid = 2;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Kod ten zwraca następujące wyniki:

productid	unitprice
2	20.00

Wiersz dla produktu 2 jest teraz zablokowany na wyłączność.

W oknie Connection 2 uruchamiamy poniższy kod, by ustawić poziom izolacji *READ COMMITTED* i odpytać wiersz dla produktu 2.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Warto pamiętać, że ten poziom izolacji jest poziomem domyślnym; o ile poprzednio nie zmieniliśmy poziomu izolacji sesji, nie musimy go ustawiać wprost. Instrukcja *SELECT* jest aktualnie zablokowana, ponieważ do odczytu wymagane jest nałożenie blokady współdzielonej, a to żądanie blokady współdzielonej nie jest zgodne z blokadą wyłączną utrzymywaną w oknie Connection 1.

Następnie w oknie Connection 1 uruchomimy poniższy kod, by zatwierdzić transakcje.

```
COMMIT TRAN;
```

Teraz przechodzimy do okna Connection 2 i widzimy, że uzyskaliśmy następujące dane wyjściowe:

productid	unitprice
2	20.00

Inaczej niż w przypadku poziomu *READ UNCOMMITTED*, dla poziomu izolacji *READ COMMITTED* nie uzyskujemy brudnych odczytów – możemy odczytywać tylko zatwierdzone zmiany.

Z punktu widzenia czasu trwania blokad, dla poziomu izolacji *READ COMMITTED* odczytujący utrzymuje blokadę współdzieloną do zakończenia działań na zasobie. Blokada nie jest utrzymywana do zakończenia transakcji; w rzeczywistości nie jest nawet utrzymywana do zakończenia instrukcji. Oznacza to, że pomiędzy dwoma odczytami tego samego zasobu danych w tej samej transakcji na zasób nie jest nakładana blokada. Z tego względu inna transakcja może modyfikować zasób pomiędzy tymi dwoma odczytami i odczytujący dla każdego odczytu może uzyskać różne wartości. Zjawisko to nazywane jest niepowtarzalnymi odczytami lub niespójną analizą. W przypadku większości aplikacji sytuacja taka jest do zaakceptowania, ale niektóre zastosowania nie mogą do niej dopuszczać.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

Dodatkowo trzeba się upewnić, że wszystkie otwarte transakcje zostały zakończone.

Poziom izolacji *REPEATABLE READ*

Jeśli chcemy zapewnić, że nikt nie zmieni wartości pomiędzy kolejnymi odczytami wykonywanymi w tej samej transakcji, musimy podnieść poziom izolacji do poziomu *REPEATABLE READ* (odczyty powtarzalne). Na tym poziomie odczytujący musi nałożyć blokadę współdzieloną, a ponadto blokada ta jest utrzymywana do zakończenia transakcji. Oznacza to, że zaraz po nałożeniu blokady współdzielonej na zasób danych przez odczytującego do momentu zakończenia przez niego transakcji nikt nie uzyska blokady wyłącznej, by modyfikować zasób. W ten sposób zagwarantowane jest uzyskiwanie powtarzalnych odczytów (lub inaczej spójność analiz).

Poniższy przykład ilustruje uzyskiwanie powtarzalnych odczytów. W oknie Connection 1 uruchamiamy kod, by ustawić poziom izolacji sesji *REPEATABLE READ*, otworzyć transakcję i odczytać wiersz produktu 2.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Kod ten zwraca poniższe wyniki, pokazujące bieżącą cenę produktu 2.

productid	unitprice
2	19.00

Blokada współdzielona jest nadal utrzymywana w oknie Connection 1 dla wiersza produktu 2, ponieważ dla poziomu *REPEATABLE READ* blokady współdzielone są utrzymywane do zakończenia transakcji. W oknie Connection 2 uruchamiamy poniższy kod, by dokonać próby zmodyfikowania wiersza dla produktu 2.

```
UPDATE Production.Products
SET unitprice += 1.00
WHERE productid = 2;
```

Zwróćmy uwagę, że próba zostaje zablokowana, ponieważ żądanie modyfikującego, dotyczące nałożenia blokady wyłącznej, jest w konflikcie z blokadą współdzieloną nałożoną przez odczytującego. Jeśli odczytujący działałby na poziomie izolacji *READ UNCOMMITTED* lub *READ COMMITTED*, nie utrzymywałby blokady współdzielonej i modyfikacja zostałaby przeprowadzona.

W oknie Connection 1 uruchamiamy poniższy kod, by po raz drugi odczytać wiersz dla produktu 2 i zatwierdzić transakcję.

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;

COMMIT TRAN;
```

Kod ten zwraca następujący wynik:

productid	unitprice
2	19.00

Drugi odczyt dał taki sam wynik jak przy pierwszym odczycie. Teraz, po zatwierdzeniu transakcji odczytującego i zwolnieniu blokady współdzielonej, modyfikujący w oknie Connection 2 może nałożyć blokadę wyłączną i zaktualizować wiersz.

Poziom *REPEATABLE READ* zapobiega powstawaniu innego zjawiska nazwanego utracone modyfikacje (niższe poziomy nie chronią przed występowaniem tego zjawiska). Utracone modyfikacje mają miejsce, jeśli dwie transakcje odczytują wartość, dokonują obliczeń w oparciu o odczytane wartości, a następnie aktualizują wartość. Ponieważ niższe poziomy izolacji (niższe niż *REPEATABLE READ*) nie utrzymują blokad nałożonych na odczytywany zasób, obie transakcje mogą aktualizować wartość, a „wygrywa” ta transakcja, która ostatnia zmodyfikuje wartość, zastępując zmiany wprowadzone przez drugą transakcję. Na poziomie *REPEATABLE READ* obie strony utrzymują swoje blokady współdzielone po pierwszym odczycie, tak więc żadna ze stron nie uzyska blokady wyłącznej, by przeprowadzić aktualizację. Sytuacja taka doprowadza do zakleszczenia, ale zapobiega utracie modyfikacji. Zakleszczenia omówiono w dalszej części rozdziału.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

Poziom izolacji *SERIALIZABLE*

Działając na poziomie izolacji *REPEATABLE READ* odczytujący utrzymują blokady współdzielone do zakończenia transakcji. Dzięki temu mamy gwarancję uzyskania powtarzalnych odczytów wierszy, które pierwszy raz zostały odczytane w transakcji. Transakcja jednak blokuje zasoby (na przykład wiersze), które zapytanie wyszukało podczas pierwszego uruchomienia, ale nie zasoby, które jeszcze nie istniały w tym momencie. Możliwa jest sytuacja, w której inna transakcja wstawi do tabeli nowe wiersze, które spełniają warunki filtru zapytania odczytującego. W rezultacie drugi odczyt w tej samej transakcji może zwrócić także nowe wiersze. Te nowe wiersze nazywane są *fantomami*, a takie odczyty *odczytami fantomowymi*.

Aby zapobiec odczytom fantomowym, trzeba podnieść poziom izolacji do poziomu *SERIALIZABLE* (podlegający serializacji). Pod wieloma względami poziom izolacji *SERIALIZABLE* działa podobnie do poziomowi *REPEATABLE READ*: mianowicie, poziom ten wymaga, by odczytujący nakładał blokadę współdzieloną i utrzymywał blokadę do zakończenia transakcji. Jednak poziom izolacji *SERIALIZABLE* wnosi dodatkowy aspekt – logicznie ten poziom izolacji zmusza odczytującego do blokowania całego

zakresu kluczy spełniającego warunki filtru zapytania. Oznacza to, że odczytujący nie tylko blokuje istniejące wiersze (zakwalifikowane przez filtr), ale także wiersze, które mogłyby zostać dodane w przyszłości. Mówiąc bardziej precyzyjnie, blokuje próby innych transakcji dodania wierszy, które spełniają warunki filtru.

Poniższy przykład ilustruje mechanizm zapobiegania odczytom fantomowym przez poziom izolacji *SERIALIZABLE*. W oknie Connection 1 uruchamiamy poniższy kod, by ustawić poziom izolacji *SERIALIZABLE*, otworzyć transakcję i odpytać wszystkie produkty z kategorii 1.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRAN
  SELECT productid, productname, categoryid, unitprice
  FROM Production.Products
  WHERE categoryid = 1;
```

Uzyskujemy następujące wyniki, które pokazują 12 produktów w kategorii 1.

productid	productname	categoryid	unitprice
-----	-----	-----	-----
1	Product HHYDP	1	18.00
2	Product RECZE	1	19.00
24	Product QOGNU	1	4.50
34	Product SWNJY	1	14.00
35	Product NEVTJ	1	18.00
38	Product QDOMO	1	263.50
39	Product LSOFL	1	18.00
43	Product ZZZHR	1	46.00
67	Product XLXQF	1	14.00
70	Product TOONT	1	15.00
75	Product BWRLG	1	7.75
76	Product JYGFE	1	18.00

(12 row(s) affected)

W oknie Connection 2 uruchamiamy poniższy kod, by dokonać próby wstawienia nowego produktu z kategorii 1.

```
INSERT INTO Production.Products
  (productname, supplierid, categoryid,
   unitprice, discontinued)
VALUES('Product ABCDE', 1, 1, 20.00, 0);
```

Dla wszystkich poziomów izolacji niższych niż *SERIALIZABLE* taka próba zakończy się sukcesem. W przypadku poziomu izolacji *SERIALIZABLE* próba zostanie zablokowana.

W oknie Connection 1 uruchamiamy poniższy kod, by po raz drugi odczytać produkty z kategorii 1 i zatwierdzić transakcję.

```
SELECT productid, productname, categoryid, unitprice
FROM Production.Products
WHERE categoryid = 1;

COMMIT TRAN;
```

Uzyskujemy te same wyniki, co poprzednio, bez fantomów. Teraz, kiedy zatwierdzona jest transakcja odczytującego, zwolniona jest blokada współdzielona zakresu kluczy, modyfikujący w oknie Connection 2 może nałożyć blokadę wyłączną i wstawić wiersz.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DELETE FROM Production.Products
WHERE productid > 77;
```

Poniższy kod ustawia domyślny poziom izolacji dla wszystkich otwartych połączeń.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Poziomy izolacji oparte na wersjonowaniu wierszy

Przy korzystaniu z technologii wersjonowania wierszy SQL Server może przechowywać poprzednie wersje zatwierdzonych wierszy w bazie danych *tempdb*. SQL Server obsługuje dwa poziomy izolacji nazwane *SNAPSHOT* (migawka) i *READ COMMITTED SNAPSHOT* (odczyt zatwierdzonych migawek), oparte na wersjonowaniu wierszy. Poziom *SNAPSHOT* pod względem logicznym jest podobny do poziomowi izolacji *SERIALIZABLE*, jeśli chodzi o problemy spójności, jakie mogą lub nie mogą mieć miejsca; poziom *READ COMMITTED SNAPSHOT* jest podobny do poziomowi izolacji *READ COMMITTED*. Odczytujący, którzy korzystają z poziomów izolacji opartych na wersjach wierszy, nie nakładają jednak blokad współdzielonych, tak więc nie oczekują w sytuacjach, kiedy żądane dane są zablokowane na wyłączność. Odczytujący nadal mogą uzyskać poziom spójności danych podobny do poziomów *SERIALIZABLE* i *READ COMMITTED*, odpowiednio. SQL Server udostępnia odczytującym starszą wersję wiersza, jeśli bieżąca wersja nie jest tą, którą powinni zobaczyć (nie jest jeszcze zatwierdzona).

Zwróćmy uwagę, że jeśli włączymy dowolny z poziomów izolacji opartych na migawkach (które są domyślnie włączone w Azure SQL Database), dla instrukcji *DELETE* i *UPDATE* konieczne będzie skopiowanie wersji wiersza sprzed zmiany do bazy danych *tempdb*; instrukcja *INSERT* nie powoduje tworzenia wersji w bazie danych *tempdb*, ponieważ w tym przypadku nie istnieje wcześniejsza wersja wiersza. Trzeba zdawać sobie sprawę, że włączenie jednego z poziomów izolacji opartych na wersjonowaniu wierszy może mieć negatywny wpływ na wydajność aktualizowania i usuwania danych. Wydajność odczytujących zazwyczaj jest lepsza, niekiedy bardzo znacząco, ponieważ nie muszą uzyskiwać blokad współdzielonych i nie muszą oczekiwać, jeśli na dane nałożone są blokadę wyłączną.

Poziom izolacji *SNAPSHOT*

W przypadku poziomu izolacji *SNAPSHOT* (migawka), podczas odczytywania danych zagwarantowane jest, że odczytujący uzyska ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie rozpoczęcia transakcji. Oznacza to, że mamy zagwarantowane uzyskanie zatwierdzonych odczytów i odczytów powtarzalnych oraz

że zagwarantowany jest brak odczytów fantomów – tak jak w przypadku poziomu izolacji *SERIALIZABLE*. Jednak zamiast stosowania blokad współdzielonych, ten poziom izolacji bazy na wersjach wierszy.

Jak już wspomniałem, poziomy izolacji oparte na migawkach negatywnie wpływają na wydajność podczas aktualizowania i usuwania danych, niezależnie od tego, czy modyfikacja jest, czy nie jest wykonywana w sesji uruchomionej pod kontrolą jednego z poziomów izolacji opartych na migawkach. Z tego powodu, aby zezwolić na działanie transakcji przy użyciu poziomu izolacji *SNAPSHOT* w instancji SQL Server dla siedziby (działanie takie jest domyślnie włączone dla wersji Azure SQL Database), trzeba najpierw włączyć opcję na poziomie bazy danych, uruchamiając poniższy kod w dowolnym otwartym oknie zapytania.

```
ALTER DATABASE TSQV4 SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Działanie poziomu izolacji *SNAPSHOT* ilustruje poniższy przykład. Kod uruchamiamy w oknie Connection 1, by otworzyć transakcję, zaktualizować cenę produktu 2 poprzez dodanie wartości 1.00 do jego aktualnej ceny 19.00 i odpytać wiersz produktu, by pokazać nową cenę.

```
BEGIN TRAN;  
  
UPDATE Production.Products  
    SET unitprice += 1.00  
WHERE productid = 2;  
  
SELECT productid, unitprice  
FROM Production.Products  
WHERE productid = 2;
```

Pokazane poniżej wyniki działania kodu pokazują nową cenę produktu – 20.00.

productid	unitprice
2	20.00

Zwróćmy uwagę, że nawet jeśli transakcja w oknie Connection 1 działa pod kontrolą poziomu izolacji *READ COMMITTED*, system SQL Server musi skopiować do bazy danych *tempdb* wersję wiersza, zanim wykona aktualizację (cena wynosząca 19.00). Dzieje się tak, ponieważ poziom izolacji *SNAPSHOT* został włączony na poziomie bazy danych. Jeśli ktoś rozpoczyna transakcję przy użyciu poziomu izolacji *SNAPSHOT*, ta sesja przed aktualizacją może zażądać utworzenia wersji. Dla przykładu uruchomimy poniższy kod w oknie Connection 2, by ustawić poziom izolacji *SNAPSHOT*, otworzyć transakcję i odpytać wiersz produktu 2.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
  
BEGIN TRAN;  
  
SELECT productid, unitprice  
FROM Production.Products  
WHERE productid = 2;
```

Jeśli transakcja działa pod kontrolą poziomu izolacji *SERIALIZABLE*, zapytanie zostanie zablokowane. Ponieważ jednak działa na poziomie *SNAPSHOT*, uzyskujemy ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie uruchomienia transakcji. Wersja ta (cena wynosząca 19.00) nie jest jednak aktualną wersją (cena wynosząca 20.00), tak więc SQL Server pobierze odpowiednią wersję z magazynu wersji, a kod zwraca następujący wynik.

productid	unitprice
2	19.00

Powracamy do okna Connection 1 i zatwierdzamy transakcję, która zmodyfikowała wiersz.

```
COMMIT TRAN;
```

W tym momencie aktualna wersja wiersza, czyli cena wynosząca 20.00, jest wersją zatwierdzoną. Jeśli jednak ponownie odczytamy dane w oknie Connection 2, powinniśmy nadal uzyskać ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie uruchomienia transakcji (cena wynosząca 19.00). Uruchomimy poniższy kod w oknie Connection 2, by odczytać ponownie dane, a następnie zatwierdzić transakcję.

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

```
COMMIT TRAN;
```

Jak oczekiwaliśmy, uzyskujemy wyniki informujące, że cena produktu wynosi 19.00.

productid	unitprice
2	19.00

W oknie Connection 2 uruchamiamy poniższy kod, by otworzyć nową transakcję, odpytać dane i zatwierdzić transakcję.

```
BEGIN TRAN
```

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

```
COMMIT TRAN;
```

Tym razem ostatnią zatwierdzoną wersją wiersza, która była dostępna w momencie uruchomienia transakcji, jest wiersz, dla którego cena wynosi 20.00. Z tego względu uzyskujemy następujący wynik:

productid	unitprice
2	20.00

Teraz, kiedy dla żadnej transakcji nie jest potrzebna wersja wiersza z ceną wynoszącą 19.00, uruchamiany co minutę proces oczyszczania usunie wersję z bazy danych *tempdb*. Można zauważyć, że bardzo długo trwające transakcje mogą uniemożliwić to oczyszczanie i spowodować znaczne powiększenie bazy *tempdb*, aż do jej przepełnienia.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

Wykrywanie konfliktów

Poziom izolacji *SNAPSHOT* zapobiega konfliktom podczas aktualizowania, ale inaczej niż w przypadku poziomów *REPEATABLE READ* i *SERIALIZABLE*, na których możliwe są zakleszczenia, na poziomi izolacji *SNAPSHOT* transakcja nie zostanie wykonana, wskazując wykrycie konfliktu aktualizacji. Poziom *SNAPSHOT* potrafi wykryć konflikty aktualizacji analizując magazyn wersji – potrafi określić, czy inna transakcja zmodyfikowała dane pomiędzy odczytem a zapisem, które miały miejsce w bieżącej transakcji.

Poniższy przykład jest ilustracją sytuacji, w której nie ma konfliktu aktualizacji, a po nim pokazuję przykład, w którym pojawia się taki konflikt.

W oknie Connection 1 uruchamiamy poniższy kod, by ustawić poziom izolacji *SNAPSHOT*, otworzyć transakcję i odczytać wiersz produktu 2.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Uzyskujemy następujące wyniki:

productid	unitprice
2	19.00

Zakładając, że wykonujemy pewne obliczenia w oparciu o odczytane dane, uruchamiamy poniższy kod, nadal będąc w oknie Connection 1, by zaktualizować cenę poprzednio odczytanego produktu definiując nową cenę 20.00 i zatwierdzamy transakcję.

```
UPDATE Production.Products
SET unitprice = 20.00
WHERE productid = 2;
```

```
COMMIT TRAN;
```

Żadna inna transakcja nie zmodyfikowała wiersza pomiędzy naszymi odczytem, obliczeniami i zapisem; z tego względu nie powstały konflikty aktualizowania i system SQL Server zezwolił na wykonanie aktualizacji.

Uruchamiamy poniższy kod, by produktowi 2 przywrócić poprzednią cenę 19.00.

```
UPDATE Production.Products
  SET unitprice = 19.00
WHERE productid = 2;
```

Następnie w oknie Connection 1 uruchamiamy poniższy kod, by ponownie otworzyć transakcję i odczytać wiersz produktu 2.

```
BEGIN TRAN;

  SELECT productid, unitprice
  FROM Production.Products
  WHERE productid = 2;
```

Otrzymujemy wyniki, które pokazują, że cena produktu wynosi 19.00.

productid	unitprice
2	19.00

Tym razem w oknie Connection 2 uruchamiamy poniższy kod, by zaktualizować cenę produktu 2 na 25.00.

```
UPDATE Production.Products
  SET unitprice = 25.00
WHERE productid = 2;
```

Zakładamy, że wykonaliśmy jakieś obliczenia w oknie Connection 1 w oparciu o cenę 19.00, którą odczytaliśmy. W oparciu o te obliczenia w oknie Connection 1 próbujemy zaktualizować cenę produktu na 20.00.

```
UPDATE Production.Products
  SET unitprice = 20.00
WHERE productid = 2;
```

SQL Server wykrył, że tym razem inna transakcja zmodyfikowała dane pomiędzy naszym odczytem a zapisem; z tego względu transakcja zostaje zakończona niepowodzeniem i pojawił się następujący komunikat o błędzie:

```
Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use
snapshot isolation to access table 'Production.Products' directly or indirectly
in database 'TSQV4' to update, delete, or insert the row that has been
modified or deleted by another transaction. Retry the transaction or change the
isolation level for the update/delete statement.
(Transakcja poziomu izolacji Snapshot została przerwana ze względu na konflikt
aktualizacji. Nie można używać poziomu izolacji migawki do uzyskania dostępu
do tabeli 'Production.Products' bezpośrednio lub pośrednio w bazie danych
'TSQV4', by zaktualizować, usunąć lub wstawić wiersz zmodyfikowany lub
```

usunięty przez inną transakcję. Można spróbować ponownie wykonać transakcję lub zmienić poziom izolacji dla instrukcji aktualizacji/usuwania.)

Oczywiście możemy użyć odpowiedniej obsługi błędów, by w momencie wykrycia konfliktu aktualizacji ponownie wykonać całą transakcję.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
UPDATE Production.Products  
SET unitprice = 19.00  
WHERE productid = 2;
```

Zamykamy wszystkie połączenia. Trzeba pamiętać, że jeśli nie zamkniemy wszystkich połączeń, wyniki uzyskiwane w przykładach mogą być inne niż te prezentowane w rozdziale.

Poziom izolacji *READ COMMITTED SNAPSHOT*

Poziom izolacji *READ COMMITTED SNAPSHOT* (odczyty zatwierdzonych migawek) również opiera swoje działanie na wersjach wiersza. Różni się od poziomu *SNAPSHOT* tym, że zamiast dostarczać odczytującemu ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie rozpoczęcia transakcji, odczytujący uzyskuje ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie uruchomienia instrukcji. Poziom *READ COMMITTED SNAPSHOT* nie wykrywa konfliktów aktualizacji, tak więc jest bardzo podobny pod względem logiki działania do poziomu izolacji *READ COMMITTED*, za wyjątkiem tego, że odczytujący nie muszą nakładać blokad współdzielonych i nie oczekują, jeśli na żądany zasób nałożona jest blokada wyłączna.

Aby umożliwić używanie poziomu izolacji *READ COMMITTED SNAPSHOT* w bazie danych systemu SQL Server dla siedziby (działanie to jest domyślnie włączone w Azure SQL Database), trzeba włączyć inną flagę bazy danych *READ COMMITTED SNAPSHOT*. Uruchomienie poniższego kodu umożliwia używanie tego poziomu izolacji w bazie danych *TSQV4*.

```
ALTER DATABASE TSQV4 SET READ_COMMITTED_SNAPSHOT ON;
```

Aby kod ten zadziałał prawidłowo, musimy mieć wyłączny dostęp do bazy danych *TSQV4*.

Interesujący aspekt włączenia tej flagi bazy danych jest taki, że, inaczej niż w przypadku poziomu *SNAPSHOT*, flaga ta rzeczywiście zmienia sens lub semantykę z poziomu izolacji *READ COMMITTED* na poziom *READ COMMITTED SNAPSHOT*. Oznacza to, że jeśli flaga ta jest włączona, domyślnym poziomem izolacji jest *READ COMMITTED SNAPSHOT*, chyba że jawnie zmienimy poziom izolacji sesji.

Aby zilustrować działanie poziomu *READ COMMITTED SNAPSHOT*, otwieramy dwa połączenia. W oknie Connection 1 uruchamiamy poniższy kod, by otworzyć transakcję, zaktualizować wiersz produktu 2 i odczytać wiersz, pozostawiając otwartą transakcję.

```
USE TSQLV4;
BEGIN TRAN;

    UPDATE Production.Products
        SET unitprice += 1.00
    WHERE productid = 2;

    SELECT productid, unitprice
    FROM Production.Products
    WHERE productid = 2;
```

Uzyskujemy następujący wynik, który pokazuje, że cena produktu została zmieniona i wynosi 20.00.

productid	unitprice
2	20.00

W oknie Connection 2 otwieramy transakcję i odczytujemy wiersz produktu 2, pozostawiając transakcję otwartą.

```
BEGIN TRAN;

    SELECT productid, unitprice
    FROM Production.Products
    WHERE productid = 2;
```

Uzyskujemy ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie uruchomienia instrukcji (19.00).

productid	unitprice
2	19.00

W oknie Connection 1 uruchamiamy poniższy kod, by zatwierdzić transakcję.

```
COMMIT TRAN;
```

Teraz uruchamiamy kod w oknie Connection 2, by ponownie odczytać wiersz produktu 2 i zatwierdzić transakcję.

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;

COMMIT TRAN;
```

Jeśli kod ten uruchomiony byłby pod kontrolą poziomu izolacji *SNAPSHOT*, otrzymalibyśmy cenę 19.00; ponieważ jednak kod działał na poziomie *READ COMMITTED SNAPSHOT*, otrzymaliśmy ostatnią zatwierdzoną wersję wiersza, która była dostępna w momencie uruchomienia instrukcji (20.00), a nie w momencie uruchomienia transakcji (19.00).


```

productid  unitprice
-----
2          20.00

```

Pamiętamy, że to zjawisko nazywane jest niepowtarzalnymi odczytami lub niespójnymi analizami.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę danych.

```

UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;

```

Zamykamy wszystkie połączenia. Następnie otwieramy nowe połączenie i uruchamiamy poniższy kod, by w bazie danych *TSQLV4* wyłączyć poziomy izolacji oparte na wersjach wierszy.

```

ALTER DATABASE TSQLV4 SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE TSQLV4 SET READ_COMMITTED_SNAPSHOT OFF;

```

Podsumowanie poziomów izolacji

Tabela 10-3 zawiera podsumowanie problemów logicznej spójności danych, które mogą lub nie mogą się zdarzać na każdym poziomie izolacji. Pokazuje też, czy dany poziom izolacji wykrywa konflikty aktualizacji i czy poziom izolacji korzysta z wersji wierszy.

TABELA 10-3 Podsumowanie poziomów izolacji

Poziom izolacji	Dopuszczanie niezatwierdzonych odczytów?	Dopuszczanie niepowtarzalnych odczytów?	Dopuszczanie utraconych aktualizacji?	Dopuszczanie odczytów fantomów?	Wykrywanie konfliktów aktualizacji?	Używanie wersji wiersza?
<i>READ UNCOMMITTED</i>	Tak	Tak	Tak	Tak	Nie	Nie
<i>READ COMMITTED</i>	Nie	Tak	Tak	Tak	Nie	Nie
<i>READ COMMITTED SNAPSHOT</i>	Nie	Tak	Tak	Tak	Nie	Tak
<i>REPEATABLE READ</i>	Nie	Nie	Nie	Tak	Nie	Nie
<i>SERIALIZABLE</i>	Nie	Nie	Nie	Nie	Nie	Nie
<i>SNAPSHOT</i>	Nie	Nie	Nie	Nie	Tak	Tak

Zakleszczenia

Zakleszczenie (*deadlock*) to sytuacja, w której co najmniej dwa procesy blokują się nawzajem. Przykładem zakleszczenia się dwóch procesów może być sytuacja, kiedy proces A blokuje proces B, a proces B blokuje proces A. Przykładem zakleszczenia się większej liczby procesów może być sytuacja, kiedy proces A blokuje proces B, proces B blokuje proces C, a proces C blokuje proces A. W każdej sytuacji SQL Server wykrywa zakleszczenie i interweniuje, kończąc jedną z transakcji. Gdyby system SQL Server nie zareagował, procesy pozostałyby zakleszczone na wieczność.

O ile nie zostanie to określone inaczej, SQL Server wybiera do zakończenia tę transakcję, która wykonała mniej pracy (opierając się na aktywności zapisanej w dzienniku transakcji), ponieważ jej wycofanie będzie wymagało mniejszego nakładu. SQL Server pozwala jednak nam ustawiać opcję sesji nazwaną *DEADLOCK_PRIORITY* za pomocą jednej z 21 wartości z zakresu od -10 do 10 . Proces o najniższym priorytecie zakleszczenia jest wybierany jako „ofiara” zakleszczenia, niezależnie od tego, ile pracy zostało wykonane przez transakcję; w przypadkach nierozstrzygniętych decyduje ilość wykonanej pracy.

Poniższy przykład pokazuje proste zakleszczenie. Później pokażę, jak unikać występowania zakleszczeń w systemie.

Otwieramy dwa połączenia z bazą danych TSQLV4. W oknie Connection 1 uruchamiamy poniższy kod, by otworzyć nową transakcję i zaktualizować wiersz tabeli *Production.Products* dla produktu 2, po czym pozostawiamy transakcję otwartą.

```
USE TSQLV4;

BEGIN TRAN;

    UPDATE Production.Products
        SET unitprice += 1.00
    WHERE productid = 2;
```

W oknie Connection 2 uruchamiamy poniższy kod, by otworzyć nową transakcję, zaktualizować wiersz tabeli *Sales.OrderDetails* dla produktu 2 i pozostawiamy transakcję otwartą.

```
BEGIN TRAN;

    UPDATE Sales.OrderDetails
        SET unitprice += 1.00
    WHERE productid = 2;
```

W tym momencie transakcja w oknie Connection 1 nałożyła blokadę wyłączną na wiersz produktu 2 w tabeli *Production.Products*, a transakcja w oknie Connection 2 nałożyła blokady na wiersze produktu 2 w tabeli *Sales.OrderDetails*. Obie instrukcje się powiodły i jeszcze nie pojawiło się żadne zablokowanie.

W oknie Connection 1 uruchamiamy poniższy kod, by wykonać zapytanie dotyczące produktu 2 w tabeli *Sales.OrderDetails* i zatwierdzić transakcję:

```
SELECT orderid, productid, unitprice
FROM Sales.OrderDetails
WHERE productid = 2;
```

COMMIT TRAN;

Transakcja w oknie Connection 1 wymaga blokady współdzielonej, by można było przeprowadzić odczyty. Ponieważ inna transakcja utrzymuje blokadę wyłączną dla tego samego zasobu, transakcja w oknie Connection 1 zostaje zablokowana. W tym momencie mamy sytuację blokowania, ale jeszcze nie jest to zakleszczenie. Oczywiście istnieje możliwość, że transakcja w oknie Connection 2 zostanie zakończona i zwolnione wszystkie blokady, co pozwoli transakcji w oknie Connection 1 nałożyć żądane blokady.

Następnie w oknie Connection 2 uruchamiamy poniższy kod, by wykonać zapytanie o wiersz produktu 2 w tabeli *Product.Production* i zatwierdzić transakcję.

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

COMMIT TRAN;

Aby można było wykonać odczyt, transakcja w oknie Connection 2 musi nałożyć blokadę współdzieloną na wiersz produktu 2 w tabeli *Product.Production*, tak więc żądanie to jest teraz w konflikcie z wyłączną blokadą utrzymywaną dla tego samego zasobu w oknie Connection 1. Oba procesy blokują się nawzajem – czyli mamy do czynienia z zakleszczeniem. System SQL Server rozpoznaje zakleszczenie (zazwyczaj w ciągu kilku sekund), wybiera jeden z dwóch procesów jako ofiarę zakleszczenia i kończy transakcję, generując poniższy komunikat o błędzie:

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 52) was deadlocked on lock resources with another
process and has been chosen as the deadlock victim. Rerun the transaction.
```

(Nastąpiło zakleszczenie transakcji (Proces ID 52) spowodowane blokadą nałożoną przez inny proces; transakcja została wybrana jako ofiara zakleszczenia. Należy ponownie uruchomić transakcję.)

W tym przykładzie system SQL Server do zakończenia wybrał transakcję w oknie Connection 1 (pokazaną tu jako proces ID 52). Ponieważ nie ustawiliśmy priorytetów zakleszczenia, a obie transakcje miały podobną ilość wykonanej pracy, dowolna z transakcji mogła być zakończona.

Zakleszczenia są kosztownymi sytuacjami, ponieważ wiążą się z odwracaniem działań, które już zostały wykonane. Aby unikać powstawania zakleszczeń w systemie, możemy kierować się poniższymi zaleceniami.

Oczywiste jest to, że im dłuższe transakcje, tym dłużej trwają blokady, co zwiększa prawdopodobieństwo wystąpienia zakleszczeń. Powinniśmy utrzymywać możliwie

krótkie transakcje, przenosząc na zewnątrz transakcji te działania, które pod względem logicznym nie są częścią tej samej jednostki pracy.

Typowe zakleszczenie ma miejsce, gdy dwie transakcje uzyskują dostęp do zasobów w odwrotnej kolejności. W naszym przykładzie, w oknie Connection 1 najpierw uzyskiwany jest dostęp do wiersza w tabeli *Production.Products*, a następnie do wiersza w tabeli *Sales.OrderDetails*, natomiast w oknie Connection 2 najpierw sięgamy do wiersza w tabeli *Sales.OrderDetails*, a potem do wiersza w tabeli *Production.Products*. Ten rodzaj zakleszczenia nie miałby miejsca, jeśli obie transakcje uzyskiwałyby dostęp do zasobów w tej samej kolejności. Poprzez zmianę kolejności w jednej z transakcji możemy zapobiec takiemu rodzajowi zakleszczenia, oczywiście przy założeniu, że nie ma to znaczenia dla logiki aplikacji.

Podany przykład zakleszczenia zawiera rzeczywisty konflikt logiczny, jako że obie strony próbują uzyskać dostęp do tych samych wierszy. Jednak zakleszczenia często powstają także przy braku prawdziwego konfliktu logicznego, a są spowodowane brakiem prawidłowego indeksowania, pozwalającego obsłużyć filtry zapytania. Załóżmy na przykład, że obie instrukcje w transakcji w oknie Connection 2 mają filtrować produkt 5. Teraz, jako że instrukcje w oknie Connection 1 obsługują produkt 2, a instrukcje w oknie Connection 2 obsługują produkt 5, nie powinno być żadnego konfliktu. Jeśli jednak żadne indeksy dla kolumny *productid* w tabelach nie obsługują filtru, SQL Server musi skanować (i blokować) wszystkie wiersze tabeli. Działanie takie, rzecz jasna, może prowadzić do zakleszczenia. Krótko mówiąc, przy dobrym projekcie indeksu trudniej o sytuacje, w których powstają zakleszczenia, których nie cechuje rzeczywisty konflikt logiczny.

Inną kwestią, która pozwala unikać zakleszczeń, jest wybór poziomu izolacji. Instrukcje *SELECT* w przykładzie muszą nakładać blokady współdzielone, ponieważ działają na poziomie izolacji *READ COMMITTED*. Jeśli użyjemy poziomu *READ COMMITTED SNAPSHOT*, odczytujący nie muszą nakładać blokad współdzielonych i takie zakleszczenia (wynikające z nakładania blokad współdzielonych) mogą być eliminowane.

Po ukończeniu ćwiczeń uruchamiamy poniższy kod, by wyczyścić bazę we wszystkich połączeniach.

```
UPDATE Production.Products
  SET unitprice = 19.00
WHERE productid = 2;

UPDATE Sales.OrderDetails
  SET unitprice = 19.00
WHERE productid = 2
  AND orderid >= 10500;

UPDATE Sales.OrderDetails
  SET unitprice = 15.20
WHERE productid = 2
  AND orderid < 10500;
```

Podsumowanie

Rozdział ten zawiera omówienie zagadnień związanych z transakcjami i współbieżnością. Dowiedzieliśmy się, czym są transakcje i jak system SQL Server nimi zarządza. Wyjaśniłem także, jak SQL Server izoluje dane przetwarzane w jednej transakcji przed niespójnym użyciem przez inną transakcję oraz jak rozwiązywać problemy, które powstają w sytuacjach blokowania. Opisałem również, jak kontrolować poziomy spójności uzyskiwanych danych poprzez wybór poziomu izolacji oraz jaki wpływ na współbieżność ma dokonany wybór. Opisałem cztery poziomy izolacji, które nie korzystają z wersji wierszy oraz dwa poziomy, dla których wersje wierszy są podstawą działania. Na koniec przedstawiłem problem zakleszczenia i przedstawiłem zalecenia praktyczne, które pozwalają zmniejszyć liczbę pojawiających się zakleszczeń.

Ćwiczenia

Podrozdział ten zawiera ćwiczenia, które ułatwiają lepsze przyswojenie tematyki opisanej w rozdziale. Ćwiczenia w większości poprzednich rozdziałów dotyczyły zadań, dla których należało opracować rozwiązanie w postaci zapytania lub instrukcji języka T-SQL. Ćwiczenia w tym rozdziale są inne. Przedstawiono tu instrukcje, które służą rozwiązywaniu problemów związanych z blokowaniem i zakleszczeniami oraz pozwalające zaobserwować działania różnych poziomów izolacji. Z tego względu, inaczej niż w pozostałych rozdziałach, nie zamieszczono tu oddzielnego podrozdziału „Rozwiązania”.

We wszystkich ćwiczeniach tego rozdziału należy korzystać z połączenia z przykładową bazą danych *TSQV4*.

Ćwiczenie 1

Ćwiczenia 1-1 do 1-6 dotyczą blokowania. Zakładają one, że używamy poziomu izolacji *READ COMMITTED* (blokowanie). Jest to poziom domyślny dla produktu SQL Server w siedzibie. Aby wykonać te ćwiczenia w Azure SQL Database, należy wyłączyć wersjonowanie wierszy.

Ćwiczenie 1-1

Otworzyć trzy połączenia w programie SQL Server Management Studio (w ćwiczeniach będziemy odwoływać się do nich poprzez nazwy Connection 1, Connection 2 i Connection 3). W oknie Connection 1 uruchomić poniższy kod, by zaktualizować wiersze w tabeli *Sales.OrderDetails*.

```
BEGIN TRAN;  
UPDATE Sales.OrderDetails
```

```
SET discount = 0.05
WHERE orderid = 10249;
```

Ćwiczenie 1-2

W oknie Connection 2 uruchomić poniższy kod, by odpytać tabelę *Sales.OrderDetails*; okno Connection 2 będzie zablokowane.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Ćwiczenie 1-3

Uruchomić poniższy kod w oknie Connection 3 i zidentyfikować blokady oraz identyfikatory procesów zaangażowane w łańcuch blokowania.

```
SELECT -- użyj *, by poznać inne atrybuty
    request_session_id AS spid,
    resource_type AS restype,
    resource_database_id AS dbid,
    resource_description AS res,
    resource_associated_entity_id AS resid,
    request_mode AS mode,
    request_status AS status
FROM sys.dm_tran_locks;
```

Ćwiczenie 1-4

Zastąpić identyfikatory procesów 52 i 53 identyfikatorami procesów zaangażowanych w łańcuch blokowania, które zostały znalezione w poprzednim ćwiczeniu. Uruchomić poniższy kod, by uzyskać informacje o połączeniu, sesji i blokowaniu dla procesów zaangażowanych w łańcuch blokowania.

```
-- Informacje o połączeniu:
SELECT -- use * to explore
    session_id AS spid,
    connect_time,
    last_read,
    last_write,
    most_recent_sql_handle
FROM sys.dm_exec_connections
WHERE session_id IN(52, 53);

-- Informacje o sesji
SELECT -- use * to explore
    session_id AS spid,
    login_time,
    host_name,
    program_name,
    login_name,
```

```

    nt_user_name,
    last_request_start_time,
    last_request_end_time
FROM sys.dm_exec_sessions
WHERE session_id IN(52, 53);

-- Blokowanie
SELECT -- use * to explore
    session_id AS spid,
    blocking_session_id,
    command,
    sql_handle,
    database_id,
    wait_type,
    wait_time,
    wait_resource
FROM sys.dm_exec_requests
WHERE blocking_session_id > 0;

```

Ćwiczenie 1-5

Uruchomić poniższy kod, by uzyskać tekst poleceń SQL dla połączeń zaangażowanych w łańcuch blokowania.

```

SELECT session_id, text
FROM sys.dm_exec_connections
    CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(52, 53);

```

Ćwiczenie 1-6

W oknie Connection 1 uruchomić poniższy kod, by wycofać transakcję.

```
ROLLBACK TRAN;
```

W oknie Connection 2 będzie można zauważyć, że zapytanie *SELECT* zwróciło dwa wiersze pozycji zamówienia i że te wiersze nie zostały zmodyfikowane.

Przypomnijmy, że jeśli zachodzi konieczność zakończenia transakcji blokującej, możemy użyć polecenia *KILL* z odpowiednim identyfikatorem procesu. Zamknąć wszystkie połączenia.

Ćwiczenie 2

Ćwiczenia 2-1 do 2-6 dotyczą poziomów izolacji.

Ćwiczenie 2-1

Ćwiczenie to dotyczy poziomu izolacji *READ UNCOMMITTED* i składa się z kilku podpunktów.

Ćwiczenie 2-1a

Otworzyć dwa nowe połączenia (ćwiczenie to odwołuje się do nich za pomocą nazw Connection 1 i Connection 2).

Ćwiczenie 2-1b

W oknie Connection 1 uruchomić poniższy kod, by zaktualizować wiersze w tabeli *Sales.OrderDetails* i odpytać je.

```
BEGIN TRAN;

    UPDATE Sales.OrderDetails
        SET discount += 0.05
    WHERE orderid = 10249;

    SELECT orderid, productid, unitprice, qty, discount
    FROM Sales.OrderDetails
    WHERE orderid = 10249;
```

Ćwiczenie 2-1c

W oknie Connection 2 uruchomić poniższy kod, by ustawić poziom izolacji *READ UNCOMMITTED* i wykonać zapytanie do tabeli *Sales.OrderDetails*.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Można zauważyć, że otrzymaliśmy zmodyfikowaną, niezatwierdzoną wersję wierszy.

Ćwiczenie 2-1d

W oknie Connection 1 uruchomić poniższy kod, by wycofać transakcję.

```
ROLLBACK TRAN;
```

Ćwiczenie 2-2

Ćwiczenie to dotyczy poziomu izolacji *READ COMMITTED*.

Ćwiczenie 2-2a

W oknie Connection 1 uruchomić poniższy kod, by zaktualizować wiersze w tabeli *Sales.OrderDetails* i odczytać je.

```
BEGIN TRAN;
```



```
UPDATE Sales.OrderDetails
    SET discount += 0.05
WHERE orderid = 10249;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Ćwiczenie 2-2b

W oknie Connection 2 uruchomić poniższy kod, by ustawić poziom izolacji *READ COMMITTED* i wykonać zapytanie tabeli *Sales.OrderDetails*.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails -- WITH (READCOMMITTEDLOCK)
WHERE orderid = 10249;
```

Zwróćmy uwagę, że teraz nastąpiło zablokowanie.

Ćwiczenie 2-2c

W oknie Connection 1 uruchomić poniższy kod, by zatwierdzić transakcję.

```
COMMIT TRAN;
```

Ćwiczenie 2-2d

Przejsz do okna Connection 2 i zwrócić uwagę, że otrzymaliśmy zmodyfikowaną, zatwierdzoną wersję wierszy.

Ćwiczenie 2-2e

Uruchomić poniższy kod, by wyczyścić bazę.

```
UPDATE Sales.OrderDetails
    SET discount = 0.00
WHERE orderid = 10249;
```

Ćwiczenie 2-3

Ćwiczenie to dotyczy poziomu izolacji *REPEATABLE READ*.

Ćwiczenie 2-3a

W oknie Connection 1 uruchomić poniższy kod, by ustawić poziom izolacji *REPEATABLE READ*, otworzyć transakcję i odczytać dane z tabeli *Sales.OrderDetails*.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRAN;
```

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Uzyskujemy dwa wiersze o wartości rabatu 0.00.

Ćwiczenie 2-3b

W oknie Connection 2 uruchomić poniższy kod i zwrócić uwagę, że nastąpiła blokada.

```
UPDATE Sales.OrderDetails
SET discount += 0.05
WHERE orderid = 10249;
```

Ćwiczenie 2-3c

W oknie Connection 1 uruchomić poniższy kod, by ponownie odczytać dane i zatwierdzić transakcję.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

```
COMMIT TRAN;
```

Ponownie otrzymaliśmy dwa wiersze o wartości rabatu 0.00, czyli uzyskaliśmy powtarzalne odczyty. Zwróćmy uwagę, że jeśli kod działa pod kontrolą niższego poziomu izolacji (na przykład *READ UNCOMMITTED* lub *READ COMMITTED*), instrukcja *UPDATE* nie będzie blokowana, a my uzyskamy niepowtarzalne odczyty.

Ćwiczenie 2-3d

Przejsz do okna Connection 2 i zwrócić uwagę, że aktualizacja została zakończona.

Ćwiczenie 2-3e

Uruchomić poniższy kod, by wyczyścić bazę.

```
UPDATE Sales.OrderDetails
SET discount = 0.00
WHERE orderid = 10249;
```

Ćwiczenie 2-4

Ćwiczenie to dotyczy poziomu izolacji *SERIALIZABLE*.

Ćwiczenie 2-4a

W oknie Connection 1 uruchomić poniższy kod, by ustawić poziom izolacji *SERIALIZABLE* i wykonać zapytanie tabeli *Sales.OrderDetails*.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRAN;

    SELECT orderid, productid, unitprice, qty, discount
    FROM Sales.OrderDetails
    WHERE orderid = 10249;
```

Ćwiczenie 2-4b

W oknie Connection 2 uruchomić poniższy kod, by wstawić wiersz do tabeli *Sales.OrderDetails* o takim samym identyfikatorze zamówienia, który był filtrowany przez poprzednie zapytanie i zwrócić uwagę, że nastąpiło zablokowanie.

```
INSERT INTO Sales.OrderDetails
    (orderid, productid, unitprice, qty, discount)
VALUES(10249, 2, 19.00, 10, 0.00);
```

Zwrócić uwagę, że dla niższych poziomów izolacji (*READ UNCOMMITTED*, *READ COMMITTED* lub *REPEATABLE READ*) ta instrukcja *INSERT* nie zostałaby zablokowana.

Ćwiczenie 2-4c

W oknie Connection 1 uruchomić poniższy kod, by ponownie wykonać zapytanie tabeli *Sales.OrderDetails* i zatwierdzić transakcję.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;

COMMIT TRAN;
```

Uzyskujemy taki sam wynik jak w poprzednim zapytaniu tej samej transakcji, a ponieważ instrukcja *INSERT* została zablokowana, nie uzyskujemy żadnych odczytów fantomowych.

Ćwiczenie 2-4d

Powrócić do okna Connection 2 i zwrócić uwagę, że zakończyła się instrukcja *INSERT*.

Ćwiczenie 2-4e

Uruchomić poniższy kod, by wyczyścić bazę.

```
DELETE FROM Sales.OrderDetails
WHERE orderid = 10249
    AND productid = 2;
```

Ćwiczenie 2-4f

W obu oknach Connection 1 i Connection 2 uruchomić poniższy kod, by ustawić domyślny poziom izolacji.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Ćwiczenie 2-5

Ćwiczenie to dotyczy poziomu izolacji *SNAPSHOT*.

Ćwiczenie 2-5a

Jeśli ćwiczenie wykonywane jest w instancji SQL Server dla siedziby, uruchomić poniższy kod, by umożliwić działanie poziomu izolacji *SNAPSHOT* w bazie danych *TSQLV4* (opcja ta domyślnie jest włączona dla wersji Azure SQL Database):

```
ALTER DATABASE TSQLV4 SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Ćwiczenie 2-5b

W oknie Connection 1 uruchomić poniższy kod, by otworzyć transakcję, zaktualizować wiersze w tabeli *Sales.OrderDetails* i odczytać je.

```
BEGIN TRAN;
```

```
    UPDATE Sales.OrderDetails  
        SET discount += 0.05  
    WHERE orderid = 10249;
```

```
    SELECT orderid, productid, unitprice, qty, discount  
    FROM Sales.OrderDetails  
    WHERE orderid = 10249;
```

Ćwiczenie 2-5c

W oknie Connection 2 uruchomić poniższy kod, by ustawić poziom izolacji *SNAPSHOT* i wykonać zapytanie tabeli *Sales.OrderDetails*. Zwrócić uwagę, że nie nastąpiło zablokowanie – zamiast blokowania, otrzymaliśmy wcześniejszą, spójną wersję danych, która była dostępna w momencie uruchomienia transakcji (wartość rabatu wynosząca 0.00).

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
    SELECT orderid, productid, unitprice, qty, discount  
    FROM Sales.OrderDetails  
    WHERE orderid = 10249;
```

Ćwiczenie 2-5d

Przejsć do okna Connection 1 i zatwierdzić transakcję.

```
COMMIT TRAN;
```

Ćwiczenie 2-5e

Przejsć do okna Connection 2 i ponownie wykonać zapytanie danych; zwrócić uwagę, że wartość rabatu nadal wynosi 0.00.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Ćwiczenie 2-5f

W oknie Connection 2 zatwierdzić transakcję i ponownie odczytać dane; zwrócić uwagę, że teraz wartość rabatu wynosi 0.05.

```
COMMIT TRAN;
```

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Ćwiczenie 2-5g

Uruchomić poniższy kod, by wyczyścić bazę.

```
UPDATE Sales.OrderDetails
SET discount = 0.00
WHERE orderid = 10249;
```

Zamknąć wszystkie połączenia.

Ćwiczenie 2-6

Ćwiczenie to dotyczy poziomu izolacji *READ COMMITTED SNAPSHOT*.

Ćwiczenie 2-6a

Jeśli ćwiczenie wykonywane jest w oparciu o instancję SQL Server dla siedziby, włączyć opcję *READ_COMMITTED_SNAPSHOT* w bazie danych TSQV4 (opcja ta jest domyślnie włączona w wersji Azure SQL Database).

```
ALTER DATABASE TSQV4 SET READ_COMMITTED_SNAPSHOT ON;
```

Ćwiczenie 2-6b

Otworzyć dwa nowe połączenia (będziemy odwoływać się do nich po nazwach Connection 1 i Connection 2).

Ćwiczenie 2-6c

W oknie Connection 1 uruchomić poniższy kod, by otworzyć transakcję, zaktualizować wiersze w tabeli *Sales.OrderDetails* i odczytać je.

```
BEGIN TRAN;
```

```
UPDATE Sales.OrderDetails
    SET discount += 0.05
WHERE orderid = 10249;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Ćwiczenie 2-6d

W oknie Connection 2 uruchomić poniższy kod, który teraz działa pod kontrolą poziomu izolacji *READ COMMITTED SNAPSHOT*, ponieważ włączona została flaga bazy danych *READ_COMMITTED_SNAPSHOT*. Warto zwrócić uwagę, że nie nastąpiło blokowanie – w zamian uzyskaliśmy wcześniejszą, spójną wersję danych, która była dostępna w momencie uruchomienia instrukcji (wartość rabatu wynosi 0.00).

```
BEGIN TRAN;
```

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Ćwiczenie 2-6e

Przejsz do okna Connection 1 i zatwierdzić transakcję.

```
COMMIT TRAN;
```

Ćwiczenie 2-6f

Przejsz do okna Connection 2, ponownie wykonać zapytanie danych i zatwierdzić transakcję; zwrócić uwagę, że uzyskaliśmy nową wartość rabatu wynoszącą 0.05.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

```
COMMIT TRAN;
```

Ćwiczenie 2-6g

Uruchomić poniższy kod, by wyczyścić bazę.

```
UPDATE Sales.OrderDetails  
    SET discount = 0.00  
WHERE orderid = 10249;
```

Zamknąć wszystkie połączenia.

Ćwiczenie 2-6h

Jeśli ćwiczenie wykonywane jest instancji SQL Server dla siedziby, przywrócić flagom bazy danych ich wartości domyślne, wyłączając poziomy izolacji korzystające z wersji wierszy.

```
ALTER DATABASE TSQLV4 SET ALLOW_SNAPSHOT_ISOLATION OFF;  
ALTER DATABASE TSQLV4 SET READ_COMMITTED_SNAPSHOT OFF;
```

Ćwiczenie 3

Ćwiczenie 3 (kroki 1 do 7) dotyczy zakleszczeń.

Ćwiczenie 3-1

Otworzyć dwa nowe połączenia (w ćwiczeniu będziemy odwoływać się do nich po nazwach Connection 1 i Connection 2).

Ćwiczenie 3-2

W oknie Connection 1 uruchomić poniższy kod, by otworzyć transakcję i zaktualizować wiersz produktu 2 w tabeli *Production.Products*.

```
BEGIN TRAN;  
  
    UPDATE Production.Products  
        SET unitprice += 1.00  
    WHERE productid = 2;
```

Ćwiczenie 3-3

W oknie Connection 2 uruchomić poniższy kod, by otworzyć transakcję i zaktualizować wiersz produktu 3 w tabeli *Production.Products*.

```
BEGIN TRAN;  
  
    UPDATE Production.Products  
        SET unitprice += 1.00  
    WHERE productid = 3;
```

Ćwiczenie 3-4

W oknie Connection 1 uruchomić poniższy kod, by odczytać produkt 3. Nastąpi zablokowanie (pamiętajmy o usunięciu znaczników komentarza wskazówki, jeśli połączenie dotyczy SQL Database).

```
SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 3;
```

```
COMMIT TRAN;
```

Ćwiczenie 3-5

W oknie Connection 2 uruchomić poniższy kod, by wykonać zapytanie produktu 2. Nastąpi zablokowanie oraz wygenerowany zostanie komunikat o błędzie zakleszczenia w oknie Connection 1 lub w oknie Connection 2.

```
SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

```
COMMIT TRAN;
```

Ćwiczenie 3-6

Co można zasugerować jako metodę zapobiegania tego rodzaju zakleszczeniu? Wskazówka: opis rozwiązania znaleźć można w podrozdziale „Zakleszczenia”.

Ćwiczenie 3-7

Uruchomić poniższy kod, by wyczyścić bazę danych.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

```
UPDATE Production.Products
SET unitprice = 10.00
WHERE productid = 3;
```


ROZDZIAŁ 11

Obiekty programowalne

Rozdział ten zawiera skrótowy opis obiektów programowalnych, pozwalając zapoznać Czytelnika z możliwościami języka T-SQL i używanymi w nim koncepcjami. Tematyka rozdziału obejmuje zmienne, wsady, elementy przepływu, kursory, tabele tymczasowe, procedury, w tym funkcje definiowane przez użytkownika, procedury składowane, wyzwalacze i dynamiczne instrukcje SQL. Celem rozdziału jest zaprezentowanie ogólnego opisu, bez zagłębiania się w szczegóły techniczne. Będę skupiać się na aspektach logicznych i możliwościach obiektów programowalnych, a nie na poznawaniu wszystkich elementów kodu i ich szczegółów technicznych, gdyż te stanowią materiał na oddzielną książkę, a mówiąc ściślej, wiele książek, których zresztą nie brakuje na rynku.

Zmienne

Zmienne umożliwiają tymczasowe przechowywanie wartości danych, by można je było później użyć w tym samym wsadzie, w którym zmienne zostały zadeklarowane. Wsady (*ang. batch*) opisane zostały w dalszej części rozdziału; na razie wystarczy znać ogólną definicję, czyli wiedzieć, że jest to jedna lub wiele instrukcji T-SQL wysłanych do wykonania do systemu SQL Server jako pojedyncza jednostka.

Do zadeklarowania jednej lub wielu zmiennych posługujemy się instrukcją *DECLARE*, a instrukcja *SET* służy do przypisania wartości pojedynczej zmiennej. Na przykład poniższy kod deklaruje zmienną nazwaną *@i*, której typ danych to *INT* i przypisuje tej zmiennej wartość 10.

```
DECLARE @i AS INT;  
SET @i = 10;
```

Alternatywnie można zadeklarować i zainicjować zmienną w tej samej instrukcji, jak w poniższym kodzie:

```
DECLARE @i AS INT = 10;
```

Gdy przypisujemy wartość do zmiennej skalarnej, wartość musi być wynikiem wyrażenia skalarnego. Wyrażenie może być podzapytaniem skalarным. Dla przykładu, poniższy kod deklaruje zmienną nazwaną *@empname* i przypisuje jej wyniki podzapytania skalarnego, które zwraca pełną nazwę pracownika o identyfikatorze 3.

```
USE TSQLV4;
```

```
DECLARE @empname AS NVARCHAR(31);
SET @empname = (SELECT firstname + N' ' + lastname
                FROM HR.Employees
                WHERE empid = 3);
```

```
SELECT @empname AS empname;
```

Kod ten zwraca następujące wyniki:

```
empname
-----
Judy Lew
```

Instrukcja *SET* w danym momencie może operować tylko na jednej zmiennej, tak więc jeśli zachodzi potrzeba przypisania wartości do wielu atrybutów, trzeba użyć wielu instrukcji *SET*. Może to wiązać się z niepotrzebnym nakładem, kiedy chcemy pobrać wiele wartości atrybutów z tego samego wiersza. Dla przykładu, poniższy kod stosuje dwie oddzielne instrukcje *SET* do pobrania imienia i nazwiska pracownika o identyfikatorze 3 do dwóch oddzielnych zmiennych.

```
DECLARE @firstname AS NVARCHAR(10), @lastname AS NVARCHAR(20);
SET @firstname = (SELECT firstname
                  FROM HR.Employees
                  WHERE empid = 3);
SET @lastname = (SELECT lastname
                  FROM HR.Employees
                  WHERE empid = 3);

SELECT @firstname AS firstname, @lastname AS lastname;
```

Kod ten generuje następujące wyniki:

```
firstname  lastname
-----
Judy       Lew
```

Język T-SQL obsługuje także niestandardowe przypisywanie *SELECT*, które pozwala wykonać odczyt danych i przypisać wiele wartości uzyskanych z tego samego wiersza do wielu zmiennych w pojedynczej instrukcji, jak w poniższym przykładzie:

```
DECLARE @firstname AS NVARCHAR(10), @lastname AS NVARCHAR(20);

SELECT
    @firstname = firstname,
    @lastname = lastname
FROM HR.Employees
WHERE empid = 3;

SELECT @firstname AS firstname, @lastname AS lastname;
```

Przypisanie *SELECT* ma przewidywalne działanie, jeśli zakwalifikowany jest tylko jeden wiersz. Jeśli jednak zapytanie zakwalifikowało większą liczbę wierszy, kod ten

nie zakończy się błędem. Przypisania będą miały miejsce dla każdego zakwalifikowanego wiersza i dla każdego wiersza, do którego uzyskiwany jest dostęp, wartości bieżącego wiersza zastępują istniejące wartości w zmiennych. Po zakończeniu przypisywania *SELECT* wartościami w zmiennych są te, które zostały pobrane z ostatniego wiersza, do którego SQL Server uzyskał dostęp. Na przykład poniższe przypisanie *SELECT* ma dwa zakwalifikowane wiersze.

```
DECLARE @empname AS NVARCHAR(31);

SELECT @empname = firstname + N' ' + lastname
FROM HR.Employees
WHERE mgrid = 2;

SELECT @empname AS empname;
```

Informacje o pracowniku, które na końcu znajdują się w zmiennej po zakończeniu przypisania *SELECT*, zależą od kolejności, w której SQL Server uzyska dostęp do tych wierszy – a nad tą kolejnością nie mamy żadnej kontroli. Kiedy uruchomiłem ten kod, uzyskałem następujące wyniki:

```
empname
-----
Sven Mortensen
```

Instrukcja *SET* jest bezpieczniejsza niż przypisanie *SELECT*, ponieważ do pobierania danych z tabeli wymaga stosowania podzapytania skalarne. Przypomnijmy, że podzapytanie skalarne zakończy się błędem, jeśli zwraca więcej niż jedną wartość. Na przykład poniższy kod nie zadziała:

```
DECLARE @empname AS NVARCHAR(31);

SET @empname = (SELECT firstname + N' ' + lastname
                FROM HR.Employees
                WHERE mgrid = 2);

SELECT @empname AS empname;
```

Ponieważ zmienna nie została przypisana, jej wartością pozostaje znacznik *NULL*, który jest wartością domyślną dla niezainicjowanych zmiennych. Kod ten zwraca następujące wyniki:

```
Msg 512, Level 16, State 1, Line 3
Subquery returned more than 1 value. This is not permitted when the subquery
follows =, !=, <, <=, >, >= or when the subquery is used as an expression.

(Z podzapytania została zwrócona więcej niż 1 wartość. Jest to niedozwolone,
jeśli podzapytanie następuje po znaku =, !=, <, <=, > lub >= albo gdy jest
używane jako wyrażenie.)
empname
-----
NULL
```

Wsady

Wsad (*ang. batch*) to jedna lub wiele instrukcji T-SQL wysłanych jako pojedyncza jednostka przez aplikację kliencką do systemu SQL Server w celu ich wykonania. Wsad jako jedna jednostka (w całości) poddawany jest procesowi analizy (sprawdzenie składni), rozpoznawania (sprawdzenie istnienia kolumn i obiektów odniesień), sprawdzenia uprawnień i optymalizacji.

Nie należy mylić transakcji i wsadów. Transakcja jest niepodzielną (atomową) jednostką pracy. Wsad może zawierać wiele transakcji, a transakcja może być przekazywana w częściach jako wiele wsadów. Kiedy transakcja jest anulowana lub wycofywana w „połowie drogi”, system SQL Server odwraca częściowe działanie, które miało miejsce od rozpoczęcia transakcji, niezależnie od tego, gdzie wsad się rozpoczął.

Klienckie interfejsy API (Application Programming Interface), takie jak *ADO.NET*, udostępniają metody przekazywania wsadu do systemu SQL Server w celu wykonania kodu. Narzędzia użytkowe systemu SQL Server, takie jak SQL Server Management Studio, *SQLCMD* czy *OSQL*, udostępniają polecenie klienckie *GO*, które sygnalizuje koniec wsadu. Trzeba pamiętać, że *GO* jest poleceniem klienckim, a nie instrukcją języka T-SQL.

Wsad jako jednostka analizy

Wsad jest zbiorem poleceń, które są analizowane i wykonywane jako jedna jednostka. Jeśli analiza wypada pomyślnie, system SQL Server spróbuje wykonać wsad. W przypadku wykrycia błędu składni cały wsad nie jest przekazywany do wykonania przez system SQL Server. Przykładowo poniższy kod zawiera trzy wsady, a w drugim z nich jest błąd składniowy (*FOM* zamiast *FROM* w drugi zapytaniu).

```
-- Prawidłowy wsad
PRINT 'Pierwszy wsad';
USE TSQLV4;
GO
-- Nieprawidłowy wsad
PRINT 'Drugi wsad';
SELECT custid FROM Sales.Customers;
SELECT orderid FOM Sales.Orders;
GO
-- Prawidłowy wsad
PRINT 'Trzeci wsad';
SELECT empid FROM HR.Employees;
```

Ponieważ drugi wsad zawiera błąd składni, cały wsad nie zostaje przekazany do wykonania do systemu SQL Server. Analiza pierwszego i trzeciego wsadu nie wykazuje błędów składni i dlatego wsady te są przekazywane do wykonania. Kod ten generuje następujące wyniki, które pokazują, że cały drugi wsad nie został wykonany.

```
Pierwszy wsad
Msg 102, Level 15, State 1, Line 4
Incorrect syntax near 'Sales'.
(Nieprawidłowa składnia w pobliżu 'Sales')
Trzeci wsad
```

```
empid
-----
2
7
1
5
6
8
3
9
4

(9 row(s) affected)
```

Wsady i zmienne

Zmienna jest lokalna względem wsadu, w którym została zdefiniowana. Jeśli próbujemy odnieść się do zmiennej, która została zdefiniowana w innym wsadzie, wygenerowany zostanie błąd, wskazujący, że zmienna nie została zdefiniowana. Na przykład poniższy kod deklaruje zmienną i drukuje jej zawartość w jednym wsadzie, a następnie próbuje wydrukować jego zawartość w innym wsadzie.

```
DECLARE @i AS INT;
SET @i = 10;
-- Wykonanie powiodło się
PRINT @i;
GO

-- Wykonanie nie powiodło się
PRINT @i;
```

Odwołanie do zmiennej w pierwszej instrukcji *PRINT* jest prawidłowe, ponieważ występuje w tym samym wsadzie, w którym zmienna została zadeklarowana, jednak drugie odniesienie się do zmiennej nie jest poprawne. Z tego względu pierwsza instrukcja *PRINT* zwraca wartość zmiennej (10), natomiast wykonanie drugiej instrukcji się nie udaje. Poniżej pokazano informacje wyjściowe zwracane przez ten kod:

```
10
Msg 137, Level 15, State 2, Line 3
Must declare the scalar variable "@i".

(Musi być zadeklarowana zmienna skalarna "@i")
```

Instrukcje, których nie można łączyć w tym samym wsadzie

Następujące instrukcje nie mogą być łączone z innymi instrukcjami w tym samym wsadzie: *CREATE DEFAULT*, *CREATE FUNCTION*, *CREATE PROCEDURE*, *CREATE RULE*, *CREATE SCHEMA*, *CREATE TRIGGER* i *CREATE VIEW*. Na przykład w poniższym kodzie w tym samym wsadzie umieszczono instrukcję *DROP*, a po niej instrukcję *CREATE VIEW* i dlatego kod ten jest nieprawidłowy.

```
DROP VIEW IF EXIST Sales.MyView;

CREATE VIEW Sales.MyView
AS

SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY YEAR(orderdate);
GO
```

Próba uruchomienia kodu powoduje wygenerowanie następującego komunikatu o błędzie:

```
Msg 111, Level 15, State 1, Line 3
'CREATE VIEW' must be the first statement in a query batch.
('CREATE VIEW' musi być pierwszą instrukcją w partii zapytań.)
```

Aby rozwiązać ten problem, oddzielamy instrukcję *DROP* od *CREATE VIEW*, umieszczając je w różnych wsadach poprzez dodanie polecenia *GO* po instrukcji *DROP*.

Wsad jako jednostka rozpoznawania

Wsad jest też jednostką rozpoznawania, co oznacza, że sprawdzanie istnienia obiektów i kolumn dokonywane jest na poziomie wsadu. Proces ten określany jest też mianem *wiązania* (*binding*). Warto pamiętać o tym podczas projektowania granic wsadu. Jeśli stosujemy do obiektu zmiany schematu i spróbujemy wykonywać operacje na danych w tym samym wsadzie, system SQL Server może jeszcze nie zdawać sobie sprawy z wprowadzonych zmian schematu i instrukcja na danych nie zostanie wykonana ze względu na błąd rozpoznawania. Problem ten ilustruje poniższy przykład. Następnie przedstawię najlepsze praktyki pozwalające unikać tego problemu.

Uruchomimy poniższy kod, by w bieżącej bazie danych utworzyć tabelę nazwaną *T1* z jedną kolumną nazwaną *col1*.

```
DROP TABLE IF EXIST dbo.T1;
CREATE TABLE dbo.T1(col1 INT);
```

Następnie spróbujemy do *T1* dodać kolumnę nazwaną *col2* i w tym samym wsadzie odczytać nową kolumnę.

```
ALTER TABLE dbo.T1 ADD col2 INT;
SELECT col1, col2 FROM dbo.T1;
```

Chociaż wydaje się, że kod jest idealnie bezbłędny, wsad generuje błąd w trakcie fazy rozpoznawania, wyświetlając następujący komunikat:

```
Msg 207, Level 16, State 1, Line 2
Invalid column name 'col2'.
```

(Nieprawidłowa nazwa kolumny 'col2'.)

W fazie rozpoznawania instrukcji *SELECT* tabela *T1* ma tylko jedną kolumnę i odwołanie do kolumny *col2* powoduje błąd. Najlepszym sposobem unikania tego rodzaju problemów jest separacja instrukcji *DDL* i *DML* i umieszczanie ich w oddzielnych wsadach, jak w poniższym kodzie:

```
ALTER TABLE dbo.T1 ADD col2 INT;
GO
SELECT col1, col2 FROM dbo.T1;
```

Opcja *GO n*

Polecenie *GO* tak naprawdę nie jest poleceniem języka T-SQL; jest poleceniem używanym przez narzędzia klienckie systemu SQL Server, takie jak SSMS, do zaznaczenia końca wsadu. Polecenie to obsługuje argument wskazujący, ile razy chcemy wykonać wsad. Aby zilustrować działanie polecenia *GO* z argumentem, najpierw przy użyciu poniższego kodu utworzymy tabelę *T1*:

```
DROP TABLE IF EXIST dbo.T1;
CREATE TABLE dbo.T1(col1 INT IDENTITY);
```

Kolumna *col1* automatycznie uzyskuje wartości na podstawie właściwości *identity*. Zwróćmy uwagę, że kod zadziała równie dobrze, jeśli do wygenerowania wartości użyjemy domyślnego ograniczenia na podstawie obiektu sekwencji. Następnie uruchamiamy poniższy kod, by wyłączyć domyślne dane wyjściowe generowane przez instrukcje *DML*, które wskazują, ile wierszy dotyczyły instrukcje.

```
SET NOCOUNT ON;
```

Na koniec uruchamiamy poniższy kod, by zdefiniować wsad przy użyciu instrukcji *INSERT DEFAULT VALUES* i deklarujemy 100-krotne wykonanie wsadu.

```
INSERT INTO dbo.T1 DEFAULT VALUES;
GO 100

SELECT * FROM dbo.T1;
```

Zapytanie zwraca 100 wierszy o wartościach od 1 do 100 w kolumnie *col1*.

Elementy kontroli przepływu wykonania

Elementy kontroli przepływu wykonania umożliwiają sterowanie działaniem kodu. Język T-SQL udostępnia bardzo podstawowe formy kontroli przepływu w postaci takich elementów jak *IF ... ELSE* oraz *WHILE*.

Element kontroli przepływu *IF ... ELSE*

Element *IF ... ELSE* umożliwia kontrolę wykonania kodu w oparciu o predykat. Specyfikujemy instrukcję lub blok instrukcji, które są wykonywane, jeśli predykat przyjmuje wartość *TRUE* oraz opcjonalnie podajemy instrukcję lub blok instrukcji wykonywanych, jeśli predykat przyjmuje wartość *FALSE* bądź *UNKNOWN*.

Dla przykładu, poniższy kod sprawdza, czy dzień dzisiejszy jest ostatnim dniem roku (inaczej mówiąc, czy rok w dzisiejszej dacie będzie inny niż rok w dacie jutrzejszej). Jeśli warunek ten jest prawdziwy, kod drukuje komunikat informujący, że dzisiaj mamy koniec roku; jeśli warunek nie jest spełniony („else”), kod drukuje komunikat stwierdzający, że dzień dzisiejszy nie jest ostatnim dniem roku.

```
IF YEAR(SYSDATETIME()) <> YEAR DATEADD(day, 1, SYSDATETIME())
    PRINT 'Dzisiejszy dzień to ostatni dzień roku.';
ELSE
    PRINT 'Dzisiejszy dzień nie jest ostatnim dniem roku.';
```

W tym przykładzie użyliśmy instrukcji *PRINT* do zademonstrowania, która część kodu jest wykonywana, a która nie, ale rzecz jasna równie dobrze można użyć innych instrukcji.

Warto pamiętać, że język T-SQL stosuje trójwartościową logikę i dlatego blok *ELSE* jest aktywowany, kiedy predykat ma wartość *FALSE* lub *UNKNOWN*. W przypadkach, kiedy predykat może przyjmować wartości *FALSE* i *UNKNOWN* (na przykład, jeśli używane są znaczniki *NULL*) i potrzebne jest różne traktowanie tych przypadków, musimy zapewnić bezpośrednie sprawdzanie znaczników *NULL* za pomocą predykatu *IS NULL*.

Jeśli przepływ działania wymaga użycia większej liczby przypadków niż dwa, elementy *IF ... ELSE* mogą być zagnieżdżane. Na przykład, poniższy kod w różny sposób obsługuje następujące trzy sytuacje:

1. Dzisiejszy dzień to ostatni dzień roku.
2. Dzisiejszy dzień to ostatni dzień miesiąca, ale nie ostatni dzień roku.
3. Dzisiejszy dzień nie jest ostatnim dniem miesiąca.

```
IF YEAR(SYSDATETIME()) <> YEAR DATEADD(day, 1, SYSDATETIME())
    PRINT 'Dzisiejszy dzień to ostatni dzień roku.';
ELSE
    IF MONTH(SYSDATETIME()) <> MONTH DATEADD(day, 1, SYSDATETIME())
        PRINT 'Dzisiejszy dzień to ostatni dzień miesiąca, ale nie roku.';
    ELSE
        PRINT 'Dzisiejszy dzień nie jest ostatnim dniem miesiąca.';
```


Jeśli trzeba uruchomić kilka instrukcji w sekcjach *IF* lub *ELSE*, trzeba użyć bloku instrukcji. Granice bloku instrukcji oznaczamy za pomocą słów kluczowych *BEGIN* i *END*. Na przykład poniższy kod pokazuje, jak uruchamiać jeden rodzaj procesu, jeśli mamy pierwszy dzień miesiąca, i inny rodzaj procesu, jeśli nie jest to pierwszy dzień miesiąca.

```
IF DAY(SYSDATETIME()) = 1
BEGIN
    PRINT 'Dzisiejszy dzień to pierwszy dzień miesiąca.';
    PRINT 'Uruchomienie procesu dla pierwszego dnia miesiąca.';
    /* ... tu umieszczany jest kod procesu... */
    PRINT 'Zakończony proces bazy danych dla pierwszego dnia miesiąca.';
END
ELSE
BEGIN
    PRINT 'Dzisiejszy dzień nie jest pierwszym dniem miesiąca.';
    PRINT 'Uruchomienie procesu innego niż proces dla pierwszego dnia miesiąca.';
    /* ... tu umieszczany jest kod procesu... */
    PRINT 'Zakończony proces nie-dla-pierwszego dnia miesiąca.';
END
```

Element kontroli przepływu *WHILE*

Język T-SQL udostępnia element *WHILE*, pozwalający na wykonywanie kodu w pętli. Element *WHILE* powtarza wykonanie instrukcji lub bloku instrukcji, dopóki predykat wyspecyfikowany po słowie kluczowym *WHILE* ma wartość *TRUE*. Kiedy predykat uzyska wartość *FALSE* lub *UNKNOWN*, pętla zostaje zakończona.

Język T-SQL nie udostępnia wbudowanego elementu tworzenia pętli, która wykonana jest określoną liczbę powtórzeń, ale bardzo łatwo możemy utworzyć taki element za pomocą pętli *WHILE* i zmiennej. Dla przykładu, poniższy kod pokazuje, jak napisać pętlę, która iterowana jest 10 razy.

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END;
```

Kod deklaruje zmienną całkowitą nazwaną *@i*, która służy jako licznik pętli i inicjuje zmienną za pomocą wartości 1. Następnie kod wprowadza pętlę, która jest iterowana, dopóki zmienna jest mniejsza lub równa 10. W każdej iteracji kod w treści pętli drukuje bieżącą wartość zmiennej *@i* i następnie zwiększa ją o 1. Kod ten zwraca wyniki pokazujące, że pętla została wykonana dziesięciokrotnie.

```
1
2
3
4
```

```

5
6
7
8
9
10

```

Jeśli w pewnym momencie chcemy przerwać bieżącą pętlę i kontynuować instrukcję, która znajduje się po instrukcjach pętli, używamy polecenia *BREAK*. Na przykład poniższy kod wychodzi z pętli, jeśli wartość zmiennej *@i* równa jest 6.

```

DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    IF @i = 6 BREAK;
    PRINT @i;
    SET @i = @i + 1;
END;

```

Kod ten wygeneruje pokazane poniżej wyniki wskazujące na pięciokrotną iterację pętli i przerwanie jej działania na początku szóstej iteracji.

```

1
2
3
4
5

```

Oczywiście kod ten nie jest specjalnie sensowny; jeśli chcemy, by pętla iterowana była tylko pięć razy, po prostu specyfikujemy predykat *@i <= 5*. Kod ten jest jedynie ilustracją użycia polecenia *BREAK*.

Jeśli w pewnym punkcie treści pętli chcemy pominąć resztę działań bieżącej iteracji i sprawdzić ponownie predykat pętli, posługujemy się poleceniem *CONTINUE*. Na przykład poniższy kod pokazuje, jak pominąć w pętli działanie szóstej iteracji w miejscu, gdzie występuje instrukcja *IF* i wykonać do końca pozostałe działania pętli.

```

DECLARE @i AS INT = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    IF @i = 6 CONTINUE;
    PRINT @i;
END;

```

Dane wyjściowe tego kodu pokazują, że, z wyjątkiem szóstej iteracji, dla wszystkich pozostałych wydrukowana została wartość zmiennej *@i*.

```

1
2
3
4
5

```

```
7
8
9
10
```

Jako inny przykład użycia pętli *WHILE*, poniższy kod tworzy tabelę *dbo.Numbers* i wypełnienia ją tysiącem wierszy o wartościach od 1 do 1000 w kolumnie *n*.

```
SET NOCOUNT ON;
DROP TABLE IF EXIST dbo.Numbers;
CREATE TABLE dbo.Numbers(n INT NOT NULL PRIMARY KEY);
GO

DECLARE @i AS INT = 1;
WHILE @i <= 1000
BEGIN
    INSERT INTO dbo.Numbers(n) VALUES(@i);
    SET @i = @i + 1;
END
```

Kursory

W rozdziale 2 „Zapytania do pojedynczej tabeli” wyjaśniłem, że zapytanie bez klauzuli *ORDER BY* zwraca zbiór (lub wielozbiór), natomiast zapytanie zawierające tę klauzulę zwraca konstrukcję, która w standardzie SQL nazywana jest *kursorem* – nierelacyjny wynik z zagwarantowaną kolejnością wierszy. W kontekście materiału omawianego w rozdziale 2, użycie terminu „kursor” dotyczyło tego pojęcia. Standard SQL i język T-SQL obsługuje także obiekt nazywany *kursorem*, który umożliwia przetwarzanie pojedynczych wierszy zbioru wyników zapytania w żądanej kolejności, czyli pozwala na inne działanie, niż w przypadku użycia zapytań bazujących na zbiorach – zwykłych zapytaniach bez kursora, w których operacje wykonywane są na całym zbiorze lub wielozbiorze i nie mogą być zależne od kolejności.

Chciałbym wyraźnie podkreślić, że naszym domyślnym wyborem powinno być stosowanie zapytań opartych na zbiorach; tylko w sytuacji, kiedy istnieją istotne ku temu powody, powinniśmy rozważać użycie kursorów. Zalecenie to wynika z kilku czynników:

1. Po pierwsze i przede wszystkim, stosowania kursorów oznacza działanie niezgodne z modelem relacyjnym, którego podstawą jest teoria zbiorów.
2. Operacje wykonywane przez kursor rekord po rekordzie wnoszą dodatkowy koszt. W porównaniu do operacji wykonywanych na zbiorze, w przypadku działań kursora pojawia się pewien dodatkowy koszt związany z operacjami wykonywanymi na każdym rekordzie. Gdy porównamy zapytanie oparte na zbiorach i kod kursora, które w tle wykonują podobne przetwarzanie fizyczne, zazwyczaj działanie kodu kursora okaże się wielokrotnie wolniejsze, niż działanie kodu wykorzystującego zbiory.

3. W przypadku kursora spora część kodu dotyczy technicznych aspektów rozwiązania – inaczej mówiąc, sposobu przetwarzania danych (deklarowanie kursora, otwieranie kursora, tworzenie pętli poprzez rekordy kursora i zwalnianie kursora). W przypadku rozwiązań bazujących na zbiorze skupiamy się głównie na aspektach logicznych rozwiązania – innymi słowy na tym, *co chcemy uzyskać*, a nie na tym, *w jaki sposób to uzyskujemy*. Z tego względu rozwiązania oparte na kursorach są zwykle dłuższe, mniej czytelne i trudniejsze w utrzymaniu.

Dla większości osób nie jest łatwe nauczenie się myślenia w kategoriach zbiorów. Bardziej intuicyjne są pojęcia związane z kursorami – przetwarzanie tylko jednego rekordu na raz i w określonej kolejności. W rezultacie kursory są często stosowane, a nawet można powiedzieć, że są nadużywane – kursory są używane tam, gdzie istnieją lepsze rozwiązania oparte na zbiorach. Warto poświęcić czas, by nauczyć się myślenia w kategoriach zbiorów. Może to zabrać sporo czasu i wysiłku, czasem nawet trwa latami, ale skoro skorzystamy z języka, który opiera się na modelu relacyjnym, z pewnością jest to właściwy sposób myślenia.

Zakładając, że udało się przekonać Czytelnika, że preferowanym wyborem powinny być rozwiązania oparte na zbiorach, istotne jest również zrozumienie sytuacji wyjątkowych – kiedy powinniśmy stosować kursor. Jednym z takich przykładów jest sytuacja, kiedy trzeba stosować pewne zadanie do każdego wiersza pewnej tabeli czy widoku – może na przykład zachodzić potrzeba wykonania niektórych zadań administracyjnych dla każdego indeksu lub tabeli bazy danych. W takim przypadku sensowne jest użycie kursora do kolejnego iterowania indeksu lub nazw tabel i wykonywania odpowiedniego zadania na każdym elemencie.

Inny przykład, w którym warto rozważyć stosowanie kursora, to sytuacja, kiedy rozwiązanie oparte na zbiorze ma kiepską wydajność i wyczerpaliśmy już wszystkie metody optymalizowania. Jak już wspomniałem wcześniej, rozwiązania oparte na zbiorach zazwyczaj są znacznie szybsze, jednak w niektórych sytuacjach szybsze mogą być rozwiązania z użyciem kursora. Typowym przykładem będzie obliczanie sum kroczących. W rozdziale 7 „Zaawansowane zagadnienia tworzenia zapytań” przedstawiono bardzo sprawne rozwiązanie bazujące na zbiorach wykorzystujące funkcje okna. Jeśli jednak używamy starszej wersji systemu SQL Server, rozwiązanie sum kroczących bazujące na zbiorach nie jest dobrze zoptymalizowane i wiąże się z wielokrotnym skanowaniem danych. Kwestie optymalizacji wykraczają poza tematykę tej książki, tak więc nie będę tutaj szczegółowo rozwijać tego zagadnienia; na tym etapie wystarczy, że będziemy zdawać sobie sprawę, że rozwiązanie z zastosowaniem kursora w tym przypadku wymaga jedynie jednego skanowania danych i dlatego może być szybsze, niż rozwiązania bazujące na zbiorach uruchamiane w wersjach wcześniejszych niż SQL Server 2012.

Korzystanie z kursorów obejmuje następujące etapy ogólne:

1. Zadeklarowanie kursora w oparciu o zapytanie.

2. Otwarcie kursora.
3. Pobranie wartości atrybutów z pierwszego rekordu kursora do zmiennych.
4. Do osiągnięcia końca kursora (dopóki wartość funkcji nazwanej @@FETCH_STATUS wynosi 0), przechodzenie w pętli przez rekordy kursora; w każdej iteracji pętli wartości atrybutu pobierane są z bieżącego rekordu kursora do zmiennych i wykonywane jest przetwarzanie potrzebne dla bieżącego wiersza.
5. Zamknięcie kursora.
6. Zwolnienie zasobów kursora.

Poniższy przykład kodu z użyciem kursora dla każdego klienta i miesiąca oblicza sumę kroczącą ilości towarów na podstawie widoku *Sales.CustOrders*.

```
SET NOCOUNT ON;

DECLARE @Result TABLE
(
    custid      INT,
    ordermonth  DATETIME,
    qty         INT,
    runqty      INT,
    PRIMARY KEY(custid, ordermonth)
);

DECLARE
    @custid      AS INT,
    @prvcustid   AS INT,
    @ordermonth  DATETIME,
    @qty         AS INT,
    @runqty      AS INT;

DECLARE C CURSOR FAST_FORWARD /* tylko do odczytu, tylko do przodu */ FOR
    SELECT custid, ordermonth, qty
    FROM Sales.CustOrders
    ORDER BY custid, ordermonth;

OPEN C;

FETCH NEXT FROM C INTO @custid, @ordermonth, @qty;

SELECT @prvcustid = @custid, @runqty = 0;

WHILE @@FETCH_STATUS = 0
BEGIN
    IF @custid <> @prvcustid
        SELECT @prvcustid = @custid, @runqty = 0;

    SET @runqty = @runqty + @qty;

    INSERT INTO @Result VALUES(@custid, @ordermonth, @qty, @runqty);

    FETCH NEXT FROM C INTO @custid, @ordermonth, @qty;
END

CLOSE C;
```

```

DEALLOCATE C;

SELECT
    custid,
    CONVERT(VARCHAR(7), ordermonth, 121) AS ordermonth,
    qty,
    runqty
FROM @Result
ORDER BY custid, ordermonth;

```

Kod deklaruje kursor w oparciu o zapytanie, które zwraca wiersze z widoku *CustOrders* uporządkowane według identyfikatora klienta i miesiąca zamówienia, po czym po kolei iteruje poszczególne rekordy. Kod zachowuje bieżącą sumę kroczącą ilości w zmiennej nazwanej *@runqty*, która jest resetowana, ilekroć znaleziony zostaje nowy klient. Dla każdego wiersza kod oblicza bieżącą sumę kroczącą poprzez dodawanie bieżącej ilości z danego miesiąca (*@qty*) do zmiennej *@runqty* i wstawia wiersz z identyfikatorem klienta, miesiącem zamówienia, bieżącą miesięczną ilością i skumulowaną ilością do zmiennej tablicowej nazwanej *@Result*. Po przetworzeniu wszystkich rekordów kod odpytuje zmienną tablicową, by zaprezentować skumulowane sumy.

Poniżej przedstawiono wyniki działania tego kodu (pokazane w skróconej postaci):

custid	ordermonth	qty	runqty

1	2015-08	38	38
1	2015-10	41	79
1	2016-01	17	96
1	2016-03	18	114
1	2016-04	60	174
2	2014-09	6	6
2	2015-08	18	24
2	2015-11	10	34
2	2016-03	29	63
3	2014-11	24	24
3	2015-04	30	54
3	2015-05	80	134
3	2015-06	83	217
3	2015-09	102	319
3	2016-01	40	359
...			
90	2015-07	5	5
90	2015-09	15	20
90	2015-10	34	54
90	2016-02	82	136
90	2016-04	12	148
91	2014-12	45	45
91	2015-07	31	76
91	2015-12	28	104
91	2016-02	20	124
91	2016-04	81	205

(636 row(s) affected)

Jak pokazałem w rozdziale 7, język T-SQL obsługuje rozszerzone funkcje okna, które udostępniają tworzenie eleganckich i bardzo sprawnych rozwiązań dla obliczeń kumulowanych, co pozwala uniknąć stosowania kursorów. Poniższy przykład pokazuje, jak można rozwiązać to samo zadanie za pomocą funkcji okna.

```
SELECT custid, ordermonth, qty,  
       SUM(qty) OVER(PARTITION BY custid  
                     ORDER BY ordermonth  
                     ROWS UNBOUNDED PRECEDING) AS runqty  
FROM Sales.CustOrders  
ORDER BY custid, ordermonth;
```

Tabele tymczasowe

Jeśli zachodzi potrzeba chwilowego przechowania danych w tabelach, w niektórych sytuacjach lepiej jest nie używać tabel trwałych. Przyjmijmy, że potrzebne są dane, które są widoczne jedynie w bieżącej sesji lub nawet tylko w bieżącym wsadzie – na przykład, trzeba przechować tymczasowe dane podczas przetwarzania danych, jak w przykładzie kodu kursora z poprzedniego podrozdziału. Innym przykładem może być sytuacja, gdy użytkownik nie ma uprawnień do tworzenia tabel w bieżącej bazie danych.

System SQL Server obsługuje trzy rodzaje tabel tymczasowych, które mogą być czasem wygodniejsze w użyciu niż tabele trwałe: lokalne tabele tymczasowe, globalne tabele tymczasowe i zmienne tablicowe. W kolejnych podrozdziałach opisano na przykładach sposób użycia tych tabel.

Lokalne tabele tymczasowe

Lokalną tabelę tymczasową tworzymy nadając jej nazwę z prefiksem w postaci znaku hash (#), na przykład *#T1*. Trzeba tu podkreślić, że wszystkie trzy rodzaje tabel tymczasowych są tworzone w bazie danych *tempdb* – określenie „lokalny” nie odnosi się do miejsca przechowywania tabeli, ale jej zasięgu.

Lokalna tabela tymczasowa jest widoczna jedynie w sesji, w której została utworzona, na poziomie tworzącym tabelę i na wszystkich poziomach podrzędnych w stosie wywołania (wewnętrzne procedury, funkcje, wyzwalacze i wsady dynamiczne). Lokalna tabela tymczasowa jest usuwana automatycznie przez system SQL Server, gdy poziom tworzący w stosie wywołania wychodzi poza zakres. Spróbuję to wyjaśnić na przykładzie. Załóżmy, że procedura składowana nazwana *Proc1* wywołuje procedurę nazwaną *Proc2*, która z kolei wywołuje procedurę *Proc3*, a ta wywołuje procedurę nazwaną *Proc4*. Procedura *Proc2* tworzy tabelę tymczasową *#T1* przed wywołaniem procedury *Proc3*. Tabela *#T1* jest widoczna dla procedury *Proc2*, *Proc3* i *Proc4*, ale nie dla procedury *Proc1* oraz zostanie automatycznie usunięta, gdy zakończy się procedura *Proc2*. Jeśli tabela tymczasowa zostaje utworzona we wsadzie na najbardziej

zewnętrznym poziomie zagnieżdżenia sesji (inaczej mówiąc, jeśli wartość funkcji @@NESTLEVEL wynosi 0), jest także widoczna dla wszystkich kolejnych wsadów i jest automatycznie usuwana przez SQL Server dopiero wtedy, gdy nastąpi rozłączenie sesji tworzącej tabelę.

Może się pojawić pytanie, jak SQL Server zapobiega konfliktom nazw, gdy dwie sesje tworzą lokalne tabele tymczasowe o tej samej nazwie. SQL Server dodaje wewnętrznie do nazwy tabeli sufiks, który powoduje, że nazwa jest jednoznaczna w bazie danych *tempdb*. Programiści nie muszą się tym zajmować – do tabeli odwołujemy się przy użyciu wprowadzonej nazwy bez wewnętrznego sufiksu i tylko nasza sesja będzie miała dostęp do tej tabeli.

Jedną z oczywistych sytuacji, w jakiej lokalne tabele tymczasowe są przydatne, jest proces, w którym trzeba chwilowo przechowywać wyniki pośrednie – jak w trakcie działania pętli – a później odczytywać dane.

Inną sytuacją sensownego wykorzystania tabel tymczasowych jest potrzeba wielokrotnego uzyskiwania dostępu do wyników, które są rezultatem kosztownego przetwarzania. Załóżmy na przykład, że trzeba złączyć tabelę *Sales.Orders* i *Sales.OrderDetails*, posumować ilości dla poszczególnych zamówień według roku zamówienia i złączyć dwie instancje posumowanych danych, by porównać łączną ilość dla każdego roku z wartością w roku poprzednim. Tabele *Orders* i *OrderDetails* w testowej bazie danych są nieduże, ale w rzeczywistych zastosowaniach tabele takie mogą zawierać miliony wierszy. Jedną z możliwości jest użycie wyrażeń tablicowych, ale pamiętamy, że wyrażenia tablicowe są wirtualne. Duży nakład pracy związany ze skanowaniem wszystkich danych, złączenie tabel *Orders* i *OrderDetails* oraz posumowanie danych przy użyciu wyrażeń tablicowych będzie musiał zostać wykonany dwukrotnie. Zamiast tego, lepiej będzie wykonać tę pracę tylko raz – przechowując wyniki w lokalnej tabeli tymczasowej – a następnie złączyć dwie instancje tabeli tymczasowej, zwłaszcza że wynik tych kosztownych obliczeń to bardzo mały zbiór – tylko jeden wiersz dla każdego roku zamówienia.

Poniższy kod ilustruje ten scenariusz użycia lokalnej tabeli tymczasowej.

```
IF OBJECT_ID('tempdb.dbo.#MyOrderTotalsByYear') IS NOT NULL
    DROP TABLE dbo.#MyOrderTotalsByYear;
GO

CREATE TABLE #MyOrderTotalsByYear
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty       INT NOT NULL
);

INSERT INTO #MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(0.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
JOIN Sales.OrderDetails AS OD
```



```

        ON OD.orderid = O.orderid
    GROUP BY YEAR(orderdate);

SELECT Cur.orderyear, Cur.qty AS curyearqty, Prv.qty AS prvyearqty
FROM dbo.#MyOrderTotalsByYear AS Cur
    LEFT OUTER JOIN dbo.#MyOrderTotalsByYear AS Prv
        ON Cur.orderyear = Prv.orderyear + 1;

```

Kod ten generuje następujące wyniki:

orderyear	curyearqty	prvyearqty
2015	25489	9581
2016	16247	25489
2014	9581	NULL

W celu zweryfikowania tego, że lokalna tabela tymczasowa jest widoczna tylko w sesji, która ją utworzyła, spróbujemy uzyskać dostęp do niej z innej sesji.

```
SELECT orderyear, qty FROM dbo.#MyOrderTotalsByYear;
```

Wyświetlony zostaje następujący komunikat o błędzie:

```

Msg 208, Level 16, State 0, Line 1
Invalid object name '#MyOrderTotalsByYear'.

(Nieprawidłowa nazwa obiektu '#MyOrderTotalsByYear'.)

```

Po ukończeniu wracamy do pierwotnej sesji i usuwamy tabelę tymczasową.

```

IF OBJECT_ID('tempdb.dbo.#MyOrderTotalsByYear') IS NOT NULL
    DROP TABLE dbo.#MyOrderTotalsByYear;

```

Czyszczenie zasobów zaraz po zakończeniu ich wykorzystywania jest zaleceniem ogólnym programowania.

Globalne tabele tymczasowe

UWAGA W czasie opracowywania tej książki globalne tabele tymczasowe nie były obsługiwane przez Azure SQL Database, tak więc, jeśli chcemy uruchamiać przykłady kodu z tego podrozdziału, trzeba korzystać z instancji SQL Server dla siedziby.



Globalna tabela tymczasowa jest widoczna we wszystkich sesjach. Globalne tabele tymczasowe są usuwane automatycznie przez system SQL Server, kiedy sesja tworząca tabelę rozłącza się i nie istnieją do niej żadne aktywne odniesienia. Globalną tabelę tymczasową stworzymy nadając jej nazwę przy użyciu prefiksu w postaci dwóch znaków hash (#), jak na przykład ##T1.

Globalne tabele tymczasowe są przydatne, kiedy chcemy współużytkować dane z innymi użytkownikami. Nie są wymagane żadne specjalne uprawnienia i każdy uzyskuje pełny dostęp zarówno dla poziomu DDL, jak i DML. Oczywiście fakt, że każdy

ma pełny dostęp, oznacza też, że każdy może zmienić lub nawet usunąć tabelę, tak więc trzeba uważnie analizować rozwiązania alternatywne.

Dla przykładu, poniższy kod tworzy globalną tabelę tymczasową nazwaną `##Globals` z kolumnami nazwanymi `id` i `val`.

```
CREATE TABLE dbo.##Globals
(
    id sysname NOT NULL PRIMARY KEY,
    val SQL_VARIANT NOT NULL
);
```

Zadaniem tabeli w tym przykładzie jest imitowanie zmiennych globalnych, które nie są obsługiwane w systemie SQL Server. Typem danych kolumny `id` jest `sysname` (typ używany przez SQL Server wewnętrznie do reprezentowania identyfikatorów), a typem danych kolumny `val` jest `SQL_VARIANT` (typ ogólny umożliwiający przechowywanie wartości prawie wszystkich typów podstawowych).

Do tabeli każdy może wstawiać wiersze. Na przykład uruchomimy poniższy kod, by wstawić wiersz reprezentujący zmienną nazwaną `i` oraz zainicjujemy ją przy użyciu wartości całkowitej 10.

```
INSERT INTO dbo.##Globals(id, val) VALUES(N'i', CAST(10 AS INT));
```

Każdy użytkownik może modyfikować i pobierać dane z tabeli. Dla przykładu możemy w dowolnej sesji wywołać poniższy kod, by odczytać bieżącą wartość zmiennej `i`.

```
SELECT val FROM dbo.##Globals WHERE id = N'i';
```

Kod generuje następujący wynik:

```
val
-----
10
```



UWAGA Warto pamiętać, że system SQL Server automatycznie usunie tabelę zaraz po rozłączeniu sesji tworzącej globalną tabelę tymczasową, o ile nie istnieją żadne aktywne odniesienia do tej tabeli.

Jeśli chcemy, by globalna tabela tymczasowa była tworzona, ilekroć uruchamiany jest system SQL Server i by SQL Server nie usuwał jej automatycznie, trzeba utworzyć tabelę poprzez procedurę składowaną, która jest oznaczona jako procedura startowa (informacje szczegółowe na ten temat znaleźć można wyszukując hasło „`sp_proccop`tion” w dokumentacji SQL Server Books Online pod adresem URL: <http://msdn.microsoft.com/en-us/library/ms181720.aspx>).

Zmienne tablicowe

Pod pewnymi względami zmienne tablicowe są podobne do lokalnych tabel tymczasowych, a pod pewnymi nie. Zmienne tablicowe deklarujemy podobnie jak inne zmienne – przy użyciu instrukcji *DECLARE*.

Podobnie jak w przypadku lokalnych tabel tymczasowych, zmienne tablicowe mają fizyczną prezentację w postaci tabeli w bazie danych *tempdb*, wbrew błędnemu, choć rozpowszechnionemu pogładowi, że zmienne istnieją tylko w pamięci. Również podobnie do lokalnych tabel tymczasowych zmienne tablicowe są widoczne tylko dla sesji, która je utworzyła, ale mają bardziej ograniczony zakres: jest to jedynie bieżący wsad. Zmienne tablicowe nie są widoczne we wsadach podrzędnych na stosie wywołań ani w następnych wsadach sesji.

Jeśli jawnie zdefiniowana transakcja jest wycofywana, zmiany w tabelach tymczasowych w tej transakcji są również wycofywane; natomiast zmiany wykonane w zmiennych tablicowych przez instrukcje ukończone w transakcji nie są odwracane. Odwracane są jedynie zmiany wprowadzone przez aktywne polecenie, którego wykonanie się nie powiodło lub które zostało przerwane przed zakończeniem.

Tabele tymczasowe i zmienne tablicowe różnią się także pod względem optymalizacji, ale ta tematyka wykracza poza zakres książki. W tym momencie wystarczy stwierdzić tylko tyle, że z punktu widzenia wydajności zazwyczaj sensowne jest używanie zmiennych tablicowych w sytuacjach, gdy mają one pomieścić niewiele danych (tylko kilka wierszy), a w pozostałych przypadkach lepiej stosować tabele tymczasowe.

Przykładowo poniższy kod wykorzystuje zmienną tablicową zamiast lokalnej tabeli tymczasowej, by porównać łączną ilość towarów w zamówieniach dla każdego roku z ilością w roku poprzednim.

```
DECLARE @MyOrderTotalsByYear TABLE
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty       INT NOT NULL
);

INSERT INTO @MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(0.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);

SELECT Cur.orderyear, Cur.qty AS curyearqty, Prv.qty AS prvyearqty
FROM @MyOrderTotalsByYear AS Cur
    LEFT OUTER JOIN @MyOrderTotalsByYear AS Prv
        ON Cur.orderyear = Prv.orderyear + 1;
```

Kod ten zwraca następujące wyniki:

orderyear	curyearqty	prvyearqty
2014	9581	NULL
2015	25489	9581
2016	16247	25489

Warto zauważyć, że zamiast stosowania zmiennej tablicowej lub tabeli tymczasowej i samozłączenia bardziej efektywną metodą realizacji tego zadania będzie użycie funkcji *LAG*, jak w poniższym przykładzie kodu:

```
DECLARE @MyOrderTotalsByYear TABLE
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty       INT NOT NULL
);

INSERT INTO @MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);

SELECT orderyear, qty AS curyearqty,
    LAG(qty) OVER(ORDER BY orderyear) AS prvyearqty
FROM @MyOrderTotalsByYear;
```

Typy tablicowe

Język T-SQL udostępnia również typy tablicowe. Umożliwiają one przechowanie definicji (schematu) tabeli w bazie danych; można jej następnie użyć jako definicji zmiennych tablicowych lub parametrów wejściowych procedur składowanych i funkcji definiowanych przez użytkownika.

Na przykład poniższy kod tworzy typ tablicowy nazwany *dbo.OrderTotalsByYear* w bieżącej bazie danych.

```
DROP TYPE IF EXIST dbo.OrderTotalsByYear;

CREATE TYPE dbo.OrderTotalsByYear AS TABLE
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty       INT NOT NULL
);
```

Po utworzeniu typu tablicowego, ilekroć zachodzi potrzeba zadeklarowania zmiennej tablicowej w oparciu o definicję typu tablicowego, nie musimy powtarzać tego kodu – wystarczy po prostu podać *dbo.OrderTotalsByYear* jako typ zmiennej, jak w poniższym przykładzie:

```
DECLARE @MyOrderTotalsByYear AS dbo.OrderTotalsByYear;
```

Jako bardziej pełny przykład, poniższy kod deklaruje zmienną nazwaną `@MyOrderTotalsByYear` nowego typu tablicowego, wykonuje zapytanie do tabel `Orders` i `OrderDetails`, by obliczyć łączną ilość towarów w zamówieniach dla poszczególnych lat, przechowuje wynik w zmiennej tablicowej i odczytuje tę zmienną, by zaprezentować jej zawartość.

```
DECLARE @MyOrderTotalsByYear AS dbo.OrderTotalsByYear;

INSERT INTO @MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);

SELECT orderyear, qty FROM @MyOrderTotalsByYear;
```

Kod ten zwraca następujące wyniki:

orderyear	qty
2014	9581
2015	25489
2016	16247

Możliwość stosowania typów tablicowych to nie tylko korzyść w postaci skrócenia kodu. Jak już wspomniałem, można ich używać jako typ parametrów wejściowych procedur składowanych i funkcji, co jest wyjątkowo przydatną cechą.

Dynamiczny kod SQL

SQL Server umożliwia konstruowanie kodu języka T-SQL jako ciągu znaków, a następnie wykonanie tego kodu jako wsadu. Cecha ta nazywana jest *dynamycznym kodem SQL* (*dynamic SQL*). SQL Server udostępnia dwie metody wykonywania dynamicznego kodu SQL: przy użyciu polecenia `EXEC` (skrót od `EXECUTE`) oraz przy użyciu procedury składowanej `sp_executesql`. W dalszej części podrozdziału wyjaśnię różnice pomiędzy nimi i przykłady zastosowania obu metod.

Dynamiczny kod SQL jest przydatny w następujących sytuacjach:

- **Automatyzacja zadań administracyjnych** Przykładem może być odczytywanie metadanych, a następnie konstruowanie i wykonywanie instrukcji `BACKUP DATABASE` dla każdej bazy danych w instancji.
- **Poprawa wydajności niektórych zadań** Na przykład konstruowanie sparametryzowanych zapytań ad-hoc, które mogą ponownie użyć poprzednio zbuforowanych planów wykonania (więcej informacji na ten temat w dalszej części rozdziału).

- **Konstruowanie elementów kodu na podstawie faktycznych danych** Na przykład dynamiczne konstruowanie zapytania *PIVOT*, jeśli nie wiemy wcześniej, jakie elementy powinny pojawić się w klauzuli *IN* operatora *PIVOT*.



UWAGA Trzeba szczególnie uważać przy włączaniu informacji wprowadzanych przez użytkownika jako części naszego kodu. Hackerzy mogą podejmować próby wprowadzania kodu, którego wcale nie chcielibyśmy uruchamiać – tego typu działania nazywamy iniekcją (wstrzyknięciem) SQL. Najlepszym środkiem przeciwdziałania iniekcjom SQL jest unikanie bezpośredniego włączania informacji wprowadzanych przez użytkownika jako części naszego kodu (na przykład przy użyciu parametrów). Jeśli składamy informacje wejściowe użytkownika jako część naszego kodu, trzeba rzetelnie przeprowadzić inspekcję wejścia i wykrywać próby iniekcji SQL. Bardzo dobry artykuł opisujący te zagadnienia znaleźć można w dokumentacji SQL Server Books Online pod hasłem „SQL Injection”.

Polecenie *EXEC*

Polecenie *EXEC* akceptuje jako wejście ciąg znaków umieszczony w nawiasach jako wejście i wykonuje wsad kodu znajdujący się wewnątrz tego ciągu znaków. Polecenie *EXEC* jako wejście obsługuje zarówno zwykłe ciągi znaków, jak i ciągi Unicode.

W poniższym przykładzie w zmiennej *@sql* przechowywany jest ciąg znaków z instrukcją *PRINT*, a następnie polecenie *EXEC* używane jest do wywołania wsadu kodu przechowywanego w tej zmiennej.

```
DECLARE @sql AS VARCHAR(100);
SET @sql = 'PRINT ''Ten komunikat został wydrukowany za pomocą dynamicznego
wsadu SQL.''';
EXEC(@sql);
```

Zwróćmy uwagę na użycie dwóch pojedynczych znaków cudzysłowu do reprezentowania jednego pojedynczego znaku cudzysłowu wewnątrz ciągu. Kod ten zwraca następujący wynik.

Ten komunikat został wydrukowany za pomocą dynamicznego wsadu SQL.

Procedura składowana *sp_executesql*

Procedura składowana *sp_executesql* jest alternatywą dla polecenia *EXEC*. Jest to bezpieczniejsza i bardziej elastyczna konstrukcja, ponieważ posiada interfejs – czyli obsługuje parametry wejściowe i wyjściowe. Inaczej niż w przypadku polecenia *EXEC*, jako wsad wejściowy kodu procedura *sp_executesql* obsługuje tylko ciągi znaków Unicode.

Możliwość używania parametrów wejściowych i wyjściowych w dynamicznym kodzie SQL ułatwia napisanie bezpieczniejszego i efektywniejszego kodu. Z punktu widzenia bezpieczeństwa, parametry pojawiające się w kodzie nie mogą być

traktowane jako część kodu do wykonania – mogą być traktowane wyłącznie jako operandy wyrażeń. Tak więc przy użyciu parametrów możemy zmniejszyć podatność kodu na iniekcje SQL.

Procedura składowana *sp_executesql* zapewnia zazwyczaj większą efektywność działania niż polecenie *EXEC*, ponieważ parametryzacja pomaga w wielokrotnym wykorzystywaniu buforowanych planów wykonywania. Plan wykonywania jest to plan fizycznego przetwarzania tworzony dla przez SQL Server, stanowiący zbiór instrukcji opisujących, do jakich obiektów trzeba uzyskać dostęp, w jakiej kolejności, jakich użyć indeksów, przy użyciu jakiej metody uzyskiwania dostępu, rodzaj używanych algorytmów złączenia itd. Upraszczając, jednym z wymagań dla ponownego użycia poprzednio skompilowanego i zbuforowanego planu jest to, by ciąg zapytania (kod) był taki sam jak ten, dla którego w buforze istnieje plan. Najlepszym sposobem sprawnego ponownego wykorzystywania planów wykonywania jest stosowanie procedur składowanych z parametrami. W ten sposób, nawet jeśli zmienia się wartości parametrów, ciąg (kod) zapytania pozostaje ten sam. Jeśli jednak z jakiegoś powodu decydujemy się doraźnie użyć kodu ad hoc zamiast procedury składowanej, w przypadku użycia procedury składowanej *sp_executesql* możemy nadal korzystać z parametrów i w ten sposób zwiększać szanse na ponowne użycie planu.

Procedura *sp_executesql* ma dwa parametry wejściowe oraz sekcję przypisań. W pierwszym parametrze nazywanym *@stmt* specyfikujemy ciąg znaków Unicode, zawierający wsad kodu do uruchomienia. Ciąg znaków Unicode przechowujący deklaracje parametrów wejściowych i wyjściowych dostarczamy w drugim parametrze nazywanym *@params*. Następnie specyfikujemy przypisania parametrów wejściowych i wyjściowych oddzielane przecinkami.

Poniższy przykład konstruuje wsad kodu z zapytaniem do tabeli *Sales.Orders*. W przykładzie tym użyty został parametr wejściowy o nazwie *@orderid*, użyty w filtrze zapytania.

```
DECLARE @sql AS NVARCHAR(100);

SET @sql = N'SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = @orderid;';

EXEC sp_executesql
    @stmt = @sql,
    @params = N'@orderid AS INT',
    @orderid = 10248;
```

Kod ten generuje następujący wynik:

orderid	custid	empid	orderdate
10248	85	5	2014-07-04 00:00:00.000

W tym przykładzie przypisaliśmy wartość 10248 do parametru wejściowego, ale jeśli nawet uruchomimy kod ponownie przy użyciu innej wartości, ciąg zapytania pozostanie ten sam. W ten sposób zwiększamy szanse na ponowne użycie poprzednio utworzonego planu.

PIVOT w dynamicznym kodzie SQL

Podrozdział ten zawiera tematy zaawansowane i opcjonalne i jest on przewidziany dla Czytelników, którzy bardzo dobrze opanowali techniki przestawiania danych i zagadnienia dynamicznego kodu SQL. W rozdziale 7 pokazałem stosowanie operatora *PIVOT* do przestawiania danych. Jak pamiętamy, w przypadku statycznego zapytania musimy wiedzieć wcześniej, które wartości wyspecyfikować w klauzuli *IN* operatora *PIVOT*. Poniższy kod jest przykładem zapytania statycznego z użyciem operatora *PIVOT*.

```
SELECT *
FROM (SELECT shipperid, YEAR(orderdate) AS orderyear, freight
      FROM Sales.Orders) AS D
      PIVOT(SUM(freight) FOR orderyear IN([2014],[2015],[2016])) AS P;
```

Kod ten wykonuje zapytanie do tabeli *Sales.Orders* i przestawia dane tak, by identyfikatory dostawców zwracane były w wierszach, lata zamówień w kolumnach, a łączna ilość znajdowała się na przecięciu wiersza dostawcy z rokiem zamówienia. Kod ten generuje następujące wyniki:

shipperid	2014	2015	2016
3	4233.78	11413.35	4865.38
1	2297.42	8681.38	5206.53
2	3748.67	12374.04	12122.14

W przypadku zapytania statycznego musimy wcześniej wiedzieć, jakie wartości (w tym przypadku lata zamówień) wyspecyfikować w klauzuli *IN* operatora *PIVOT*. Oznacza to, że co roku trzeba poprawiać kod. Zamiast tego możemy wykonać zapytanie w celu pobrania lat z danych, skonstruować dynamiczny kodu SQL na podstawie odczytanych lat i wykonać dynamiczny wsad SQL, jak poniżej:

```
DECLARE
    @sql          AS NVARCHAR(1000),
    @orderyear AS INT,
    @first       AS INT;

DECLARE C CURSOR FAST_FORWARD FOR
    SELECT DISTINCT(YEAR(orderdate)) AS orderyear
    FROM Sales.Orders
    ORDER BY orderyear;

SET @first = 1;

SET @sql = N'SELECT *
FROM (SELECT shipperid, YEAR(orderdate) AS orderyear, freight
```



```

        FROM Sales.Orders) AS D
    PIVOT(SUM(freight) FOR orderyear IN(');

OPEN C;

FETCH NEXT FROM C INTO @orderyear;

WHILE @@fetch_status = 0
BEGIN
    IF @first = 0
        SET @sql = @sql + N', '
    ELSE
        SET @first = 0;

    SET @sql = @sql + QUOTENAME(@orderyear);

    FETCH NEXT FROM C INTO @orderyear;
END

CLOSE C;

DEALLOCATE C;

SET @sql = @sql + N')) AS P;';

EXEC sp_executesql @stmt = @sql;

```

UWAGA Istnieją bardziej efektywne metody konkatencji ciągów niż użycie kursora, na przykład użycie agregacji CLR (Common Language Runtime) i opcji *FOR XML PATH*, ale są to bardziej zaawansowane metody, których opis wykracza poza zakres tej książki.



Procedury

Procedury to programowalne obiekty, które opakowują kod wykonujący jakieś obliczenia, zwracający pewne wyniki lub realizujący inne działania. System SQL Server obsługuje trzy rodzaje procedur: funkcje zdefiniowane przez użytkownika, procedury składowane i wyzwalacze.

W SQL Server mamy do dyspozycji wybór pomiędzy opracowaniem procedury w języku T-SQL a zaprogramowaniem jej przy użyciu kodu Microsoft .NET, wykorzystującego integrację CLR w produkcie. Ponieważ ta książka omawia język T-SQL, przytoczone przykłady będą dotyczyć użycia T-SQL. Mówiąc ogólnie, jeśli realizowane zadanie dotyczy głównie operacji na danych, język T-SQL jest zazwyczaj lepszą opcją. Natomiast kod .NET jest zazwyczaj lepszym wyborem, jeśli zadanie bardziej wiąże się z konstruowaniem logiki iteracyjnej, operacji na ciągach czy intensywnych obliczeniach.

Funkcje definiowane przez użytkownika

Zadanie funkcji definiowanych przez użytkownika (UDF) to opakowanie logiki, która wykonuje pewne obliczenia, z możliwością wykorzystywania parametrów wejściowych oraz zwrócenie wyników.

System SQL Server obsługuje skalarne funkcje UDF oraz funkcje UDF zwracające wartości w postaci tabeli (funkcje tablicowe). Pierwsze (skalarne) funkcje UDF zwracają pojedynczą wartość, drugie zwracają tabelę. Jedną z korzyści stosowania funkcji UDF polega na możliwości ich wykorzystywania w zapytaniach. Skalarne funkcje UDF mogą występować w dowolnym miejscu zapytania, w którym może występować wyrażenie zwracające pojedynczą wartość (na przykład lista *SELECT*). Tablicowe funkcje UDF występują w klauzuli *FROM* zapytania. W tym podrozdziale przedstawię przykład skalarnej funkcji UDF.

Funkcje UDF nie mogą dawać żadnych „efektów ubocznych”. Oznacza to oczywiście, że funkcje UDF nie mogą dokonywać żadnych zmian schematu ani modyfikować danych. Jednak mniej oczywiste jest to, że są również inne działania powodujące efekty uboczne. Przykładem mogą być skutki powstające przy wywołaniu funkcji *RAND* zwracającej przypadkową wartość lub funkcji *NEWID* tworzącej globalnie unikalny identyfikator (*GUID*). Ilekroć wywołujemy funkcję *RAND* bez wyspecyfikowania ziarna (ang. *seed*), SQL Server generuje przypadkową wartość ziarna, która opiera się na poprzednim wywołaniu funkcji *RAND*. Z tego powodu SQL Server musi przechowywać wewnętrznie informacje o każdym wywoływaniu funkcji *RAND*. Podobnie każde wywołanie funkcji *NEWID* powoduje, że system musi ustawić pewne informacje, które są uwzględniane przy następnym wywołaniu funkcji *NEWID*. Ponieważ funkcje *RAND* i *NEWID* dają efekty uboczne, nie możemy ich stosować w funkcjach UDF.

Dla przykładu, poniższy kod tworzy funkcję UDF nazwaną *dbo.GetAge*, która zwraca wiek osoby o wyspecyfikowanej dacie urodzenia (argument *@birthdate*) dla podanej daty zdarzenia (argument *@eventdate*).

```

DROP FUNCTION IF EXIST dbo.GetAge;
GO

CREATE FUNCTION dbo.GetAge
(
    @birthdate AS DATE,
    @eventdate AS DATE
)
RETURNS INT
AS
BEGIN
    RETURN
        DATEDIFF(year, @birthdate, @eventdate)
        - CASE WHEN 100 * MONTH(@eventdate) + DAY(@eventdate)
                < 100 * MONTH(@birthdate) + DAY(@birthdate)
              THEN 1 ELSE 0
        END;

```

```
END;
GO
```

Funkcja oblicza wiek jako różnicę lat pomiędzy rokiem urodzenia a rokiem zdarzenia minus 1 rok w przypadkach, dla których miesiąc i dzień zdarzenia są mniejsze, niż miesiąc i dzień daty urodzenia. Wyrażenie $100 * month + day$ to sztuczka pozwalająca na połączenie miesiąca i daty w celu prostszego porównania. Na przykład, dla dwunastego dnia lutego wyrażenie generuje liczbę całkowitą 212.

Zwróćmy uwagę, że funkcja może zawierać więcej, niż tylko klauzulę *RETURN*. Może obejmować złożony kod z elementami przepływu, obliczenia i inne elementy. Funkcja musi jednak zawierać klauzulę *RETURN*, która zwraca wartość.

W celu zilustrowania użycia funkcji UDF w zapytaniu posłużymy się poniższym kodem, który wykonuje zapytanie do tabeli *HR.Employees* i wywołuje funkcję *GetAge* na liście *SELECT* do obliczenia wieku każdego pracownika w dniu dzisiejszym.

```
SELECT
    empid, firstname, lastname, birthdate,
    dbo.GetAge(birthdate, SYSDATETIME()) AS age
FROM HR.Employees;
```

Jeśli przykładowo uruchomimy to zapytanie 12 lutego 2012, uzyskamy następujące wyniki:

empid	firstname	lastname	birthdate	age
1	Sara	Davis	1968-12-08	47
2	Don	Funk	1972-02-19	43
3	Judy	Lew	1983-08-30	32
4	Yael	Peled	1957-09-19	58
5	Sven	Mortensen	1975-03-04	40
6	Paul	Suurs	1983-07-02	32
7	Russell	King	1980-05-29	35
8	Maria	Cameron	1978-01-09	38
9	Patricia	Doyle	1986-01-27	30

(9 row(s) affected)

Warto zauważyć, że przy wykonaniu zapytania w innym systemie, wartości prezentowane w kolumnie *age* będą zależne od dnia uruchomienia (daty).

Procedury składowane

Procedury składowane są programami przechowywanymi po stronie serwera, zawierającymi kod T-SQL. Procedury składowane mogą stosować parametry wejściowe i wyjściowe, potrafią zwracać zbiory wyników zapytań i mogą wywoływać kod powodujący różne efekty uboczne. Procedury składowane mogą modyfikować dane, ale także można za ich pośrednictwem wprowadzać zmiany schematów.

W porównaniu do kodu ad hoc, stosowanie procedur składowanych ma wiele zalet:

- **Hermetyzacja logiki** Jeśli zachodzi potrzeba zmiany implementacji procedury składowanej, możemy ją zmodyfikować w jednym miejscu bazy danych, a procedura zostanie zmieniona dla wszystkich jej użytkowników.
- **Lepsza kontrola zabezpieczeń** Użytkownikom możemy przydzielać uprawnienia do wykonywania procedury bez przyznawania bezpośredniego uprawnienia do wykonywania bazowej czynności. Załóżmy na przykład, że chcemy zezwolić niektórym użytkownikom na usuwanie klientów bazy danych, ale nie chcemy przydzielić im bezpośredniego uprawnienia do usuwania wierszy w tabeli *Customers*. Tak więc chcemy zapewnić, że żądania usunięcia klienta będą zatwierdzane – na przykład poprzez sprawdzenie, czy klient ma otwarte zamówienia lub niespłacony dług – a także może zachodzić potrzeba przeprowadzania inspekcji żądań. Poprzez nieprzydzielenie bezpośrednich uprawnień usuwania wierszy z tabeli *Customers*, a zamiast tego przydzielenie uprawnień do wykonywania procedury, która obsługuje to zadanie, zapewniamy, że mają miejsce wszystkie wymagane sprawdzenia poprawności i inspekcji. Ponadto procedury składowane ułatwiają unikanie iniekcji kodu SQL, w szczególności, kiedy procedura zastępuje parametrami kod ad hoc przekazywany z aplikacji klienckiej.
- **Wewnątrz procedury można umieścić cały kod obsługi błędów, która w tle podejmuje w odpowiednim momencie działania naprawcze** Obsługa błędów została omówiona w dalszej części rozdziału.
- **Procedury składowane poprawiają wydajność** Wcześniej wspomniałem o możliwości ponownego użycia wcześniej skompilowanych i zbuforowanych planów wykonywania. Zapytania w procedurach składowanych są zwykle sparаметryzowane, dzięki czemu wyższe jest prawdopodobieństwo ponownego wykorzystania planów wykonywania. Inną zaletą używania procedur składowanych z punktu widzenia wydajności jest zmniejszenie ruchu sieciowego. Aplikacja kliencka musi przekazać tylko nazwę procedury i jej argumenty. Serwer przetwarza cały kod procedury i do wywołującego zwraca jedynie dane wyjściowe. Na etapach pośrednich nie ma więc żadnego ruchu związanego z przesyłaniem danych tam i z powrotem.

Jako prosty przykład, poniższy kod tworzy procedurę składowaną nazwaną *Sales.GetCustomerOrders*. Procedura jako argumenty wejściowe akceptuje identyfikator klienta (*@custid*) i zakres dat (*@fromdate* i *@todate*). Jako zbiór wyników zwraca wiersze z tabeli *Sales.Orders* reprezentujące zamówienia złożone przez żądanego klienta w żądanym zakresie dat oraz liczbę zaangażowanych wierszy jako parametr wyjściowy (*@numrows*).

```
DROP PROC IF EXIST Sales.GetCustomerOrders;
GO
```

```
CREATE PROC Sales.GetCustomerOrders
    @custid AS INT,
```

```

    @fromdate AS DATETIME = '19000101',
    @todate AS DATETIME = '99991231',
    @numrows AS INT OUTPUT
AS
SET NOCOUNT ON;

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid
    AND orderdate >= @fromdate
    AND orderdate < @todate;

SET @numrows = @@rowcount;
GO

```

Podczas wykonywania procedury, jeśli nie podamy wartości parametru *@fromdate* (data początkowa), procedura użyje wartości domyślnej *19000101*, a jeśli nie wyspecyfikujemy wartości parametru *@todate* (data końcowa), użyje wartości domyślnej *99991231*. Słowo kluczowe *OUTPUT* służy do wskazania, że *@numrows* to parametr wyjściowy. Polecenie *SET NOCOUNT ON* zostało użyte do wyeliminowania komunikatu wskazującego, ile wierszy dotyczyły instrukcje DML wewnątrz procedury, takie jak instrukcja *SELECT*.

Poniżej pokazano przykład wykonania tej procedury, żądającego informacji o zamówieniach złożonych przez klienta o identyfikatorze 1 w roku 2015. Kod przypisuje wartość parametru wyjściowego *@numrows* do zmiennej lokalnej *@rc* i zwraca ją, by pokazać, ile wierszy dotyczyło zapytanie.

```

DECLARE @rc AS INT;

EXEC Sales.GetCustomerOrders
    @custid = 1,
    @fromdate = '20150101',
    @todate = '20160101',
    @numrows = @rc OUTPUT;

SELECT @rc AS numrows;

```

Kod zwraca następujące wyniki, wyświetlając trzy zakwalifikowane zamówienia:

orderid	custid	empid	orderdate
10643	1	6	2015-08-25 00:00:00.000
10692	1	4	2015-10-03 00:00:00.000
10702	1	4	2015-10-13 00:00:00.000

numrows

3

Ponownie uruchamiamy kod, przekazując identyfikator klienta, który nie istnieje w tabeli *Orders* (na przykład identyfikator klienta o wartości 100). Uzyskujemy poniższe wyniki, które pokazują, że nie zostało znalezione żadne zamówienie.

```

orderid    custid    empid    orderdate
-----
numrows
-----
0

```

Rzecz jasna, jest to tylko prosty przykład. Procedury składowane pozwalają na wykonywanie znacznie bardziej skomplikowanych działań.

Wyzwalacze

Wyzwalacz (ang. *trigger*) jest specjalnym rodzajem procedury składowanej – takiej, która nie może być wykonywana wprost, ale jest wywoływana po wystąpieniu pewnego zdarzenia. Ilekroć określone zdarzenie ma miejsce, uruchamia się kod wyzwalacza. System SQL Server obsługuje powiązania wyzwalaczy z dwoma rodzajami zdarzeń – zdarzenia operacji na danych (wyzwalacze DML), takie jak instrukcja *INSERT*, oraz zdarzenia definiowania danych (wyzwalacze DDL), takie jak instrukcja *CREATE TABLE*.

Wyzwalacze mają wiele zastosowań, jak inspekcja lub wymuszanie reguł integralności, które nie mogą być zapewnione przez ograniczenia czy wymuszanie zasad.

Wyzwalacz stanowi część transakcji, która obejmuje zdarzenie powodujące jego uruchomienie. Wykonując polecenie *ROLLBACK TRAN* wewnątrz kodu wyzwalacza, spowodujemy wycofanie wszystkich zmian, które miały miejsce w wyzwalaczu, a także wszystkich zmian, które miały miejsce w transakcji powiązanej z wyzwalaczem.

Wyzwalacze w systemie SQL Server są uruchamiane dla instrukcji, a nie dla modyfikowanego wiersza.

Wyzwalacze DML

System SQL Server obsługuje dwa rodzaje wyzwalaczy DML – *after (po)* oraz *instead of (zamiast)*. Wyzwalacz *after* uruchamia się po zakończeniu zdarzenia z nim związanego i może być definiowany tylko dla tabel trwałych. Wyzwalacz *instead of* jest uruchamiany zamiast zdarzenia z nim związanego i może być definiowany dla trwałych tabel i widoków.

W kodzie wyzwalacza możemy uzyskiwać dostęp do specjalnych tabel nazywanych *inserted* (wstawione) i *deleted* (usunięte), zawierających wiersze, na które oddziaływała modyfikacja powodująca uruchomienie wyzwalacza. Tabela *inserted* przechowuje nowy obraz wierszy, na które oddziaływały akcje *INSERT* i *UPDATE*. Tabela *deleted* przechowuje obraz wierszy, na które oddziaływały akcje *DELETE* i *UPDATE*. Przypomnijmy, że działania wstawiania, aktualizacji i usuwania mogą być wywoływane przez instrukcje *INSERT*, *UPDATE* i *DELETE*, a także przez instrukcję *MERGE*. W przypadku wyzwalaczy *instead of*, tabele *inserted* i *deleted* zawierają wiersze, które miały być poddane oddziaływaniu modyfikacji uruchamiających wyzwalacz.

Poniższy prosty przykład wyzwalacza *after* przeprowadza inspekcję wstawienia wiersza do tabeli. Najpierw wykonamy kod tworzący w bieżącej bazie danych tabelę *dbo.T1* i drugą tabelę *dbo.T1_Audit*, która będzie przechowywać informacje inspekcji zdarzeń wstawiania do tabeli *T1*.

```
DROP TABLE IF EXIST dbo.T1_Audit, dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL PRIMARY KEY,
    datacol VARCHAR(10) NOT NULL
);

CREATE TABLE dbo.T1_Audit
(
    audit_lsn INT NOT NULL IDENTITY PRIMARY KEY,
    dt DATETIME NOT NULL DEFAULT(SYSDATETIME()),
    login_name sysname NOT NULL DEFAULT(ORIGINAL_LOGIN()),
    keycol INT NOT NULL,
    datacol VARCHAR(10) NOT NULL
);
```

W tabeli inspekcji kolumna *audit_lsn* ma właściwość *identity* i reprezentuje numer seryjny dziennika inspekcji. Kolumna *dt* reprezentuje datę i godzinę wstawienia przy użyciu domyślnego wyrażenia *SYSDATETIME()*. Kolumna *login_name* przy użyciu domyślnego wyrażenia *ORIGINAL_LOGIN()* będzie przechowywać nazwę logowania użytkownika, który wykonał wstawienie.

Następnie uruchamiamy poniższy kod, by utworzyć wyzwalacz *AFTER INSERT* *trg_T1_insert_audit* dla tablicy *T1*, monitorujący wstawienia.

```
CREATE TRIGGER trg_T1_insert_audit ON dbo.T1 AFTER INSERT
AS
SET NOCOUNT ON;

INSERT INTO dbo.T1_Audit(keycol, datacol)
SELECT keycol, datacol FROM inserted;

GO
```

Jak widzimy, wyzwalacz wstawia do tabeli inspekcji wyniki zapytania do tabeli *inserted*. Wartości kolumn w tabeli inspekcji, które nie są wprost wymienione w instrukcji *INSERT*, są generowane przez opisane wcześniej wyrażenia domyślne. Aby przetestować wyzwalacz, uruchomimy następujący kod:

```
INSERT INTO dbo.T1(keycol, datacol) VALUES(10, 'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(30, 'x');
INSERT INTO dbo.T1(keycol, datacol) VALUES(20, 'g');
```

Wyzwalacz uruchamia się po każdej instrukcji. Teraz wykonamy zapytanie do tabeli inspekcji.

```
SELECT audit_lsn, dt, login_name, keycol, datacol
FROM dbo.T1_Audit;
```

Otrzymamy poniższe wyniki (oczywiście w systemie Czytelnika wartości *dt* i *login_name*, które odzwierciedlają datę i godzinę wstawień oraz nazwę logowania, będą inne).

audit_lsn	dt	login_name	keycol	datacol
1	2012-02-12 09:04:27.713	K2\Gandalf	10	a
2	2012-02-12 09:04:27.733	K2\Gandalf	30	x
3	2012-02-12 09:04:27.733	K2\Gandalf	20	g

Po ukończeniu ćwiczenia uruchamiamy poniższy kod, by wyczyścić bazę danych.

```
DROP TABLE IF EXIST dbo.T1_Audit, dbo.T1;
```

Wyzwalacze DDL

Wyzwalacze *DDL* używane są do inspekcji, wymuszania zasad i zarządzania zmianami. Wersja SQL Server dla siedziby obsługuje tworzenie wyzwalaczy *DDL* w dwóch zakresach: w zakresie bazy danych i w zakresie serwera, w zależności od rodzaju zdarzenia. Wersja Azure SQL Database aktualnie obsługuje tylko wyzwalacze zakresu bazy danych.

Wyzwalacz *database* tworzymy dla zdarzenia o zakresie bazy danych, takiego jak *CREATE TABLE*, natomiast wyzwalacz *all server* tworzony jest dla zdarzenia o zakresie serwera, takiego jak *CREATE DATABASE*. SQL Server obsługuje tylko wyzwalacze *DDL* typu *after*; nie są natomiast obsługiwane wyzwalacze *DDL* typu *instead of*.

Wewnątrz wyzwalacza możemy uzyskać informacje o zdarzeniu, które spowodowało uruchomienie wyzwalacza poprzez zapytanie do funkcji nazwanej *EVENTDATA*. Funkcja ta zwraca informacje o zdarzeniu w postaci wartości *XML*. Do wyciągnięcia z wartości *XML* atrybutów zdarzenia, takich jak czas wysłania, typ zdarzenia czy nazwa logowania, możemy użyć wyrażenia *XQuery*.

Poniższy kod tworzy tabelę *dbo.AuditDDLEvents*, która przechowuje informacje o inspekcji.

```
IF OBJECT_ID('dbo.AuditDDLEvents', 'U') IS NOT NULL
    DROP TABLE dbo.AuditDDLEvents;

CREATE TABLE dbo.AuditDDLEvents
(
    audit_lsn          INT          NOT NULL IDENTITY,
    posttime           DATETIME    NOT NULL,
    eventtype          sysname     NOT NULL,
    loginname          sysname     NOT NULL,
    schemaname         sysname     NOT NULL,
    objectname         sysname     NOT NULL,
    targetobjectname   sysname     NULL,
    eventdata          XML         NOT NULL,
    CONSTRAINT PK_AuditDDLEvents PRIMARY KEY(audit_lsn)
);
```


Zwróćmy uwagę, że tabela zawiera kolumnę *eventdata*, której typ danych to XML. Oprócz poszczególnych atrybutów, które wyzwalacz pobiera z informacji o zdarzeniu i przechowuje w tych atrybutach, tabela ta przechowuje także pełne informacje o zdarzeniu w kolumnie *eventdata*.

Poniższy kod utworzy dla bazy danych wyzwalacz inspekcji *trg_audit_ddl_events* przy użyciu grupy zdarzeń *DDL_DATABASE_LEVEL_EVENTS*, która reprezentuje wszystkie zdarzenia DDL na poziomie bazy danych.

```
CREATE TRIGGER trg_audit_ddl_events
  ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS
AS
SET NOCOUNT ON;

DECLARE @eventdata AS XML = eventdata();

INSERT INTO dbo.AuditDDLEvents(
  posttime, eventtype, loginname, schemaname,
  objectname, targetobjectname, eventdata)
VALUES(
  @eventdata.value('(/EVENT_INSTANCE/PostTime)[1]',      'VARCHAR(23)'),
  @eventdata.value('(/EVENT_INSTANCE/EventType)[1]',      'sysname'),
  @eventdata.value('(/EVENT_INSTANCE/LoginName)[1]',      'sysname'),
  @eventdata.value('(/EVENT_INSTANCE/SchemaName)[1]',      'sysname'),
  @eventdata.value('(/EVENT_INSTANCE/ObjectName)[1]',      'sysname'),
  @eventdata.value('(/EVENT_INSTANCE/TargetObjectName)[1]', 'sysname'),
  @eventdata);

GO
```

Kod wyzwalacza najpierw przechowuje informacje o zdarzeniu uzyskane z funkcji *EVENTDATA* w zmiennej *@eventdata*. Następnie wstawia wiersz do tabeli inspekcji wraz z atrybutami pobranymi z informacji o zdarzeniu przy użyciu wyrażeń XQuery przez metodę *.value* oraz wartość XML, zawierającą pełne informacje o zdarzeniu.

Aby sprawdzić wyzwalacz, uruchamiamy poniższy kod, który zawiera kilka instrukcji DDL.

```
CREATE TABLE dbo.T1(col1 INT NOT NULL PRIMARY KEY);
ALTER TABLE dbo.T1 ADD col2 INT NULL;
ALTER TABLE dbo.T1 ALTER COLUMN col2 INT NOT NULL;
CREATE NONCLUSTERED INDEX idx1 ON dbo.T1(col2);
```

Następnie wykonamy zapytanie do tabeli inspekcji.

```
SELECT * FROM dbo.AuditDDLEvents;
```

Uzyskamy następujące wyniki (podzielone na dwie części, by poprawić ich czytelność). Ponownie w systemie Czytelnika wartości w atrybutach *posttime* i *loginname*, będą odzwierciedlały czas wysłania i nazwę logowania jego środowiska:

audit_lsn	posttime	eventtype	loginname
1	2012-02-12 09:06:18.293	CREATE_TABLE	K2\Gandalf

```

2          2012-02-12 09:06:18.413 ALTER_TABLE K2\Gandalf
3          2012-02-12 09:06:18.423 ALTER_TABLE K2\Gandalf
4          2012-02-12 09:06:18.423 CREATE_INDEX K2\Gandalf

```

audit_lsn	schemaname	objectname	targetobjectname	eventdata
1	dbo	T1	NULL	<EVENT_INSTANCE>...
2	dbo	T1	NULL	<EVENT_INSTANCE>...
3	dbo	T1	NULL	<EVENT_INSTANCE>...
4	dbo	idx1	T1	<EVENT_INSTANCE>...

Po ukończeniu ćwiczenia uruchamiamy poniższy kod, by wyczyścić bazę.

```

DROP TRIGGER IF EXIST trg_audit_ddl_events ON DATABASE;
DROP TABLE IF EXIST dbo.AuditDDLEvents;

```

Obsługa błędów

System SQL Server udostępnia narzędzia obsługi błędów występujących w czasie wykonywania kodu T-SQL. Głównym narzędziem używanym do obsługi błędów jest konstrukcja nazwana *TRY...CATCH*. SQL Server udostępnia także zbiór funkcji, które pozwalają uzyskać informacje na temat błędu. Omówienie rozpocznę od prostego przykładu użycia konstrukcji *TRY...CATCH*, po czym przejdę do bardziej szczegółowych przykładów.

Użycie konstrukcji *TRY...CATCH* polega na umieszczeniu kodu T-SQL w bloku *TRY* (pomiędzy słowami kluczowymi *BEGIN TRY* i *END TRY*), zaś cały kod obsługi błędów znajduje się w sąsiadującym bloku *CATCH* (pomiędzy słowami kluczowymi *BEGIN CATCH* i *END CATCH*). Jeśli w bloku *TRY* nie wystąpi błąd, blok *CATCH* jest po prostu pomijany. Jeśli w bloku *TRY* pojawił się błąd, kontrola przepływu jest przekazywana do odpowiedniego bloku *CATCH*. Warto zauważyć, że jeśli konstrukcja *TRY...CATCH* przechwyci błąd i go obsłuży, z punktu widzenia wywołującego żaden błąd się nie pojawi.

Uruchomimy poniższy kod, by zademonstrować przypadek braku błędu w bloku *TRY*.

```

BEGIN TRY
    PRINT 10/2;
    PRINT 'Brak błędów';
END TRY
BEGIN CATCH
    PRINT 'Błąd';
END CATCH;

```

Cały kod w bloku *TRY* działa poprawnie i dlatego blok *CATCH* zostaje pominięty. Kod ten generuje następujący wynik:

```

5
Brak błędów

```

Następnie uruchomimy podobny kod, ale tym razem wykonane będzie dzielenie przez 0 – pojawia się błąd.

```
BEGIN TRY
    PRINT 10/0;
    PRINT 'Brak błędów';
END TRY
BEGIN CATCH
    PRINT 'Błąd';
END CATCH;
```

Kiedy pojawił się błąd *dzielenia przez zero* w pierwszej instrukcji *PRINT* bloku *TRY*, kontrola została przekazana do odpowiedniego bloku *CATCH*. Druga instrukcja *PRINT* w bloku *TRY* nie została wykonana. Z tego względu kod ten generuje następujące dane wyjściowe:

Błąd

Zazwyczaj obsługa błędów wymaga wykonania pewnej pracy w bloku *CATCH* w celu znalezienia przyczyny błędu i wybrania odpowiedniego działania. SQL Server udostępnia informacje o błędach za pośrednictwem zestawu funkcji. Funkcja *ERROR_NUMBER* zwraca liczbę całkowitą przypisaną do błędu. Blok *CATCH* zazwyczaj zawiera kod sterowania przepływem, który sprawdza numer błędu, by określić, jakie podjąć działania. Funkcja *ERROR_MESSAGE* zwraca tekst komunikatu o błędzie. Aby uzyskać listę numerów błędów i odpowiednich komunikatów o błędzie, możemy odpytać widok katalogowy *sys.messages*. Funkcje *ERROR_SEVERITY* i *ERROR_STATE* zwracają ważność błędu i stan błędu. Funkcja *ERROR_LINE* zwraca numer wiersza, w którym pojawił się błąd. Na koniec, funkcja *ERROR_PROCEDURE* zwraca nazwę procedury, w której błąd wystąpił, lub *NULL*, jeśli błąd nie pojawił się w procedurze.

W celu pokazania bardziej szczegółowego przykładu obsługi błędów z uwzględnieniem funkcji błędów, najpierw uruchomimy poniższy kod, by w bieżącej bazie danych utworzyć tabelę nazwaną *dbo.Employees*.

```
DROP TABLE IF EXIST dbo.Employees;
CREATE TABLE dbo.Employees
(
    empid    INT          NOT NULL,
    empname  VARCHAR(25)  NOT NULL,
    mgrid    INT          NULL,
    CONSTRAINT PK_Employees PRIMARY KEY(empid),
    CONSTRAINT CHK_Employees_empid CHECK(empid > 0),
    CONSTRAINT FK_Employees_Employees
        FOREIGN KEY(mgrid) REFERENCES dbo.Employees(empid)
);
```

Poniższy kod wstawia nowy wiersz do tabeli *Employees* w bloku *TRY* i jeśli pojawia się błąd, pokazuje, jak zidentyfikować błąd przy użyciu funkcji *ERROR_NUMBER*

w bloku *CATCH*. Kod stosuje kontrolę przepływu do zidentyfikowania i obsługi błędów, które chcemy obsługiwać w bloku *CATCH*, a w innych przypadkach przerzuca go dalej.

Kod drukuje także wartości innych funkcji błędów, po prostu po to, by pokazać, jakie informacje na temat błędu są dostępne.

```
BEGIN TRY

    INSERT INTO dbo.Employees(empid, empname, mgrid)
        VALUES(1, 'Emp1', NULL);
    -- Wypróbować także dla empid = 0, 'A', NULL

END TRY
BEGIN CATCH

    IF ERROR_NUMBER() = 2627
    BEGIN
        PRINT '    Obsługa naruszenia PK...';
    END
    ELSE IF ERROR_NUMBER() = 547
    BEGIN
        PRINT '    Obsługa naruszenia ograniczenia CHECK/FK...';
    END
    ELSE IF ERROR_NUMBER() = 515
    BEGIN
        PRINT '    Obsługa nieprawidłowej wartości NULL...';
    END
    ELSE IF ERROR_NUMBER() = 245
    BEGIN
        PRINT '    Obsługa błędu konwersji...';
    END
    ELSE
    BEGIN
        PRINT 'Obsługa nieznanego błędu...';
        THROW; -- tylko system SQL Server 2012
    END

    PRINT '    Numer błędu      : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
    PRINT '    Komunikat błędu: ' + ERROR_MESSAGE();
    PRINT '    Ważność błędu   : ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
    PRINT '    Stan błędu      : ' + CAST(ERROR_STATE() AS VARCHAR(10));
    PRINT '    Wiersz błędu    : ' + CAST(ERROR_LINE() AS VARCHAR(10));
    PRINT '    Procedura błędu  : ' + COALESCE(ERROR_PROCEDURE(), 'Not within
proc');

END CATCH;
```

Kiedy uruchomimy ten kod po raz pierwszy, dodanie nowego wiersza do tabeli *Employees* udaje się i dlatego blok *CATCH* jest pomijany. Otrzymujemy następujący wynik:

```
(1 row(s) affected)
```

Kiedy uruchomimy ten kod po raz drugi, wykonanie instrukcji *INSERT* się nie powiedzie, kontrola zostanie przekazana do bloku *CATCH* i rozpoznany zostanie błąd naruszenia klucza podstawowego. Uzyskujemy następujące informacje wyjściowe:

Obsługa naruszenia PK...

Numer błędu : 2627

Komunikat błędu : Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object 'dbo.Employees'.

Ważność błędu: 14

Stan błędu : 1

Wiersz błędu : 3

Procedura błędu : Not within proc (nie w procedurze)

Aby przejrzeć inne błędy, uruchamiamy kod z wartościami 0, 'A' i *NULL* jako identyfikator pracownika.

Dla celów demonstracji użyłem instrukcji *PRINT* jako działań po zidentyfikowaniu błędu. Oczywiście obsługa błędu zazwyczaj wiąże się z innymi działaniami, a nie tylko z wydrukowaniem komunikatu rozpoznania błędu.

Zwróćmy uwagę, że mamy możliwość utworzenia procedury składowanej, która opakowuje kod obsługi błędów do wielokrotnego użycia, jak w poniższym przykładzie:

```
DROP PROC IF EXIST dbo.ErrInsertHandler;
GO

CREATE PROC dbo.ErrInsertHandler
AS
SET NOCOUNT ON;

IF ERROR_NUMBER() = 2627
BEGIN
    PRINT 'Obsługa naruszenia PK...';
END
ELSE IF ERROR_NUMBER() = 547
BEGIN
    PRINT 'Obsługa naruszenia ograniczenia CHECK/FK...';
END
ELSE IF ERROR_NUMBER() = 515
BEGIN
    PRINT 'Obsługa nieprawidłowej wartości NULL...';
END
ELSE IF ERROR_NUMBER() = 245
BEGIN
    PRINT 'Obsługa błędu konwersji...';
END

PRINT '    Numer błędu      : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
PRINT '    Komunikat błędu: ' + ERROR_MESSAGE();
PRINT '    Ważność błędu   : ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT '    Stan błędu      : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT '    Wiersz błędu    : ' + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT '    Procedura błędu : ' + COALESCE(ERROR_PROCEDURE(), 'Not within
```

```
proc');  
GO
```

W bloku *CATCH* sprawdzamy, czy numer błędu jest jednym z tych, które chcemy obsługiwać lokalnie, a w takim przypadku po prostu wykonujemy procedurę składowaną; w przeciwnym razie przekazujemy błąd dalej do kodu wywołującego.

```
BEGIN TRY  
  
    INSERT INTO dbo.Employees(empid, empname, mgrid)  
        VALUES(1, 'Emp1', NULL);  
  
END TRY  
BEGIN CATCH  
  
    IF ERROR_NUMBER() IN (2627, 547, 515, 245)  
        EXEC dbo.ErrInsertHandler;  
    ELSE  
        THROW;  
  
END CATCH;
```

W ten sposób możemy w jednym miejscu utrzymywać kod obsługi błędu do wielokrotnego wykorzystywania.

Podsumowania

W tym rozdziale przedstawiłem ogólny opis obiektów programowalnych w celu zapoznania Czytelników z możliwościami systemu SQL Server w tym obszarze i pokazania podstawowych pojęć. Omówiłem zmienne, wsady, elementy sterowania przepływem wykonania, kursory, tabele tymczasowe, dynamiczny kod SQL, funkcje definiowane przez użytkownika, procedury składowane, wyzwalacze i obsługę błędów – całkiem sporo zagadnień. Rozdział ten zarazem kończy tę książkę. Jeśli Czytelnik jest już gotów na spotkanie z bardziej zaawansowanymi zagadnieniami języka T-SQL, kolejnym naturalnym krokiem będzie moja książka *Zapytania w języku T-SQL* (APN Promise, 2015).

Rozpoczynamy

Dodatek ten ułatwia skonfigurowanie środowiska niezbędnego do tego, by lektura książki przyniosła Czytelnikowi jak największe korzyści.

Wszystkie przytoczone w książce przykłady kodu można uruchamiać w wersji Microsoft SQL Server dla siedziby (*box*), zaś większość można wykonać w środowisku Microsoft Azure SQL Database. Informacje szczegółowe na temat różnic pomiędzy tymi wersjami znaleźć można w rozdziale 1 „Podstawy zapytań i programowania T-SQL”, w podrozdziale „Odmiany ABC produktu SQL Server”.

W pierwszym punkcie „Rozpoczynamy pracę w Azure SQL Database” podaję łącze do witryny Web, w której można znaleźć informacje niezbędne do rozpoczęcia korzystania z produktu SQL Database.

W drugim punkcie „Instalowanie SQL Server w wersji dla siedziby” przyjęto założenie, że Czytelnik do uruchamiania przykładów kodu chce korzystać z lokalnej instancji SQL Server i że jeszcze nie ma takiej instalacji. Opisałem tam proces instalacji oprogramowania SQL Server 2016. Jeśli ktoś już ma działającą instancję SQL Server, śmiało może pominąć czytanie tego podrozdziału.

Trzeci punkt, „Pobieranie i instalowanie SQL Server Management Studio” zawiera instrukcje pobierania i instalacji narzędzia SSMS.

Czwarty punkt, „Pobieranie kodu źródłowego i przykładowej bazy danych” udostępnia łącza do witryny Web, z której można pobrać kod źródłowy dla książki; dodatkowo zawarłem w nim instrukcje instalowania używanej w książce przykładowej bazy danych w instancji SQL Server w wersji dla siedziby i w Azure SQL Database.

W piątym podrozdziale „Używanie narzędzia SQL Server Management Studio” wyjaśniam, jak tworzyć i uruchamiać kod T-SQL w systemie SQL Server przy użyciu narzędzia SQL Server Management Studio (SSMS).

Ostatni podrozdział „Korzystanie z dokumentacji SQL Server Books Online” opisuje tę witrynę i wyjaśnia jej rolę w uzyskiwaniu informacji na temat języka T-SQL.

Rozpoczynamy pracę w Azure SQL Database

Jeśli przytoczone w książce przykłady chcemy uruchamiać w Azure SQL Database, potrzebny będzie dostęp do serwera Azure SQL Database przy użyciu konta z uprawnieniami do tworzenia nowej bazy danych, albo poprosić administratora, by utworzył dla nas testową bazę danych. Jeśli nie mamy jeszcze dostępu do Azure SQL Database, użyteczne informacje na temat rozpoczynania pracy z tym produktem znaleźć można na stronie głównej Windows Azure pod adresem: <https://azure.microsoft.com>.

Do utworzenia subskrypcji Azure niezbędne jest konto Microsoft (dawniej nazywane Windows Live ID). Jeśli nie mamy takiego konta, można je utworzyć na stronie <https://signup.live.com>. Jeśli mamy już subskrypcję produktu Azure, możemy połączyć się z portalem zarządzania – Microsoft Azure Portal – pod adresem <https://portal.azure.com/>, umożliwiającym zarządzanie serwerami i bazami danych Azure SQL Database.

Strona główna Microsoft Azure umożliwia rozpoczęcie pracy poprzez zakup subskrypcji lub wypróbowanie wersji testowej oraz udostępnia różne zasoby, takie jak portal zarządzania, społeczności i wsparcie techniczne.

Po uzyskaniu dostępu do Azure SQL Database trzeba wykonać instrukcje opisujące pobieranie kodu źródłowego i instalowania bazy danych, zamieszczone w dalszej części Dodatku.

Instalowanie produktu SQL Server w wersji dla siedziby

Podrozdział ten jest przeznaczony dla osób, które chcą uruchamiać przykłady kodu i ćwiczenia w lokalnej instancji SQL Server i nie mają jeszcze takiej instalacji. Można tu użyć dowolnego wydania systemu SQL Server 2016 lub późniejszego. Zakładając, że Czytelnik nie ma jeszcze dostępu do instancji SQL Server, kolejne podpunkty pokazują, gdzie można uzyskać produkt SQL Server i jak go zainstalować.

Ćwiczenie 1. Uzyskanie produktu SQL Server

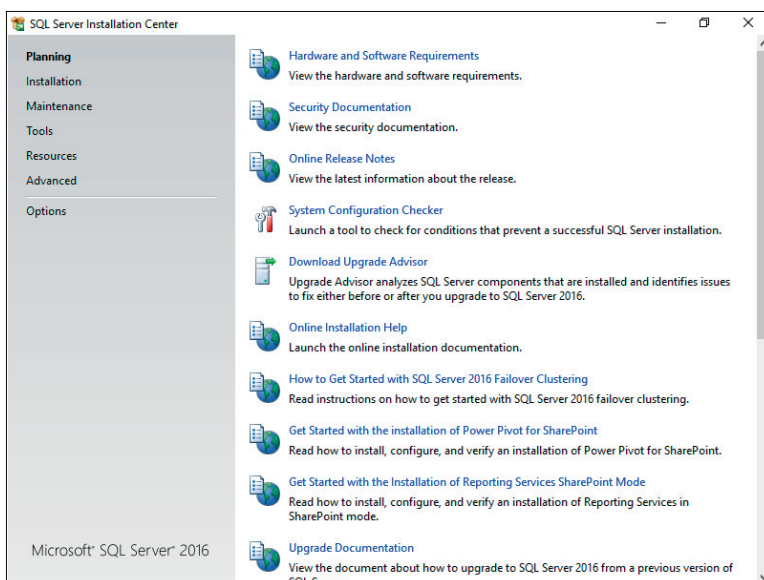
Jak wspomniałem, można używać dowolnej wersji SQL Server 2016, aby wykonywać ćwiczenia i przykłady kodu pokazane w książce. Jeśli posiadamy subskrypcję MSDN (Microsoft Developer Network), do celów szkoleniowych możemy posługiwać się produktem SQL Server 2016 Developer, który można pobrać ze strony <https://msdn.microsoft.com/subscriptions/downloads>. Wydanie Developer jest również dostępne bez opłat dla członków Visual Studio Dev Essentials. Szczegółowe informacje zawiera artykuł <https://blogs.technet.microsoft.com/dataplatforminsider/2016/03/31/microsoft-sql-server-developer-edition-is-now-free>. Inną opcją jest użycie wersji próbnej SQL Server 2016, którą można pobrać z witryny <https://www.microsoft.com/sql>. W dalszej części dodatku zademonstruję instalację wydania próbnego SQL Server 2016.

Ćwiczenie 2. Instalowanie silnika bazy danych

Po zainstalowaniu wszystkich wymaganych wstępnie programów możemy przejść do zainstalowania produktu.

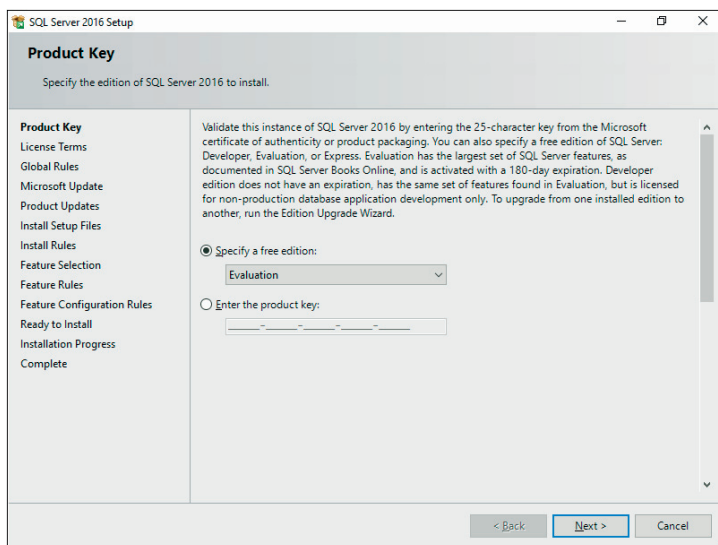
W celu zainstalowania silnika bazy danych:

1. Uruchom program **setup.exe** z folderu instalacyjnego SQL Server. Powinno pojawić się okno dialogowe SQL Server Installation Center, pokazane na rysunku A-1.



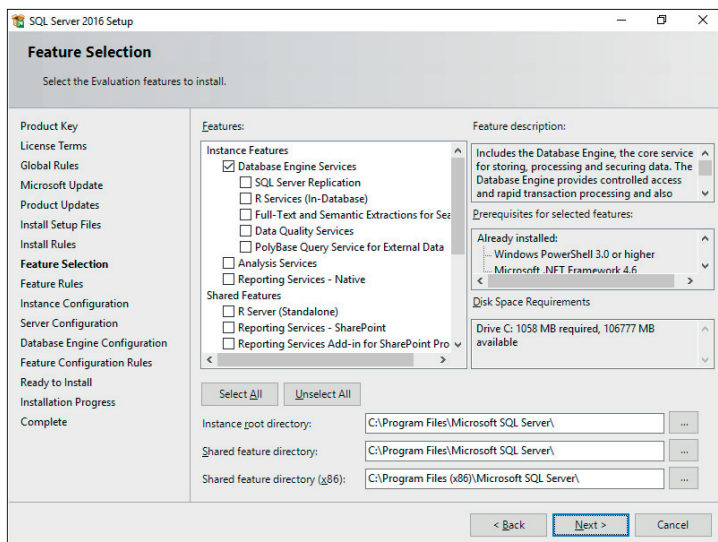
RYСУNEK A-1 SQL Server Installation Center

2. W panelu z lewej strony wybierz Installation. Zwróć uwagę, że zawartość ekranu się zmieniła.
3. W panelu z prawej strony wybierz opcję New SQL Server Stand-Alone Installation Or Add Features To An Existing Installation (instalacja autonomiczna nowego serwera SQL lub dodanie funkcji do instalacji istniejącej). Pojawi się okno Product Key (klucz produktu), pokazane na rysunku A-2.
4. Upewnij się, że w sekcji Specify A Free Edition (wybierz bezpłatne wydanie) zaznaczona jest opcja Evaluation (testowa), po czym kliknij Next (dalej), aby kontynuować. Wyświetlone zostanie okno dialogowe License Terms (warunki licencyjne).
5. Potwierdź, że akceptujesz warunki licencji, po czym kliknij Next. Pojawi się okno dialogowe Microsoft Update.



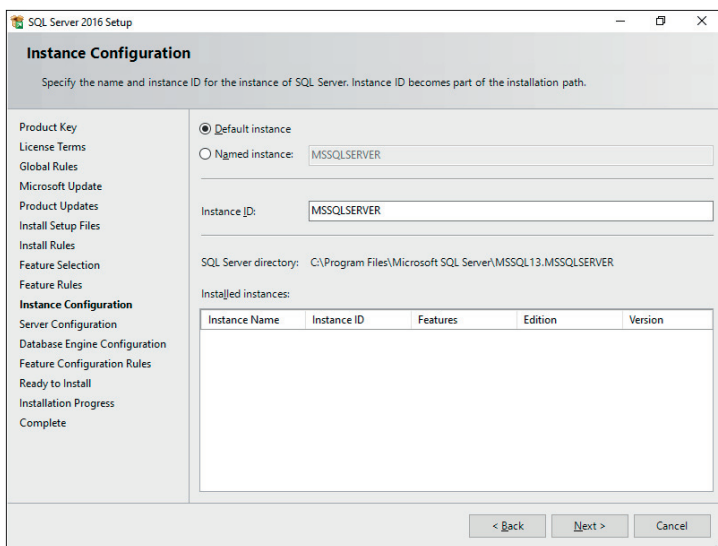
RYSUNEK A-2 Okno dialogowe Product Key

6. Potwierdź, że chcesz użyć usługi Microsoft Update do sprawdzenia dostępności aktualizacji i poprawek (jest to zdecydowanie zalecane działanie), po czym kliknij Next. Pojawi się okno dialogowe Install Rules (reguły instalacji).
7. Upewnij się, że nie są sygnalizowane żadne problemy. Kliknij Next. Pojawi się okno dialogowe Feature Selection (wybór funkcjonalności). Wybierz elementy, które chcesz zainstalować, jak na rysunku A-3.



RYSUNEK A-3 Okno dialogowe Feature Selection

8. Zaznacz funkcjonalność Database Engine Services. Dla ćwiczeń i przykładów zawartych w tej książce nie są potrzebne żadne inne funkcjonalności i dodatki.
9. Kliknij Next, aby kontynuować. Pojawi się okno dialogowe Instance Configuration (konfigurowanie instancji), pokazane na rysunku A-4.



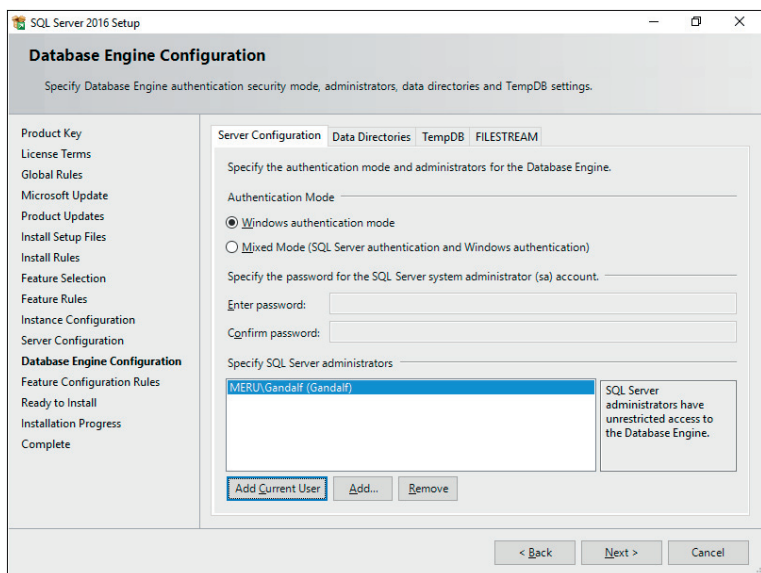
RYСУNEK A-4 Okno dialogowe Instance Configuration

Jeśli koncepcja instancji SQL Server nie wydaje się zbyt zrozumiała, szczegóły na ten temat można znaleźć w rozdziale 1, w punkcie „Architektura SQL Server”.

10. Jeśli na używanym komputerze nie ma jeszcze domyślnej instancji SQL Server i chcesz skonfigurować nową instancję jako domyślną, wystarczy upewnić się, że zaznaczona jest opcja Default Instance. Jeśli wolisz skonfigurować nową instancję jako nazwaną, wybierz opcję Named Instance, po czym wpisz nazwę dla tej instancji (na przykład **SQL2016**). Nazwa instancji nie może zawierać spacji ani znaków specjalnych poza dywizem. Później, przy łączeniu się z SQL Server, w przypadku łączenia się z instancją domyślną wystarczy podać samą nazwę komputera (na przykład, **MERU**), zaś kombinację nazwa_komputera\nazwa_instancji w przypadku instancji nazwanej (na przykład **MERU\SQL2016**).
11. Kliknij Next. Pojawi się okno dialogowe Server Configuration (konfiguracja serwera).

Na potrzeby przykładów i ćwiczeń zawartych w tej książce nie trzeba zmieniać żadnych domyślnych ustawień w częściach Service Accounts (konta usług) ani Collation (ustawienia językowe). Więcej informacji na temat ustawień językowych zawiera rozdział 2, „Zapytania do pojedynczej tabeli” w podpunkcie „Opcje sortowania (*collation*)”.

12. Kliknij Next. Pojawi się okno dialogowe Database Engine Configuration (konfiguracja silnika bazy danych).
13. Na karcie Server Configuration upewnij się, że w sekcji Authentication Mode (tryb uwierzytelniania) wybrana jest opcja Windows Authentication Mode (tryb uwierzytelniania Windows). W sekcji Specify SQL Server Administrators (określ administratorów SQL Server) kliknij Add Current User (dodaj bieżącego użytkownika), aby przypisać aktualnie zalogowanemu użytkownikowi (czyli sobie) rolę serwerową System Administrator (sysadmin), jak na rysunku A-5. Administratorzy SQL Server mają nieograniczony dostęp do silnika bazy danych SQL Server.

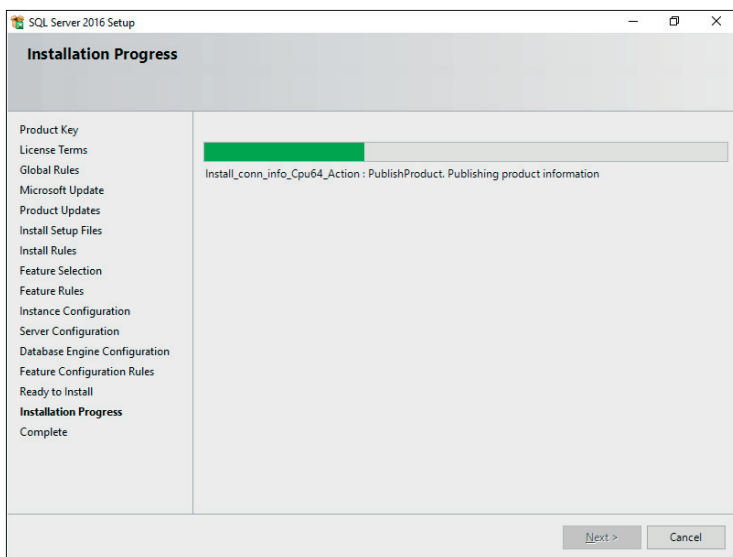


RYСУNEK A-5 Okno dialogowe Database Engine Configuration

Oczywiście zamiast *MERU\Gandalf* zobaczysz swoją nazwę użytkownika.

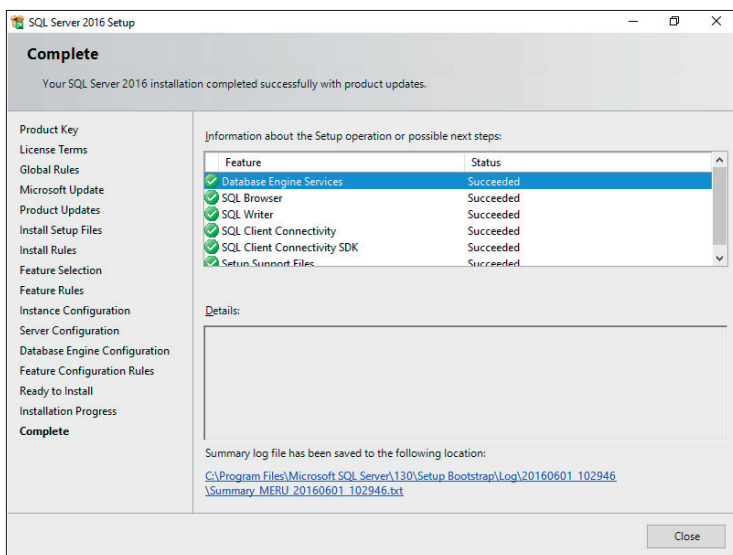
Jeśli wolisz zmienić domyślne ustawienia programu instalacyjnego dotyczące lokalizacji katalogów danych, możesz to zrobić na karcie Data Directories. Na potrzeby ćwiczeń i przykładów z tej książki nie trzeba nic konfigurować na kartach TempDB ani FILESTREAM.

14. Kliknij Next. Pojawi się okno dialogowe Ready To Install (gotowy do instalacji), zawierające podsumowanie wyborów instalacyjnych.
15. Przejrzyj wyświetlane informacje, aby upewnić się, że wszystkie wybrane opcje są właściwe, po czym kliknij Install, aby rozpocząć rzeczywisty proces instalacyjny. Przez cały czas trwania instalacji wyświetlane jest okno dialogowe Installation Progress. Okno to zawiera ogólny pasek postępu, a także sygnalizuje status instalacji każdej wybranej funkcji (rysunek A-6.)



RYSUNEK A-6 Okno dialogowe Installation Progress

16. Po zakończeniu instalacji pojawi się okno dialogowe Complete (zakończzone), pokazane na rysunku A-7.



RYSUNEK A-7 Okno dialogowe Complete

Okno to powinno wskazywać, że instalacja zakończyła się sukcesem.

17. Kliknij Close, aby zakończyć.

Pobieranie i instalowanie SQL Server Management Studio

Bez względu na to, czy zamierzamy używać lokalnej wersji SQL Server, czy Azure SQL Database jako silnika bazy danych, należy pobrać i zainstalować oprogramowanie SQL Server Management Studio (określane zazwyczaj akronimem SSMS) – narzędzie klienckie, które umożliwia pisanie, debugowanie i uruchamianie kodu T-SQL w silniku bazy danych. Pakiet instalacyjny tego narzędzia można pobrać ze strony <https://msdn.microsoft.com/en-us/library/mt238290.aspx>. Po zakończeniu pobierania wystarczy uruchomić program instalacyjny. Proces instalacji nie wymaga żadnej interakcji z użytkownikiem poza kliknięciem przycisku Install w celu rozpoczęcia i przycisku Close na końcu.

Pobieranie kodu źródłowego i instalowanie przykładowej bazy danych

Aby pobrać kod źródłowy, należy odwiedzić witrynę Web powiązaną z tą książką: <http://www.ksiazki.promise.pl/asp/produkt.aspx?pid=112055>. Na stronie tej znajduje się zakładka Dodatkowe informacje, w której znajdziemy łącze do pobrania pojedynczego skompresowanego pliku, zawierającego cały kod źródłowy wykorzystywany w książce, a także plik skryptu TSQLV4.sql, który tworzy przykładową bazę danych. Plik ten należy rozpakować do dowolnego lokalnego folderu (na przykład C:\Podstawy_TSQL).

Po rozpakowaniu znajdziemy pliki skryptów (do trzech) powiązane z poszczególnymi rozdziałami książki:

- Jeden plik zawiera kod źródłowy odpowiedniego rozdziału. Zamieściłem go dla wygody Czytelnika, na wypadek, gdyby nie chciał przepisywać całych (niekiedy dość długich) przykładów występujących w książce. Nazwa pliku odpowiada numerowi i tytułowi rozdziału.
- Drugi plik zawiera ćwiczenia dla danego rozdziału. Nazwa pliku uzupełniona jest sufiksem „ćwiczenia”.
- Trzeci plik zawiera rozwiązania ćwiczeń dla odpowiedniego rozdziału. Nazwa pliku uzupełniona jest sufiksem „rozwiązania”.

Do otwierania plików i uruchamiania zawartego w nich kodu należy używać narzędzia SSMS. Jeśli narzędzie SSMS nie zostało jeszcze zainstalowane, należy to zrobić, zgodnie z instrukcjami zawartymi w punkcie „Pobieranie i instalowanie SQL Server Management Studio”. W kolejnym punkcie wyjaśnię, jak należy posługiwać się narzędziem SSMS.

Plik źródłowy zawiera również plik tekstowy o nazwie *orders.txt*, który można wykorzystać podczas wykonywania przykładów i ćwiczeń z rozdziału 8, „Modyfikowanie

danych”. Na koniec dołączyłem plik skryptu o nazwie *TSQIV4.sql*, który tworzy przykładową bazę danych wykorzystywaną w tej książce, czyli *TSQIV4*.

Aby utworzyć przykładową bazę w lokalnej instancji produktu SQL Server (w siedzibie), należy po prostu uruchomić ten skrypt, mając aktywne połączenie z docelową instancją SQL Server. Poniższa procedura krok po kroku pokazuje tworzenie bazy danych przy użyciu skryptu.

Aby utworzyć i wypełnić przykładową bazę danych w wersji pudełkowej (w siedzibie) produktu SQL Server:

1. Podwójnie kliknij plik **TSQIV4.sql** w Eksploratorze plików, aby otworzyć plik w narzędziu SSMS. Pojawi się okno dialogowe Connect To Database Engine (połącz z silnikiem bazy danych).
2. Upewnij się, że w polu Server Name (nazwa serwera) widoczna jest nazwa instancji, z którą chcesz się połączyć. Na przykład, jeśli komputer nosi nazwę *MERU* i instancja jest domyślna, w polu tym należy wpisać **MERU**; w przypadku instancji nazwanej pole powinno zawierać tekst podobny do **MERU\SQI2016** (jeśli instancja ma nazwę *SQI2016* na komputerze *MERU*).
3. W polu Authentication (uwierzytelnienie) upewnij się, że zaznaczona jest opcja Windows Authentication. Kliknij Connect.
4. Po uzyskaniu połączenia z SQL Server naciśnij klawisz F5, aby uruchomić skrypt. Po zakończeniu wykonywania w panelu Messages powinien pojawić się komunikat Command(s) Completed Successfully. W panelu Available Databases (dostępne bazy danych) powinieneś zobaczyć bazę danych *TSQIV4*.

Aby utworzyć i wypełnić przykładową bazę danych w Azure SQL Database

1. Podwójnie kliknij plik *TSQIV4.sql* w Eksploratorze plików, aby otworzyć plik w narzędziu SSMS. Pojawi się okno dialogowe Connect To Database Engine (połącz z silnikiem bazy danych).
2. W polu Server Name (nazwa serwera) wpisz nazwę serwera Azure SQL Database, z którym chcesz się połączyć – na przykład *myserver.database.windows.net*.
3. Upewnij się, że w polu Authentication (uwierzytelnienie) wybrana jest opcja SQL Authentication i że wpisane są właściwa nazwa logowania i hasło. Kliknij Options.
4. Na karcie Connection Properties (właściwości połączenia) wpisz **master** w polu tekstowym Connect To Database (połącz z bazą danych), po czym kliknij Connect.
5. Pomiń instrukcje zawarte w Części A skryptu (przeznaczone dla lokalnej instalacji SQL Server), i wykonaj kolejno instrukcje zawarte w Części B – cały ten blok jest objęty znakami komentarza */* */* i nie zostaje wykonany automatycznie.

Najważniejszą instrukcją jest ta, która nakazuje wykonać poniższe polecenie, aby utworzyć bazę danych *TSQLV4*:

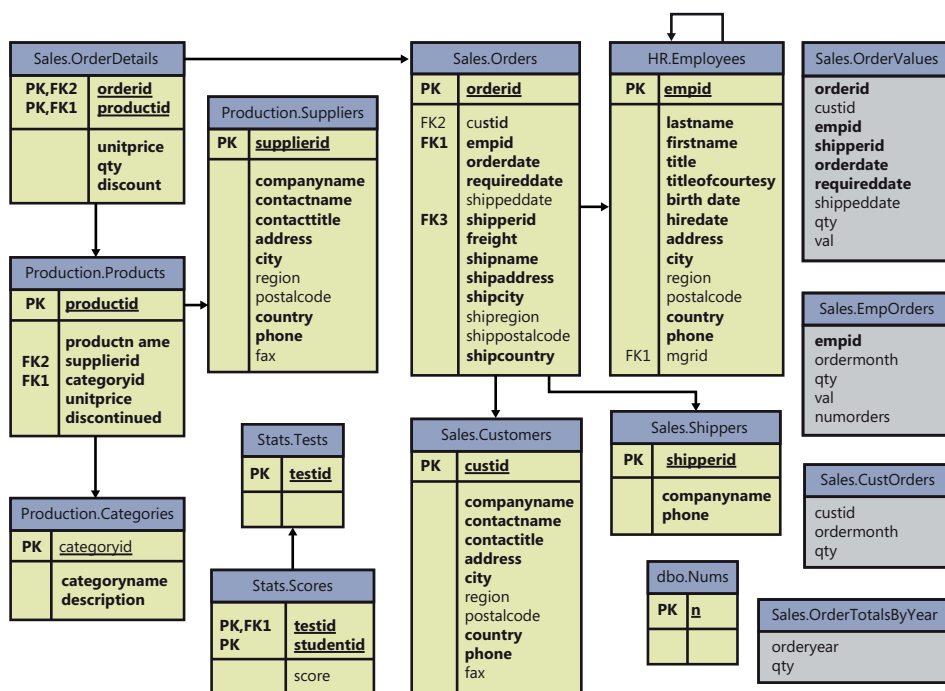
```
CREATE DATABASE TSQLV4;
```

6. Prawym klawiszem myszy kliknij w pustym obszarze panelu zapytań i wybierz polecenie Connection | Change Connection. Pojawi się okno dialogowe Connect To Database Engine (łączenie z bazą danych). Wpisz **TSQLV4** jako docelową bazę danych i kliknij Connect. Powinieneś widzieć wpis bazy danych *TSQLV4* w panelu Available Databases (dostępne bazy danych).

Alternatywnie możesz po prostu wybrać *TSQLV4* w panelu Available Databases.

7. Zaznacz kod w Części C skryptu (zaczynając od komentarza *Tworzenie schematów* aż do końca pliku skryptu). Naciśnij F5, aby uruchomić zaznaczony kod. Po zakończeniu wykonywania w panelu Messages pojawi się komunikat *Command(s) Completed Successfully*. Zwróć uwagę, że przy powolnych łączach wykonanie kodu może zająć kilka minut.

Model danych bazy *TSQLV4* jest przedstawiony na rysunku A-8.



RYSUNEK A-8 Model danych bazy *TSQLV4*

Posługiwanie się programem SQL Server Management Studio

SQL Server Management Studio (SSMS) jest narzędziem klienckim, w którym można projektować i uruchamiać kod T-SQL na serwerze SQL Server. Celem tego punktu nie jest przedstawienie kompletnego opisu funkcjonalności SSMS, ale zapewnienie dobrego punktu startowego.

UWAGA Narzędzie SSMS jest okresowo aktualizowane, zatem wygląd interfejsu może się różnić od ekranów pokazanych w tym dodatku.

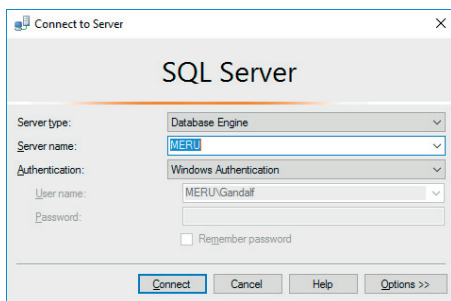


Aby rozpocząć pracę w SSMS:

1. Uruchom narzędzie SSMS w grupie programów Microsoft SQL Server.
2. Jeśli jest to pierwsze uruchomienie SSMS, zalecam określić opcje rozruchowe, tak by środowisko zostało skonfigurowane w odpowiedni sposób.
 - a. Jeśli pojawi się okno dialogowe Connect To Server, na razie kliknij Cancel (anuluj).
 - b. Wybierz polecenie Tools | Options, aby otworzyć okno dialogowe Options. W sekcji Environment | Startup (środowisko | rozruch), ustaw opcję At Startup (podczas rozruchu) na Open Object Explorer And Query Window (otwórz panel Object Explorer i okno zapytań). Wybór ten spowoduje, że przy każdym uruchomieniu narzędzia SSMS otworzy się panel Object Explorer i nowe okno zapytania.

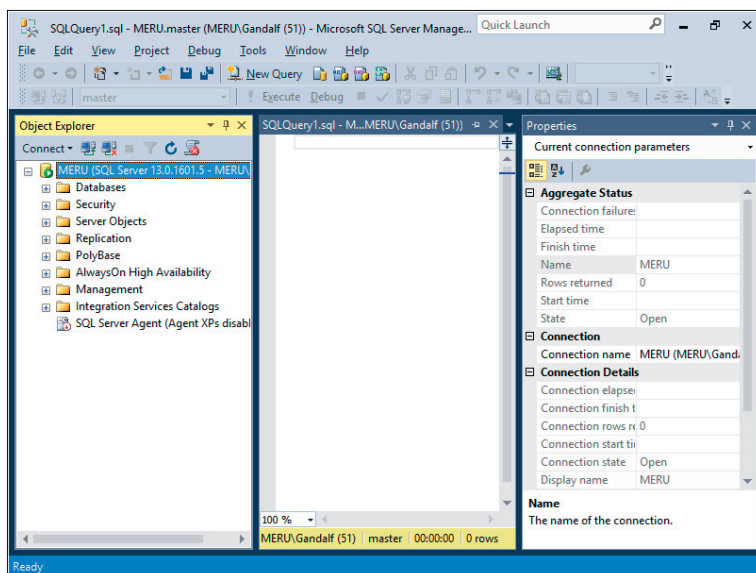
Object Explorer jest narzędziem, w którym można zarządzać programem SQL Server i graficznie przeglądać definicje obiektów, zaś okno zapytania jest miejscem, w którym tworzymy i wykonujemy kod T-SQL. Warto przejrzeć drzewo obiektów panelu, aby poznać dostępne opcje, ale tylko nieliczne z nich mają znaczenie na tym etapie. Gdy zdobędziesz już pewne doświadczenie w pracy z SSMS, zapewne wiele z tych opcji nabierze większego sensu i zapewne będziesz chciał zmienić przynajmniej niektóre z nich.
 - c. Po zakończeniu przeglądania okna dialogowego Options kliknij OK, aby zatwierdzić wybory.
3. Zamknij SSMS i uruchom go ponownie, aby się upewnić, że rzeczywiście otworzy się panel Object Explorer i nowe okno zapytania. Powinieneś zobaczyć okno dialogowe Connect To Server, pokazane na rysunku A-9.

W tym oknie należy wpisać szczegóły instancji SQL Server, z którą chcesz się połączyć.



RYСУNEK A-9 Okno dialogowe Connect To Server

4. Wpisz nazwę serwera w polu Server Name; jeśli wykonywałeś już jakieś połączenia, pole to udostępni listę tych serwerów. W przypadku Azure SQL Database konieczne jest podanie pełnej nazwy DNS serwera w postaci *twójserwer.database.windows.net* (zastępując *twójserwer* nazwą własnego serwera Azure).
5. Wybierz tryb uwierzytelniania z listy Authentication, zgodnie z typem loginu, którego używasz do połączenia (Windows Authentication lub SQL Server Authentication). Jeśli używasz tego pierwszego (co jest działaniem zalecanym), nie musisz podawać nazwy logowania i hasła. W przeciwnym przypadku trzeba podać nazwę logowania SQL server oraz hasło. W przypadku korzystania z Azure SQL Database dodatkowo kliknij Options i wpisz **TSQLV4** w polu Connect To Dataset okna dialogowego Connection Properties.
6. Kliknij Connect. Narzędzie SSMS się uruchomi, jak pokazano na rysunku A-10.

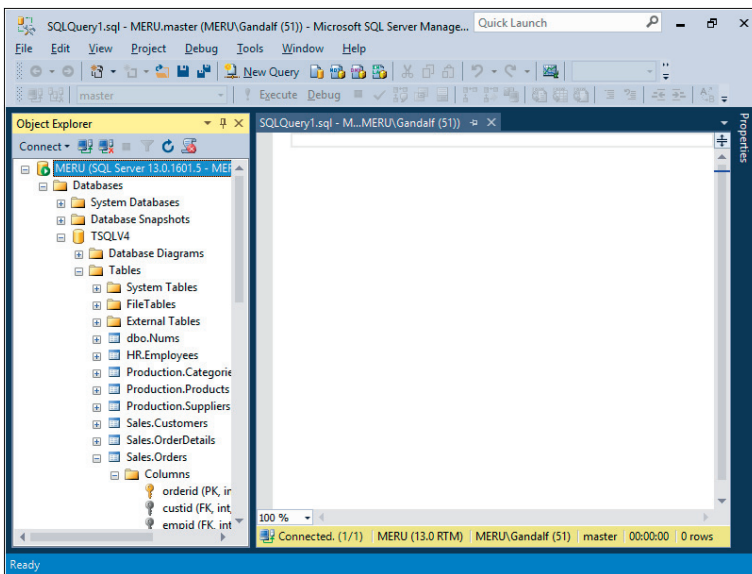


RYСУNEK A-10 Początkowy ekran SSMS

Panel Object Explorer pojawia się z lewej strony, na prawo od niego wyświetlone jest okno zapytania, zaś po prawej stronie pojawi się panel Properties (właściwości). Panel ten można ukryć, klikając przycisk Auto Hide (w prawym górnym rogu okna panelu, na lewo od ikony X). Dostosuj rozmiary panelu Object Explorer i okna zapytania, jak ci wygodnie. Choć tematyka tej książki skupia się na tworzeniu kodu T-SQL, a nie na zarządzaniu serwerem SQL Server, zdecydowanie zachęcam do poznania zawartości tego panelu, przeglądając drzewo obiektów i klikając różne węzły prawym klawiszem myszy. Zapewne szybko uznasz Object Explorer za bardzo wygodne narzędzie graficznego przeglądania baz danych i obiektów w tych bazach, co pokazuje rysunek A-11.

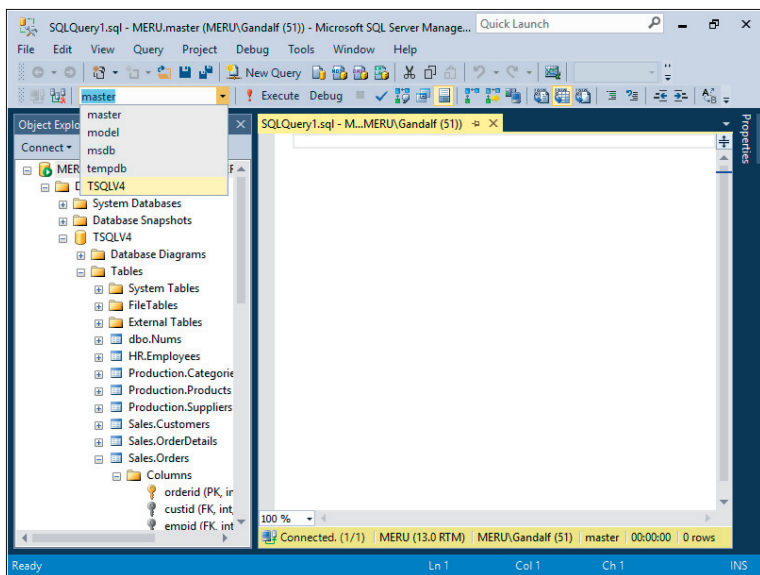
Zwróć uwagę, że można przeciągać elementy z panelu Object Explorer do okna zapytania.

WSKAZÓWKA Jeśli przeciągniesz folder Columns dla tabeli z panelu Object Explorer do okna zapytania, SQL Server wyliczy wszystkie kolumny tej tabeli, rozdzielane przecinkami.



RYСУNEK A-11 Object Explorer

W oknie zapytań wpisujemy i wykonujemy kod T-SQL. Uruchamiany kod jest wykonywany względem bazy danych, z którą jesteś połączony w tym momencie. Bazę danych można wybrać z listy rozwijanej Available Databases, pokazanej na rysunku A-12.



RYСУNEK A-12 Rozwinięta lista Available Databases

7. Upewnij się, że jesteś połączony z bazą przykładową *TSQLV4*.

Zwróć uwagę, że w dowolnym momencie możesz zmienić serwer i bazę danych, z którą jesteś połączony, klikając prawym klawiszem myszy w pustym obszarze okna zapytań i wybierając Connection | Change Connection z menu podręcznego.

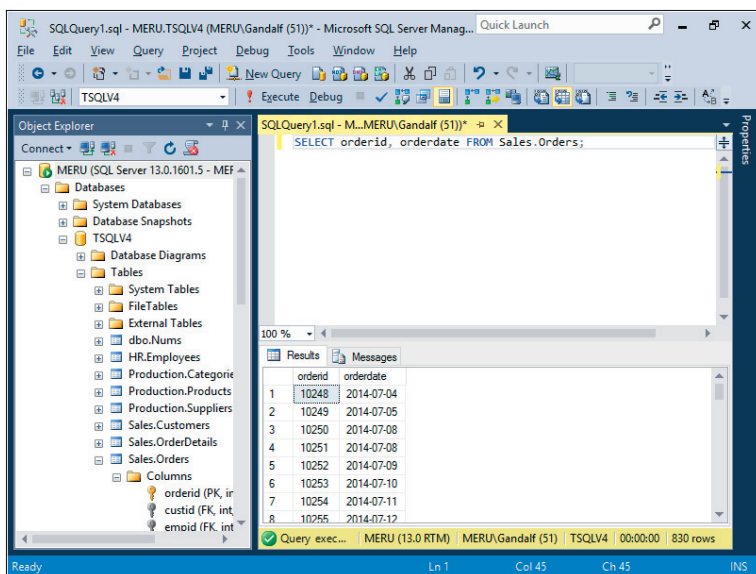
8. Teraz można już zacząć tworzyć kod T-SQL. Na początek wpisz następujący kod w oknie zapytania:

```
SELECT orderid, orderdate FROM Sales.Orders;
```

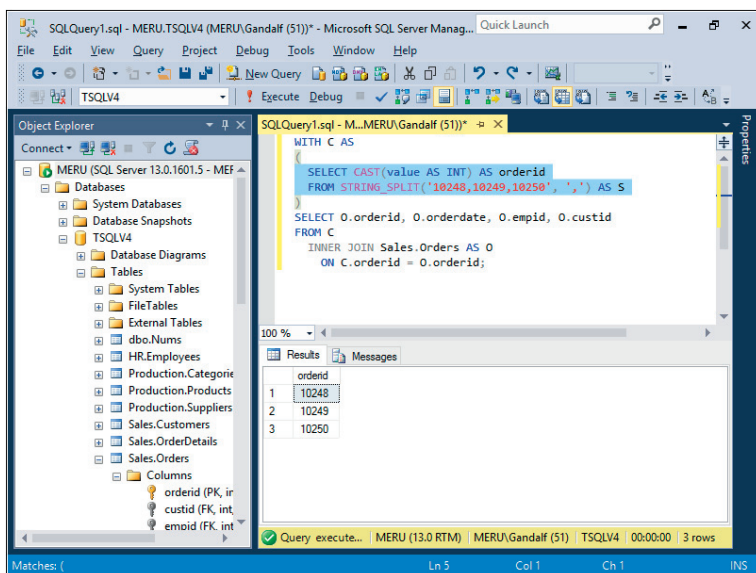
9. Naciśnij F5, aby wykonać wpisany kod. Alternatywnie możesz kliknąć przycisk Execute (ikonę z czerwonym wykrzyknikiem). Pojawi się panel Results z wynikami tego zapytania, jak pokazano na rysunku A-13.

Miejsce docelowe danych wynikowych można kontrolować przy użyciu polecenia Query | Results To (zapytanie | wyniki do) lub klikając odpowiednie ikony w pasku narzędzi SQL Editor. Dostępne są następujące opcje: Results To Grid (wyniki w siatce, opcja domyślna), Results To Text (wyniki jako tekst) oraz Results To File (wyniki do pliku).

Zwróć uwagę, że jeśli jakaś część kodu jest zaznaczona, tak jak na rysunku A-14, przy wywołaniu wykonania SQL Server przetworzy tylko zaznaczoną część. SQL Server wykonuje cały kod zawarty w skrypcie tylko wtedy, jeśli żaden fragment nie jest zaznaczony.



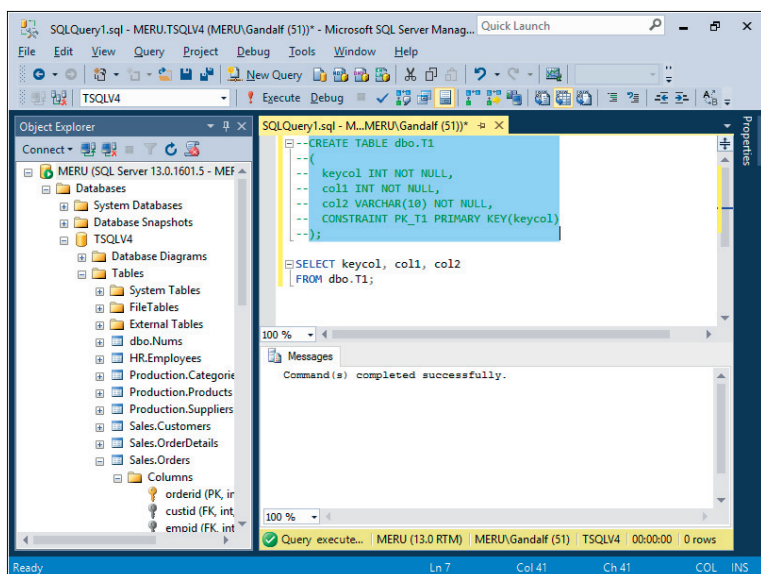
RYSUNEK A-13 Wykonanie pierwszego zapytania



RYSUNEK A-14 Wykonywanie tylko zaznaczonego kodu



WSKAZÓWK Jeśli naciśniesz i przytrzymasz przycisk Alt przed rozpoczęciem zaznaczania kodu, można zaznaczyć prostokątny blok, który nie musi się zaczynać na początku wierszy kodu, jak na rysunku A-15, co może być przydatne przy kopiowaniu lub wykonywaniu specyficznych fragmentów. Naciśnięcie klawiszy Tab lub Shift+Tab powoduje przesunięcie całego prostokąta do przodu lub wstecz. Jeśli w tym momencie zaczniesz coś pisać, tekst ten zostanie powtórzony we wszystkich zaznaczonych wierszach. Wypróbuj to!



RYСУNEK A-15 Zaznaczanie prostokątnego bloku

Na koniec, zanim zabierzesz się za samodzielne poznawanie możliwości i działania narzędzia SSMS, chciałbym przypomnieć, że cały kod źródłowy dla tej książki jest dostępny do pobrania z witryny powiązanej z książką. Szczegóły zawiera wcześniejszy punkt tego dodatku, „Pobieranie kodu źródłowego i instalowanie przykładowej bazy danych”. Zakładając, że pobrałeś kod źródłowy i rozpakowałeś pliki do folderu na dysku lokalnym, każdy z plików skryptów można otworzyć przy użyciu polecenia menu File | Open | File lub klikając ikonę Open File w standardowym pasku narzędzi. Alternatywnie można podwójnie kliknąć plik skryptu w Eksploratorze plików, co spowoduje otwarcie go w narzędziu SSMS.

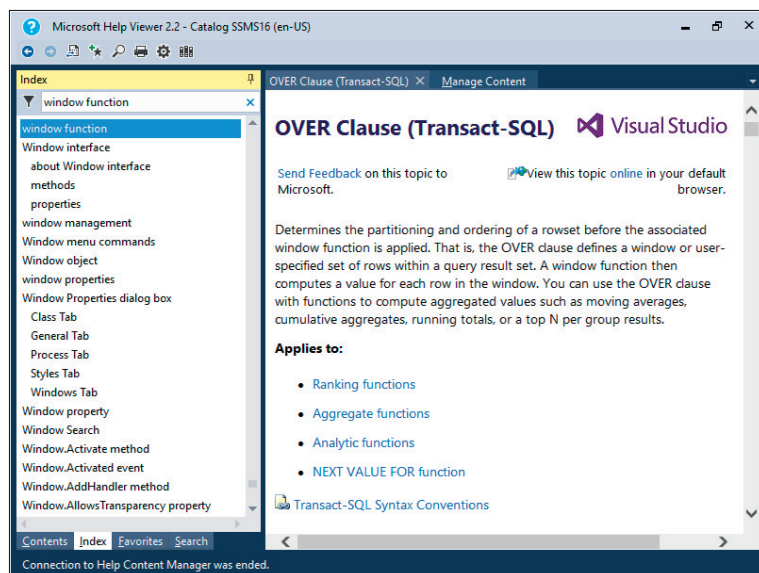
Korzystanie z SQL Server Books Online

Microsoft SQL Server Books Online to dostępna online dokumentacja SQL Server udostępniana przez firmę Microsoft. Zawiera ona ogromną liczbę użytecznych informacji. Podczas projektowania kodu T-SQL powinniśmy myśleć o Books Online jako najlepszym przyjacielu – oczywiście po tej książce o podstawach T-SQL.

Dostęp do Books Online można uzyskać z menu Help narzędzia SSMS, klikając View Help. Domyślnie SSMS pobiera zawartość z Internetu. Możliwe jest również zainstalowanie dokumentacji lokalnie i przeglądanie jej przy użyciu narzędzia Help Viewer. W tym celu należy wybrać polecenie Add and Remove Help Content (dodawanie i usuwanie zawartości pomocy) w menu Help lub nacisnąć klawisze Ctrl+Alt+F1. Przykładowo ja zainstalowałem wszystkie elementy, które w nazwie zawierają akronim SQL.

Nauka posługiwania się Books Online nie jest zbyt skomplikowana i nie zamierzam obrażać inteligencji Czytelników, objaśniając rzeczy oczywiste. Zamieszczenie punktu poświęconego Books w tym dodatku wynika raczej z chęci zwrócenia uwagi na istnienie tej wspaniałej dokumentacji i podkreślenie jej ważności, a nie wyjaśnienia, jak z niej korzystać. Zbyt często zdarza się, że ktoś pyta innych użytkowników o jakieś zagadnienie dotyczące SQL Server, gdy odpowiedź można bez wysiłku znaleźć w Books Online.

Pokażę jednak kilka sposobów, jakimi można wyszukiwać informacje w Books Online. Jednym z okien, którego najczęściej używam w programie Help Viewer do wyszukiwania informacji, jest zakładka Index, pokazana na rysunku A-16.

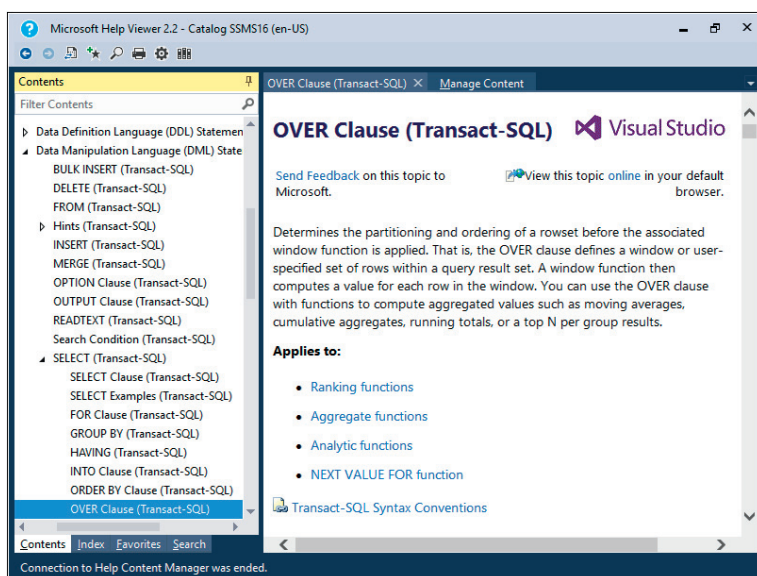


RYSUNEK A-16 Okno Index narzędzia Help Viewer

W polu wyszukiwania wpisujemy jedno lub kilka słów kluczowych dotyczących poszukiwanego zagadnienia. W miarę wpisywania tematu (na przykład **window function**) Help Viewer umieszcza na początku posortowanej listy zagadnień najlepiej pasujące tematy. Przykładowo można wpisać któreś ze słów kluczowych T-SQL, dla którego szukamy informacji o składni, ale także inne zagadnienia.

Znaleziony temat można dodać do listy Help Favorites, klikając przycisk Add To Favorites w pasku narzędzi, dzięki czemu łatwiej będzie do niego wrócić w przyszłości. Można również powiązać bieżący element pomocy z odpowiednim tematem w karcie Content (zawartość), klikając przycisk Show Topic In Contents.

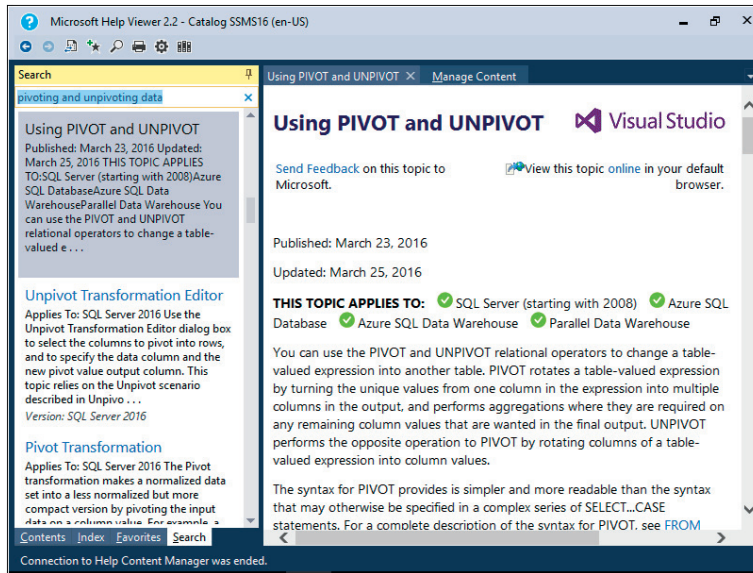
Można wreszcie szukać zagadnienia w samej karcie Contents, przechodząc drzewo tematów pokazane na rysunku A-17.



RYСУNEK A-17 Okno Contents narzędzia Help Viewer

Jeśli chcesz po prostu przejrzeć, co jest dostępne w języku T-SQL, należy przejść do drzewa folderów Transact-SQL Reference, które znajduje się wewnątrz folderu SQL Shared Language Reference folder.

Innym użytecznym narzędziem jest okno Search, pokazane na rysunku A-18. Okno to pozwala wyszukiwać artykuły na podstawie występujących w nich słów. Poszukiwania te są bardziej abstrakcyjne, niż przeszukiwania indeksu – przypominają raczej korzystanie z wyszukiwarki internetowej. Zauważmy wreszcie, że jeśli chcemy znaleźć określone słowo w aktualnie otwartym artykule, należy kliknąć przycisk Find In Topic w pasku narzędzi lub nacisnąć klawisze Ctrl+F, aby aktywować pasek Find.



RYСУNEK A-18 Okno Search narzędzia Help Viewer

WSKAZÓWKA Na koniec chciałbym dodać jeszcze jedną radę. Jeśli potrzebujesz pomocy na temat elementu składni w trakcie pisania kodu w SQL Server Management Studio, upewnij się, że kursor znajduje się gdzieś w tym elemencie kodu i naciśnij F1. Spowoduje to załadowanie Books Online i otwarcie strony opisującej składnię dla tego elementu, oczywiście o ile taki temat pomocy w ogóle istnieje.



Informacje o autorze



ITZIK BEN-GAN jest wykładowcą i współzałożycielem firmy SolidQ, firmy szkoleniowej i konsultingowej koncentrującej się na zagadnieniach baz danych. Już w roku 1999 uzyskał tytuł Microsoft MVP w dziedzinie SQL Server i od tego czasu na całym świecie prowadzi wiele szkoleń poświęconych programowaniu oraz konstruowaniu i dostrajaniu zapytań w języku T-SQL. Jest autorem i współautorem kilku książek na temat języka T-SQL. Opublikował także szereg artykułów dla magazynu *SQL Server Pro*, a także dla MSDN i *The SolidQ Journal*. Itzik Ben-Gan często występuje na konferencjach technicznych, jak Tech-Ed, SQL PASS, SQL

Server Connections, prowadzi własne prezentacje dla różnych grup użytkowników SQL Server i uczestniczy w wydarzeniach zorganizowanych przez SolidQ. Opracował i przeprowadza regularnie na całym świecie kursy SolidQ's Advanced T-SQL i T-SQL Fundamentals. Więcej informacji można znaleźć na jego stronie: <http://tsql.solidq.com/>.

Indeks

Symbole i cyfry

- (odejmowanie) 60-61
- (ujemne) 60-61
- % (procent), symbol wieloznaczny 84-85
- () , nawiasy 60-61
- * (gwiazdka) w klauzuli SELECT 49
- * (mnożenie) 60-61
- + (dodatnie) 60-61
- + (dodawanie) 60-61
- + (złączanie ciągów) 60-61
- = (przypisanie) 60-61
- =, >, <, >=, <=, <>, !=, !>, !< (operatory porównywania) 60-61
- @identity, funkcja 287-289
- [^lista znaków lub zakres], symbol wieloznaczny 87
- [lista znaków], symbol wieloznaczny 85
- [znak-znak], symbol wieloznaczny 85-87
- _ (podkreślenie), symbol wieloznaczny 85
- 1NF, pierwsza postać normalna 10
- 2NF, druga postać normalna 10-13
- 3NF, trzecia postać normalna 11-13

A

- ABC, odmiany (Appliance, Box, Cloud) 15-18
- ACID 358-361
- administracyjne zadania a dynamiczny kod SQL 423-428
- agregujące funkcje. *Zobacz także* COUNT; przestawianie danych; funkcje okna a znaczniki NULL 40-43
- agregacje kroczące a podzapytania 168-169
- agregujące funkcje okna 251-254
- aliasy
 - a klauzula SELECT 44-49
 - a tabele pochodne 186-189
 - a złączenia krzyżowe 122
- klauzula ORDER BY 50
- samozłączenia 124
- tabele pochodne i zagnieżdżanie 190-191
- wbudowane funkcje tablicowe (TVF) 207
- wspólne wyrażenia tablicowe (CTE), przypisywanie w 192-193
- ALL, a duplikaty 222
- ALTER DATABASE, identyfikatory w cudzysłowach 73-75
- ALTER PROC 429
- ALTER SEQUENCE 291
- ALTER TABLE
 - definiowanie integralności danych 27-31
 - obiekty sekwencji 293-296
- ALTER TABLE ADD CONSTRAINT 297-300
- ALTER TABLE DROP CONSTRAINT 297-300
- ALTER VIEW 199
- American National Standards Institute (ANSI) 3
- analiza dla punktu w czasie. *Zobacz* temporalne tabele
- analizowanie kodu, wsady 406
- AND
 - pierwszeństwo 60-61
 - predykaty 58
 - stosowanie 59
 - w instrukcji MERGE 310
- ANSI (American National Standards Institute) 3
- Appliance, Box, Cloud (odmiany ABC) 15-18
- APPLY
 - odwrotne przestawianie danych 261-264
 - wyrażenia tablicowe, omówienie 207-211

argumenty, zapytania do tabel pochodnych
i 189-190

AS, klauzula 44-49

a tabelę pochodną 185-186

AT TIME ZONE 99-100

zapytania do tabel temporalnych 347

atomowość transakcji 358-361

atrybut nie tworzący klucza, postaci
normalne 10-13

atrybuty

filtrowanie w złączeniach

zewnętrznych 138-139

formy normalne 9-13

tezy, predykaty i relacje 7-8

atrybuty zbiorów

postaci normalne 9-13

tezy, predykaty i relacje 7-8

użycie terminu 7

automatyzowanie zadań administracyjnych

a dynamiczny kod SQL 423-428

autonumerowanie

przypisujące UPDATE 305-306

AVG 40-43

Zobacz także funkcje okna

Azure SQL Data Warehouse 18

Azure SQL Database

ABC, odmiany (Appliance, Box,
Cloud) 16-18

blokady i blokowanie 361-362

globalne tabele tymczasowe 419

izolacja oparta na wersjonowaniu
wierszy 380

izolacja, omówienie 372-374

łącze witryny i informacje
dostępowe 441-442

READ COMMITTED SNAPSHOT 384

źródłowy kod, instalowanie 448-450

B

bazodanowe systemy, typy 13-15

bazy danych SQL Server

omówienie architektury 19-23

rozszerzenia nazw plików (.mdf, .ldf, .ndf) 23

schematy i obiekty 23-24

BEGIN 411

BEGIN TRAN (TRANSACTION) 357-361

BETWEEN

korzystanie z 58

pierwszeństwo 60-61

zapytania do tabel temporalnych 345

bezpieczeństwo

procedury składowane 429

schematy i obiekty bazy danych SQL
Server 23-24

bitemporalne tabele 333

Zobacz także temporalne tabele

blocking_session_id 369-370

blokady. *Zobacz także* izolacja transakcji

ćwiczenia 391-402

możliwe do blokowania typy

zasobów 363-364

rozwiązywanie problemów 364-372

tryby i kompatybilność 361-363

zakleszczenia 378, 388-391

blokowanie. *Zobacz* izolacja transakcji;
blokady

brakujące wartości 8-9

w złączeniach zewnętrznych 136-138

brudne odczyty. *Zobacz* izolacja transakcji

BULK INSERT 286

C

CACHE, obiekty sekwencji i 290-291

całość, definicja 4

Cancel Executing Query, polecenie 370-372

Cantor, Georg 4

CASCADE 29

CASE, wyrażenia

omówienie cech 61-64

przestawianie w zapytaniach
grupujących 256-257

CAST

funkcje daty i czasu 96-98

literały daty i czasu 91-92

skorelowane podzapytania i 165

CATCH, obsługa błędów 436-440

CHAR 59, 71-72

CHARINDEX 78

CHECK

CASE, wyrażenia 61-64
 korzystanie oddzielnie z dat i czasu 92-94
 CHECK OPTION, w widokach 204-206
 chmurowe przetwarzanie, a odmiany
 ABC 16-18

CHOOSE, wyrażenia CASE 64

COALESCE, wyrażenia CASE 64

Codd, Edgar F. 6-7

COLLATE 72-73

COLUMNPROPERTY 109

COMMIT TRAN

transakcje, omówienie 357-361
 tryby blokad i kompatybilność 361-363

COMPRESS 82

CONCAT 75-77

CONSTRAINT 297-300

CONTINUE 412-413

CONVERT

funkcje daty i czasu 96-98
 korzystanie oddzielnie z dat i czasu 94
 literały daty i czasu 91-92

COUNT. *Zobacz także* funkcje okna

GROUP BY i 40-43

zewnętrzne złączenia, używanie w 141-144

CREATE DATABASE, wyzwalacze DDL
 (zdarzenia definicji danych) 433-436

CREATE DEFAULT 408

CREATE FUNCTION 408

CREATE PROCEDURE 408

CREATE RULE 408

CREATE SCHEMA 408

CREATE SEQUENCE 290-291

CREATE TABLE

definiowanie integralności danych 27-31
 korzystanie z 25
 wyzwalacze DDL (zdarzenia definicji
 danych) 433-436

CREATE TRIGGER 408

CREATE VIEW 408

CROSS APPLY

a JOIN 207
 odwrotne przestawianie danych 262-264
 omówienie 207-211

CUBE 266-268

CURRENT_TIMESTAMP 95-96

CYCLE w obiektach sekwencji 290-291

Ć

ćwiczenia

DELETE 323-327

DELETE, rozwiązania 327-331

funkcje okien 272-275

funkcje okien, rozwiązania 277-280

INSERT 323-327

INSERT, rozwiązania 327-331

JOIN 144-149

JOIN, rozwiązania 149-152

MERGE 323-327

MERGE, rozwiązania 327-331

OFFSET-FETCH 323-327

OFFSET-FETCH, rozwiązania 327-331

OUTPUT 323-327

OUTPUT, rozwiązania 327-331

podzapytania 175-179

podzapytania, rozwiązania 179-184

przestawianie i odwrotne przestawianie
 danych 272-275

przestawianie i odwrotne przestawianie
 danych, rozwiązania 277-280

temporalne tabele 348-351

temporalne tabele, rozwiązania 351-355

TOP 323-327

TOP, rozwiązania 327-331

transakcje, poziomy izolacji
 i blokady 391-402

UPDATE 323-327

UPDATE, rozwiązania 327-331

wyrażenia SELECT 110-115

wyrażenia SELECT, rozwiązania 115-120

wyrażenia tablicowe 211-216, 323-327

wyrażenia tablicowe, rozwiązania 216-220,
 327-331

zbiory grupujące 272-275

zbiory grupujące, rozwiązania 277-280

ćwiczenia, zasoby do wykorzystania

Azure SQL Database, rozpoczynanie
 pracy 441-442

ćwiczenia, zasoby do wykorzystania (cd.)

instalowanie SQL Server 442-447
 SQL Server Books Online 456-459
 SQL Server Management Studio, korzystanie
 z 450-456
 źródłowy kod, pobieranie
 i instalowanie 448-450

D

dane czasowe

agregujące funkcje okna 253-254
 AT TIME ZONE 99-100
 brakujące wartości w złączeniach
 zewnętrznych 136-138
 CAST, CONVERT i PARSE, funkcje 96-98
 DATEADD 100
 DATEDIFF i DATEDIFF_BIG 101-102
 DATENAME 103
 DATEPART 102
 EOMONTH 106
 FROMPARTS 103-106
 funkcje bieżącej daty i czasu 95-96
 ISDATE 103
 korzystanie oddzielnie z dat i czasu 92-94
 literały 88-92
 SWITCHOFFSET 98
 typy 87-88
 YEAR, MONTH i DAY 102-103

dane daty i czasu

agregujące funkcje okna 253-254
 AT TIME ZONE 99-100
 bieżąca data i czas, funkcje 95-96
 brakujące dane w złączeniach
 zewnętrznych 136-138
 CAST, CONVERT i PARSE 96-98
 DATEADD 100
 DATEDIFF oraz DATEDIFF_BIG 101-102
 DATENAME 103
 DATEPART 102
 EOMONTH 106
 filtrowanie zakresów dat 94-95
 FROMPARTS 103-106
 ISDATE 103
 korzystanie oddzielnie 92-94

literały 88-92
 SWITCHOFFSET 98
 typy 87-88
 YEAR, MONTH oraz DAY 102-103

dane znakowe

identyfikatory w cudzysłowach 73-75
 opcje sortowania 72-75
 typy danych, omówienie 71-72

Darwen, Hugh 6-7

Data Lake 18

DATABASEPROPERTYEX 109

DATALLENGTH 77-78

DATE

korzystanie oddzielnie z dat i czasu 92-94
 literały 88-92
 typy danych 87-88

Date, Chris 6-7

DATEADD 100

DATEDIFF i DATEDIFF_BIG 101-102

DATEFORMAT 89

DATENAME 103

DATEPART 102

DATETIME

korzystanie oddzielnie z dat i czasu 92-94
 literały 88-92
 typy danych 87-88

DATETIME2

literały 88-92
 temporalne tabele, tworzenie 334-338
 typy danych 87-88

DATETIMEOFFSET

a SWITCHOFFSET 98
 literały 88-92
 typy danych 87-88

DAY 102-103

DBCC CHECKIDENT 289

DEADLOCK_PRIORITY 388

DECLARE

tymczasowe zmienne tablicowe 421-422
 zmienne i 403

DECOMPRESS 82

deklaratywna integralność danych 27-31

DELETE

ćwiczenia 323-327

- DML, wyzwalacze (zdarzenia manipulacji danymi) 432-433
- omówienie 296-300
- OUTPUT i 318-319
- poziomy izolacji z wersjonowaniem wierszy 380
- rozwiązania ćwiczeń 327-331
- widoki i 199
- wyrażenia tablicowe, modyfikowanie danych przy użyciu 311-313
- złączenia i 299-300
- DENSE_RANK, rankingowa funkcja okna 244-248
- DESC 52
- DISTINCT
 - a funkcje agregujące 41-43
 - a klauzula ORDER BY 52
 - duplikaty 47-49, 222
 - EXCEPT (DISTINCT) 228-229
 - funkcje okna, ustalanie rankingu 246-248
 - INTERSECT (DISTINCT) 225
 - UNION (DISTINCT) 222-225
 - wielowartościowe podzapytania 160
- DML. *Zobacz* język manipulacji danymi (DML)
- dodatni (+) 60-61
- dodawanie (+) 60-61
- dopełnianie, klauzula ORDER BY 54-55
- DROP CONSTRAINT a obiekt sekwencji 293-296
- DROP IF EXISTS, tworzenie tabel 25
- duplikaty a klauzula DISTINCT 47-49, 222
- DW (data warehouse, hurtownia danych) 13-15
- dynamiczne instrukcje SQL 423-428
- dziennik transakcji a trwałość danych 359
- E**
- ekstrakcja, transformacja i ładowanie (ETL) 15
- elementy sterowania przepływem wykonania 410-413
- IF...ELSE 410-411
- WHILE 411-413
- ELSE
 - IF...ELSE, element sterowania przepływem 410-411
 - wyrażenia CASE, omówienie 61-64
- ELSE NULL, wyrażenia CASE 61-64
- ENCRYPTION a widoki 202-203
- END 411
- Entity Relationship Modeling (ERM) 9
- EOMONTH 106
- ERROR_LINE 437
- ERROR_MESSAGE 437
- ERROR_NUMBER 437
- ERROR_PROCEDURE 437
- ERROR_SEVERITY 437
- ERROR_STATE 437
- ESCAPE, znak 87
- ETL (ekstrakcja, transformacja i ładowanie) 15
- EVENTDATA, wyzwalacze DDL (zdarzenia definicji danych) 433-436
- EXCEPT
 - ćwiczenia 234-237
 - korzystanie z 228-230
 - pierwszeństwo 230-231
 - rozwiązania ćwiczeń 237-240
- EXCEPT (DISTINCT) 228-229
- EXCEPT ALL 228-230
- EXEC
 - dynamiczne instrukcje SQL 423-426
 - INSERT EXEC 284-285
- EXISTS
 - a skorelowane podzapytania 165-167
 - podzapytania, problem z NULL 171
- F**
- FALSE
 - a ograniczenia CHECK 29-31
 - NULL, omówienie 64-69
 - podzapytania, problem z NULL 170-171
 - w elemencie sterowania przepływem IF... ELSE 410-411
- WHERE 39
- WHILE, element sterowania przepływem 411-413

filtry. *Zobacz także* podzapytania

filtry TOP, omówienie 53-55

HAVING, omówienie 43-44

logika predykatów, omówienie 6

OFFSET-FETCH, omówienie 55-56

WHERE, omówienie 38-39

WITH TIES 55

zakleszczenia, unikanie 390

zakresy dat 94-95

FIRST_VALUE, offset funkcji okna 248-251

fn_helpcollations 72-75

FOR SYSTEM_TIME CONTAINED

IN 345-347

FOR SYSTEM_TIME, zapytania do tabel

temporalnych 341-347

FOR XML, widoki i klauzula ORDER

BY 200-201

FORMAT 81

FROM. *Zobacz także* JOIN

logiczny porządek przetwarzania

zapytania 34

omówienie 36-37

tabele pochodne 185-186

FROMPARTS 103-106

funkcje

CHARINDEX 78

COMPRESS i DECOMPRESS 82

CONCAT, złączanie ciągów

tekstowych 75-77

FORMAT 81

funkcje dotyczące daty i czasu 95-106

LEFT i RIGHT 77

LEN i DATALENGTH 77-78

PATINDEX 78

REPLACE 78-79

REPLICATE 79-80

RTRIM i LTRIM 81

STRING_SPLIT 84

STUFF 80

SUBSTRING 77

UPPER i LOWER 80-81

funkcje okna

agregujące funkcje okna 251-254

ćwiczenia 272-275

offsetowe funkcje okna 248-251

omówienie 56-58, 241-244

rankingowe funkcje okna 244-248

rozwiązania ćwiczeń 277-280

funkcje zdefiniowane przez użytkownika

(UDF), jako procedury 428-429

funkcje zwracające tabelę (TVF),

wbudowane (inline) 198-199

G

GENERATED ALWAYS AS ROW

END 334-338

GENERATED ALWAYS AS ROW

START 334-338

generowanie globalnie unikatowych

identyfikatorów (GUID), a INSERT

SELECT 284

GETDATE 95-96

GETUTCDATE 95-96

globalne tabele tymczasowe 419-421

głównego klucza, ograniczenie, integralność
danych 27

GO 406-410

GROUP BY. *Zobacz także* zbiory grupujące

CUBE 268

GROUPING SETS 266-268

HAVING i 43-44

logiczny porządek przetwarzania 34

omówienie 39-43

pochodne tabele, przypisywanie

aliasów 187-189

przstawianie danych w zapytaniach

grupujących 256-257

ROLLUP 268-269

GROUPING 266, 269-272

GROUPING SETS 266-268

GROUPING_ID 266, 269-272

gwiazda, schemat hurtowni danych 14-15

gwiazdka (*) w klauzuli SELECT 49

GZIP, algorytm 82

H

HAVING

GROUP BY i 40

logiczny porządek przetwarzania
 zapytania 34
 omówienie 43-44
 wyrażenia CASE, omówienie 61-64

HDInsight 16

historyczne dane. *Zobacz* temporalne tabele

HOLDLOCK 372

hurtownie danych 13-15

I

IaaS (infrastruktura jako usługa)

ABC, odmiany (Appliance, Box,
 Cloud) 16-18

ID sesji, rozwiązywanie problemów
 z blokowaniem 365-372

IDENT_CURRENT 287-289

identity, właściwość 286-289

IDENTITY_INSERT 288-289

identyfikatory w cudzysłowach 73-75

identyfikatory, nazwy

oddzielanie 37

w cudzysłowach 73-75

IF...ELSE, element sterowania
 przepływem 410-411

IIF, wyrażenia CASE 64

IMPLICIT_TRANSACTIONS 357-358

IN

korzystanie z 58

pierwszeństwo 60-61

podzapytania, problem z NULL 170-171

wielowartościowe podzapytania,
 przykłady 158-162

INCREMENT BY, w obiektach
 sekwencji 290-291

informacyjne widoki schematu, zapytania do
 metadanych 108

INFORMATION_SCHEMA 108

infrastruktura jako usługa (IaaS) 16-18

In-Memory OLTP 23

INSERT

a MERGE 309

a OUTPUT 316-318

ćwiczenia 323-327

poziomy izolacji z wersjonowaniem 380

rozwiązania ćwiczeń 327-331

widoki i 199

właściwość identity 286-289

wyrażenia tablicowe, modyfikowanie danych
 przy użyciu 311-313

wyzwalacze DML (zdarzenia manipulacji
 danymi) 432-433

INSERT EXEC 284-285

INSERT SELECT 284

INSERT VALUES 281-284

inspekcja, wyzwalacze DDL (zdarzenia
 definicji danych) 433-436

Zobacz także temporalne tabele

INT 60

integralność danych

definiowanie 27-31

ograniczenia 9

International Organization for
 Standardization (ISO) 3

INTERSECT

ćwiczenia 234-237

korzystanie z 225-228

pierwszeństwo 230-231

rozwiązania ćwiczeń 237-240

INTERSECT (DISTINCT) 225

INTERSECT ALL 225-228

INTO, SELECT INTO 285-286

IS NOT NULL 65-69

IS NULL

omówienie 65-69

wyrażenia CASE 64

ISDATE 103

ISO (International Organization for
 Standardization) 3

ISO/ANSI SQL-89, składnia

złączenia krzyżowe 123

złączenia wewnętrzne 127-129

ISO/ANSI SQL-92, składnia

złączenia krzyżowe 122

złączenia wewnętrzne 126-127

izolacja transakcji 358-361

ćwiczenia 391-402

omówienie 372-374

READ COMMITTED 375-378

izolacja transakcji (*cd.*)

READ COMMITTED SNAPSHOT 384-388
 READ UNCOMMITTED 374-375
 REPEATABLE READ 377-378
 SERIALIZABLE 378-379
 SNAPSHOT 380-384
 SNAPSHOT, wykrywanie konfliktów 382-384
 zakleszczenia 388-391

J

język definicji danych (DDL) a wsady 409

język manipulacji danymi (DML) 281

wsady i 409

zagnieżdżanie DML 321-323

JOIN

CROSS APPLY i 207

ćwiczenia 144-149

DELETE i 299-300

kontra podzapytania 159

naturalne złączenia 130

omówienie 122

OUTER APPLY i 208

pochodne tabele, wielokrotne odwoływanie się do 191

rozwiązania ćwiczeń 149-152

samo-złączenia 123-124

UPDATE i 302-305

zapytania z wielokrotnymi złączeniami 133

złączenia krzyżowe 122-126

złączenia nierównościowe 130-131

złączenia wewnętrzne 126-129

złączenia zewnętrzne

agregacja COUNT, wykorzystanie 141-144

brakujące wartości, dołączanie 136-138

filtrowanie atrybutów z niezachowanej strony 138-139

podstawy 133-136

zapytania z wieloma złączeniami 139-141

złączenia złożone 129-130

K

kandydujące klucze

Zobacz także ograniczenia

normalne formy 10-11

omówienie cech 9

kandydujące klucze, właściwość

identity 286-289

katalogowe widoki, zapytania do

metadanych 106-108

klucze a zapytania SQL 47-49

kolejność porządkowania w klawiuli ORDER BY 52

kompresowanie danych znakowych 71-72

konstruktory wierszy 305

kroczące agregacje, podzapytania i 168-169

krotki 6-7

krzyżowe złączenia

a CROSS APPLY 207

ćwiczenia 144-149

omówienie 122-126

rozwiązania ćwiczeń 149-152

kursory 50

jako obiekty programowalne 413-416

L-Ł

LAG 422

offset funkcji okna 248-251

LANGUAGE/DATEFORMAT 89-90

LAST_VALUE, offset funkcji okna 248-251

.ldf (Log Data File) 23

LEAD, offset funkcji okna 248-251

LEFT 77

LEN 77-78

LIKE

korzystanie z 59, 84-87

pierwszeństwo 60-61

literały dla daty i czasu 88-92

LOCK_ESCALATION 364

LOCK_TIMEOUT 370-372

login uwierzytelniany przez SQL Server 21

login uwierzytelniany przez system

Windows, bazy danych SQL Server 21

login, bazy danych SQL Server 21

lokalne tabele tymczasowe 417-418

LOWER 80-81

LTRIM 81

łańcuch blokad 367

M

magazynowanie, dane znakowe i 71-72
masowe przetwarzanie równoległe (MPP) 16
master, baza danych, architektura SQL
 Server 20-23
MAX 40-43, 72
MAXVALUE w obiekcie sekwencji 290-291
.mdf (Master Data File) 23
MERGE
 ćwiczenia 323-327
 DML, wyzwacze (zdarzenia manipulacji danymi) 432-433
 OUTPUT i 320-321
 rozwiązania ćwiczeń 327-331
 scalanie danych, omówienie 306-310
 wyrażenia tablicowe, modyfikowanie danych przy użyciu 311-313
metadane, zapytania do 106-109
Microsoft Analytics Platform System (APS) 16-18
Microsoft Azure SQL Data Warehouse 18
Microsoft Azure SQL Database. *Zobacz* Azure SQL Database
Microsoft SQL Server. *Zobacz* SQL Server
MIN 40-43
minihurtownia danych 14-15
MINVALUE w obiekcie sekwencji 290-291
mnożenie (*) 60-61
model relacyjny
 brakujące wartości 8-9
 normalizacja 9-13
 ograniczenia 9
 omówienie 6-13
 tezy, predykaty i relacje 7-8
model, baza danych, architektura SQL
 Server 20-23
MONTH 94-95, 102-103
msdb, baza danych, architektura SQL
 Server 20-23
MTD, agregująca funkcja okna 253-254

N

N (National), prefiks typu danych 59, 71-72

następne wartości, zwracanie
 w podzapytaniu 167-168
naturalne złączenia 130
nawiasy () 60-61
nazwy kolumn
 aliasy, przypisywanie we wspólnych wyrażeniach tablicowych (CTE) 192-193
 błędy podstawiania
 w podzapytaniach 171-174
 nazwy identyfikatorów 37
 pochodne tabele i 186
nazwy obiektów, kwalifikowanie przy użyciu schematu 36
nazwy, schematy i obiekty 23-24
 Zobacz także aliasy
NCHAR 59, 71-72
.ndf (Not Master Data File) 23
NEWID 284, 428
NEXT VALUE FOR 291-296
niepowtarzalne odczyty 377, 385-388
nierównościowe złączenia. *Zobacz*
 równościowe i nierównościowe złączenia
niespójna analiza 377, 385-388
niestandardowe sekwencje, przypisania
 UPDATE i 305-306
niezależność od języka 3
niezatwierdzone odczyty 374-375
nieznane, brakujące wartości 8-9
NO ACTION, ograniczenia klucza obcego 29
NOLOCK 372
normalizacja 9-13
 kończenie wyrażen średnikiem 36
 modele relacyjne, niezależność od języka 3
 standardy do użycia 3
 teoria zbiorów, omówienie 4
 tezy, predykaty i relacje 7-8
NOT
 łączenie wyrażen logicznych 59
 pierwszeństwo 60-61
 podzapytania, problem z NULL 170-171
 wielowartościowe podzapytania i 159
NOT EXISTS, problem z NULL 171
NOT IN, problem z NULL 170-171
NOT NULL a grupowanie 269-272

NTILE, rankingowa funkcja okna 244-248
 NULL

- brakujące wartości, omówienie 8-9
- DISTINCT i duplikaty 222
- funkcje agregujące 41-43
- GROUPING i GROUPING_ID 269-272
- IF...ELSE, element sterowania przepływem 410-411
- INSERT VALUES i 282
- integralność danych 27-31
- INTERSECT i 225
- omówienie 64-69
- podzapytania, problem z 169-171
- scalanie ciągów tekstowych 75-77
- SET NULL 29
- skalarne podzapytania i 157
- tabele, tworzenie 26
- wartości zmiennych i 406
- wielowartościowe podzapytania 159
- wyrażenia CASE 256
- zewnętrzne złączenia w wielokrotnych złączeniach 139-141
- zewnętrzne złączenia, filtrowanie atrybutów 138-139
- zewnętrzne złączenia, podstawy 134-136

NUMERIC 60

NVARCHAR 59, 71-72

O

obcego klucza, ograniczenie. *Zobacz także*
 ograniczenia
 integralność danych, definiowanie 28-29
 omówienie 9

obiekt sekwencji 290-296

obiekt, definicja 4

obiekty programowalne

- dynamiczne instrukcje SQL 423-428
- elementy sterowania przepływem wykonania 410-413
- kursory 413-416
- obsługa błędów 436-440
- procedury 428
 - funkcje zdefiniowane przez użytkownika 428-429

- procedury składowane 429-430

- wyzwalacze 432-436

- tabele tymczasowe 417-423

- wsady i 406-410

- zmiennne 403-406

obiekty w architekturze SQL Server 23-24

- Zobacz także* obiekty programowalne

OBJECT_ID, tworzenie tabel 25

OBJECTPROPERTY 109

obsługa błędów

- procedury składowane 429

- programowalne obiekty 436-440

- transakcje, omówienie 358-361

- zakleszczenia 390-391

odczyty fantomowe 378-379

- SNAPSHOT 380

odejmowanie (-) 60-61

odwołania, tabela w ograniczeniu klucza

- obcego 28-29

odwołująca się tabela w ograniczeniu klucza

- obcego 28-29

odwrotna kolejność dostępu,

- zakleszczenie 390

odwrotne przestawianie danych 260-265

- ćwiczenia 272-275

- przy użyciu APPLY 261-264

- przy użyciu UNPIVOT 264-265

OFFSET-FETCH

- ćwiczenia 323-327

- modyfikowanie danych przy

- użyciu 313-316

- omówienie 55-56

- rozwiązania ćwiczeń 327-331

- tabele pochodne 186

- widoki i klauzula ORDER BY 200-201

offsetowe funkcje okna 248-251

ograniczenia

- CHECK, definiowanie 29-31

- klucza głównego, definiowanie 27

- klucza obcego, definiowanie 28-29

- omówienie 9

- unikalności, definiowanie 28

- wartości domyślnej, definiowanie 31

ograniczenia domyślne, definiowanie 31

okresowe tabele oparte na czasie
aplikacji 333

Zobacz także temporalne tabele

OLTP (transakcyjne przetwarzanie w trybie
online), bazy danych 13-15

ON

zewnętrzne złączenia w wielokrotnych
złączeniach 139-141

zewnętrzne złączenia, podstawy 134-136

ON DELETE CASCADE 29

opcje sortowania (collation) 20, 72-75

operacje jednoczesne (all-at-once) 69-70

operatory arytmetyczne 59-61

operatory logiczne 59

operatory porównania 59-61

operatory zbiorowe

ćwiczenia 234-237

EXCEPT 228-230

INTERSECT 225-228

nieobsługiwane frazy logiczne, metody
obejścia 232-234

omówienie 221-222

pierwszeństwo 230-231

rozwiązania ćwiczeń 237-240

UNION 222-225

operatory. *Zobacz także* APPLY; JOIN; PIVOT;

UNPIVOT

omówienie 58-61

reguły pierwszeństwa 60-61

scalanie ciągów tekstowych 75-77

złożone operatory a UPDATE 301-302

OR

łączenie wyrażeń logicznych 59

pierwszeństwo 60-61

predykaty i 58

ORDER BY

filtry TOP i 54-55

funkcje okna, omówienie 56-58, 242-243

GROUP BY i 40

INTERSECT ALL i 225-228

logiczny porządek przetwarzania
zapytania 34

obchodzenie nieobsługiwanych fraz
logicznych 232-234

OFFSET-FETCH, omówienie 55-56

offsetowe funkcje okna 249-251

omówienie 49-52

operatory zbiorowe 221

tabele pochodne 186

widoki, wyrażenia tablicowe 199-201

wyrażenia CASE, omówienie 61-64

organizacje standaryzujące 3

OUTER APPLY

a JOIN 208

omówienie 207-211

OUTPUT

ćwiczenia 323-327

DELETE i 318-319

INSERT i 316-318

MERGE i 320-321

omówienie 316

rozwiązania ćwiczeń 327-331

UPDATE i 319-320

zagnieżdżone wyrażenia DML 321-323

OVER

funkcje okna, omówienie 56-58, 241-244

obiekty sekwencji 292-296

puste nawiasy () i 251-252

P

PaaS (platforma jako usługa)

ABC, odmiany (Appliance, Box,
Cloud) 16-18

Parallel Data Warehouse (PDW) 16

parametry tablicowe (TVP) 422-423

PARSE

data i czas, funkcje 96-98

data i czas, literały 91-92

PARTITION BY

funkcje okna, omówienie 56-58, 243

INTERSECT ALL i 225-228

partycje, offsetowe funkcje okna 248-251

PATINDEX 78

PERCENT, filtry TOP 54

PERIOD FOR SYSTEM_TIME, tworzenie
tabeli temporalnej 334-338

pętle, wyrażenie WHILE 411-413

PIVOT

dynamiczne instrukcje SQL 423, 426-428
 przedstawianie danych, omówienie 257-259
 plany wykonania SQL Server 424-426
 platforma jako usługa (PaaS) 16-18
 płatek śniegu, typ wymiarów 14
 podkreślenie (), symbol wieloznaczny 85
 podzapytania niezależne
 definicja 155
 skalarne podzapytania 156-157
 wielowartościowe podzapytania 158-162
 podzapytania. *Zobacz także* wyrażenia
 tablicowe
 agregacje kroczące 168-169
 ćwiczenia 175-179
 funkcje okna 242
 kontra złączenia 159
 nazwy kolumn, błędy podstawień 171-174
 niezależne
 definicja 155
 skalarne podzapytania 156-157
 wielowartościowe podzapytania 158-162
 omówienie 155
 poprzednie lub następne wartości,
 zwracanie 167-168
 problem ze znacznikami NULL 169-171
 rozwiązania ćwiczeń 179-184
 skalarne podzapytania, definicja 155
 skorelowane
 EXISTS i 165-167
 omówienie 162-165
 tablicowe podzapytania, definicja 155
 wielowartościowe podzapytania,
 definicja 155
 pojemnik 4
 PolyBase 16
 połączone tabele historii 334-338
 pomijanie opcji OFFSET-FETCH 55-56
 postaci normalne. *Zobacz* normalizacja
 predykaty. *Zobacz także* poszczególne
 predykaty
 LIKE, predykat 84-87
 logika predykatów 6
 omówienie 58-61

relacyjny model, omówienie 7-8
 znaczniki NULL, omówienie 64-69
 proceduralna integralność danych 27-31
 procedury
 funkcje zdefiniowane przez
 użytkownika 428-429
 omówienie 428
 procedury składowane 429-430
 wyzwalacze 432-436
 procedury składowane 429-430
 wyzwalacze 432-436
 procent (%), symbol wieloznaczny 84-85
 proste wyrażenia CASE 61-64
 prywatna chmura, odmiany ABC 16-18
 przedstawianie danych
 ćwiczenia 272-275
 odwrotne przedstawianie danych 260-265
 omówienie 254-256
 przy użyciu operatora PIVOT 257-259
 rozwiązania ćwiczeń 277-280
 w zapytaniach z grupowaniem 256-257
 przeszukiwane wyrażenia CASE 61-64
 przykłady kodu, zasoby do wykorzystania
 Azures SQL Database, rozpoczynanie
 pracy 441-442
 instalowanie SQL Server 442-447
 SQL Server Books Online 456-459
 SQL Server Management Studio, korzystanie
 z 450-456
 źródłowy kod, pobieranie
 i instalowanie 448-450
 przypisanie (=) 60-61
 przywracanie wcześniejszej zawartości tabel.
 Zobacz temporalne tabele
 publiczna chmura, odmiany ABC 16-18

R

RAND 428
 RANK, rankingowa funkcja okna 244-248
 RDBMS (relacyjne systemy zarządzania
 bazami danych)
 ABC, odmiany (Appliance, Box,
 Cloud) 15-18
 definicja 1

- model relacyjny, omówienie 6-13
 - niezależność od języka 3
 - READ COMMITTED**
 - a SNAPSHOT 381, 384
 - domyślne poziomy izolacji 362
 - izolacja, omówienie 372-378, 388
 - wykrywanie konfliktów, SNAPSHOT 382-384
 - READ COMMITTED LOCKS** 384
 - READ COMMITTED SNAPSHOT**
 - domyślne poziomy izolacji 362, 372-374
 - izolacja, omówienie 380, 384-388
 - wykrywanie konfliktów, SNAPSHOT 382-384
 - READ UNCOMMITTED**, omówienie 372-375, 388
 - relacyjna zmienna, stosowanie terminu 7
 - relacyjne systemy zarządzania bazami danych (RDBMS)
 - ABC, odmiany (Appliance, Box, Cloud) 15-18
 - definicja 1
 - model relacyjny, omówienie 6-13
 - niezależność od języka 3
 - REPEATABLE READ** 372-374, 377-378, 388
 - REPLACE** 78-79
 - REPLICATE** 79-80
 - Resource, baza danych, architektura SQL Server 20
 - RESTART WITH**, obiekt sekwencji 291
 - RIGHT** 77
 - ROLLBACK TRAN (TRANSACTION)**
 - omówienie 357-361
 - tryby blokad i kompatybilność 361-363
 - wyzwalacze 432
 - ROLLUP** 266-269
 - ROW_NUMBER**
 - EXCEPT ALL i 229-230
 - funkcje okna, omówienie 56-58
 - INTERSECT ALL i 225-228
 - rankingowe funkcje okna 244-248
 - ROWS BETWEEN**
 - funkcje okna, omówienie 243
 - offsetowe funkcje okna 249-251
 - rozpoznawanie (wiązanie), we wsadach 409
 - rozdzielczość, definicja 4
 - rozdzielanie wielkości liter, dane znakowe 72
 - rozdzielanie znaków diakrytycznych, dane znakowe 72
 - rozwiązania. *Zobacz* ćwiczenia
 - rozwiązywanie problemów
 - blokada i blokowanie 364-372
 - OUTPUT, korzystanie z 316
 - tablicowe wyrażenia, modyfikowanie danych przy użyciu 311-313
 - równościowe i nierównościowe
 - złączenia 130-131
 - ćwiczenia 144-149
 - rozwiązania ćwiczeń 149-152
 - RTRIM** 81
- S**
- samo-złączenia 123-124
 - scalanie ciągów tekstowych 75-77
 - SCHEMABINDING**, widoki i 203-204
 - schematy
 - architektura SQL Server 23-24
 - dbo, schemat 25
 - informacyjne widoki schematu, zapytania do metadanych 108
 - nazwy identyfikatorów 37
 - nazwy obiektów, kwalifikowanie przy użyciu schematu 36
 - SCOPE_IDENTITY** 287-289
 - sekwencje liczb całkowitych a złączenia krzyżowe 124-126
 - SELECT**
 - funkcje okna, omówienie 243-244
 - INSERT SELECT 284
 - przestawianie przy użyciu zapytań grupujących 256-257
 - rankingowe funkcje okna 246-248
 - zmiennie, stosowanie 404-406
 - SELECT * FROM**, widoki i 199
 - SELECT ***, widoki i 199
 - SELECT INTO** 285-286

SELECT, instrukcja. *Zobacz także* wyrażenia
tablicowe

a widoki 199

ćwiczenia 110-115

dane daty i czasu, korzystanie z 87-106

elementy 33-36

filtr OFFSET-FETCH, omówienie 55-56

filtry TOP, omówienie 53-55

FROM, omówienie 36-37

funkcje okna, omówienie 56-58

GROUP BY, omówienie 39-43

HAVING, omówienie 43-44

LIKE, predykat 84-87

logiczny porządek przetwarzania
zapytania 34

metadane, odpytywanie 106-109

operacje jednoczesne (all-at-once) 69-70

operatory i funkcje, omówienie 75-84

ORDER BY, omówienie 49-52

pochodne tabele, przypisywanie
aliasów 187-189

predykaty i operatory, przegląd 58-61

rozwiązania ćwiczeń 115-120

SELECT, omówienie 44-49

WHERE, omówienie 38-39

wyrażenia CASE, omówienie 61-64

znaczniki NULL, omówienie 64-69

znakowe dane, korzystanie z 71-75

SEQUEL (Structured English QUery
Language) 3

SERIALIZABLE

a poziom SNAPSHOT 381

poziomy izolacji 372-374, 378-379, 388

SERVERPROPERTY 109

SET

identyfikatory w cudzysłowach 73-75

UPDATE i 301-302

zmienne 403-406

SET DEFAULT 29

SET NOCOUNT ON 430

SET NULL 29

skalarne podzapytania

definicja 155

niezależne podzapytania, przykłady 156-157

zmienne skalarne 403-406

skalarne UDF (funkcje zdefiniowane przez
użytkownika) 428-429

składnia

wsad jako jednostka analizy 406

złączenia krzyżowe 122-123

złączenia wewnętrzne 126-129

skorelowane podzapytania. *Zobacz także*
podzapytania

EXISTS i 165-167

omówienie 162-165

słownikowe sortowanie danych
znakowych 72

SMALLDATETIME

korzystanie oddzielnie z dat i czasu 92-94

literały 88-92

typy danych 87-88

SNAPSHOT

izolacja, omówienie 380-384, 388

READ COMMITTED SNAPSHOT 384-388

wykrywanie konfliktów 382-384

sortowanie a klauzula ORDER BY 49-52

sp_columns 109

sp_executesql, procedura
składowana 424-426

sp_help 108-109

sp_helpconstraint 109

sp_tables 108

spójność transakcji 358-361

Zobacz także izolacja transakcji

SQL (Structured Query Language),
omówienie 1-4

historia i zastosowania 3-4

logika predykatów 6

brakujące wartości 8-9

model relacyjny 6-13

ograniczenia 9

typy systemów bazodanowych 13-15

SQL Data Warehouse 18

SQL Server

instalacja produktu w siedzibie
(box) 442-447

instalowanie silnika bazy danych 442-447

omówienie architektury 15-24

ABC, odmiany (Appliance, Box, Cloud) 15-18
 bazy danych 19-23
 fizyczny układ 21-23
 instancje 18-19
 schematy i obiekty 23-24
 reguły pierwszeństwa operatorów 60-61
 rozszerzenia nazw plików 23
 SQL Server Books Online 456-459
 unikatowe indeksy 27-28
 źródłowy kod, pobieranie i instalowanie 448-450
 SQL Server Management Studio (SSMS)
 ID sesji, rozwiązywanie problemów z blokowaniem 365-372
 pobieranie i instalowanie 448
 rozpoczynanie pracy 450-456
 temporalne tabele, tworzenie 335
 SQL_VARIANT 296
 START WITH, obiekt sekwencji 290-291
 STRING_SPLIT 84
 Structured Query Language (SQL), omówienie 1-4
 STUFF 80
 styl kodowania w T-SQL 26-27
 SUBSTRING 77
 SUM 40-43
 Zobacz także funkcje okna
 SWITCHOFFSET 98
 symbole wieloznaczne
 % (procent), symbol wieloznaczny 84-85
 [^lista znaków lub zakres], symbol wieloznaczny 87
 [lista znaków], symbol wieloznaczny 85
 [znak-znak], symbol wieloznaczny 85-87
 _ (podkreślenie), symbol wieloznaczny 85
 predykat LIKE 84-87
 SYSDATETIME
 domyślne ograniczenia, definiowanie 31
 funkcje bieżącej daty i czasu 95-96
 INSERT SELECT i 284
 SYSDATETIMEOFFSET 95-96
 sys.dm_as_waiting_tasks 369-370
 sys.dm_exec_connections 367

sys.dm_exec_input_buffer 368
 sys.dm_exec_requests 369-370
 sys.dm_exec_sessions 368
 sys.dm_exec_sql_text 367
 sys.dm_tran_locks 367
 SYSTEM_TIME, zapytania do tabel temporalnych 341-347
 SYSTEM_VERSIONING, tworzenie tabel temporalnych 334-338
 SYSUTCDATETIME 95-96

Ś

średnik, używanie do kończenia wyrażeń 36

T

tabele

definiowanie integralności danych 27-31
 nazwy identyfikatorów 37
 ograniczenia CHECK, definiowanie 29-31
 ograniczenia klucza głównego, definiowanie 28
 ograniczenia obcego klucza, definiowanie 28-29
 ograniczenia unikatowego klucza, definiowanie 28
 tworzenie tabel 24-27

tabele pochodne

aliasy nazw kolumn, przypisywanie 187-189
 argumenty, używanie 189-190
 ćwiczenia 211-216
 omówienie 185-186
 rozwiązywania ćwiczeń 216-220
 wielokrotne odwołania 191
 zagnieżdżanie 190-191

tabele tymczasowe

globalne tabele tymczasowe 419-421
 lokalne tabele tymczasowe 417-418
 omówienie 417
 typy tabel 422-423
 zmienne tablicowe 421-422

tablicowe funkcje zdefiniowane przez użytkownika (UDF) 428-429

tablicowe podzapytania, definicja 155
 Zobacz także wyrażenia tablicowe

tempdb, naza danych, SQL Server

lokalne tabele tymczasowe 417-418
 omówienie architektury 20-23
 SNAPSHOT 380-384
 tymczasowe zmienne tablicowe 421-422

temporalne tabele

ćwiczenia 348-351
 modyfikowanie danych 338-341
 odpytywanie danych 341-347
 omówienie 333
 rozwiązania ćwiczeń 351-355
 tworzenie 334-338

teoria zbiorów

logika predykatów 6
 omówienie 4

tezy, model relacyjny 7-8**THEN w wyrażeniach CASE 62-64****TIME**

korzystanie oddzielnie z dat i czasu 92-94
 literały 88-92
 typy danych 87-88

TODATETIMEOFFSET 98-99**TOP (100) PERCENT, widoki i 200-201****TOP, filtry**

ćwiczenia 323-327
 modyfikowanie danych przy użyciu 313-316
 omówienie 53-55
 pochodne tabele i 186
 rozwiązania ćwiczeń 327-331
 widoki i klauzula ORDER BY 200-201

transakcje

blokady i blokowanie 361-364
 rozwiązywanie problemów 364-372
 ćwiczenia 391-402
 izolacja
 omówienie 372-374
 podsumowanie 388
 READ COMMITTED 375-378
 READ COMMITTED SNAPSHOT 384-388
 READ UNCOMMITTED 374-375
 REPEATABLE READ 377-378
 SERIALIZABLE 378-379
 SNAPSHOT 380-384

SNAPSHOT, wykrywanie

konfliktów 382-384

kontra wsady 406

omówienie 357-361

zakleszczenia 388-391

transakcyjne przetwarzanie w trybie online (OLTP), bazy danych 13-15**TRUE**

IF...ELSE, element sterowania
 przepływem 410-411

ograniczenia CHECK 29-31

WHILE, element sterowania
 przepływem 411-413

znaczenie 39

znaczniki NULL 64-69

TRUNCATE 296-300**trwałość transakcji 358-361****TRY...CATCH, obsługa błędów 436-440****TRY_, funkcje daty i czasu 96-98****T-SQL**

całościowe użycie 4

informacje podstawowe 1-4

logika predykatów 6

model relacyjny 6-13

brakujące wartości 8-9

normalizacja 9-13

ograniczenia 9

tezy, predykaty i relacje 7-8

niezależność od języka 3

standardy SQL 4

styl kodowania 26-27

teoria zbiorów 4-6

typy systemów bazodanowych 13-15

typy danych

dane znakowe 71-72

operatory, pierwszeństwo
 wykonywania 59-61

prefiks N, stosowanie 59

typy tez, predykatów i relacji 8**U**

UDF (funkcje zdefiniowane przez
 użytkownika), jako procedury 428-429

ujemne (-) 60-61

Unicode, typy danych 59

dane znakowe 71-72

unikatowość, ograniczenie a integralność
danych 28

unikatowy indeks w SQL Server

ograniczenia klucza głównego 27

ograniczenia unikatowości, definiowanie 28

UNION

ćwiczenia 234-237

korzystanie z 222-225

pierwszeństwo 230-231

rozwiązania ćwiczeń 237-240

UNION (DISTINCT) 222-225

UNION ALL 222-223

UNIQUE a znaczniki NULL 69

UNKNOWN

IF...ELSE, element sterowania
przepływem 410-411

ograniczenia CHECK i 29-31

podzapytania, problem ze znacznikami
NULL 170-171

skalarne podzapytania 157

WHILE, element sterowania
przepływem 411-413

złączenia zewnętrzne, filtrowanie
atrybutów 138-139

złączenia zewnętrzne, w wielokrotnych
złączeniach 139-141

znaczenie 39

znaczniki NULL 64-69

UNPIVOT 264-265

UPDATE

ćwiczenia 323-327

DML, wyzwalacze (zdarzenia manipulacji
danymi) 432-433

MERGE i 309

omówienie 300-302

oparte na złączeniu 302-305

OUTPUT i 319-320

poziomy izolacji oparte na wersjonowaniu
wierszy 380

przypisanie UPDATE 305-306

rozwiązania ćwiczeń 327-331

widoki 199

wrażenia tablicowe, modyfikowanie danych
przy użyciu 311-313

UPPER 80-81

uprawnienia, schematy i obiekty bazy
danych 23-24

USE, tworzenie tabel 25

użytkownik bazy danych, loginy SQL
Server 21

V

VALUES 282-284

VAR, element 71-72

VARCHAR 26, 59, 71-72

W

wbudowane (inline) funkcje tablicowe (TVF)

ćwiczenia 211-216

omówienie 198-199, 206-207

rozwiązania ćwiczeń 216-220

wcześniejsza wersja tabeli. *Zobacz* temporalne
tabele

wcześniejsze wartości, zwracanie
w podzapytaniu 167-168

wersjonowane systemowe tabele temporalne.
Zobacz temporalne tabele

wersjonowanie wierszy

poziomy izolacji i 358-359

READ COMMITTED SNAPSHOT 384-388

SNAPSHOT 380-384

WHEN MATCHED THEN 308-310

WHEN NOT MATCHED THEN 308-310

WHEN, wyrażenia CASE 62-64

WHERE

logiczny porządek przetwarzania
zapytania 34

omówienie 38-39

tabele pochodne, argumenty 189-190

tabele pochodne, przypisywanie aliasów 187

UPDATE i 301-302

UPDATE oparte na złączeniu 302-305

wrażenia CASE 61-64

WHERE (cd.)

- złączenia zewnętrzne, filtrowanie atrybutów 138-139
- złączenia zewnętrzne, podstawy 134-136
- WHILE, element sterowania przepływem 411-413
- wiązanie, wsady i 409
- widoki jako wyrażenia tablicowe
 - CHECK OPTION 204-206
 - ćwiczenia 211-216
 - ENCRYPTION 202-203
 - omówienie 198-199
 - ORDER BY i 199-201
 - rozwiązania ćwiczeń 216-220
 - SCHEMABINDING 203-204
- wielowartościowe podzapytania
 - definicja 155
 - niezależne podzapytania, przykłady 158-162
- wielowyrażeniowa funkcja tablicowa (TVF) 206
- wielozbiór 4
- wiersze 6
- wirtualne maszyny (VM), a odmiany ABC (Appliance, Box, Cloud) 16-18
- WITH NOCHECK 31
- WITH TIES, klauzula ORDER BY 54-55
- WITH, wspólne wyrażenia tablicowe (CTE) 192
- wsady
 - dynamiczne instrukcje SQL 423-428
 - jako jednostka analizy składni 406
 - jako jednostka rozpoznawania (wiązania) 409
 - kontra transakcje 406
 - omówienie cech 406
 - zmienne 403, 408
- wspólne wyrażenia tablicowe (CTE)
 - aliasy nazw kolumn, przypisywanie 192-193
 - argumenty, używanie 192-193
 - ćwiczenia 211-216
 - definiowanie wielu CTE 193-195
 - omówienie 192
 - rekurencyjne wyrażenia CTE 195-198
 - rozwiązania ćwiczeń 216-220

- wielokrotne odwołania 195
- współbieżność a transakcje, omówienie 357-361
- wydajność
 - architektura SQL Server, układ fizyczny 21-23
 - dynamiczne instrukcje SQL 423-428
 - filtry zapytań 39
 - nazwy obiektów, kwalifikowanie przy użyciu schematu 36
 - poziomy izolacji z wersjonowaniem wierszy 380
 - procedury składowane 429
 - sp_executesql, procedura składowana 424-426
- wykrywanie konfliktów, poziom izolacji SNAPSHOT 382-384
- wyłączne blokady 361-363
- wymuszanie zasad, wyzwalacze DDL (zdarzenia definicji danych) 433-436
- wyrażenia tablicowe
 - APPLY, operator 207-211
 - ćwiczenia 211-216, 323-327
 - modyfikowanie danych przy użyciu 311-313
 - omówienie 185
 - rozwiązania ćwiczeń 216-220, 327-331
- tabele pochodne
 - aliasy kolumn, przypisywanie 187-189
 - argumenty, używanie w 189-190
 - omówienie 185-186
 - wielokrotne odwołania 191
 - zagnieżdżanie 190-191
- wbudowane funkcje zwracające tabelę (TVF), omówienie 198-199, 206-207
- widoki
 - CHECK OPTION 204-206
 - ENCRYPTION 202-203
 - omówienie 198-199
 - ORDER BY 199-201
 - SCHEMABINDING 203-204
- wspólne wyrażenia tablicowe (CTE)
 - aliasy kolumn, przypisywanie 192-193
 - argumenty, używanie w 192-193

- definiowanie wielu CTE 193-195
- omówienie 192
- rekurencyjne wyrażenia CTE 195-198
- wielokrotne odwołania w 195
- wyrażenia wektorowe 305
- wyzwalacze 432-436
- wyzwalacze DDL (zdarzenia definicji danych) 434-436
- wyzwalacze DML (zdarzenia manipulacji danymi) 432-433

X

XACT_ABORT 358

Y

YEAR 102-103

- filtrowanie zakresów dat 94-95

YTD, agregująca funkcja okna 253-254

Z

zagnieżdżone podzapytania 155

- pochodne tabele i 190-191

zagnieżdżone wyrażenia DML 321-323

zakleszczenia 378, 388-391

zamknięty świat, założenie 7

zapytania z wielokrotnymi złączeniami

- ćwiczenia 144-149
- korzystanie z 133
- rozwiązania ćwiczeń 149-152
- złączenia zewnętrzne w 139-141

zapytania. *Zobacz także* SELECT;

- podzapytania; wyrażenia tablicowe
- aliasy, stosowanie 44-49
- ćwiczenia 110-115
- dane daty i czasu, korzystanie z 87-106
- filtr OFFSET-FETCH, omówienie 55-56
- filtr TOP, omówienie 53-55
- fraza zapytania, definicja 36
- FROM, omówienie 36-37
- funkcje okna, omówienie 56-58
- GROUP BY, omówienie 39-43
- HAVING, omówienie 43-44
- identyfikatory, oddzielanie 37
- klauzule zapytania, definicja 36

- LIKE, predykat 84-87
- logiczny porządek przetwarzania
- zapytania 33-36
- metadanych 106-109
- operacje jednoczesne (all-at-once) 69-70
- operatory i funkcje, omówienie 75-87
- ORDER BY, omówienie 49-52
- predykaty i operatory, przeład 58-61
- rozwiązania ćwiczeń 115-120
- SELECT, klauzula, omówienie 44-49
- skalarne podzapytania 155
- w tabelach temporalnych 341-347
- WHERE, omówienie 38-39
- wyrażenia CASE, omówienie 61-64
- zagnieżdżone i zewnętrzne zapytania, definicja 155
- znaczniki NULL, omówienie 64-69
- znakowe dane, korzystanie z 71-75
- zarządzanie zmianami, wyzwalacze DDL (zdarzenia dotyczące danych) 433-436
- zasoby, typy blokwalne 363-364
- zawarte w sobie bazy danych, architektura SQL Server 21
- zbiory grupujące. *Zobacz także* GROUP BY
- CUBE 268
- ćwiczenia 272-275
- GROUPING i GROUPING_ID, funkcje 269-272
- GROUPING SETS, klauzula pomocnicza 266-268
- omówienie 265-266
- ROLLUP 268-269
- rozwiązania ćwiczeń 277-280
- zdarzenia definicji danych (wyzwalacze DDL) 432-436
- zdarzenia manipulacji danych (wyzwalacze DML) 432-433
- zewnętrzne zapytania 155
- Zobacz także* podzapytania; wyrażenia tablicowe
- złączanie tekstów (+) 60-61
- złączenia wewnętrzne 126-129
- ćwiczenia 144-149
- rozwiązania ćwiczeń 149-152

złączenia zewnętrzne

agregacja COUNT, wykorzystanie 141-144

brakujące wartości, dołączanie 136-138

ćwiczenia 144-149

filtrowanie atrybutów z niezachowanej
strony 138-139

podstawy 133-136

rozwiązania ćwiczeń 149-152

w zapytaniach z wielokrotnymi
złączeniami 139-141

złączenia złożone 129-130

ćwiczenia 144-149

rozwiązania ćwiczeń 149-152

zmienne jako obiekty programowalne

omówienie 403-406

tymczasowe zmienne tablicowe 421-422
wsady i 408

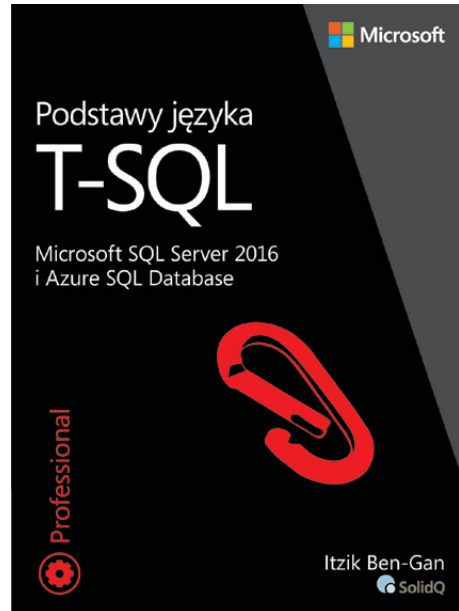
zmienne skalarne 403-406

Polecamy także

978-83-7541-305-2 **Podstawy języka T-SQL: Microsoft SQL Server 2016 i Azure SQL Database**

Książka wyjaśnia kluczowe koncepcje języka T-SQL i pomaga w wykorzystaniu tej wiedzy w praktycznych zastosowaniach. Książka przedstawia zasady działania T-SQL i logikę działającą w tle, a także kluczowe zagadnienia, takie jak zapytania jednotabelowe, złączenia, podzapytania, wyrażenia tablicowe i operatory zbiorów. Przedstawiono również bardziej zaawansowane zagadnienia, takie jak funkcje okna, tworzenie tabel przestawnych i grupowanie zbiorów. Wyjaśnione zostaną również techniki modyfikowania danych, tabele temporalne i obsługa transakcji, a także przegląd obiektów programowalnych.

Itzik Ben-Gan ma tytuł MVP w dziedzinie SQL Server od 1999 roku, jest wykładowcą i współzałożycielem firmy SolidQ, firmy szkoleniowej i konsultingowej koncentrującej się na zagadnieniach baz danych. Jest autorem kilku książek na temat języka T-SQL i twórcą kilku zaawansowanych kursów szkoleniowych w dziedzinie projektowania i optymalizowania baz danych.



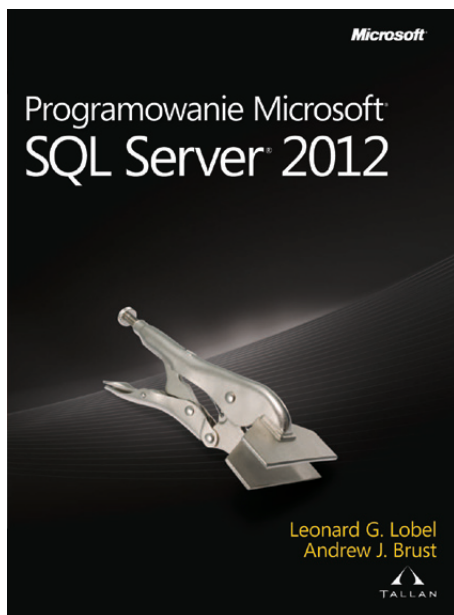
978-83-7541-098-3 **Microsoft SQL Server 2012: Optymalizacja kwerend T-SQL przy użyciu funkcji okna**

Wykorzystanie zaawansowanych funkcji okna do zwiększania wydajności i przyspieszania kwerend T-SQL

- Optymalizowanie kwerend przy użyciu nowych narzędzi
- Pełne wykorzystanie potencjału funkcji analitycznych, takich jak m.in. RANK, CUME_DISC czy LAG
- Implementowanie funkcji hipotetycznego zbioru oraz rozkładu odwrotnego w standardowym języku SQL
- Stosowanie strategii usprawniających stronicowanie, filtrowanie, przestawianie oraz obsługę sekwencji
- Skracanie czasu działania kwerend przy użyciu partycjonowania, kolejności i indeksów pokrywających
- Korzystanie z nowych operatorów optymalizujących, takich jak Window Spool
- Realizowanie standardowych zadań, takich jak sumy bieżące, mediany i luki
- ... i wiele innych!

Więcej informacji i ofert: www.ksiazki.promise.pl

Polecamy także



978-83-7541-095-2 Programowanie Microsoft SQL Server 2012

Przenieś umiejętności programowania baz danych na nowy poziom, pozwalający na tworzenie własnych aplikacji. Ten podręcznik pokazuje, jak programowo projektować, testować i wdrażać bazy danych SQL Server, dostarczając wielu praktycznych wskazówek, rzeczywistych zastosowań i przykładowych kodów źródłowych. To obowiązkowa lektura dla każdego programisty SQL Server, pragnącego poznać sposoby projektowania i tworzenia efektywnych aplikacji.

Leonard Lobel jest posiadaczem certyfikatu MVP dla serwera SQL Server, specjalistą z dziedziny .NET, głównym konsultantem w firmie Tallan, Inc., a także współzałożycielem i dyrektorem generalnym firmy Sleek Technologies, Inc.

Andrew Brust jest założycielem i dyrektorem generalnym firmy Blue Badge Insights, oferującej swoim klientom usługi w zakresie analizy, strategii i doradztwa ułatwiającego przechodzenie na technologie oferowane przez firmę Microsoft.

978-83-7541-106-5 Microsoft SQL Server 2012 Krok po kroku

Samodzielnie opanuj posługiwanie się oprogramowaniem SQL Server 2012 – krok po kroku. Idealny dla początkujących administratorów i projektantów baz danych, podręcznik ten przedstawia praktyczne ćwiczenia i techniki, które pozwolą zarządzać bazami danych, tworzyć raporty i wdrażać systemy BI.

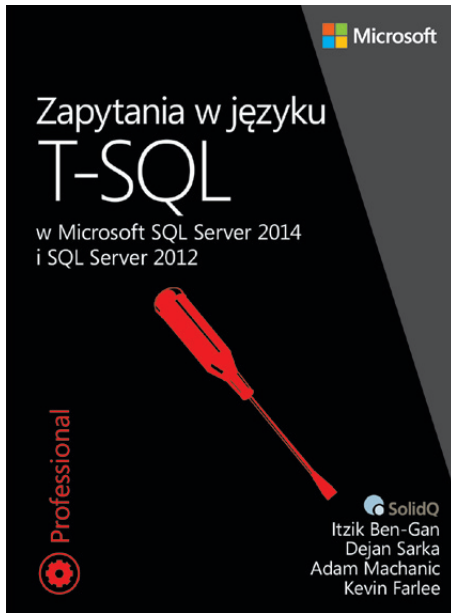
- konfigurowanie i aktualizowanie SQL Server
- kluczowe komponenty i narzędzia
- narzędzia T-SQL do wykonywania zmian w danych
- upraszczanie tworzenia zapytań
- zarządzanie obciążeniem i zużyciem zasobów
- utrzymywanie, odzyskiwanie i zabezpieczanie baz danych

Patrick LeBlanc zajmuje stanowisko Data Platform Technical Solution Professional w firmie Microsoft, pracując bezpośrednio z klientami biznesowymi wykorzystującymi oprogramowanie SQL Server. Założył witrynę poświęconą nauczaniu technologii SQL Server; jest współautorem kilku książek na temat SharePoint oraz rozwiązań BI (business intelligence).



Więcej informacji i ofert: www.ksiazki.promise.pl

Polecamy także



978-83-7541-158-4 **Zapytania w języku T-SQL w Microsoft SQL Server 2014 i SQL Server 2012**

Czterech wiodących ekspertów prezentuje pogłębiony przegląd wewnętrznej architektury T-SQL i zaawansowane, praktyczne techniki optymalizowania reaktywności i zużycia zasobów. Dzięki właściwemu rozumieniu języka i jego podstaw autorzy przedstawiają unikatowe rozwiązania, tworzone i dostrajane przez lata.

Itzik Ben-Gan, SQL Server MVP od roku 1999, współtwórca SolidQ oraz cykliów szkoleniowych Advanced T-SQL Querying, Programming and Tuning oraz T-SQL Fundamentals courses.

Dejan Sarka, MCT, SQL Server MVP, konsultant w dziedzinie baz danych/BI, autor lub współautor 11 książek.

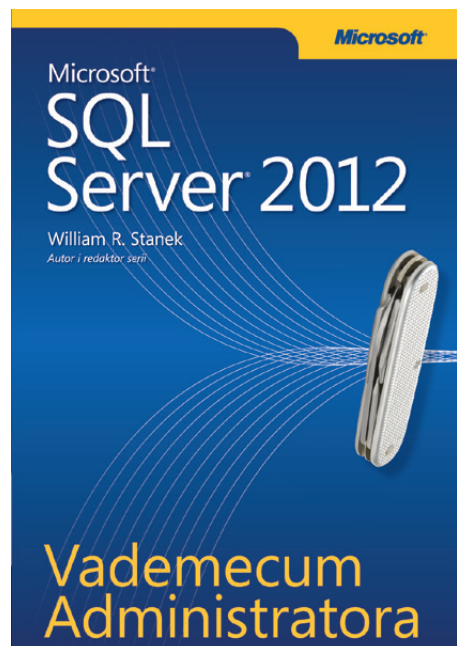
Adam Machanic, SQL Server MVP, programista, autor i wykładowca, twórca nagrodzonej procedury składowanej sp_WholsActive.

Kevin Farlee, Storage Engine Program Manager w zespole SQL Server.

978-83-7541-094-5 **Vademecum Administratora Microsoft SQL Server 2012**

Podręczny i precyzyjny poradnik kieszonkowy dostarcza odpowiedzi na wiele pytań związanych z administrowaniem serwerem SQL Server 2012. Zamieszczone w nim tabele, instrukcje i listy pozwolą Ci skupić się na podstawowych działaniach oraz na wykonywanych codziennie zadaniach. Poradnik ten zawiera w skondensowanej formie wszystkie informacje potrzebne administratorowi do rozwiązywania napotykaných problemów i dobrego wykonywania swojej pracy – zarówno przy własnym biurku, jak i w siedzibie klientów.

William R. Stanek ma tytuł Microsoft MVP i ponad 20-letnie doświadczenie w zarządzaniu systemami oraz zaawansowanym programowaniu. Jest wielokrotnie nagradzanym autorem ponad 100 książek, w tym Vademecum administratora Windows® 7 oraz Windows Server® 2008 Inside Out. Jest głównym redaktorem serii poradników Vademecum administratora (Pocket Consultant).



Więcej informacji i ofert: www.ksiazki.promise.pl

Polecamy także



978-83-7541-157-7 **Usługi Microsoft Azure Programowanie aplikacji**

Microsoft Azure to platforma chmurowa, oferująca gotowe do subskrypcji i użycia własne aplikacje chmurowe oraz dostęp do usług w modelach IaaS i PaaS. Autorzy przedstawiają różne kategorie usług dostępnych w chmurze Azure, a następnie przechodzą do demonstracji praktycznego wykorzystania rozmaitych usług chmury Azure poprzez budowanie czterech „z życia wziętych” aplikacji. Pokazana została zarówno konfiguracja składników działających w chmurze, jak i tworzenie właściwego kodu.

Dr inż. **Zbigniew Fryźlewicz** jest adiunktem w Katedrze Informatyki na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej. zajmuje się technologiami internetowymi, usługami webowymi i programowaniem wspólnym.

Mgr inż. **Łukasz Leśniczek** jest programistą specjalizującym się w technologiach platformy .NET Framework. Zajmuje się projektowaniem, implementowaniem i wdrażaniem aplikacji webowych, zarówno w warstwie front-end, jak i back-end.

978-83-7541-153-9 **Microsoft Azure SQL Database Krok po kroku**

Praktyczny przewodnik po podstawach SQL Database

Rozszerz swoje umiejętności i samodzielnie poznaj podstawy platformy Microsoft Azure SQL Database. Jeśli jesteś doświadczonym projektantem oprogramowania lub specjalistą od baz danych, ale nowicjuszem w dziedzinie chmury Microsoft Azure lub SQL Database, znajdziesz tu wskazówki, ćwiczenia i przykłady kodu potrzebne do opanowania podstawowych zagadnień i technik.

Leonard G. Lobel, Microsoft MVP w dziedzinie SQL Server, jest głównym konsultantem w firmie Tallan, Inc., Microsoft National Systems Integrator and Gold Competency Partner, i jednym z wiodących ekspertów branżowych w zakresie .NET i SQL Server.

Eric D. Boyd, MVP w dziedzinie Microsoft Azure, jest założycielem i dyrektorem generalnym responsiveX, firmy konsultingowej skupiającej się na aplikacjach i usługach w chmurze. Jest częstym wykładawcą na konferencjach branżowych, regionalnych i w lokalnych grupach użytkowników.



Więcej informacji i ofert: www.ksiazki.promise.pl