Microsoft

Microsoft® SQL Server® 2008 od środka:

Programowanie w języku T-SQL

Itzik Ben-Gan
Dejan Sarka, Greg Low
Ed Katibah, Isaac Kunen
i Roger Wolter

Microsoft® SQL Server™ 2008 od środka: Programowanie w języku T-SQL
Edycja polska Microsoft Press

Tytuł oryginału: Inside Microsoft® SQL Server™ 2008: T-SQL Programming

Polish edition by APN PROMISE Sp. z o.o. Warszawa 2010

APN PROMISE Sp. z o.o., ul. Kryniczna 2, 03-934 Warszawa
tel. +48 22 35 51 642, fax +48 22 35 51 699
e-mail: mspress@promise.pl

ISBN: 978-83-7541-063-1

Przekład: Natalia Chounlamany
Redakcja: Marek Włodarz
Korekta: Magdalena Swoboda, Anna Wojdanowicz
Skład i łamanie: MAWart Marek Włodarz

Moim bratnim duszom, Inie i Mickey'owi
– Itzik Ben-Gan

Spis treści

Przedmowa

Na początek kilka słów o autorze książki. Itzik Ben-Gan jest mentorem, konsultantem, prezenterem, szkoleniowcem i pisarzem. Wszystkie te role łączy jeden temat przewodni, a mianowicie Microsoft SQL Server. To jednak nie wszystko. Itzik ściśle współpracuje z wieloma programistami SQL Server z Redmond i otrzymał tytuł MVP (Microsoft Valued Professional). Wspomniane doświadczenie oraz umiejętności pozwoliły Itzikowi napisać książkę, która przedstawia zagadnienie programowania SQL Server z unikalnej perspektywy. Autor wie, jakie rozwiązania działają, a jakie nie. Zna metody gwarantujące wysoką wydajność, a także te obniżające efektywność. Praca szkoleniowca pozwoliła mu poznać najczęściej zadawane pytania oraz sposoby zrozumiałego prezentowania zaawansowanych koncepcji programowania SQL Server.

Itzik zaprosił kilku autorów do udziału w tworzeniu tej książki. Najnowsze technologie SQL Server zostały zaprezentowane przy współpracy z zespołem ds. rozwoju SQL Server. Rozdział poświęcony danym przestrzennym został napisany przez Eda Katibaha (o pseudonimie „Spatial Ed") oraz Isaaca Kunena, natomiast rozdział dotyczący funkcji Service Broker przez Rogera Woltera. Dejan Sarka pomógł w objaśnieniu technologii CLR i XML oraz stworzył fascynujący rozdział poświęcony wsparciu dla danych tymczasowych w modelu relacyjnym, w którym sygnalizuje programistom SQL Server potencjalną przydatność operatorów relacyjnych PACK oraz UNPACK – nadal niedostępnych w systemie SQL Server. Greg Low prezentuje różnorodne możliwości śledzenia dostępu oraz modyfikacji danych i metadanych. Dejan oraz Greg są weteranami SQL Server i razem z Itzikiem pracują w firmie Solid Quality Mentors.

Osobiście jestem zwolennikiem praktycznego podejścia do nauki programowania. Ta książka zawiera wiele przykładów, które są zaprezentowane w sposób umożliwiający uruchamianie ich na własnym serwerze SQL Server. Czytelnicy, którzy nie mają dostępu do systemu SQL Server, mogą pobrać wersję ewaluacyjną produktu SQL Server 2008 ze strony *http://www.microsoft.com/sql* (do tego celu niezbędny jest identyfikator Windows Live ID, z wersji ewaluacyjnej Enterprise można korzystać przez 180 dni). Zaleca się uruchamianie przykładów w edycji Enterprise lub Developer SQL Server. Nie trzeba przepisywać fragmentów kodu prezentowanych w książce, wystarczy pobrać kod źródłowy z witryny *http://www.InsideTSQL.com*.

Czytelnikom, którzy nie mają doświadczenia w stosowaniu języka SQL, zaleca się rozpoczęcie nauki od wcześniej opublikowanej książki *Microsoft SQL Server 2008: T-SQL Fundamentals*. Osobom, które nie miały okazji korzystać z systemu SQL Server, ale pracowały już z innymi produktami wspierającymi technologię SQL, zaleca się lekturę innej książki z tej serii: *Microsoft SQL Server 2008 od środka: Zapytania T-SQL*. Oczywiście można przejść od razu do tej publikacji, która koncentruje się na możliwościach programowania w systemie SQL Server. Aczkolwiek gdy okaże się, że prezentowane w tej książce przykłady zawierają instrukcje o implementacji charakterystycznej dla SQL Server, wspomniane książki mogą okazać się pomocne.

Jestem przekonany, że nawet zaawansowani programiści SQL Server odnajdą w tej książce nowe, efektywniejsze metody realizowania zadań. Zgadzam się z Dejanem, że definiowane przez użytkowników typy CLR są rzadko stosowane w systemach produkcyjnych. I uwaga ta dotyczy nie tylko definiowanych przez użytkowników typów, ale również funkcji, wyzwalaczy oraz procedur składowanych CLR. Ta książka przedstawia wiele przykładowych rozwiązań, demonstrując ich implementację w wersji C# oraz Microsoft Visual Basic, czyli najpopularniejszych językach CLR. Autorzy zalecają rozważne stosowanie obiektów CLR ze względu na ich wpływ na wydajność. Itzik nie tylko prezentuje ogólne dyrektywy dotyczące wydajności, ale informuje również, ile trwało wykonywanie alternatywnych rozwiązań na jego komputerze. Oczywiście czytelnicy mogą przetestować rozwiązania na własnych maszynach.

Czynnik wydajności jest brany pod uwagę nie tylko w przypadku konstrukcji CLR. Każdy rozdział książki zawiera wskazówki prowadzące do podniesienia efektywności. Na przykład w rozdziale 7 „Tabele tymczasowe i zmienne tabelaryczne" można dowiedzieć się, kiedy lepiej jest zastosować tabele tymczasowe, a kiedy zmienne tabelaryczne. Itzik wykorzystuje proste przykłady, interpretując plany wykonania zapytań i prezentując sposób zastosowania liczników we/wy do porównania alternatywnych metod realizowania tego samego zadania.

Wspomniałem już, że napisany przez Dejana Rozdział 12 „Wsparcie dla danych tymczasowych w modelu relacyjnym" jest fascynujący. Dlaczego? Podzielę się z Wami pewnym sekretem. Jakiś czas temu rozważaliśmy zaimplementowanie specjalnych funkcji wspierających obsługę tymczasowych danych w SQL Server. Podjęliśmy intensywne działania, a zespół programistów SQL Server nawiązał współpracę z wiodącymi ośrodkami naukowymi. Jeden z liderów technicznych zmienił nawet numer rejestracyjny swojego samochodu na „TIME DB". Co stało się z tym projektem? Implementacja była kosztowna i złożona. Wielokrotnie dokonywana ocena alternatyw nie pozwoliła wyłonić zdecydowanego zwycięscy. I stale pojawiał się ten sam kontrargument: „przecież można osiągnąć ten sam cel w inny sposób". Za każdym razem, gdy ktoś próbował podważyć to twierdzenie, ktoś inny publikował fragment kodu demonstrujący, że można zrealizować konkretne zadanie przy użyciu *istniejących* funkcji SQL Server. Jednak nie znam nikogo, kto zaprezentowałby to zagadnienie w sposób tak kompleksowy, jak Dejan w rozdziale 12 tej książki!

Miałem okazję pracować w tym samym zespole co Roger Wolter, gdy był on odpowiedzialny za rozwój nowej funkcji Service Broker w SQL Server 2005. Napisany przez Rogera rozdział 16 stanowi doskonałe odzwierciedlenie jego osobowości: dokładny i konkretny w perfekcyjnej formie. Początkujący użytkownicy funkcji Service Broker mogą rozważyć rozpoczęcie lektury od końca rozdziału, gdzie prezentowane są scenariusze zastosowania funkcji Service Broker, a także krótkie porównanie funkcji Service Broker z funkcjami wspomagającymi komunikację w rozwiązaniach Microsoft Message Queue (MSMQ), BizTalk oraz Windows Communication Foundation (WCF). Brazylijski bank Itau oraz serwis MySpace to dwaj przykładowi klienci SQL Server, którzy wykorzystują funkcję Service Broker do różnych celów. Bank Itau stosuje funkcję Service Broker w procesie przetwarzania wsadowego. Natomiast w MySpace funkcja Service Broker

tworzy platformę komunikacji między setkami serwerów SQL Servers wspierającymi pracę witryny społecznościowej MySpace.com.

Jestem przekonany, że ta książka jest warta przeczytania i okaże się pomocna dla początkujących oraz bardziej doświadczonych użytkowników SQL Server. Stanowi ona nieocenione źródło wiedzy dla programistów, architektów baz danych oraz administratorów.

Lubor Kollar
Group Program Manager
Zespół SQL Server Customer Advisory
Microsoft, Redmond, Washington USA.

Podziękowania

Wiele osób przyczyniło się do powstania obydwu książek *Microsoft SQL Server 2008 od środka: Zapytania w języku T-SQL* oraz *Microsoft SQL Server 2008 od środka: Programowanie w języku T-SQL* i chciałbym podziękować im za pomoc. Niektóre osoby były bezpośrednio zaangażowane w proces tworzenia lub edycji książki, inne dostarczyły wsparcia, porad bądź inspiracji.

Dziękuję współautorom książki *Zapytania T-SQL** Luborowi Kollar, Dejanowi Sarka oraz Stevowi Kass, a także współautorom książki *Programowanie T-SQL* Dejanowi Sarka, Rogerowi Wolter, Gregowi Low, Edowi Katibah oraz Isaacowi Kunen – praca z Wami była dla mnie zaszczytem. Jestem pod wrażeniem Waszej wiedzy i doświadczenia, a lektura Waszych publikacji to prawdziwa przyjemność. Dziękuję za to, że zgodziliście się wziąć udział w tym projekcie.

Podziękowania dla Lubora nie tylko za bezpośredni udział (napisanie rozdziału książki *Zapytania T-SQL* i przedmowy do książki *Programowanie T-SQL*), ale także za wsparcie, porady, przyjaźń oraz inspirację. Mam nadzieję, że wkrótce znowu będziemy mieli okazję spędzić ze sobą więcej czasu, spacerując, pijąc oraz rozmawiając o SQL i nie tylko.

Podziękowania dla Dejko – Twoja wiedza o modelu relacyjnym jest godna uznania. Każda nasza rozmowa stanowi dla mnie okazję do zdobycia nowych informacji lub pogłębienia dotychczasowej wiedzy. Podoba mi się to, że nie ulegasz trendom i nie podążasz ślepo za opiniami osób uważanych za ekspertów dziedzinowych. Masz własne zdanie i potrafisz spojrzeć na wiele kwestii z innej perspektywy. Dziękuję za to, że zgodziłeś się współuczestniczyć w tworzeniu tej książki, a także za twoją przyjaźń. Czas spędzony z Tobą jest dla mnie zawsze przyjemnością. Musimy kiedyś zmierzyć się ponownie z tą listą piw, w końcu to już prawie 10 lat!

Podziękowania dla redaktora technicznego Steva Kass, który dysponuje unikatowym połączeniem talentu matematycznego, SQL oraz literackiego. Mam świadomość, że niełatwo było zajmować się jednocześnie redakcją książki i pisaniem własnych rozdziałów, tym większa jest moja wdzięczność. Pewnie moje słowa wprowadzą Cię w zakłopotanie, ale obserwowanie geniusza w akcji było dla mnie bardzo interesującym doświadczeniem. Przywodziło mi na myśl fragment powieści Williama Goldmana zatytułowanej *Narzeczona księcia*, w którym opisana została praca Domingo Montoya nad mieczem dla sześciopalczastego człowieka.

Umachandarowi Jayachandran (UC) ogromnie dziękuję za pomoc w redakcji wybranych rozdziałów. Twoja wiedza o T-SQL jest godna podziwu i bardzo się cieszę, że mogłeś wziąć udział w projekcie. Chciałbym również podziękować Bobowi Beauchemin za recenzję rozdziału poświęconego danym przestrzennym. Lubię czytać Twoje

* Dla zachowania przejrzystości przy odwoływaniu się do obu pozycji stosowane są skrócone wersje tytułów, odpowiednio *Zapytania T-SQL* oraz *Programowanie T-SQL* (przyp. red.).

artykuły; Twoje poglądy na temat możliwości programowania SQL Server są zawsze interesujące i na czasie.

Podziękowania dla Cesara Galindo-Legaria. Czuję się zaszczycony, że zgodziłeś się napisać przedmowę do książki *Zapytania T-SQL*. Zaprojektowany przez Ciebie zespół optymalizator SQL Server jest po prostu rewelacyjny. Nadal staram się zrozumieć jego działanie i za każdym razem, gdy uda mi się rozwikłać kolejny fragment zagadki, jestem zachwycony możliwościami tego oprogramowania. Twoja głęboka wiedza, miłe usposobienie i skromność stanowią dla mnie inspirację.

Podziękowania dla pracowników wydawnictwa Microsoft Press, a w szczególności koordynatora produktu Kena Jonesa. Jestem ci wdzięczny za personalne podejście, bardzo lubię nasze Guinnessowe sesje. Zadanie pogodzenia potrzeb wszystkich osób wydawało mi się niewykonalne, ale tobie jakoś udało się osiągnąć ten cel.

Podziękowania dla redaktora nadzorującego Sally Stickney za umożliwienie realizacji tego projektu. Wiem, że książka *Zapytania T-SQL* była Twoim ostatnim projektem w wydawnictwie Microsoft Press przed rozpoczęciem nowej kariery i tym bardziej mam nadzieję, że miło wspominasz to doświadczenie. Życzę powodzenia na nowej drodze.

Podziękowania dla redaktorki projektu Denise Bankaitis, która prawdopodobnie poświęciła tej serii książek więcej czasu niż jakikolwiek inny pracownik Microsoft Press. Dziękuję za efektywne zarządzanie projektem i zapewnienie sprawnej realizacji zadań. Praca z Tobą była przyjemnością.

Chciałbym również podziękować menedżerom projektu DeAnn Montoya oraz Ashley Schneider z zespołu redakcyjnego S4Carlisle, a także redaktorce Becke McKay. Wiem, że poświęciłyście naszym tekstom wiele godzin, doceniam to.

Podziękowania dla Solid Quality Mentors – praca w tak rewelacyjnej firmie i grupie ludzi to najwspanialsze doświadczenie w mojej dotychczasowej karierze. Czuję się tak, jakby wszystkie działania podejmowane w życiu zawodowym doprowadziły mnie do tego wspaniałego miejsca, w którym mogę zrealizować swoje powołanie, ucząc technologii SQL. Fernando Guerrero, Brianie Moran, Douglasie McDowell – to Waszej pracy firma zawdzięcza swój rozwój i sukces, możecie być z siebie dumni. Praca w Solid Quality Mentors zapewnia mi poczucie satysfakcji oraz radości, że mogę przebywać w grupie przyjaciół, którym ufam i których szanuję.

Podziękowania dla kolegów z firmy: Rona Talmage, Andrew J. Kelly, Eladio Rincón, Dejana Sarka, Herberta Albert, Fritzai Lechnitz, Gianluca Hotz, Erika Veerman, Jay Hackney, Daniela Seara, Davida Mauri, Andrei Benedetti, Miguela Egea, Adolfa Wiernik, Javiera Loria, Rushabha Mehta, Grega Low, Petera Myers, Randiego Dyess i wielu innych. Dziękuję Jeanne Reeves oraz Glen McCoin za umożliwienie realizacji szkoleń oraz całemu zespołowi za wsparcie. Chciałbym także podziękować Kathy Blomstrom za zarządzanie naszymi projektami oraz doskonałą redakcję.

Wyrazy wdzięczności dla zespołu ds. rozwoju SQL Server, który jest odpowiedzialny za implementację i optymalizację języka T-SQL: Michaela Wang, Michaeala Rys, Erica Hanson, Umachandara Jayachandran (UC), Tobiasa Thernström, Jima Hogg, Isaaca Kunen, Krzysztofa Kozielczyka, Cesara Galindo-Legaria, Craiga Freedman, Conora Cunningham, Yavora Angelov, Susan Price i wielu innych. Jesteśmy poniekąd skazani

na używanie owoców Waszej pracy i jak dotąd efekty są rewelacyjne! Aczkolwiek ostrzegam, że nie damy Wam spokoju, dopóki nie otrzymamy pełnej implementacji klauzuli OVER.

Chciałbym podziękować także Dubiemu Lebelowi oraz Assafowi Fraenkelowi z izraelskiego oddziału Microsoft oraz Amiemu Levinowi za pomoc w nadzorowaniu pracy izraelskiej grupy użytkowników SQL Server.

Podziękowania dla zespołu *SQL Server Magazine*: Megan Bearly, Sheili Molnar, Mary Waterloo, Michele Crockett, Mika Otey, Lavon Peters, Anne Grubb – możliwość współtworzenia tak wspaniałego czasopisma to dla mnie ogromny zaszczyt. Gratulacje z okazji dziesięciolecia magazynu! Trudno uwierzyć, że to już 10 lat – czas płynie tak szybko, gdy dobrze się bawimy.

Dziękuję kolegom SQL Server MVP: Erlandowi Sommarskog, Alejandro Mesa, Aaronowi Bertrand, Tiborowi Karaszi, Stevowi Kass, Dejanowi Sarka, Royowi Harvey, Toniemu Rogerson, Marcello Poletti (Marc), Paulowi Randall, Bobowi Beauchemin, Adamowi Machanic, Simonowi Sabin, Tomowi Moreau, Hugonowi Kornelis, Davidowi Portas, Davidowi Guzman, Paulowi Nielsen i wielu innym: Wasz wkład w działanie społeczności SQL Server jest nieoceniony. Nasze dyskusje oraz wymiany poglądów pozwoliły mi zdobyć wiele cennych informacji.

Podziękowania dla kolegów SQL Server MCT: Tibora Karaszi, Chrisa Randall, Teda Malone i innym. Nasza współpraca trwa już bardzo długo i cieszę się, że nadal należycie do grupy szkoleniowców SQL. Łączy nas zamiłowanie do nauczania. Wy najlepiej rozumiecie, jaką satysfakcję może przynosić przekazywanie wiedzy.

Podziękowania dla moich studentów – bez Was moja praca nie miałaby sensu. Nauczanie jest moim powołaniem i celem, który przyświeca praktycznie całej mojej działalności związanej z technologią SQL, także pisaniu książek. Wasze pytania inspirują mnie do odnajdowania odpowiedzi, a zatem w dużym stopniu Wam zawdzięczam swoją wiedzę.

Podziękowania dla moich rodziców Emilii i Gabriela Ben-Gan oraz rodzeństwa Iny Aviram i Michaela Ben-Gan za nieustające wsparcie. To nie przypadek, że większość z nas wybrała zawód nauczyciela, aczkolwiek realizacja powołania wymaga ode mnie częstych podróży. Brakuje mi Was, gdy jestem daleko i z utęsknieniem czekam na nasze spotkanie.

Lilach, to Ty musisz wytrzymywać ze mną cały czas i wysłuchiwać moich pomysłów dotyczących SQL. To niczym pranie mózgu – w końcu zaczniesz zadawać dodatkowe pytania i zanim się obejrzysz, będziesz nawet czytać moje książki. Oczywiście z własnej nieprzymuszonej woli. Przynajmniej taki jest plan… Dziękuję za nadanie znaczenia mojemu życiu i za wspieranie mnie w trudnych momentach towarzyszących tworzeniu książek.

Wprowadzenie

Niniejsza książka – wraz z należącą do tej samej serii książką *Microsoft SQL Server 2008 od środka: Zapytania w języku T-SQL* – prezentuje zaawansowane metody definiowania i optymalizowania zapytań T-SQL oraz programowania w systemie Microsoft SQL Server 2008. Zostały one napisane z myślą o doświadczonych programistach i administratorach baz danych, którzy zajmują się tworzeniem oraz optymalizowaniem kodu w SQL Server 2008. Dla uproszczenia wspomniane publikacje będą nazywane w skrócie *Zapytania T-SQL* oraz *Programowanie T-SQL* lub prostu *te książki*.

Osoby, które przeczytały wcześniejszą edycję książek poświęconą wersji SQL Server 2005, mogą odnaleźć w tym wydaniu omówienie wielu nowych zagadnień, funkcji i udoskonaleń wprowadzonych w wersji SQL Server 2008, a także uaktualnione i wzbogacone omówienie starszej funkcjonalności.

Te książki koncentrują się na praktycznych, często spotykanych problemach i prezentują różne metody radzenia sobie z nimi. Przedstawionych zostanie wiele udoskonalonych technik, które umożliwią specjalistom IT stosowanie naturalnych i efektywnych rozwiązań.

Te książki prezentują możliwości tworzenia zapytań bazujących na zbiorach, a następnie objaśniają, na czym polega przewaga tej techniki nad proceduralnym programowaniem z wykorzystaniem kursorów. Jednocześnie uczą rozpoznawać te nieliczne sytuacje, w których lepiej jest zastosować to drugie podejście.

Wcześniejsza książka z tej serii zatytułowana *Zapytania T-SQL* koncentruje się na definiowaniu i optymalizowaniu zapytań bazujących na zbiorach i autorzy tej publikacji zalecają uprzednie zapoznanie się z nią. Książka *Programowanie T-SQL* koncentruje się na programowaniu proceduralnym i bazuje na założeniu, że czytelnik posiadł umiejętność tworzenia zapytań.

Pięć pierwszych rozdziałów książki *Zapytania T-SQL* stanowi wprowadzenie do tematu logicznego oraz fizycznego przetwarzania. Wiedza ta pozwala czerpać maksymalne korzyści z informacji prezentowanych w pozostałych częściach obu książek.

Pierwszy rozdział jest poświecony logicznemu przetwarzaniu zapytań. Dokładnie opisuje logiczne fazy tego procesu, unikalne aspekty definiowania zapytań SQL oraz specjalny sposób myślenia, który trzeba zaadaptować, programując w relacyjnym środowisku zorientowanym na zbiory.

Drugi rozdział zawiera wprowadzenie do teorii zbiorów i logiki predykatów – matematycznych filarów, na których bazuje model relacyjny. Opanowanie tych podstaw pomaga w zrozumieniu modelu i języka. Ten rozdział został napisany przez Stevena Kassa, który pełnił również rolę głównego recenzenta technicznego obu książek. Steve dysponuje unikatowym połączeniem talentu matematycznego, informatycznego, SQL oraz literackiego, co uczyniło zeń idealnego kandydata do zaprezentowania tego zagadnienia.

Trzeci rozdział prezentuje model relacyjny. Zrozumienie modelu relacyjnego jest niezbędne do odpowiedniego projektowania baz danych i pomaga w pisaniu dobrego

kodu. W tym rozdziale zdefiniowane zostaną takie pojęcia z zakresu algebry relacyjnej, jak relacje, krotki i operatory. Następnie model relacyjny zostanie przedstawiony z innej perspektywy nazywanej *rachunkiem relacyjnym*. To podejście w większym stopniu przypomina podejście biznesowe, ponieważ model logiczny jest opisywany w kategorii predykatów i propozycji. W systemach transakcyjnych ogromne znaczenie ma integralność danych, dlatego duża część rozdziału poświęcona jest opisowi różnego typu więzów integralności. W końcowej części rozdziału omówiona zostanie normalizacja – formalny proces udoskonalania projektu bazy danych. Ten rozdział został napisany przez Dejana Sarka, który jest wiodącym specjalistą w zakresie modelu relacyjnego.

Czwarty rozdział poświęcony jest optymalizacji zapytań. Opisuje metodologię dostosowywania zapytań, która została opracowana przez Solid Quality Mentors i jest stosowana w systemach produkcyjnych wdrażanych przez firmę. W tym rozdziale opisane zostaną również metody stosowania indeksów i analizowania planów wykonania. Ponadto zaprezentowane zostaną najważniejsze informacje, które pozwolą zrozumieć przykłady prezentowane w pozostałych rozdziałach obu książek. Indeksy oraz plany zapytań stanowią ważne aspekty definiowania i optymalizowania zapytań.

Piąty rozdział, również napisany przez Steva Kass, zawiera omówienie złożonych koncepcji oraz algorytmów. W centrum uwagi znajdą się wybrane algorytmy często stosowane przez aparat SQL Server. Przedstawione zostaną najczarniejsze scenariusze, jak również mniej skomplikowane problemy. Poznanie złożonych algorytmów wykorzystywanych przez aparat baz danych pomaga w przewidzeniu m.in. w jaki sposób dodawanie danych do tabel wpływać będzie na wydajność określonych zapytań. Lepsze zrozumienie sposobu przetwarzania zapytań przez aparat SQL Server pomaga w ich optymalizowaniu.

Kolejne rozdziały prezentują zaawansowane metody definiowania i optymalizowania zapytań, z uwzględnieniem zarówno aspektów logicznych, jak i fizycznych. Omówione zostaną następujące zagadnienia: zapytania podrzędne, wyrażenia tabelaryczne i funkcje szeregujące, złączenia i operacje na zbiorach, agregacje i przestawianie danych, opcje TOP i APPLY, modyfikacje danych, wykonywanie zapytań na partycjonowanych tabelach, a także grafy, drzewa, hierarchie oraz zapytania rekurencyjne.

Rozdział poświęcony wykonywaniu zapytań na partycjonowanych tabelach został napisany przez Lubora Kollara. Lubor kierował początkową fazą rozwoju tabel oraz indeksów partycjonowanych. Jego pracy zawdzięczamy wiele dostępnych obecnie funkcji. Aktualnie Lubor należy do zespołu SQL Server Customer Advisory Team (SQL CAT) i często zdarza mu się doradzać klientom, którzy posiadają duże implementacje partycjonowanych tabel oraz indeksów.

Dodatek A zawiera zagadki logiczne. W tej części czytelnicy mają okazję poprawić umiejętność myślenia logicznego przez rozwiązywanie łamigłówek. Instrukcje SQL zasadniczo bazują na logice. W związku z tym rozwijanie umiejętności logicznego myślenia pomaga w rozwiązywaniu problemów związanych z pisaniem zapytań. Co więcej, wspólne rozwiązywanie zagadek może być rozrywką dla całej rodziny. Zaprezentowane łamigłówki stanowią zestawienie zagadek logicznych, które publikowałem w serii artykułów poświęconych T-SQL w *SQL Server Magazine*. Chciałbym podziękować *SQL Server Magazine* za zgodę na udostępnienie tych zagadek czytelnikom książki.

Książka *Programowanie T-SQL* koncentruje się na programistycznych konstrukcjach T-SQL i rozszerza opis o technologie XML oraz XQuery, a także integrację CLR. Omówione zostaną następujące zagadnienia: widoki, definiowane przez użytkowników funkcje, procedury składowane, wyzwalacze, transakcje i współbieżność, obsługa wyjątków, tabele tymczasowe i zmienne tabelaryczne, kursory, dynamiczne instrukcje SQL, obsługa danych typu Data i godzina, definiowane przez użytkowników typy CLR, wsparcie dla danych tymczasowych w modelu relacyjnym, XML oraz XQuery (łącznie z otwartym schematem), dane przestrzenne, śledzenie dostępu i modyfikacji danych oraz funkcja Service Broker.

Rozdziały zawierające omówienie definiowanych przez użytkownika typów CLR, wsparcia dla danych tymczasowych w modelu relacyjnym, XML oraz XQuery zostały napisane przez Dejana Sarka. Jak wspomniałem, Dejan posiada ogromną wiedzę o modelu relacyjnym i ciekawe spostrzeżenia na temat samego modelu oraz możliwości stosowania konstrukcji prezentowanych w tym rozdziale.

Rozdział poświęcony danym przestrzennym został napisany przez Eda Katibaha oraz Isaaca Kunena. Ed oraz Isaac należą do zespołu programistycznego SQL Server i kierowali implementacją obsługi danych przestrzennych w SQL Server 2008. To dla mnie ogromny zaszczyt, że projektanci funkcji zgodzili się napisać poświęcony jej rozdział. Wsparcie dla danych przestrzennych stanowi nowość w SQL Server 2008 i wiąże się z wprowadzeniem nowych typów danych, metod oraz indeksów. Ten rozdział nie pretenduje do miana kompleksowego omówienia danych przestrzennych ani encyklopedii wszystkich metod przestrzennych, które są obecnie wspierane przez SQL Server. Pełni rolę wprowadzenia do podstawowych koncepcji przestrzennych i prezentuje kluczowe konstrukcje programistyczne niezbędne do pomyślnego stosowania nowej funkcji w SQL Server.

Rozdział poświęcony śledzeniu dostępu i modyfikacji danych został napisany przez Grega Low. Greg uzyskał tytuł SQL Server MVP i pełni funkcję dyrektora organizacji SolidQ Australia. Greg ma również wieloletnie doświadczenie w pracy z SQL Server (szkoleniu, wykładach i publikacjach) i cieszy się dużym szacunkiem w społeczności SQL Server. Ten rozdział omawia rozszerzone zdarzenia, inspekcje, śledzenie zmian i przechwytywanie danych modyfikacji. Technologie stanowiące główny temat tego rozdziału służą do śledzenia dostępu oraz modyfikacji danych i zostały wprowadzone w wersji SQL Server 2008. Na początku można odnieść wrażenie, że technologie pokrywają się lub zaprzeczają, a zalecenia dotyczące ich stosowania są niejasne. Jednak ten rozdział analizuje poszczególne technologie, prezentując ich możliwości oraz ograniczenia, a także objaśniając zamierzone zastosowanie.

Ostatni rozdział poświęcony funkcji Service Broker (SSB) został napisany przez Rogera Woltera. Roger pełni funkcję menedżera projektu w zespole ds. rozwoju SQL Server i kierował początkową fazą wprowadzania funkcji SSB do SQL Server. Nikt nie może opisać komponentu lepiej niż jego projektant. Ta z początku niedoceniana funkcja SQL Server 2005 jest obecnie stosowana w środowiskach produkcyjnych w wielu różnych aplikacjach. Ten rozdział prezentuje architekturę komponentu SSB i metody wykorzystywania go do budowania solidnych, asynchronicznych aplikacji bazodanowych. Edycja SQL Server 2008 oferuje wsparcie dla nowych funkcji dodanych do SSB i bazuje

na doświadczeniach oraz metodologiach zastosowanych w aplikacjach SSB w wersji SQL Server 2005. Najważniejsze nowe funkcje to Queue Priorities, External Activation oraz dodatkowa aplikacja do rozwiązywania problemów korzystająca z zebranych przez zespół ds. SSB obserwacji klientów, którzy wdrożyli aplikacje.

Wymagania sprzętowe oraz programowe

Do wykonywania ćwiczeń prezentowanych w tej książce oraz uruchamiania wszystkich przykładów kodu potrzebny będzie Microsoft SQL Server 2008 (zalecana edycja to Developer lub Enterprise) oraz Microsoft Visual Studio 2008 (edycja Professional lub Database). Subskrybenci MSDN mogą pobrać SQL Server 2008 oraz Visual Studio 2008 z witryny *http://msdn.microsoft.com*. Można również nieodpłatnie pobrać 180-dniową wersję ewaluacyjną edycji SQL Server 2008 Enterprise ze strony *http://www.microsoft.com/sqlserver/2008/en/us/trial-software.aspx* oraz 90-dniową wersję ewaluacyjną edycji Visual Studio 2008 Professional ze strony *http://www.microsoft.com/visualstudio/en-us/try/default.mspx*.

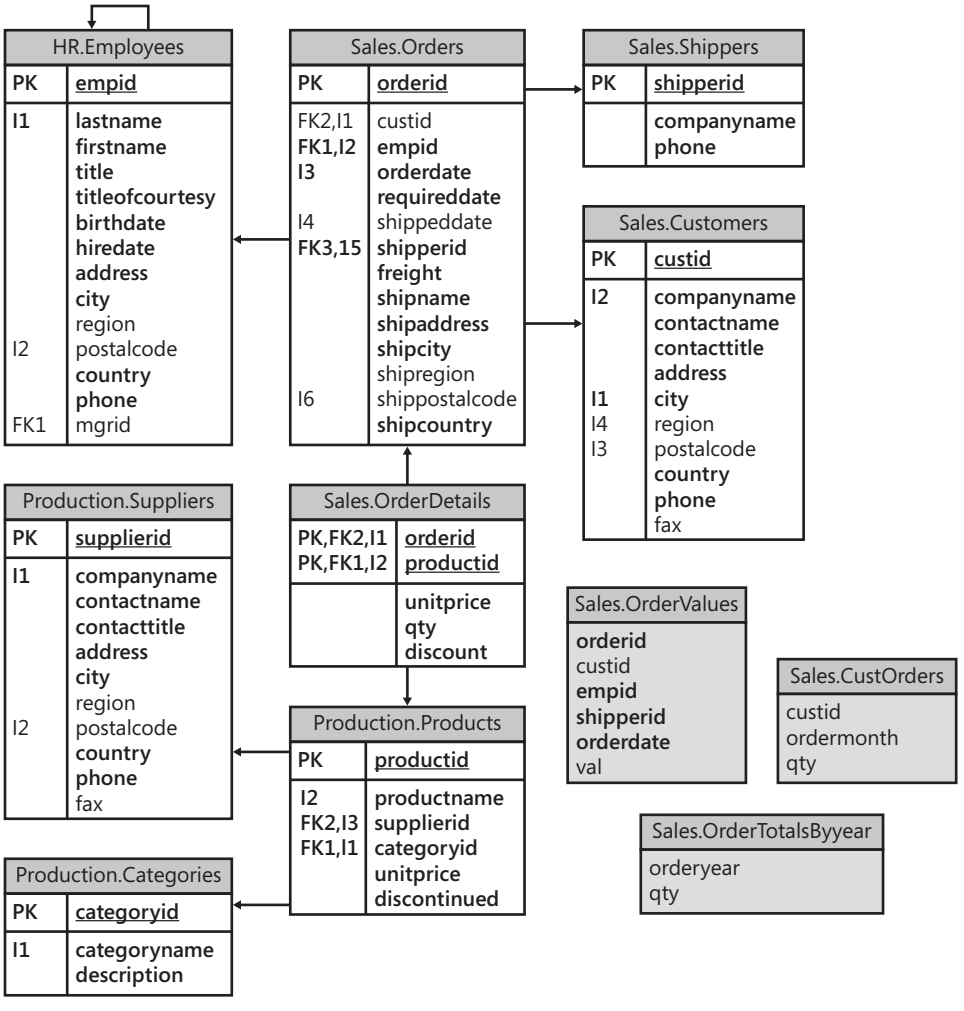
Wymagania systemowe oprogramowania SQL Server 2008 dostępne są pod adresem *http://msdn.microsoft.com/en-us/library/ms143506.aspx*, a oprogramowania Visual Studio 2008 pod adresem *http://www.microsoft.com/visualstudio/en-us/products/default.mspx*.

Materiały pomocnicze oraz przykładowa baza danych

Pomocnicza witryna sieci Web poświęcona tym książkom zawiera wszystkie prezentowane przykłady kodu, erratę, dodatkowe zasoby i nie tylko. Pomocnicza witryna sieci Web znajduje się pod adresem: *http://www.insidetsql.com*.

Witryna sieci Web zawiera przykładową bazę danych oraz dodatkowe pliki konieczne do uruchomienia przykładów kodu.

Po pobraniu kodu źródłowego należy uruchomić plik skryptu InsideTSQL2008.sql, aby stworzyć pomocniczą bazę danych InsideTSQL2008, która jest wykorzystywana w wielu przykładach. Model danych w bazie danych InsideTSQL2008 został zaprezentowany na rysunku W-1.



Rysunek W-1 Model danych w bazie danych InsideTSQL2008

Dodatkowe materiały dostępne online

Nowe i zaktualizowane materiały pomocnicze dla tej książki będą publikowane w witrynie Microsoft Press Online Developer Tools. Materiały te mogą obejmować modyfikacje zawartości książki, artykuły, łącza do pomocniczych zasobów, erratę, przykładowe rozdziały i nie tylko. Wspomniana witryna sieci Web znajduje się pod adresem *http://microsoftpresssrv.libredigital.com/serverclient/* i jest regularnie aktualizowana.

Pomoc techniczna

Dołożyliśmy wszelkich starań, aby zapewnić jak najwyższą jakość tej książki i materiałów pomocniczych zamieszczonych w witrynie sieci Web. Ewentualne poprawki oraz zmiany będą sukcesywnie dodawane do artykułu Microsoft Knowledge Base.

Wydawnictwo Microsoft Press oferuje pomoc techniczną dotyczącą wydawanych książek na następującej stronie:

http://www.microsoft.com/learning/support/books/.

Pytania i komentarze

Wszelkie uwagi dotyczące niniejszej książki bądź pytania, które pozostają bez odpowiedzi mimo zapoznania się ze wspomnianymi witrynami, proszę kierować do mnie za pośrednictwem poczty e-mail:

itzik@SolidQ.com

lub tradycyjnej poczty pod adresem:

Microsoft Press
Attn: *Inside Microsoft SQL Server 2008: T-SQL Querying and Inside Microsoft SQL Server 2008: T-SQL Programming* Editor
One Microsoft Way
Redmond, WA 98052-6399

Żaden z powyższych adresów nie udziela pomocy technicznej dotyczącej oprogramowania Microsoft.

ROZDZIAŁ 1

Widoki

Itzik Ben-Gan

Niniejsza książka stanowi kontynuację książki *Microsoft SQL Server 2008 od środka: Zapytania w języku T-SQL* (APN Promise, 2009) i przyjęto w niej założenie, że czytelnik zaznajomił się z wspomnianą publikacją lub ma adekwatną wiedzę dotyczącą zapytań. W dalszej części omówienia wspomniane książki będą nazywane w skrócie *Zapytania T-SQL* oraz *Programowanie T-SQL*. Książka *Programowanie T-SQL* koncentruje się na konstrukcjach programistycznych T-SQL. W związku z tym pierwszy rozdział poświęcony został widokom, które są zaliczane przez niektórych do konstrukcji programistycznych, a przez innych do zapytań.

Ten rozdział rozpoczyna się od krótkiego omówienia widoków oraz ich zastosowań. Następnie zaprezentowane zostaną szczegółowe metody pracy z widokami. Przedstawione zostaną między innymi techniki upraszczania zapytań przy użyciu widoków, a także podnoszenia wydajności bazy danych za pomocą indeksowanych widoków.

Co to są widoki?

Widok stanowi nazwaną tabelę wirtualną, która jest definiowana za pomocą zapytania i wykorzystywana jak tabela. W odróżnieniu od trwałych tabel, widok nie ma fizycznej reprezentacji danych, o ile nie zostaną na nim utworzone indeksy. Aby wykonać zapytanie na nieindeksowanym widoku, SQL Server musi w rzeczywistości uzyskać dostęp do odpowiednich tabel. Prezentowane w tym rozdziale informacje dotyczyć będą widoków nieindeksowanych (o ile nie zaznaczono inaczej).

Tworząc widok, należy zdefiniować jego nazwę oraz zapytanie. Microsoft SQL Server przechowuje jedynie metadane widoku opisujące obiekty, kolumny, zabezpieczenia, zależności itp. Gdy użytkownik wykonuje zapytanie na widoku w celu pobrania lub zmodyfikowania danych, procesor zapytań zastępuje odwołanie do widoku jego definicją. Innymi słowy procesor zapytań „rozwija" definicję widoku i generuje plan wykonania, który uzyskuje dostęp do bazowych obiektów.

Widoki odgrywają znaczącą rolę w bazie danych. Jednym z najważniejszych zastosowań widoków jest mechanizm abstrakcji. Widoki ułatwiają na przykład prezentowanie w zależności od sytuacji bardziej lub mniej znormalizowanego obrazu wewnętrznych danych, bez wpływania na stopień normalizacji rzeczywistych danych. Widoki mogą posłużyć do uproszczenia rozwiązań, umożliwiając zastosowanie modularnego podejścia polegającego na stopniowym rozwiązywaniu złożonych problemów. Widoki można traktować również jako swego rodzaju warstwę bezpieczeństwa, ponieważ istnieje

możliwość przyznawania uprawnień do filtrowania lub modyfikowania danych jedynie za pośrednictwem widoków, zamiast bazowej tabeli (o ile właściciel widoku jest jednocześnie właścicielem bazowych obiektów).

Widoki mogą także wpływać na wydajność, gdy zostaną dla nich utworzone indeksy. Stworzenie indeksu klastrowego na widoku powoduje, że dane zostają zmaterializowane na dysku, a widok, który zwykle pełni rolę wirtualną, nabiera fizycznego charakteru. Widoki indeksowane zostaną omówione w osobnej części tego rozdziału. Na razie wystarczy zapamiętać, że widok bez indeksu zwykle nie ma znaczącego wpływu na wydajność – pozytywnego ani negatywnego.

Podobnie jak w przypadku pozostałych wyrażeń tabelarycznych, m.in. tabeli pochodnej, wspólnego wyrażenia tabelarycznego (CTE) czy wbudowanej definiowanej przez użytkownika funkcji tabelarycznej, zapytanie definiujące widok musi spełniać trzy wymagania:

- Nie może zawierać klauzuli ORDER BY, o ile nie towarzyszy jej klauzula TOP lub FOR XML.
- Wszystkie kolumny wynikowe muszą mieć nazwy.
- Nazwy kolumn wynikowych nie mogą się powtarzać.

Zapytanie definiujące widok nie może zawierać klauzuli ORDER BY bez instrukcji TOP lub FOR XML, ponieważ widok powinien reprezentować tabelę. Tabela stanowi jednostkę logiczną, której wiersze nie są uporządkowane – w odróżnieniu od kursora charakteryzującego się uporządkowaniem wierszy. Ponadto wszystkie kolumny w prawidłowej tabeli muszą być nazwane i nazwy te muszą być unikalne. Aby zdefiniować nazwy docelowych kolumn w widoku, można podać je w nawiasie po nazwie widoku lub w postaci wbudowanych aliasów kolumn występujących po poszczególnych wyrażeniach.

W ramach ćwiczenia uruchamiamy następujący fragment kodu, aby stworzyć przykładowy widok CustsWithOrders w bazie danych InsideTSQL2008 (dodatkowe informacje o tej bazie danych znaleźć można we Wprowadzeniu):

```
SET NOCOUNT ON;
USE InsideTSQL2008;

IF OBJECT_ID('Sales.CustsWithOrders', 'V') IS NOT NULL
  DROP VIEW Sales.CustsWithOrders;
GO
CREATE VIEW Sales.CustsWithOrders
AS

SELECT custid, companyname, contactname, contacttitle,
  address, city, region, postalcode, country, phone, fax
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
GO
```

Powyższy widok zawiera dane klientów, którzy złożyli choć jedno zamówienie. Zapytanie widoku wykorzystuje predykat EXISTS do pobierania danych tych klientów, których zamówienia są przechowywane w tabeli Orders.

Wskazówka Choć zazwyczaj odradza się stosowanie znaku gwiazdki (*), można bez obaw użyć go w predykacie EXISTS. Optymalizator wie, że predykat EXISTS nie odwołuje się do określonego atrybutu wiersza i koncentruje się na fakcie istnienia, ignorując listę SELECT. Można to zauważyć, analizując plany wykonania tego typu zapytań. Jeśli istnieje indeks na filtrowanej kolumnie (w tym przykładzie O.custid), zostanie on zastosowany i nie wystąpią żadne dodatkowe operacje przeszukania. Poniższa instrukcja stanowi dodatkowy dowód na to, że lista SELECT jest ignorowana przez optymalizator:

```
IF EXISTS(SELECT 1/0) PRINT 'no error';
```

W normalnej sytuacji wykonanie zaprezentowanej instrukcji zakończyłoby się niepowodzeniem. Jednak w tym przypadku zakończy się ono pomyślnie, co świadczy o tym, że SQL Server nie przetwarza wyrażenia. Gdyby SQL Server przetwarzał wyrażenie, wygenerowany zostałby błąd dzielenia przez zero. Proces sprawdzania uprawnień do kolumn w wyrażeniu zawierającym znak * może stanowić pewne obciążenie, ale jest ono praktycznie niezauważalne. Zdaniem autora, zastosowanie znaku * jest bardziej naturalne i zrozumiałe niż określenie stałej wartości, a czytelność kodu stanowi bardzo istotny aspekt programowania.

W kolejnych częściach tego rozdziału omówione zostaną bardziej szczegółowe zagadnienia związane ze stosowaniem widoków, począwszy od argumentów przemawiających za zakazem definiowania klauzuli ORDER BY bez specyfikatora TOP lub FOR XML w zapytaniu widoku.

ORDER BY w widoku

Jak już wspominano, nie bez powodu nie można stosować w zapytaniach widoków klauzuli ORDER BY. Widok przypomina tabelę pod tym względem, iż reprezentuje jednostkę logiczną bez zdefiniowanej kolejności wierszy (w odróżnieniu od kursora, w którym rekordy są uporządkowane).

Spróbujmy uruchomić następujący fragment kodu, w którym usiłujemy zastosować klauzulę ORDER BY podczas tworzenia widoku CustsWithOrders:

```
ALTER VIEW Sales.CustsWithOrders
AS

SELECT country, custid, companyname, contactname, contacttitle,
  address, city, region, postalcode, phone, fax
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid)
ORDER BY country;
GO
```

Operacja nie powiedzie się i wygenerowany zostanie następujący komunikat o błędzie:

```
Msg 1033, Level 15, State 1, Procedure CustsWithOrders, Line 10
The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries,
and common table expressions, unless TOP or FOR XML is also specified.
```

Zgodnie z informacją stosowanie klauzuli ORDER BY nie jest całkowicie zabronione. Wymienionych zostaje kilka dozwolonych wyjątków (gdy klauzuli towarzyszy specyfikator TOP lub FOR XML). Warto mieć na uwadze, że słowa kluczowe TOP i FOR XML stanowią rozszerzenie języka T-SQL, nie są standardowymi elementami języka SQL. Konstrukcje TOP i ORDER BY bądź ORDER BY i FOR XML wchodzą w skład specyfikacji zbioru wynikowego, w odróżnieniu od samej klauzuli ORDER BY, która określa jedynie sposób prezentacji danych. W związku z tym konstrukcje TOP i ORDER BY bądź ORDER BY i FOR XML mogą być stosowane w definicji widoku, natomiast sama klauzula ORDER BY jest niedozwolona.

Jeśli istnieje potrzeba przesłania posortowanych danych do systemu klienckiego lub zadeklarowania obiektu kursora, który umożliwia przetwarzanie poszczególnych wierszy w określonej kolejności, można dodać klauzulę ORDER BY do zewnętrznego zapytania wykonywanego na widoku:

```
SELECT country, custid, companyname
FROM Sales.CustsWithOrders
ORDER BY country;
```

Warto zaznaczyć, że gdy zewnętrzne zapytanie zawiera operator TOP, klauzula ORDER BY pełni dwie role: określa, które wiersze ma wybrać opcja TOP oraz definiuje kolejność rekordów w wynikowym kursorze. Jednak gdy klauzula ORDER BY jest stosowana wraz z opcją TOP w wyrażeniu tabelarycznym (np. zapytaniu widoku), służy jedynie do określenia, które wiersze zostają wybrane przy użyciu opcji TOP. W takiej sytuacji widok nadal reprezentuje prawidłową tabelę (zbiór). Zapytanie wykonywane na widoku zwraca zbiór nieuporządkowanych wierszy, o ile zewnętrzne zapytanie nie zawiera klauzuli ORDER BY. Gdy zastosowana zostaje opcja TOP, definicja widoku (lub innego wyrażenia tabelarycznego) może zawierać klauzulę ORDER BY, ponieważ pełni ona jedynie funkcję pomocniczą dla opcji TOP. Zrozumienie tego mechanizmu pomaga w rozwijaniu prawidłowego kodu i eliminowaniu błędnych zastosowań wyrażeń tabelarycznych.

Próba stworzenia „sortowanego widoku" nie powiedzie się, ponieważ widok stanowi tabelę, a kolejność wierszy w tabeli nie jest określona. Programiści, którzy nie rozumieją, że zastosowanie połączenia klauzul TOP oraz ORDER BY w zapytaniu nie gwarantuje kolejności prezentowanych wierszy, mogą próbować przechytrzyć system w następujący sposób:

```
ALTER VIEW Sales.CustsWithOrders
AS

SELECT TOP (100) PERCENT
  country, custid, companyname, contactname, contacttitle,
  address, city, region, postalcode, phone, fax
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid)
```

```
ORDER BY country;
GO
```

Jaką rolę pełni klauzula ORDER BY w powyższym zapytaniu widoku? Sprawa jest dość skomplikowana, ponieważ opcja TOP nie stanowi standardu. Jednak z perspektywy zbioru klauzula ORDER BY nie ma żadnego znaczenia, gdyż pobierane są wszystkie wiersze spełniające kryteria filtra. Gdy na widoku wykonywane jest zapytanie, SQL Server nie musi gwarantować żadnej kolejności danych wynikowych, o ile zapytanie zewnętrzne nie zawiera klauzuli ORDER BY. Dokumentacja SQL Server 2008 Books Online zawiera pomocny opis tego mechanizmu: „Klauzula ORDER BY w definicji widoku służy jedynie do określania wierszy zwracanych przez klauzulę TOP. Klauzula ORDER BY w definicji widoku nie gwarantuje, że dane wynikowe zwrócone w efekcie wykonania zapytania na widoku będą posortowane, o ile samo zapytanie nie zawiera klauzuli ORDER BY".

Wykonajmy następujące zapytanie w systemie SQL Server 2008 (po zmodyfikowaniu widoku przez dołączenie specyfikacji TOP (100) PERCENT oraz klauzuli ORDER BY):

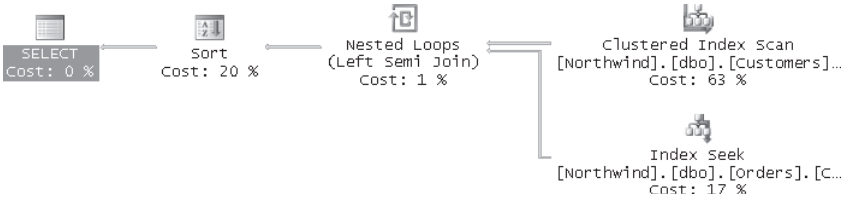
```
SELECT country, custid, companyname
FROM Sales.CustsWithOrders;
```

Poniżej zaprezentowany został wynik uruchomienia zapytania w przykładowym systemie:

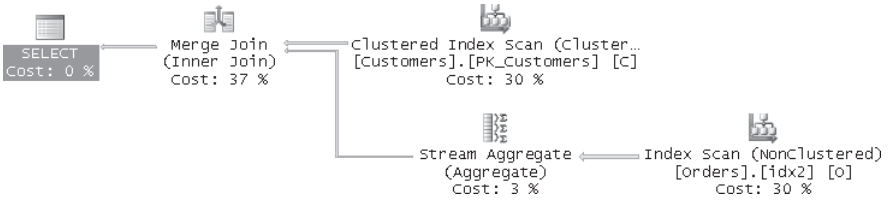
```
country         custid      companyname
--------------- ----------- ---------------
Germany         1           Customer NRZBB
Mexico          2           Customer MLTDN
Mexico          3           Customer KBUDE
UK              4           Customer HFBZG
Sweden          5           Customer HGVLZ
Germany         6           Customer XHXJV
France          7           Customer QXVLA
Spain           8           Customer QUHWH
France          9           Customer RTXGC
Canada          10          Customer EEALV
...
```

W systemach SQL Server 2008 oraz SQL Server 2005 optymalizator wykryje możliwość zignorowania specyfikacji TOP oraz klauzuli ORDER BY w definicji widoku, ze względu na obecność wyrażenia TOP (100) PERCENT. Natomiast SQL Server 2000 daje się przechytrzyć i posortuje wiersze, mimo iż zapytanie zewnętrzne nie zawiera klauzuli ORDER BY.

Po przeanalizowaniu planów wykonania w różnych wersjach systemu SQL Server można zauważyć, że w wersji 2000 (w podobnym scenariuszu zakładającym stworzenie widoku dla tabel Customers oraz Orders z przykładowej bazy danych Northwind) optymalizator posortuje dane (patrz Rysunek 1-1), natomiast w wersjach 2008 oraz 2005 nie dokona sortowania (patrz Rysunek 1-2).



RYSUNEK 1-1 Plan wykonania zapytania na widoku z klauzulą ORDER BY w wersji SQL Server 2000



RYSUNEK 1-2 Plan wykonania zapytania na widoku z klauzulą ORDER BY w wersjach SQL Server 2008 oraz SQL Server 2005

Jak można zauważyć, plan w wersji SQL Server 2000 zakłada wykorzystanie operatora sortowania do uporządkowania danych według kolumny *Country*. Natomiast optymalizator SQL Server 2008 całkowicie ignoruje połączenie klauzul TOP (100) PERCENT oraz ORDER BY. Optymalizator rozpoznał, iż klauzule TOP oraz ORDER BY nie mają żadnego znaczenia i w związku z tym nie tracił czasu na sortowanie danych według kolumny *country*. Niestety programiści przyzwyczajeni do działania wersji SQL Server 2000 mogą mylnie postrzegać to zachowanie jako błąd, mimo iż sama idea tworzenia posortowanego widoku jest nieprawidłowa. Firmy, które dokonały już migracji z systemu SQL Server 2000 do SQL Server 2005, są świadome tego problemu, ale dla organizacji, które zwlekały z aktualizacją do wersji SQL Server 2008, może być on niemiłą niespodzianką. Dlatego nowym użytkownikom wersji SQL Server 2008 zaleca się dokonanie analizy kodu i dodanie klauzuli ORDER BY do zewnętrznych zapytań w celu posortowania danych wynikowych.

UWAGA Projektant widoków w systemie SQL Server Management Studio (SSMS) pozwala na określenie kolejności sortowania, co prowadzi do wygenerowania specyfikacji TOP (100) PERCENT wraz z klauzulą ORDER BY w definicji widoku. Niewykluczone, że właśnie w ten sposób programiści odkryli to niefortunne rozwiązanie. Chociaż optymalizator zapytań w SQL Server 2008 ignoruje powyższą kombinację, projektant widoków w narzędziu SSMS nadal zachęca do stosowania tej nieskutecznej metody, która prowadzi jedynie do nieporozumień. Stanowczo odradzamy korzystywanie z tej absurdalnej techniki.

Na zakończenie uruchomimy następujący fragment kodu, aby usunąć widok CustsWithOrders:

```
IF OBJECT_ID('Sales.CustsWithOrders', 'V') IS NOT NULL
  DROP VIEW Sales.CustsWithOrders;
```

Odświeżanie widoków

SQL Server przechowuje metadane tworzonego widoku informujące o jego kolumnach, zabezpieczeniach, zależnościach itp. Zmiany obiektów, na których bazuje widok, nie są propagowane do metadanych widoku. W związku z tym zaleca się odświeżanie metadanych po wprowadzeniu tego typu modyfikacji (przy użyciu procedury składowanej *sp_refreshview*). W efekcie zmiany zostaną odzwierciedlone w widoku.

Za chwilę zaprezentujemy, co może się stać, gdy zmodyfikujemy schemat i nie odświeżymy metadanych widoku. Zaczynamy od wykonania następującego kodu w celu stworzenia pomocniczej tabeli T1 i widoku V1 w bazie danych tempdb:

```
USE tempdb;
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
GO
CREATE TABLE dbo.T1(col1 INT, col2 INT);
INSERT INTO dbo.T1(col1, col2) VALUES(1, 2);
GO
CREATE VIEW dbo.V1
AS
SELECT * FROM dbo.T1;
GO
```

Zasadniczo nie należy stosować znaku * w instrukcjach SELECT, jednak ten przykład służy wyłącznie do celów demonstracyjnych. W wyniku stworzenia widoku V1 SQL Server zapisał informacje o istniejących w danym momencie kolumnach: *col1* oraz *col2*. Uruchamiamy następujący fragment kodu w celu wykonania zapytania na widoku:

```
SELECT * FROM dbo.V1;
```

Uzyskujemy poniższe dane wynikowe zawierające obie kolumny:

```
col1        col2
----------- -----------
1           2
```

Następnie dodajemy kolumnę do tabeli T1:

```
ALTER TABLE dbo.T1 ADD col3 INT;
```

Zmiana schematu tabeli T1 nie została odzwierciedlona w metadanych widoku. SQL Server nadal sądzi, że widok zawiera tylko dwie kolumny. W konsekwencji po ponownym wykonaniu zapytania SELECT znowu uzyskamy dane wynikowe zawierające jedynie dwie kolumny:

```
col1        col2
----------- -----------
1           2
```

Aby odświeżyć metadane uruchamiamy procedurę składowaną *sp_refreshview* na widoku V1:

```
EXEC sp_refreshview 'dbo.V1';
```

Gdy ponownie wykonamy zapytanie SELECT, otrzymamy następujące dane wynikowe zawierające nową kolumnę *col3*:

```
col1        col2        col3
----------- ----------- -----------
1           2           NULL
```

To tylko jeden z przykładów ilustrujących sytuację, w której modyfikacja bazowych obiektów nie jest odzwierciedlana w metadanych widoku. Warto odświeżać metadane wszystkich widoków po modyfikacji schematów obiektów w bazie danych. Aby uniknąć żmudnego procesu wykonywania procedury składowanej *sp_refreshview* dla poszczególnych widoków, można użyć następującego zapytania:

```
SELECT
  N'EXEC sp_refreshview '
    + QUOTENAME(SCHEMA_NAME(schema_id) + N'.' + QUOTENAME(name), """)
    + ';' AS cmd
FROM sys.views
WHERE OBJECTPROPERTY(object_id, 'IsSchemaBound') = 0;
```

Zapytanie generuje listę instrukcji *sp_refreshview* dla wszystkich widoków w bazie danych, które nie są powiązane ze schematem.

Zagrożenie Należy sprawdzić kod wynikowy przed jego uruchomieniem. Choć kod powinien służyć do odświeżenia widoków, istnieje ryzyko, że złośliwy użytkownik z uprawnieniem do tworzenia widoków podstawi sfabrykowane nazwy widoków w celu przeprowadzenia niepożądanych operacji.

Na zakończenie usuwamy obiekty V1 oraz T1:

```
USE tempdb;
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
```

Rozwiązania modularne

Widoki mogą posłużyć do implementowania rozwiązań w sposób modularny. Cząstkowe problemy są rozwiązywane przy użyciu zapytań, które są następnie wykorzystywane do definiowania widoku. Ten proces ułatwia realizowanie skomplikowanych zadań, ponieważ pozwala skoncentrować się na rozwiązywaniu pojedynczych problemów.

W tym przykładzie zaprezentujemy, jak stosować podejście modularne do budowania złożonego widoku. Na początku uruchomimy następujący fragment kodu w celu stworzenia i wypełnienia tabeli Sales w bazie danych tempdb:

```
SET NOCOUNT ON;
USE tempdb;
IF OBJECT_ID('dbo.Sales', 'U') IS NOT NULL DROP TABLE dbo.Sales;
GO
CREATE TABLE dbo.Sales
(
```

```
  mnth DATE NOT NULL PRIMARY KEY,
/* Uwaga: Typ DATE został wprowadzony w SQL Server 2008.
   We wcześniejszych wersjach należy użyć typu DATETIME. /
  qty  INT  NOT NULL
);

INSERT INTO dbo.Sales(mnth, qty) VALUES
/ Uwaga: Table Value Constructor (rozszerzona klauzula VALUES) został
   wprowadzony w SQL Server 2008. We wcześniejszych wersjach należy
   użyć osobnej instrukcji INSERT VALUES dla każdego wiersza. */
  ('20071201', 100),
  ('20080101', 110),
  ('20080201', 120),
  ('20080301', 130),
  ('20080401', 140),
  ('20080501', 140),
  ('20080601', 130),
  ('20080701', 120),
  ('20080801', 110),
  ('20080901', 100),
  ('20081001', 110),
  ('20081101', 100),
  ('20081201', 120),
  ('20090101', 130),
  ('20090201', 140),
  ('20090301', 100),
  ('20090401', 100),
  ('20090501', 100),
  ('20090601', 110),
  ('20090701', 120),
  ('20090801', 110),
  ('20090901', 120),
  ('20091001', 130),
  ('20091101', 140),
  ('20091201', 100);
```

Uwaga W powyższym kodzie wykorzystywane są nowości wprowadzone w wersji SQL Server 2008: typ danych DATE oraz rozszerzona klauzula VALUES, która umożliwia wstawianie wielu wierszy za pomocą jednej instrukcji INSERT VALUES. Czytelnicy uruchamiający kod w starszej wersji SQL Server powinni użyć typu danych DATETIME i osobnych instrukcji INSERT dla poszczególnych wierszy.

Tabela zawiera po jednym wierszu dla każdego miesiąca z informacją o sumie sprzedaży (w kolumnie *qty*) oraz miesiącu (w kolumnie *mnth*). Jak można zauważyć, do przechowywania danych miesiąca wykorzystaliśmy typ DATE, który wspiera obliczenia wykonywane na datach. Pomimo iż zależy nam jedynie na informacji o miesiącu i roku, musieliśmy wpisać jakąś wartość w polu dnia i zdecydowaliśmy się na zastosowanie pierwszego dnia każdego miesiąca. Prezentując dane na ekranie, możemy wyodrębnić istotne elementy z pełnej wartości daty.

Celem tego przykładu jest wyświetlenie grup kolejnych miesięcy, które charakteryzują się tym samym trendem (wzrostowym, stałym, spadkowym lub nieznanym) oraz

zaprezentowanie trendu w kolumnie tekstowej o nazwie *trend*. Trend dla danego miesiąca można wyznaczyć, odejmując wartość *qty* z poprzedniego miesiąca od wartości *qty* dla danego miesiąca. Jeśli różnica jest dodatnia, mamy do czynienia z trendem wzrostowym (*'up'*), jeśli ujemna z trendem spadkowym (*'down'*), jeśli równa zero z trendem stałym *'same'*), a w przeciwnym przypadku z trendem nieznanym (*'unknown'*). Oto oczekiwane dane wynikowe:

```
start_range end_range trend
----------- --------- -------
200712      200712    unknown
200801      200804    up
200805      200805    same
200806      200809    down
200810      200810    up
200811      200811    down
200812      200902    up
200903      200903    down
200904      200905    same
200906      200907    up
200908      200908    down
200909      200911    up
200912      200912    down
```

Utworzenie pojedynczego zapytania, które rozwiązuje przedstawiony problem, może okazać się bardzo trudne, dlatego warto podzielić zadanie na moduły. Zaprezentujemy dwa możliwe rozwiązania: jedno bazujące na zapytaniach podrzędnych, a drugie na funkcji szeregującej.

Na początku wyliczmy znak różnicy między wartościami *qty* aktualnego i poprzedniego miesiąca. W tym celu stwórzmy widok SalesTrendSgn w następujący sposób:

```
IF OBJECT_ID('dbo.SalesTrendSgn', 'V') IS NOT NULL DROP VIEW dbo.SalesTrendSgn;
GO
CREATE VIEW dbo.SalesTrendSgn
AS

SELECT mnth, qty,
  SIGN((S1.qty -
         (SELECT TOP (1) qty
          FROM dbo.Sales AS S2
          WHERE S2.mnth < S1.mnth
          ORDER BY S2.mnth DESC))) AS sgn
FROM dbo.Sales AS S1;
GO

SELECT * FROM dbo.SalesTrendSgn;
```

Wykonanie powyższego kodu powoduje wygenerowanie następujących danych wynikowych:

```
mnth         qty         sgn
------------ ----------- -----------
2007-12-01   100         NULL
2008-01-01   110         1
2008-02-01   120         1
2008-03-01   130         1
```

```
2008-04-01   140         1
2008-05-01   140         0
2008-06-01   130         -1
2008-07-01   120         -1
2008-08-01   110         -1
2008-09-01   100         -1
2008-10-01   110         1
2008-11-01   100         -1
2008-12-01   120         1
2009-01-01   130         1
2009-02-01   140         1
2009-03-01   100         -1
2009-04-01   100         0
2009-05-01   100         0
2009-06-01   110         1
2009-07-01   120         1
2009-08-01   110         -1
2009-09-01   120         1
2009-10-01   130         1
2009-11-01   140         1
2009-12-01   100         -1
```

Funkcja SIGN zwraca 1 dla wyniku dodatniego, 0 dla wyniku równego zero, –1 dla wyniku ujemnego oraz NULL dla wyniku NULL. Kolumna *sgn* reprezentuje trend sprzedaży dla danego miesiąca. Teraz chcemy pogrupować wszystkie kolejne miesiące, które charakteryzują się tym samym trendem sprzedaży. W tym celu musimy na początku wyznaczyć czynnik grupowania, czyli wartość identyfikującą grupę. Jednym z czynników grupowania może być najwcześniejszy miesiąc w przyszłości, w którym trend różni się od trendu dla aktualnego miesiąca. Jak łatwo zauważyć, wartość ta będzie jednakowa dla wszystkich kolejnych miesięcy o tym samym trendzie.

Uruchamiamy następujący fragment kodu, aby stworzyć widok SalesGrp wyznaczający czynnik grupowania:

```
IF OBJECT_ID('dbo.SalesGrp', 'V') IS NOT NULL DROP VIEW dbo.SalesGrp;
GO
CREATE VIEW dbo.SalesGrp
AS

SELECT mnth, sgn,
  (SELECT MIN(mnth) FROM dbo.SalesTrendSgn AS V2
   WHERE V2.sgn <> V1.sgn
     AND V2.mnth > V1.mnth) AS grp
FROM dbo.SalesTrendSgn AS V1;
GO

SELECT * FROM dbo.SalesGrp;
```

Wygenerowane zostaną następujące dane wynikowe:

```
mnth         sgn         grp
------------ ----------- ----------
2007-12-01   NULL        NULL
2008-01-01   1           2008-05-01
2008-02-01   1           2008-05-01
2008-03-01   1           2008-05-01
```

```
2008-04-01   1           2008-05-01
2008-05-01   0           2008-06-01
2008-06-01   -1          2008-10-01
2008-07-01   -1          2008-10-01
2008-08-01   -1          2008-10-01
2008-09-01   -1          2008-10-01
2008-10-01   1           2008-11-01
2008-11-01   -1          2008-12-01
2008-12-01   1           2009-03-01
2009-01-01   1           2009-03-01
2009-02-01   1           2009-03-01
2009-03-01   -1          2009-04-01
2009-04-01   0           2009-06-01
2009-05-01   0           2009-06-01
2009-06-01   1           2009-08-01
2009-07-01   1           2009-08-01
2009-08-01   -1          2009-09-01
2009-09-01   1           2009-12-01
2009-10-01   1           2009-12-01
2009-11-01   1           2009-12-01
2009-12-01   -1          NULL
```

Jak łatwo zauważyć, wartości kolumny *grp* są unikatowe dla wszystkich grup sąsiadujących miesięcy o tym samym trendzie. Jedyny wyjątek stanowią dwie wartości NULL. Wartość NULL pojawiła się w wierszu 2007-12-01, ponieważ trend dla tego miesiąca nie jest znany. Natomiast wartość NULL w wierszu 2009-12-01 wynika z tego, że nie istnieją dane dla kolejnych miesięcy. Te dwie wartości NULL należą do dwóch różnych grup trendów, ale można z łatwością je rozróżnić, definiując grupę przy użyciu kolumny *sgn* (reprezentującej trend) oraz *grp*.

Ostatnie zadanie jest dość oczywiste, musimy pogrupować dane według *sgn* oraz *grp*, definiując wartość *MIN*(*mnth*) jako początek przedziału oraz wartość *MAX*(*mnth*) jako koniec przedziału. Ponadto możemy użyć wyrażenia CASE do przekonwertowania wartości *sgn* w celu przedstawienia bardziej opisowej reprezentacji trendu.

Uruchamiamy następujący fragment kodu w celu stworzenia widoku SalesTrends, który stanowi implementację tego zadania:

```
IF OBJECT_ID('dbo.SalesTrends', 'V') IS NOT NULL
  DROP VIEW dbo.SalesTrends;
GO
CREATE VIEW dbo.SalesTrends
AS

SELECT
  CONVERT(VARCHAR(6), MIN(mnth), 112) AS start_range,
  CONVERT(VARCHAR(6), MAX(mnth), 112) AS end_range,
  CASE sgn
    WHEN -1 THEN 'down'
    WHEN  0 THEN 'same'
    WHEN  1 THEN 'up'
    ELSE         'unknown'
  END AS trend
FROM dbo.SalesGrp
```

```
GROUP BY sgn, grp;
GO
```

Aby uzyskać oczekiwane dane wynikowe (zaprezentowane na początku przykładu), wystarczy wykonać zapytanie na widoku SalesTrends:

```
SELECT start_range, end_range, trend
FROM dbo.SalesTrends
ORDER BY start_range;
```

Ten sam problem można również rozwiązać w sposób bardziej efektywny, korzystając z funkcji szeregujących. Na początku tworzymy widok o nazwie SalesRN, przypisując poszczególnym wierszom z tabeli Sales numery porządkowe na podstawie kolumny *mnth*:

```
IF OBJECT_ID('dbo.SalesRN', 'V') IS NOT NULL
  DROP VIEW dbo.SalesRN;
GO
CREATE VIEW dbo.SalesRN
AS

SELECT mnth, qty, ROW_NUMBER() OVER(ORDER BY mnth) AS rn
FROM dbo.Sales;
GO

SELECT * FROM dbo.SalesRN;
```

Kod powoduje wygenerowanie następujących danych wynikowych:

```
mnth         qty         rn
------------ ----------- --------------------
2007-12-01   100         1
2008-01-01   110         2
2008-02-01   120         3
2008-03-01   130         4
2008-04-01   140         5
2008-05-01   140         6
2008-06-01   130         7
2008-07-01   120         8
2008-08-01   110         9
2008-09-01   100         10
2008-10-01   110         11
2008-11-01   100         12
2008-12-01   120         13
2009-01-01   130         14
2009-02-01   140         15
2009-03-01   100         16
2009-04-01   100         17
2009-05-01   100         18
2009-06-01   110         19
2009-07-01   120         20
2009-08-01   110         21
2009-09-01   120         22
2009-10-01   130         23
2009-11-01   140         24
2009-12-01   100         25
```

W widoku SalesTrendSgn złączamy dwa wystąpienia widoku SalesRN, aby dopasować aktualny wiersz z wierszem poprzedniego miesiąca. Następnie uzyskujemy dostęp do wartości *qty* dla aktualnego i poprzedniego miesiąca i wyliczamy znak różnicy. Oto kod nowej wersji widoku SalesTrendSgn, który bazuje na numerach wierszy z widoku SalesRN:

```
IF OBJECT_ID('dbo.SalesTrendSgn', 'V') IS NOT NULL
  DROP VIEW dbo.SalesTrendSgn;
GO
CREATE VIEW dbo.SalesTrendSgn
AS

SELECT Cur.mnth, Cur.qty, SIGN(Cur.qty – Prv.qty) AS sgn
FROM dbo.SalesRN AS Cur
  LEFT OUTER JOIN dbo.SalesRN AS Prv
    ON Cur.rn = Prv.rn + 1;
GO
```

Możemy dodatkowo zoptymalizować rozwiązanie, modyfikując widok SalesGrp wyliczający czynnik grupowania w następujący sposób:

```
IF OBJECT_ID('dbo.SalesGrp', 'V') IS NOT NULL
  DROP VIEW dbo.SalesGrp;
GO
CREATE VIEW dbo.SalesGrp
AS

SELECT mnth, sgn,
  DATEADD(month,
    -1 * ROW_NUMBER() OVER(PARTITION BY sgn ORDER BY mnth),
    mnth) AS grp
FROM dbo.SalesTrendSgn;
GO
```

Operacje prowadzące do wyznaczenia czynnika grupowania są nieco skomplikowane. Obliczamy numer wiersza (*rn*) w oparciu o kolejność *mnth*, partycjonując według *sgn* (trendu). To oznacza, że dla każdego trendu możemy mieć wiele ciągłych grup rozdzielonych lukami. Teraz zastanówmy się, w jaki sposób wartość *mnth* wzrasta w wybranym trendzie w porównaniu z wzrostem wartości *rn*. Obie wartości rosną o jeden w ramach tej samej ciągłej grupy. Gdy natrafimy na lukę, wartość *mnth* zmienia się o więcej niż jeden, natomiast *rn* nadal wzrasta o jednostkę. W związku z tym, jeśli odejmiemy *rn* miesięcy od *mnth*, wynik dla każdej z ciągłych grup będzie stały i unikalny. Jak wspomniano, zrozumienie tego procesu może przysporzyć pewnych trudności, dlatego warto wyodrębnić zapytanie SalesGrp i przetestować je na różne sposoby. Na przykład, zwracając sam numer wiersza (bez stosowania go w obliczeniu).

Na zakończenie stwórzmy widok SalesTrends, aby pogrupować dane według pól *sgn* oraz *grp*, zwracając przedziały kolejnych miesięcy o tym samym trendzie.

```
IF OBJECT_ID('dbo.SalesTrends', 'V') IS NOT NULL
  DROP VIEW dbo.SalesTrends;
GO
CREATE VIEW dbo.SalesTrends
AS
```

```
SELECT
  CONVERT(VARCHAR(6), MIN(mnth), 112) AS start_range,
  CONVERT(VARCHAR(6), MAX(mnth), 112) AS end_range,
  CASE sgn
    WHEN -1 THEN 'down'
    WHEN  0 THEN 'same'
    WHEN  1 THEN 'up'
    ELSE         'unknown'
  END AS trend
FROM dbo.SalesGrp
GROUP BY sgn, grp;
GO
```

W wyniku wykonania zapytania na widoku SalesTrends otrzymamy oczekiwane dane wynikowe:

```
SELECT start_range, end_range, trend
FROM dbo.SalesTrends
ORDER BY start_range;
```

Warto pamiętać, że SQL Server 2008 wspiera wspólne wyrażenia tabelaryczne (CTE), które również umożliwiają opracowywanie rozwiązań z wykorzystaniem modularnego podejścia. Co więcej, wyrażenia CTE stanowią w pewnym sensie wbudowane widoki, które istnieją jedynie w zasięgu zewnętrznego zapytania. Tak naprawdę widoki pośrednie zastosowane w tym przykładzie służą jedynie do implementacji podejścia modularnego. Dlatego lepszym rozwiązaniem jest stworzenie końcowego widoku SalesTrends, który będzie definiowany przy użyciu modularnych wyrażeń CTE. Uruchamiamy następujący fragment kodu w celu zmodyfikowania widoku SalesTrends tak, aby zamiast widoków pośrednich zastosowane zostały wyrażenia CTE zdefiniowane w ramach tej samej instrukcji WITH:

```
ALTER VIEW dbo.SalesTrends
AS

WITH SalesRN AS
(
  SELECT mnth, qty, ROW_NUMBER() OVER(ORDER BY mnth) AS rn
  FROM dbo.Sales
),
SalesTrendSgn AS
(
  SELECT Cur.mnth, Cur.qty, SIGN(Cur.qty – Prv.qty) AS sgn
  FROM SalesRN AS Cur
    LEFT OUTER JOIN SalesRN AS Prv
      ON Cur.rn = Prv.rn + 1
),
SalesGrp AS
(
  SELECT mnth, sgn,
    DATEADD(month,
      -1 * ROW_NUMBER() OVER(PARTITION BY sgn ORDER BY mnth),
      mnth) AS grp
  FROM SalesTrendSgn
)
```

```
SELECT
  CONVERT(VARCHAR(6), MIN(mnth), 112) AS start_range,
  CONVERT(VARCHAR(6), MAX(mnth), 112) AS end_range,
  CASE sgn
    WHEN -1 THEN 'down'
    WHEN  0 THEN 'same'
    WHEN  1 THEN 'up'
    ELSE         'unknown'
  END AS trend
FROM SalesGrp
GROUP BY sgn, grp;
GO
```

W wyniku wykonania poniższego zapytania na widoku SalesTrends otrzymamy oczekiwane dane wynikowe:

```
SELECT start_range, end_range, trend
FROM dbo.SalesTrends
ORDER BY start_range;
```

Podsumowując, modularne podejście do rozwiązywania problemów prowadzi do uproszczenia procesu i zmniejszenia ryzyka błędów.

Na zakończenie uruchamiamy następujące instrukcje porządkujące:

```
IF OBJECT_ID('dbo.SalesTrends', 'V') IS NOT NULL DROP VIEW dbo.SalesTrends;
IF OBJECT_ID('dbo.SalesGrp', 'V') IS NOT NULL DROP VIEW dbo.SalesGrp;
IF OBJECT_ID('dbo.SalesTrendSgn', 'V') IS NOT NULL DROP VIEW dbo.SalesTrendSgn;
IF OBJECT_ID('dbo.SalesRN', 'V') IS NOT NULL DROP VIEW dbo.SalesRN;
IF OBJECT_ID('dbo.Sales', 'U') IS NOT NULL DROP TABLE dbo.Sales;
```

Modyfikowanie widoków

Należy pamiętać, że widok stanowi tabelę wirtualną, a w momencie wykonania zapytania na widoku SQL Server przetwarza instrukcję SELECT (z definicji widoku) i wykonuje ją na bazowej tabeli. Widok może być wykorzystywany nie tylko w instrukcjach SELECT, ale również w operacjach modyfikacji. W przypadku wykonania operacji modyfikacji na widoku SQL Server dokonuje zmian w bazowej tabeli. Widok pełni w takim scenariuszu rolę agenta lub pośrednika. Widok może posłużyć do ograniczenia dostępu do danych, gdy dozwolone są jedynie modyfikacje przeprowadzane za pomocą widoku, nie bezpośrednio na bazowej tabeli. W tym scenariuszu widok stanowi w pewnym sensie zabezpieczenie, stojąc na straży tajności i dostępu do danych.

Widoki umożliwiają na przykład stosowanie zabezpieczeń na poziomie wierszy.

Ostrzeżenie W tym podrozdziale zaprezentowane zostaną proste metody stosowania widoków do ochrony danych na poziomie wierszy. Zaprezentowana technika nie jest idealna i przydaje się jedynie w aplikacjach, w których bezpieczeństwo nie ma dużego znaczenia. Szczegółowe omówienie zabezpieczeń na poziomie wierszy oraz wad tego mechanizmu ochrony znaleźć można w następującym artykule: *http://technet.microsoft.com/en-us/library/cc966395.aspx*.

Poniższy fragment kodu służy do stworzenia tabeli o nazwie UserData z kolumną *loginname* o domyślnej wartości określanej przy użyciu funkcji SUSER_SNAME (aktualnej nazwy użytkownika). Stworzymy widok, który prezentuje wszystkie atrybuty (poza *loginname)* tylko aktualnemu użytkownikowi, wykorzystując do tego celu filtr *loginname* = *SUSER_SNAME*(). Na zakończenie odbierzemy roli *public* prawo do wykonywania operacji DML na tabeli i nadamy prawo do realizowania tych operacji na widoku:

```
USE tempdb;
IF OBJECT_ID('dbo.CurrentUserData', 'V') IS NOT NULL
  DROP VIEW dbo.CurrentUserData;
IF OBJECT_ID('dbo.UserData', 'T') IS NOT NULL
  DROP TABLE dbo.UserData;
GO
CREATE TABLE dbo.UserData
(
  keycol    INT         NOT NULL IDENTITY PRIMARY KEY,
  loginname sysname     NOT NULL DEFAULT (SUSER_SNAME()),
  datacol   VARCHAR(20) NOT NULL,
  /* ... pozostałe kolumny ... */
);
GO
CREATE VIEW dbo.CurrentUserData
AS

SELECT keycol, datacol
FROM dbo.UserData
WHERE loginname = SUSER_SNAME();
GO

DENY SELECT, INSERT, UPDATE, DELETE ON dbo.UserData TO public;
GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.CurrentUserData TO public;
```

Po zastosowaniu tych ograniczeń użytkownicy będą mogli wyświetlać i przetwarzać jedynie własne dane.

Modyfikacje realizowane za pośrednictwem widoku podlegają następującym ograniczeniom:

- Nie można wstawiać danych przy użyciu widoku, jeśli bazowa tabela zawiera choć jedną kolumnę, która nie znajduje się w widoku i nie akceptuje pośrednich wartości. Kolumna umożliwia pośrednie przypisanie, jeśli akceptuje wartości NULL lub przypisano do niej wartość domyślną, właściwość IDENTITY bądź typ ROWVERSION.
- Jeśli widok jest definiowany przy użyciu złączenia, instrukcja UPDATE lub INSERT może wpływać tylko na jedną stronę złączenia. Innymi słowy, instrukcja INSERT musi określać listę kolumn docelowych, które należą do jednej strony złączenia. Analogicznie, wszystkie kolumny modyfikowane przez kolumnę UPDATE muszą należeć do jednej strony złączenia. Aczkolwiek można odwoływać się do pozostałych kolumn w innych częściach zapytania m.in. po prawej stronie przypisania, w filtrze zapytania. Nie można usuwać danych z widoku zdefiniowanego przy użyciu zapytania ze złączeniem.

- Nie można modyfikować kolumny będącej wynikiem obliczeń. To ograniczenie dotyczy także wyrażeń skalarnych oraz agregatów. SQL Server nawet nie podejmuje próby analizy i odwrócenia obliczenia.
- Jeśli podczas tworzenia lub modyfikowania widoku określona została opcja WITH CHECK OPTION, instrukcje INSERT lub UPDATE, które nie spełniają kryteriów filtra w zapytaniu widoku, zostaną odrzucone. Dodatkowe informacje zostaną zaprezentowane w dalszej części tego rozdziału w części „Opcje widoku".

Operacje modyfikujące dane, które nie spełniają powyższych reguł, mogą zostać zrealizowane, jeśli dla widoku zdefiniowany został wyzwalacz INSTEAD OF. Wyzwalacz INSTEAD OF pozwala zastąpić oryginalną instrukcję własnym kodem. Można na przykład napisać niestandardowy kod, aby odwrócić obliczenia na kolumnach i zastosować zmiany bezpośrednio na bazowej tabeli. Wyzwalacze zostaną omówione w rozdziale 4 „Wyzwalacze".

Należy zachować szczególną ostrożność, zezwalając na modyfikację za pośrednictwem widoku zdefiniowanego z wykorzystaniem złączenia. Użytkownicy, którzy nie zdają sobie sprawy z tego, że obiektem docelowym modyfikacji jest widok, a nie tabela, mogą być w niektórych przypadkach zaskoczeni wynikiem operacji. Na przykład, gdy modyfikują „jedną" stronę złączenia jeden-do-wielu.

W kolejnym ćwiczeniu uruchomimy następujący fragment kodu w celu stworzenia tabel Customers oraz Orders, a także widoku CustOrders stanowiącego złączenie obu tabel:

```
SET NOCOUNT ON;
USE tempdb;
IF OBJECT_ID('dbo.CustOrders', 'V') IS NOT NULL DROP VIEW dbo.CustOrders;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;
GO

CREATE TABLE dbo.Customers
(
  cid   INT         NOT NULL PRIMARY KEY,
  cname VARCHAR(25) NOT NULL,
  /* pozostałe kolumny /
)
INSERT INTO dbo.Customers(cid, cname) VALUES
  (1, 'Cust 1'),
  (2, 'Cust 2');

CREATE TABLE dbo.Orders
(
  oid INT NOT NULL PRIMARY KEY,
  cid INT NOT NULL REFERENCES dbo.Customers,
  / pozostałe kolumny */
)
INSERT INTO dbo.Orders(oid, cid) VALUES
  (1001, 1),
  (1002, 1),
  (1003, 1),
  (2001, 2),
```

```
  (2002, 2),
  (2003, 2);
GO

CREATE VIEW dbo.CustOrders
AS

SELECT C.cid, C.cname, O.oid
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON O.cid = C.cid;
GO
```

Gdy wykonamy zapytanie na widoku:

```
SELECT cid, cname, oid FROM dbo.CustOrders;
```

Uzyskamy następujące dane wynikowe:

```
cid         cname                    oid
----------- ------------------------ -----------
1           Cust 1                   1001
1           Cust 1                   1002
1           Cust 1                   1003
2           Cust 2                   2001
2           Cust 2                   2002
2           Cust 2                   2003
```

Jak można zauważyć, wartości atrybutów klienta, takie jak nazwa firmy (*cname*), powtarzają się dla poszczególnych zamówień.

Załóżmy, że użytkownik z prawem do przeprowadzania modyfikacji na widoku, chce zmienić nazwę firmy na *'Cust 42'* dla zamówienia o identyfikatorze (*oid*) równym 1001. W tym celu uruchamia następującą instrukcję UPDATE:

```
UPDATE dbo.CustOrders
  SET cname = 'Cust 42'
WHERE oid = 1001;
```

Oczywiście, gdyby obiektem docelowym modyfikacji była tabela, a nie widok, po wykonaniu zapytania na ekranie pojawiłby się tylko jeden wiersz z wartością *'Cust 42'* w kolumnie *cname*. Jednak gdy obiektem docelowym jest widok, SQL Server w pośredni sposób modyfikuje tabelę Customers. W związku z tym zmieniona została wartość atrybutu *cname* dla klienta o identyfikatorze 1. Po ponownym wykonaniu zapytania na widoku CustOrders:

```
SELECT cid, cname, oid FROM dbo.CustOrders;
```

Wyświetlone zostaną następujące dane wynikowe:

```
cid         cname                    oid
----------- ------------------------ -----------
1           Cust 42                  1001
1           Cust 42                  1002
1           Cust 42                  1003
2           Cust 2                   2001
2           Cust 2                   2002
2           Cust 2                   2003
```

Jak widać, zmieniona została wartość *cname* dla zamówienia 1001. Wartość *cname* dla zamówienia 1001 w widoku reprezentowała wartość pola *cname* z tabeli Customers dla klienta powiązanego z zamówieniem 1001 (klienta o identyfikatorze 1 zgodnie z danymi z tabeli Orders). W związku z tym widok prezentuje zmodyfikowaną wartość *cname* dla wszystkich trzech zamówień złożonych przez klienta o identyfikatorze 1.

Na zakończenie uruchamiamy następujące instrukcje porządkujące:

```
USE tempdb;
IF OBJECT_ID('dbo.CurrentUserData', 'V') IS NOT NULL
  DROP VIEW dbo.CurrentUserData;
IF OBJECT_ID('dbo.UserData', 'U') IS NOT NULL
  DROP TABLE dbo.UserData;
IF OBJECT_ID('dbo.CustOrders', 'V') IS NOT NULL
  DROP VIEW dbo.CustOrders;
IF OBJECT_ID('Sales.Orders', 'U') IS NOT NULL
  DROP TABLE Sales.Orders;
IF OBJECT_ID('Sales.Customers', 'U') IS NOT NULL
  DROP TABLE Sales.Customers;
```

Opcje widoku

Tworząc lub modyfikując widok, można określać opcje kontrolujące jego zachowanie i funkcjonalność. Opcje ENCRYPTION, SCHEMABINDING oraz VIEW_METADATA są definiowane w nagłówku widoku, natomiast opcja CHECK OPTION jest umieszczana po zapytaniu.

ENCRYPTION

Opcja ENCRYPTION może być stosowana w widokach, definiowanych przez użytkownika funkcjach, procedurach składowanych i wyzwalaczach. W przypadku braku opcji ENCRYPTION SQL Server przechowuje tekst definiujący treść obiektu/procedury w postaci zwykłego tekstu w *sys.sql_modules*. Jeśli opcja ENCRYPTION zostanie określona, tekst obiektu zostaje przekonwertowany do nieczytelnego formatu. Jednak mechanizm szyfrowania nie gwarantuje ochrony własności intelektualnej, ponieważ istnieją sposoby deszyfrowania tekstu dla obiektu zapisanego przy użyciu opcji ENCRYPTION. W SQL Server 2008 uprzywilejowani użytkownicy mogą uzyskać dostęp do kodu obiektu za pośrednictwem dedykowanego połączenia administratora (DAC), bezpośrednio w plikach bazy danych lub w pamięci za pomocą debuggera. Szczegółowe informacje na temat opcji ENCRYPTION znaleźć można w dokumentacji SQL Server Books Online.

SCHEMABINDING

Opcja SCHEMABINDING służy do powiązania widoku lub definiowanej przez użytkownika funkcji ze schematem bazowych obiektów. Po stworzeniu widoku z opcją SCHEMABINDING SQL Server uniemożliwia usunięcie lub modyfikację schematu

bazowych obiektów, jeśli spowodowałoby to uszkodzenie widoku. Aby można było zastosować tę opcję w definicji widoku, muszą zostać spełnione dwa wymagania składniowe: wszystkie obiekty muszą być określane przy użyciu dwuczęściowych nazw (np. Sales.Orders zamiast Orders), a instrukcja SELECT nie może zawierać znaku * (nazwy wszystkich kolumn muszą zostać bezpośrednio zdefiniowane).

Do zaprezentowania przykładowego zastosowania opcji ENCRYPTION oraz SCHEMABINDING posłuży nam następujący fragment kodu, który ponownie tworzy wykorzystywany już wcześniej widok CustsWithOrders:

```
USE InsideTSQL2008;
IF OBJECT_ID('Sales.CustsWithOrders') IS NOT NULL
  DROP VIEW Sales.CustsWithOrders;
GO

CREATE VIEW Sales.CustsWithOrders WITH ENCRYPTION, SCHEMABINDING
AS

SELECT custid, companyname, contactname, contacttitle,
  address, city, region, postalcode, country, phone, fax
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT 1 FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
GO
```

> **Uwaga** Jeśli widok już istnieje, zamiast usuwać i ponownie go tworzyć, lepiej jest użyć instrukcji ALTER VIEW, ponieważ dzięki temu zachowane zostaną uprawnienia.

Jak można zauważyć, znak * stosowany w zapytaniu podrzędnym EXISTS zastąpiliśmy stałą wartością 1, aby spełnione były wymagania opcji SCHEMABINDING.

Gdy spróbujemy wyświetlić tekst widoku:

```
EXEC sp_helptext 'Sales.CustsWithOrders';
```

Uzyskamy następujący efekt:

```
The text for object 'Sales.CustsWithOrders' is encrypted.
```

Natomiast gdy spróbujemy sprawdzić definicję obiektu przy użyciu funkcji OBJECT_DEFINITION:

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.CustsWithOrders'));
```

Uzyskamy wynik NULL.

Gdy spróbujemy zmodyfikować jedną z kolumn stosowanych w widoku:

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

Zobaczymy następującą informację o błędzie:

```
Msg 5074, Level 16, State 1, Line 1
The object 'CustsWithOrders' 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.
```

Komunikat sygnalizuje, że istnieją obiekty zależne od kolumny, którą usiłujemy usunąć. W SQL Server 2008 dostępne są nowe dynamiczne obiekty zarządzania, które dostarczają informacji o zależnościach obiektów i pozwalają zidentyfikować je przed próbą dokonania zmian schematu. Na przykład funkcja *sys.dm_sql_referencing_entities* umożliwia sprawdzenie, czy istnieją obiekty zależne od obiektu, który planujemy usunąć lub zmodyfikować:

```
SELECT referencing_schema_name, referencing_entity_name
FROM sys.dm_sql_referencing_entities('Sales.Customers', 'OBJECT');
```

Planując modyfikację lub usunięcie kolumny, można sprawdzić, które obiekty są od niej zależne, wykonując zapytanie na widoku *sys.sql_expression_dependencies* w następujący sposób:

```
SELECT
  OBJECT_SCHEMA_NAME(referencing_id) AS referencing_schema_name,
  OBJECT_NAME(referencing_id) AS referencing_entity_name
FROM sys.sql_expression_dependencies
WHERE referenced_schema_name = N'Sales'
  AND referenced_entity_name = N'Customers'
  AND COL_NAME(referenced_id, referenced_minor_id) = N'address';
```

Oba zapytania zwracają *'Sales'* jako nazwę schematu oraz *'CustsWithOrders'* jako nazwę obiektu odwołującego się, informując, że widok Sales.CustsWithOrders bazuje na tabeli Sales.Customers, a dokładniej na kolumnie *Sales.Customers.address*.

CHECK OPTION

Określenie opcji WITH CHECK OPTION podczas tworzenia widoku uniemożliwia wykonywanie instrukcji INSERT oraz UPDATE, które nie spełniają kryterium filtra określonego w definicji widoku. Jeśli opcja ta nie zostanie zastosowana, widok zwykle akceptuje modyfikacje, które nie spełniają kryteriów filtra. Na przykład widok CustsWithOrders umożliwia wykonanie następującej instrukcji INSERT, pomimo iż służy ona do wstawienia danych nowego klienta, który nie złożył jeszcze żadnego zamówienia:

```
INSERT INTO Sales.CustsWithOrders(
         companyname, contactname, contacttitle, address, city, region,
         postalcode, country, phone, fax)
  VALUES(N'Customer ABCDE', N'ABCDE', N'ABCDE', N'ABCDE', N'ABCDE',
         N'ABCDE', N'ABCDE', N'ABCDE', N'ABCDE', N'ABCDE');
```

Dane nowego klienta zostały dodane do tabeli Customers, lecz nie zostaną wyświetlone w wyniku wykonania zapytania na widoku, ponieważ widok zawiera jedynie dane klientów z zamówieniami:

```
SELECT custid, companyname
FROM Sales.CustsWithOrders
WHERE companyname = N'Customer ABCDE';
```

Wynikiem wykonania powyższej instrukcji będzie zbiór pusty.

Jednak po wykonaniu zapytania bezpośrednio na tabeli Customers, zobaczymy dane nowego klienta:

```
SELECT custid, companyname
FROM Sales.Customers
WHERE companyname = N'Customer ABCDE';
```

Powyższe zapytanie zwróci informacje o kliencie ABCDE.

A teraz uruchomimy następujący fragment kodu w celu dodania opcji WITH CHECK OPTION do definicji widoku:

```
ALTER VIEW Sales.CustsWithOrders WITH ENCRYPTION, SCHEMABINDING
AS

SELECT custid, companyname, contactname, contacttitle,
  address, city, region, postalcode, country, phone, fax
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT 1 FROM Sales.Orders AS O
   WHERE O.custid = C.custid)
WITH CHECK OPTION;
GO
```

> **Uwaga** Modyfikując widok, trzeba ponownie zastosować wszystkie opcje, które mają zostać zachowane (w naszym przypadku opcje ENCRYPTION oraz SCHEMABINDING). Opcje, które nie zostaną umieszczone w instrukcji ALTER, przestaną działać.

Gdy teraz spróbujemy wstawić wiersz, który nie spełnia kryteriów filtra:

```
INSERT INTO Sales.CustsWithOrders(
         companyname, contactname, contacttitle, address, city, region,
         postalcode, country, phone, fax)
  VALUES(N'Customer FGHIJ', N'FGHIJ', N'FGHIJ', N'FGHIJ', N'FGHIJ',
         N'FGHIJ', N'FGHIJ', N'FGHIJ', N'FGHIJ', N'FGHIJ');
```

Otrzymamy następującą informację 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.
```

Na zakończenie uruchamiamy następujące instrukcje porządkujące:

```
DELETE FROM Sales.Customers
WHERE custid > 91;

DBCC CHECKIDENT('Sales.Customers', RESEED, 91);
```

VIEW_METADATA

SQL Server może kontrolować żądania klienckie dotyczące przeszukania lub modyfikacji danych za pośrednictwem widoku tylko wtedy, gdy żądanie generuje kod T-SQL z widokiem jako obiektem docelowym. Natomiast systemy klienckie, które żądają metadanych w trybie przeszukania z wykorzystaniem interfejsów API DB-Library, ODBC lub OLEDB, mogą przysporzyć pewnych problemów. Metadane w trybie przeszukania

to dodatkowe informacje o bazowej tabeli oraz kolumnach w zbiorze wynikowym, które SQL Server zwraca do klienckiego interfejsu API. Jeśli system kliencki zdecyduje się na skonstruowanie instrukcji realizowanych na bazowej tabeli zamiast na widoku, żądania użytkowników mogą nie działać zgodnie z oczekiwaniami.

Załóżmy, że użytkownik ma uprawnienia do widoku, ale nie do bazowej tabeli i próbuje wykonać pewne operacje na widoku. Jeśli narzędzie klienckie skonstruuje instrukcję wykonywaną na bazowej tabeli, ponieważ zażąda metadanych w trybie przeszukiwania, instrukcja nie powiedzie się z powodu niewystarczających uprawnień. Z drugiej strony, gdy użytkownik próbuje dokonać za pośrednictwem widoku modyfikacji danych, która nie jest zgodna z określoną w definicji widoku opcją CHECK OPTION, tego typu modyfikacja mogłaby się powieść, jeśli zostałaby wykonana bezpośrednio na bazowej tabeli.

Aby SQL Server przesyłał metadane dotyczące widoku (a nie bazowej tabeli), gdy kliencki interfejs API żąda metadanych w trybie przeszukiwania, tworząc lub modyfikując widok, określamy opcję VIEW_METADATA w następujący sposób:

```
ALTER VIEW Sales.CustsWithOrders
  WITH ENCRYPTION, SCHEMABINDING, VIEW_METADATA
AS

SELECT custid, companyname, contactname, contacttitle,
  address, city, region, postalcode, country, phone, fax
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT 1 FROM Sales.Orders AS O
   WHERE O.custid = C.custid)
WITH CHECK OPTION;
GO
```

Na zakończenie uruchamiamy następujące instrukcje porządkujące:

```
IF OBJECT_ID('Sales.CustsWithOrders', 'V') IS NOT NULL
  DROP VIEW Sales.CustsWithOrders;
```

Widoki indeksowane

Należy pamiętać, że widok bez indeksu nie zawiera fizycznej reprezentacji danych, a jedynie metadane wskazujące bazowe obiekty. Aczkolwiek SQL Server zmaterializuje dane widoku, jeśli stworzony zostanie dla niego unikatowy indeks klastrowy. SQL Server nadzoruje synchronizację widoku indeksowanego z modyfikacjami tabeli bazowej. Nie można przeprowadzać synchronizacji zawartości widoku na żądanie lub zgodnie z harmonogramem. Widok indeksowany w znacznym stopniu przypomina indeks tabeli w kontekście integralności danych.

Widoki indeksowane mogą przynosić korzyści w zakresie wydajności zapytań, które zwracają dane. Mogą znacznie zredukować ilość operacji we/wy koniecznych do pobrania danych, a także czas realizowania pracochłonnych obliczeń. Znacznie zwiększona zostaje wydajność na przykład zapytań zawierających agregacje danych lub złożone złączenia. Jednak warto pamiętać, że modyfikacje tabeli bazowej, do której odwołuje

się widok indeksowany, wiążą się z koniecznością dokonania zmian w indeksowanym (i tym samym zmaterializowanym) widoku, co pociąga za sobą obniżenie wydajności modyfikacji.

Długa lista warunków i ograniczeń dotyczących tworzenia widoków indeksowanych w wielu sytuacjach wyklucza ich zastosowanie. W wersji SQL Server 2008 nie złagodzono tych wymagań.

Pierwszy tworzony indeks musi być unikalny i klastrowy. Po stworzeniu indeksu klastrowego na widoku można tworzyć dodatkowe nieklastrowe indeksy. Widok musi zostać stworzony z wykorzystaniem opcji SCHEMABINDING, w związku z czym trzeba stosować dwuczęściowe nazwy obiektów oraz zdefiniować wszystkie nazwy kolumn na liście SELECT. Jeśli zapytanie widoku agreguje dane, lista SELECT musi zawierać funkcję agregacji COUNT_BIG(*). Funkcja COUNT_BIG różni się od funkcji COUNT tym, że typ wynikowy to BIGINT. Ten licznik umożliwia serwerowi SQL Server śledzenie liczby wierszy, które zostały zagregowane w każdej z grup i służy do wyliczania innych agregacji. Niektóre opcje SET w sesji muszą znajdować się w określonym stanie. To nie koniec listy wymagań i ograniczeń. Szczegółowe informacje znaleźć można w dokumentacji SQL Server Books Online.

Przypuśćmy na przykład, że chcemy zoptymalizować zapytania, które pobierają zagregowane dane dla pracowników z tabel Orders oraz OrderDetails. Jeden ze sposobów osiągnięcia tego celu polega na stworzeniu zmaterializowanego widoku zawierającego agregacje, które są często wykonywane. Następujący fragment kodu służy do stworzenia indeksowanego widoku EmpOrders w oparciu o zapytanie, które dokonuje złączenia tabel Orders oraz OrderDetails, grupuje dane według *empid* i wyznacza sumę *qty* oraz liczbę wierszy dla każdego pracownika:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.EmpOrders', 'V') IS NOT NULL DROP VIEW dbo.EmpOrders;
IF OBJECT_ID('dbo.EmpOrders', 'U') IS NOT NULL DROP TABLE dbo.EmpOrders;
GO
CREATE VIEW dbo.EmpOrders WITH SCHEMABINDING
AS

SELECT O.empid, SUM(OD.qty) AS totalqty, COUNT_BIG(*) AS cnt
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY O.empid;
GO
CREATE UNIQUE CLUSTERED INDEX idx_uc_empid ON dbo.EmpOrders(empid);
```

Jak można zauważyć: powyższy widok został stworzony z wykorzystaniem opcji SCHEMABINDING, odwołania do tabel mają postać dwuczęściowych nazw, zastosowana została funkcja COUNT_BIG (ponieważ zapytanie wylicza agregacje), a stworzony indeks jest klastrowy oraz unikalny.

SQL Server nie generuje ponownie całego indeksu za każdym razem, gdy zmodyfikowana zostaje bazowa tabela. Stosuje inteligentniejszy sposób aktualizowania indeksu. Gdy użytkownik wstawia dane, SQL Server identyfikuje odpowiednie wiersze w widoku i inkrementuje wartość agregacji *totalqty* oraz *cnt* dla tego wiersza. Gdy użytkownik

usuwa dane, SQL Server zmniejsza te wartości. Gdy użytkownik modyfikuje dane w tabeli bazowej, SQL Server w odpowiedni sposób aktualizuje wartości agregacji.

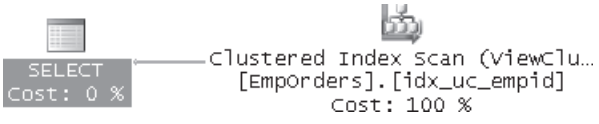
Aby zaobserwować, w jaki sposób indeksowane widoki zwiększają wydajność, uruchamiamy następujące zapytanie po włączeniu opcji STATISTICS IO oraz Include Actual Execution Plan w SSMS:

```
SET STATISTICS IO ON;
SELECT empid, totalqty, cnt FROM dbo.EmpOrders;
```

Oto wynik wykonania zapytania:

```
empid       totalqty    cnt
----------- ----------- --------------------
1           7812        345
2           6055        241
3           7852        321
4           9798        420
5           3036        117
6           3527        168
7           4654        176
8           5913        260
9           2670        107
```

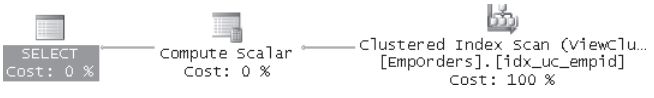
Rysunek 1-3 prezentuje plan wykonania zapytania.



Rysunek 1-3 Plan wykonania zapytania na widoku EmpOrders

Plan demonstruje, że indeks klastrowy został przeskanowany. Dla tak małego widoku, który zawiera jedynie 9 wierszy, operacje we/wy to zaledwie dwa odczyty logiczne. W edycji Enterprise SQL Server (lub edycji Developer zawierającej ten sam zestaw funkcji) optymalizator zapytań rozważy zastosowanie widoku indeksowego dla zapytań wykonywanych na widoku bez wskazówek oraz dla zapytań wykonywanych na tabeli bazowej. Na przykład następujące zapytanie powoduje wygenerowanie planu wykonania zaprezentowanego na Rysunku 1-4:

```
SELECT O.empid, SUM(OD.qty) AS totalqty, AVG(OD.qty) AS avgqty, COUNT_BIG(*) AS cnt
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY O.empid;
```



Rysunek 1-4 Plan wykonania zapytania na tabelach Orders oraz OrderDetails

Jak można zauważyć, wykorzystany został indeksowany widok i ponownie koszt operacji we/wy stanowiły zaledwie dwa odczyty logiczne. Co ciekawe, zapytanie zażądało

agregacji *AVG*(*OD.qty*), która nie stanowiła części widoku, mimo to widok indeksowany został zastosowany. Należy pamiętać, że wyliczona została suma oraz licznik. Jeśli rozwiniemy właściwości operatora Compute Scalar w planie, zobaczymy następujące wyrażenie, które służy do wyliczenia średniej na podstawie sumy i licznika:

```
[Expr1005] = Scalar Operator(CASE WHEN [InsideTSQL2008].[dbo].[EmpOrders]
   .[cnt]=(0) THEN NULL ELSE [InsideTSQL2008].[dbo].[EmpOrders].[totalqty]/
   CONVERT_IMPLICIT(int,[InsideTSQL2008].[dbo].[EmpOrders].[cnt],0) END),
   [Expr1006] = Scalar Operator([InsideTSQL2008].[dbo].[EmpOrders].[cnt])
```

Uwaga Przedstawione obserwacje dotyczą edycji Enterprise oraz Developer. W mniej zaawansowanych edycjach SQL Server widok indeksowany nie zostanie domyślnie wzięty pod uwagę, nawet w przypadku zapytań wykonywanych bezpośrednio na widoku. Aby skorzystać z indeksu, należy określić wskazówkę NOEXPAND.

SQL Server 2005 wprowadził szereg ulepszeń oraz optymalizacji w zakresie indeksowania widoków, natomiast SQL Server 2008 zaoferował szersze wsparcie dla tabel partycjonowanych. Ulepszenia dostępne w SQL Server 2005 obejmują m.in. obsługę interwałów podrzędnych (np. możliwość zastosowania indeksu, gdy zapytanie widoku zawiera filtr *col1 > 5*, a zapytanie zewnętrzne filtr *col1 > 10*) oraz wsparcie dla logicznie równoważnych wyrażeń (np. możliwość wykorzystania indeksu, gdy zapytanie widoku zawiera filtr *col1 = 5* natomiast zapytanie zewnętrzne *5 = col1*). SQL Server 2008 rozszerza wsparcie dla partycjonowanych tabel, pozwalając na przeprowadzanie na partycjonowanych tabelach dodatkowych typów operacji bez konieczności usuwania i ponownego tworzenia indeksu na widoku, co było nieuniknione w niektórych scenariuszach w systemie SQL Server 2005. SQL Server 2008 wspiera funkcję zwaną *indeksowanymi widokami dostosowanymi do partycji*, która powoduje, że procesor zapytań automatycznie aktualizuje widok indeksowany stworzony dla tabeli partycjonowanej po dodaniu nowej partycji. Dzięki temu, gdy chcemy dodać lub usunąć partycję z tabeli partycjonowanej, nie musimy już usuwać, a następnie ponownie tworzyć indeksu dla widoku.

Dodatkowe informacje Szczegółowe informacje o widokach indeksowanych w SQL Server 2008 oraz ulepszeniach wprowadzonych w tej wersji znaleźć można w artykule autorstwa Erica Hansona oraz Susan Price zatytułowanym „Improving Performance with SQL Server 2008 Indexed Views", który jest dostępny pod adresem *http://msdn.microsoft.com/en-us/library/dd171921.aspx*.

Poprawa wydajności to nie jedyne potencjalne zastosowanie widoków indeksowanych. W języku T-SQL ograniczenie UNIQUE traktuje dwie wartości NULL jako równe. Jeśli zdefiniujemy ograniczenie UNIQUE dla kolumny akceptującej wartości NULL, kolumna będzie mogła zawierać tylko jedno wystąpienie wartości NULL. Przypuśćmy, że chcemy wymusić unikalność jedynie dla określonych wartości (tzn. innych niż NULL), a jednocześnie zezwolić na wiele wartości NULL (w taki sposób powinno działać ograniczenie UNIQUE zdefiniowane przez ANSI SQL). Cel ten można osiągnąć

przy użyciu wyzwalacza, ale pociąga to za sobą pewne konsekwencje. A mianowicie modyfikacja będzie przeprowadzana dwukrotnie, a dokładniej najpierw przeprowadzana, a następnie wycofywana. Zamiast stosować wyzwalacz, możemy wymusić opisaną regułę integralności za pomocą widoku indeksowanego. W tym celu tworzymy widok indeksowany bazujący na zapytaniu, które pobiera z kolumny źródłowej jedynie wartości inne niż NULL. Jak pamiętamy, indeks klastrowy tworzony na widoku musi być unikatowy. Tego typu indeks zapobiega wstawianiu do tabeli bazowej powtarzających się, określonych wartości, ale umożliwia wstawianie wielu wartości NULL, ponieważ nie stanowią one części unikatowego indeksu.

Aby zademonstrować ten mechanizm w działaniu, uruchomimy następujący fragment kodu służący do stworzenia tabeli T1 z kolumną *keycol* oraz widoku indeksowanego bazującego na zapytaniu, które pobiera jedynie określone wartości *keycol* z tabeli T1:

```
USE tempdb;
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
GO
CREATE TABLE dbo.T1
(
  keycol  INT         NULL,
  datacol VARCHAR(10) NOT NULL
);
GO
CREATE VIEW dbo.V1 WITH SCHEMABINDING
AS

SELECT keycol FROM dbo.T1 WHERE keycol IS NOT NULL;
GO
CREATE UNIQUE CLUSTERED INDEX idx_uc_keycol ON dbo.V1(keycol);
```

Następnie wykonujemy poniższe instrukcje INSERT:

```
INSERT INTO dbo.T1(keycol, datacol) VALUES(1,    'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(1,    'b'); -- nie powiedzie się
INSERT INTO dbo.T1(keycol, datacol) VALUES(NULL, 'c');
INSERT INTO dbo.T1(keycol, datacol) VALUES(NULL, 'd');
```

Jak widzimy, druga próba wstawienia wiersza z wartością 1 w kolumnie *keycol* nie powiodła się, ale obie wartości NULL zostały zaakceptowane. Gdy wykonamy zapytanie na tabeli T1:

```
SELECT keycol, datacol FROM dbo.T1;
```

Otrzymamy następujące dane wynikowe:

```
keycol      datacol
----------- ----------
1           a
NULL        c
NULL        d
```

Jak łatwo zauważyć, obie wartości NULL zostały umieszczone w tabeli.

SQL Server 2008 umożliwia realizowanie tego zadania w jeszcze prostszy sposób przy użyciu nowej funkcji o nazwie *indeksy filtrowane*. Indeksy filtrowane to indeksy

tabeli zdefiniowane na podzbiorze wierszy tabeli bazowej, który jest identyfikowany przy użyciu predykatu określanego w klauzuli WHERE w definicji indeksu (podobnie jak w klauzuli WHERE zapytania).

Aby zrealizować to zadanie, wystarczy zdefiniować na kolumnie *keycol* unikatowy indeks filtrowany, który akceptuje tylko wiersze z wartościami innymi niż NULL w kolumnie *keycol*. W ten sposób unikatowość jest wymuszana tylko dla określonych wartości. Aby zademonstrować wykorzystanie indeksu filtrowanego do osiągnięcia tego celu, na początku usuwamy stworzony wcześniej widok, uruchamiając następujący fragment kodu:

```
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
```

Następnie tworzymy wspomniany indeks filtrowany z wykorzystaniem poniższej instrukcji:

```
CREATE UNIQUE INDEX idx_keycol_notnull ON dbo.T1(keycol)
  WHERE keycol IS NOT NULL;
```

Teraz uruchamiamy następujący fragment kodu w celu wstawienia kilku wierszy z wartościami NULL w kolumnie *keycol*, a następnie próbujemy wstawić dodatkowy wiersz z określonymi wartościami, które znajdują się już w tabeli:

```
INSERT INTO dbo.T1(keycol, datacol) VALUES(NULL, 'e');
INSERT INTO dbo.T1(keycol, datacol) VALUES(NULL, 'f');
INSERT INTO dbo.T1(keycol, datacol) VALUES(1, 'g');
```

Wiersze z wartościami NULL zostały umieszczone w tabeli, w odróżnieniu od powtarzających się, określonych wartości. Gdy wykonamy następujące zapytanie na tabeli:

```
SELECT keycol, datacol FROM dbo.T1;
```

Otrzymamy poniższe dane wynikowe:

```
keycol      datacol
----------- ----------
1           a
NULL        c
NULL        d
NULL        e
NULL        f
```

Na zakończenie uruchamiamy instrukcje porządkujące:

```
USE InsideTSQL2008;
IF OBJECT_ID('dbo.EmpOrders', 'V') IS NOT NULL DROP VIEW dbo.EmpOrders;

USE tempdb;
IF OBJECT_ID('dbo.V1', 'V') IS NOT NULL DROP VIEW dbo.V1;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
```

Podsumowanie

Widoki oferują ogromne możliwości. Pozwalają na dostarczanie niestandardowej, logicznej reprezentacji relacji, stanowią swego rodzaju zabezpieczenie, umożliwiają zastosowanie modularnego podejścia do budowania prostszych rozwiązań, a w przypadku użycia indeksów pomagają w podniesieniu wydajności zapytań. Należy jednak pamiętać, że widok reprezentuje tabelę i w związku z tym nie gwarantuje żadnej kolejności wierszy. Nie należy tworzyć „sortowanych" widoków, ponieważ sama idea zastosowania takiego rozwiązania jest błędna. Jeśli istnieje potrzeba prezentowania danych z widoku w określonej kolejności, należy zdefiniować klauzulę ORDER BY w zewnętrznym zapytaniu. Ponadto należy pamiętać o odświeżaniu metadanych widoku po wprowadzeniu zmian schematu obiektów bazowych w bazie danych.

ROZDZIAŁ 2

Funkcje definiowane przez użytkownika

Itzik Ben-Gan oraz Dejan Sarka

Funkcje definiowane przez użytkownika (User-defined Function – UDF) to konstrukcje służące do przeprowadzania obliczeń i zwracania wartości: skalarnej (pojedynczej) lub tabeli. Microsoft SQL Server 2008 umożliwia rozwijanie funkcji UDF z wykorzystaniem języka T-SQL lub wybranego języka .NET bazującego na zintegrowanej platformie CLR. Funkcje UDF mogą być umieszczane w zapytaniach, kolumnach obliczeniowych oraz ograniczeniach.

W tym rozdziale umówione zostaną różne typy funkcji UDF wspierane przez SQL Server: skalarne funkcje UDF, które zwracają pojedynczą wartość oraz tabelaryczne funkcje UDF (wbudowane i zawierające wiele instrukcji), które zwracają tabelę. Kod przykładowych funkcji UDF CLR będzie prezentowany w dwóch wersjach C# oraz Microsoft Visual Basic.

Większość procedur.NET demonstrowanych w tej książce została zaprojektowana przez Dejana Sarka.

Uwaga Niniejszy rozdział stanowi pierwszy z czterech rozdziałów (2, 3, 4 oraz 8) poświęconych obiektom CLR. Budowanie i instalowanie w bazie danych SQL Server różnego rodzaju obiektów CLR wiąże się z przeprowadzaniem tych samych, mechanicznych operacji. Aby uniknąć wielokrotnego demonstrowania rutynowych instrukcji i skoncentrować uwagę na kodzie procedur, wszystkie potrzebne informacje dotyczące instalacji obiektów CLR zostały umieszczone w Dodatku A.

Dodatek A zawiera instrukcje dotyczące tworzenia testowej bazy danych o nazwie CLRUtilities, która posłuży do testowania wszystkich rozwiązań CLR omówionych w tych czterech rozdziałach. Zaprezentowana zostanie również szczegółowa instrukcja objaśniająca, jak stworzyć, zbudować, zainstalować oraz przetestować wszystkie struktury CLR.

Dodatek zawiera kod wszystkich obiektów CLR, które są prezentowane w tych czterech rozdziałach. Zaleca się wykonanie instrukcji z Dodatku A przed zapoznaniem się z pozostałą częścią tego rozdziału, ponieważ to pozwoli stworzyć wszystkie struktury wykorzystywane w kolejnych przykładach. Następnie czytelnik może powrócić do lektury i skoncentrować się na kodzie obiektów CLR zamiast na kwestiach technicznych.

Wybrane fakty dotyczące funkcji UDF

Funkcje UDF mogą być umieszczane w zapytaniach, ograniczeniach oraz kolumnach obliczeniowych. Kod definiujący funkcję UDF nie może wpływać na stan bazy danych poza zakresem funkcji. W związku z tym kod funkcji UDF nie może modyfikować danych w tabelach ani wywoływać funkcji, które mają efekty uboczne (np. RAND). Co więcej, kod funkcji UDF może tworzyć jedynie zmienne tabelaryczne i nie może tworzyć tabel tymczasowych ani uzyskiwać do nich dostępu. Ponadto w kodzie funkcji UDF nie można stosować dynamicznych instrukcji SQL.

Tworząc lub modyfikując funkcję UDF, można określać jej opcje w nagłówku. Funkcje UDF T-SQL wspierają opcje ENCRYPTION oraz SCHEMABINDING, które zostały zaprezentowane w poprzednim rozdziale poświęconym widokom. Zarówno funkcje UDF T-SQL, jak i funkcje UDF CLR mogą być tworzone przy użyciu klauzuli EXECUTE AS, która umożliwia definiowanie kontekstu bezpieczeństwa dla wykonania funkcji. Ta opcja nie jest dostępna we wbudowanych tabelarycznych funkcjach UDF. Wbudowana tabelaryczna funkcja UDF w dużym stopniu przypomina widok z tą różnicą, że może zawierać argumenty. Składa się z pojedynczego zapytania, które definiuje tabelę wyjściową. W skalarnych funkcjach UDF (zarówno T-SQL, jak i CLR), można określać jedną z dwóch opcji: RETURNS NULL ON NULL INPUT lub CALLED ON NULL INPUT (ustawienie domyślne). Pierwsza opcja powoduje, że SQL Server w ogóle nie wywołuje funkcji, jeśli któryś z parametrów ma wartość NULL – w takiej sytuacji zbiór wynikowy jest równy NULL. Druga opcja informuje SQL Server, że funkcja ma być wywoływana nawet wtedy, gdy jeden z parametrów wejściowych jest równy NULL.

Zaleca się tworzenie funkcji UDF z zastosowaniem obu opcji: SCHEMABINDING oraz RETURNS NULL ON NULL INPUT (o ile jest to pożądane działanie). Opcja SCHEMABINDING zapobiega usuwaniu obiektów bazowych oraz modyfikacjom schematu wpływającym na wykorzystywane w funkcji kolumny. Opcja RETURNS NULL ON NULL INPUT może poprawić wydajność kodu, ponieważ powoduje pominięcie logiki funkcji i zwrócenie wartości NULL, gdy którykolwiek z parametrów wejściowych jest równy NULL.

Skalarne funkcje UDF

Skalarne funkcje UDF zwracają pojedynczą (skalarną) wartość. Mogą być stosowane tam, gdzie dozwolone są wyrażenia skalarne m.in. w zapytaniach, ograniczeniach, kolumnach obliczeniowych. Skalarne funkcje UDF wiążą się z kilkoma wymaganiami składniowymi:

- Muszą zawierać blok BEGIN/END definiujący ich treść.
- Należy wywoływać funkcję przy użyciu jej kwalifikowanej nazwy (o ile funkcja nie jest wywoływana jak procedury składowane – z wykorzystaniem instrukcji EXEC np. *EXEC myFunction 3, 4*).

- W wywołaniu nie można pomijać parametrów opcjonalnych (mających wartość domyślną), należy przynajmniej zastosować słowo kluczowe DEFAULT.

W kolejnych paragrafach omówione zostaną funkcje UDF T-SQL oraz funkcje UDF CLR.

Skalarne funkcje UDF T-SQL

Funkcje UDF T-SQL, które koncentrują się na przetwarzaniu zbiorów danych, działają zwykle szybciej niż funkcje UDF CLR, które służą do realizowania logiki proceduralnej. To spostrzeżenie dotyczy również pozostałych typów obiektów CLR, nie tylko funkcji UDF. W nagłówku funkcji określana jest jej nazwa, parametry wejściowe oraz typ zwracanych danych. Za przykład skalarnej funkcji UDF może posłużyć następująca funkcja *ConcatOrders*, która akceptuje identyfikator klienta jako parametr wejściowy i zwraca ciąg zawierający identyfikatory zamówień złożonych przez tego klienta:

```
SET NOCOUNT ON;
USE InsideTSQL2008;

IF OBJECT_ID('dbo.ConcatOrders', 'FN') IS NOT NULL
  DROP FUNCTION dbo.ConcatOrders;
GO

CREATE FUNCTION dbo.ConcatOrders
  (@custid AS INT) RETURNS VARCHAR(MAX)
AS
BEGIN
  DECLARE @orders AS VARCHAR(MAX);
  SET @orders = '';

  SELECT @orders = @orders + CAST(orderid AS VARCHAR(10)) + ';'
  FROM Sales.Orders
  WHERE custid = @custid;

  RETURN @orders;
END
GO
```

Funkcja deklaruje zmienną *@orders* i inicjalizuje ją przy użyciu pustego ciągu. W instrukcji SELECT zastosowana została specjalna składnia przypisania T-SQL, która służy do przeskanowania wybranych wierszy i przypisania wartości dla każdego z nich do zmiennej *@orders*. Wartość składa się z aktualnej zawartości zmiennej *@orders* połączonej z aktualną wartością *orderid* oraz średnikiem w roli separatora.

Aby przetestować funkcję *ConcatOrders*, uruchomimy poniższe zapytanie:

```
SELECT custid, dbo.ConcatOrders(custid) AS orders
FROM Sales.Customers;
```

Ważne Firma Microsoft nie udostępniła oficjalnej dokumentacji opisującej przedstawioną technikę agregacji-konkatenacji, która bazuje na składni przypisania w instrukcji SELECT. Zaprezentowane rozwiązanie bazuje jedynie na obserwacji. Aktualna implementacja funkcji *ConcatOrders* nie obejmuje klauzuli ORDER BY i nie gwarantuje kolejności konkatenacji. Zgodnie z wpisem na blogu Conora Cunningham z Microsoft (*http://blogs.msdn.com/sqltips/archive/2005/07/20/441053.aspx*) SQL Server powinien uwzględniać klauzulę ORDER BY. Artykuły Conora stanowią bardzo wiarygodne źródło informacji, ale trzeba podkreślić, że nie udało nam się znaleźć żadnej oficjalnej dokumentacji (poza wspomnianym wpisem na blogu), która opisywałaby działanie mechanizmu przypisywania wartości wielu wierszy w instrukcji SELECT – z klauzulą ORDER BY bądź bez niej.

Spowoduje to wygenerowanie następujących danych wynikowych, zaprezentowanych w skrótowej postaci (w celu zwiększenia czytelności zastosowano zawijanie wierszy):

```
custid      orders
----------- ------------------------------------------------------------
1           10835;10952;11011;10692;10702;10643;
2           10625;10759;10926;10308;
3           10677;10365;10682;10856;10535;10507;10573;
4           10453;10558;10743;10768;10793;10707;10741;10864;10920;10355;
            10383;10953;11016;
5           10524;10626;10689;10733;10280;10384;10444;10445;10572;10778;
            10924;10875;10654;10866;10278;10857;10672;10837;
6           10582;10509;10956;10614;10501;10853;11058;
7           10265;10436;10449;10360;10584;10628;10297;10559;10826;10679;
            10566;
8           10326;10801;10970;
9           10340;10525;10827;10663;10362;10715;10732;10470;10511;10755;
            11076;10730;10876;10932;10940;10331;10871;
11          11023;10471;10484;10947;10578;10943;10539;10599;10289;10538;
...
```

Konkatencję ciągów można osiągnąć, stosując inne bardzo efektywne rozwiązanie bazujące na opcji FOR XML PATH. To podejście jest dużo bardziej eleganckie niż metoda bazująca na przypisaniu w instrukcji SELECT, jest w pełni wspierane i zapewnia pełną kontrolę nad kolejnością wierszy. Szybka i klarowna technika konkatenacji została opracowana przez Michaela Rysa – menedżera programu w zespole ds. rozwoju Microsoft SQL Server oraz Eugena Kogana – lidera technicznego w zespole Microsoft SQL Server Engine. Wyrazy wdzięczności należą się również ekspertowi SQL Server MVP Toniemu Rogersonowi, który dodał logikę umożliwiającą obsługę znaków specjalnych. Tryb PATH stanowi prostszy sposób łączenia elementów i atrybutów niż dyrektywa EXPLICIT. Oto pełne rozwiązanie:

```
SELECT custid,
  COALESCE(
    (SELECT CAST(orderid AS VARCHAR(10)) + ';' AS [text()]
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderid
```

```
    FOR XML PATH(''), TYPE).value('.[1]', 'VARCHAR(MAX)'), '') AS orders
FROM Sales.Customers AS C;
```

Zewnętrzne zapytanie zwraca identyfikatory wszystkich klientów z tabeli Customers. Powiązane zapytanie podrzędne zwraca wszystkie identyfikatory zamówień z tabeli Orders dla aktualnego klienta w zewnętrznym wierszu. Opcja FOR XML służy do przesyłania pojedynczego wystąpienia XML złożonego z wartości zwracanych przez zapytanie. Aby osiągnąć ten cel, opcja FOR XML musi połączyć wszystkie wartości. W wyniku przypisania pustego ciągu do parametru wejściowego trybu PATH oraz zdefiniowania aliasu [*text*()] dla wyrażenia na liście SELECT uzyskujemy prostą konkatenację wartości zwracanych przez zapytanie (bez znaczników). Funkcja COALESCE służy do przekonwertowania wartości NULL do postaci pustego ciągu, w przypadku gdy klient nie złożył żadnych zamówień.

Warto mieć świadomość, że zastosowanie wyrażenia AS [text()] wiąże się z koniecznością spełnienia charakterystycznych dla XML wymagań dotyczących znaków specjalnych (m.in. *&*, < oraz >), czyli przekonwertowania ich do postaci symboli zastępczych (odpowiednio *&*, *<* oraz *>*). Prezentowane w tym przykładzie zapytanie łączy identyfikatory zamówień, które mają wartość liczbową, a zatem dane nie będą zawierały żadnych znaków specjalnych. Mimo to, aby rozszerzyć zbiór potencjalnych zastosowań zapytania, dodaliśmy do niego logikę służącą do obsługi znaków specjalnych. W skład tej logiki wchodzą: dyrektywa TYPE, która nakazuje serwerowi SQL Server zwrócić wystąpienie typu XML oraz metoda *.value*, która pobiera wartość z wystąpienia XML, konwertując symbole z powrotem do oryginalnej postaci znaków specjalnych.

Definiowane przez użytkownika agregacje (UDA) CLR również mogą rozwiązać ten problem. Szczegółowe omówienie wspomnianych technik konkatenacji ciągów, jak również innych niestandardowych agregacji zostało zaprezentowane w książce *Inside Zapytania T-SQL*.

Na zakończenie uruchamiamy następujący kod służący do przywrócenia pierwotnego stanu:

```
IF OBJECT_ID('dbo.ConcatOrders', 'FN') IS NOT NULL
  DROP FUNCTION dbo.ConcatOrders;
```

Względy wydajnościowe

Trzeba mieć świadomość, że wywoływanie skalarnych funkcji UDF w zapytaniach stanowi niebagatelne obciążenie, w szczególności gdy rolę parametrów wejściowych funkcji pełnią atrybuty zewnętrznej tabeli. Nawet gdy funkcja ma tylko klauzulę RETURN ze skalarnym wyrażeniem, nadal nie jest traktowana jak wbudowana. Proces wywoływania funkcji dla wszystkich wierszy znacząco obniża efektywność. Wystarczy uruchomić prosty test wydajności, aby przekonać się, jak duży jest koszt stosowania w zapytaniu funkcji UDF w porównaniu ze stosowaniem wyrażeń wbudowanych.

Przed przeprowadzeniem testu wydajności uruchomimy kod z Listingu 2-1 w celu stworzenia pomocniczej tabeli liczb o nazwie Nums i wypełnienia jej milionem liczb. Tabela ta będzie wielokrotnie wykorzystywana w tej książce, dlatego warto ją zachować.

Listing 2-1 Tworzenie i wypełnianie pomocniczej tabeli liczb

```
SET NOCOUNT ON;
USE InsideTSQL2008;

IF OBJECT_ID('dbo.Nums', 'U') IS NOT NULL DROP TABLE dbo.Nums;

CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);
DECLARE @max AS INT, @rc AS INT;
SET @max = 1000000;
SET @rc = 1;

INSERT INTO Nums VALUES(1);
WHILE @rc * 2 <= @max
BEGIN
  INSERT INTO dbo.Nums SELECT n + @rc FROM dbo.Nums;
  SET @rc = @rc * 2;
END

INSERT INTO dbo.Nums
  SELECT n + @rc FROM dbo.Nums WHERE n + @rc <= @max;
```

Włączmy opcję Discard Results After Execution w SQL Server Management Studio (SSMS), aby pomiar czasu nie obejmował czasu potrzebnego do wygenerowania danych wynikowych.

Uwaga Istnieją dwie metody włączania lub wyłączania opcji Discard Results After Execution w SSMS. Pierwsza z nich jest realizowana za pomocą okna dialogowego Query Options aktualnej sesji (dostępnego po wybraniu opcji Query Options z menu Query lub menu kontekstowego sesji). Ta opcja wpływa na aktualne okno edytora. Druga metoda bazuje na wykorzystaniu okna dialogowego Options (dostępnego po wybraniu opcji Options z menu Tools, a następnie zaznaczeniu w panelu nawigacyjnym po lewej stronie opcji Query Results/SQL Server i Results to Grid lub Results to Text w zależności od aktualnego ustawienia). Ta opcja wpływa jedynie na nowo otwarte okna edytora (nie na aktualnie otwarte).

Rozpoczynamy od uruchomienia zapytania na milionie wierszy z tabeli Nums z wykorzystaniem wyrażenia wbudowanego, które dodaje 1 do wartości *n*:

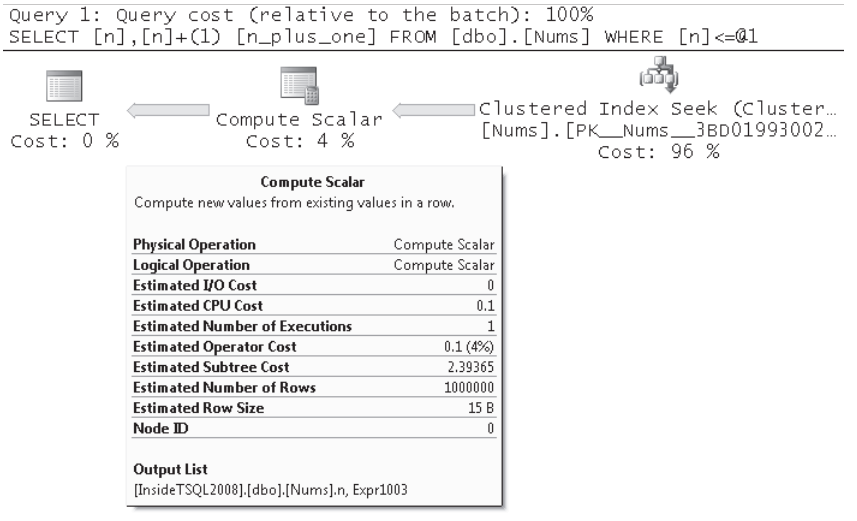
```
SELECT n, n + 1 AS n_plus_one FROM dbo.Nums WHERE n <= 1000000;
```

Rysunek 2-1 prezentuje plan wykonania zapytania.

Jak można zauważyć, analizując plan, operator Compute Scalar wylicza wartość skalarną o nazwie *Expr1003*. Okno dialogowe Properties dla tego operatora prezentuje wyrażenie służące do przypisywania tej wartości do zmiennej: [*Expr1003*] = *Scalar Operator*([*InsideTSQL2008*].[*dbo*].[*Nums*].[*n*]+(*1*)).Jest to wyrażenie wbudowane, które zostało przetworzone w ramach zapytania i nie pociąga za sobą dodatkowych obciążeń.

Pierwsze wywołanie kodu mogło obejmować fizyczne skanowanie danych. Dlatego po załadowaniu danych do pamięci podręcznej uruchamiamy zapytanie po raz drugi

i mierzymy czas wykonania. Kod zakończył działanie w przykładowym środowisku testowym po niecałej sekundzie.



Rysunek 2-1 Plan wykonania zapytania wykorzystującego wyrażenie wbudowane

Następnie tworzymy skalarną funkcję UDF o nazwie *AddOne*:

```
IF OBJECT_ID('dbo.AddOne', 'FN') IS NOT NULL
  DROP FUNCTION dbo.AddOne;
GO
CREATE FUNCTION dbo.AddOne(@i AS INT) RETURNS INT
AS
BEGIN
  RETURN @i + 1;
END
GO
```

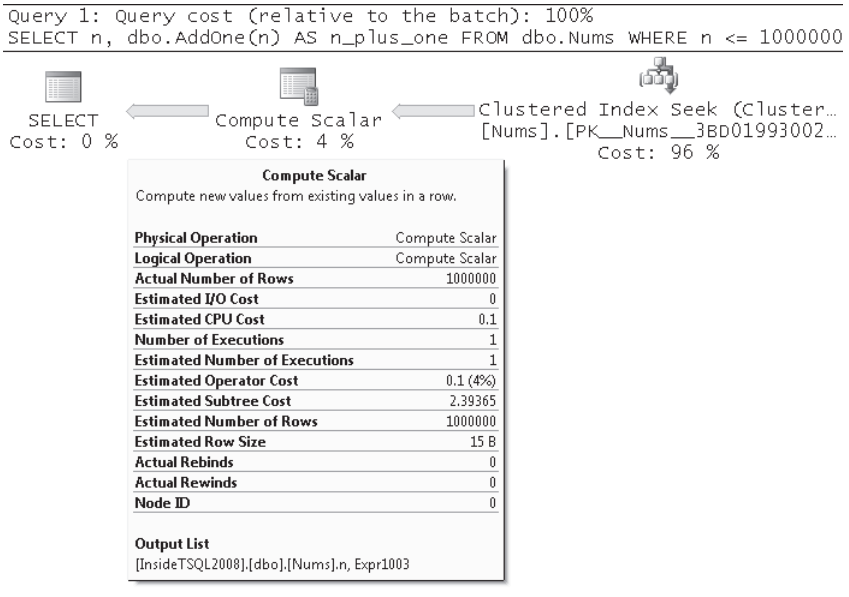
Teraz uruchamiamy zapytanie, korzystając z funkcji *AddOne*:

```
SELECT n, dbo.AddOne(n) AS n_plus_one FROM dbo.Nums WHERE n <= 1000000;
```

Rysunek 2-2 prezentuje plan wykonania zapytania.

Warto zauważyć, że w tym planie operator Compute Scalar również wylicza zmienną skalarną o nazwie *Expr1003*. Analizując okno dialogowe Properties dla tego operatora, można przekonać się, że wyrażenie przypisywane do zmiennej różni się od wyrażenia zastosowanego w poprzednim planie wykonania: [*Expr1003*] = *Scalar Operator*([*InsideTSQL2008*].[*dbo*].[*AddOne*]([*InsideTSQL2008*].[*dbo*].[*Nums*].[*n*])). Rozbieżność polega na tym, że zamiast wyrażenia *n + 1* wywołana została funkcja UDF o nazwie *AddOne*. Z pozoru może się wydawać, że nie stanowi to żadnej różnicy. W końcu wywoływana funkcja UDF zawiera to samo wyrażenie co zapytanie z poprzedniego przykładu. Jednak fakt, iż wywołanie funkcji UDF pojawia się w planie, świadczy o tym, że wyrażenie z funkcji nie stanowi wewnętrznego komponentu zapytania. Innymi słowy, SQL Server dokonuje osobnego wywołania funkcji UDF dla każdego wiersza, a każde

takie wywołanie pociąga za sobą dodatkowe obciążenie. Różnice wydajności ilustruje fakt, iż zmodyfikowane zapytanie było wykonywane w przykładowym systemie o 4 sekundy dłużej niż zapytanie z wyrażeniem wbudowanym.

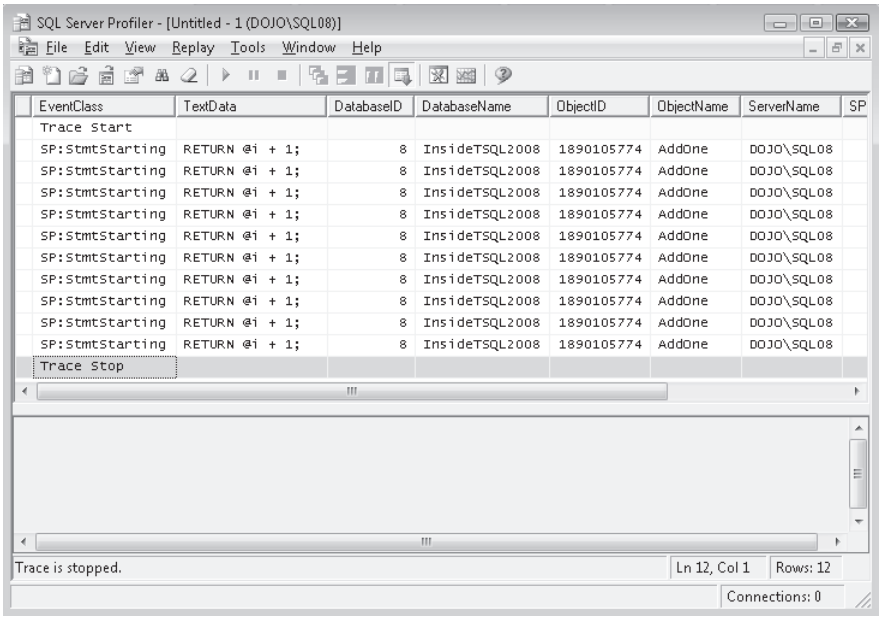


Rysunek 2-2 Plan wykonania zapytania wykorzystującego skalarną funkcję UDF

Długi czas wykonywania ostatniego zapytania wynika z obciążenia związanego z każdym wywołaniem funkcji. Można z łatwością zauważyć wysoką liczbę wywołań funkcji UDF, uruchamiając śledzenie SQL Server Profiler z wykorzystaniem zdarzenia *SP:Completed* (lub *SP:Starting*) w czasie wykonania zapytania. Aby ograniczyć rozmiar śladu, można przetestować zapytanie wykonywane na wybranych wierszach np. stosując filtr *n <= 10*. Rysunek 2-3 prezentuje zdarzenia uzyskane w wyniku śledzenia tego zapytania.

Umieszczenie kodu w funkcji UDF przynosi istotne korzyści programistyczne, m.in. uproszczenie kodu oraz możliwość wielokrotnego wykorzystywania logiki biznesowej. Jednak pociąga za sobą również negatywne konsekwencje w postaci obniżonej wydajności. Czy to oznacza konieczność wyboru między korzyściami programistycznymi a wydajnością? Niekoniecznie, ponieważ istnieje rozwiązanie, które pozwala uniknąć tego typu kompromisów i umożliwia tworzenie funkcji UDF bez obniżania wydajności zapytań.

Rozwiązanie to można stosować tylko wtedy, gdy funkcja bazuje na pojedynczym wyrażeniu (czyli nie zawiera wielu sekwencyjnie wykonywanych instrukcji). Zamiast definiowania skalarnej funkcji UDF, należy zdefiniować wbudowaną tabelaryczną funkcję UDF, która zwraca zapytanie bez klauzuli FROM, z pojedynczą kolumną bazująca na wybranym wyrażeniu. Szczegółowe omówienie wbudowanej tabelarycznej funkcji UDF zostanie zaprezentowane w dalszej części tego rozdziału w części „Tabelaryczne funkcje UDF".



RYSUNEK 2-3 Ślad w narzędziu Profiler dla wielu wywołań skalarnej funkcji UDF

Na razie wystarczy mieć świadomość, że wbudowana tabelaryczna funkcja UDF w dużym stopniu przypomina widok, który akceptuje parametry wewnętrzne. Oto wersja funkcji *AddOne* wykorzystująca wbudowaną tabelaryczną funkcję UDF:

```
IF OBJECT_ID('dbo.AddOneInline', 'IF') IS NOT NULL
  DROP FUNCTION dbo.AddOneInline;
GO
CREATE FUNCTION dbo.AddOneInline(@n AS INT) RETURNS TABLE
AS
RETURN SELECT @n + 1 AS val;
GO
```

Ponieważ funkcja UDF zwraca wartość w postaci tabeli, nie można po prostu wywołać jej w wyrażeniu, trzeba wykonać na niej zapytanie. W związku z tym, aby skonstruować wyrażenie skalarne bazujące na wywołaniu funkcji, trzeba zastosować skalarne zapytanie podrzędne:

```
SELECT n, (SELECT val FROM dbo.AddOneInline(n) AS F) AS n_plus_one
FROM dbo.Nums WHERE n <= 1000000;
```

Powyższa konstrukcja może wydawać się nieco skomplikowana, jednak oferuje ona znaczące korzyści. Wbudowane tabelaryczne funkcje UDF, jak sama nazwa wskazuje, są wbudowane. To oznacza, że SQL Server rozwija definicję funkcji UDF i włącza ją do zapytania w taki sposób, jakby odwołanie do funkcji nie istniało. Analizując plan wykonania zapytania, można zauważyć, że jest on taki sam, jak zaprezentowany na rysunku 2-1 plan wykonania zapytania, które nie zawierało żadnego wywołania funkcji, a jedynie wbudowane wyrażenie *n+1*. Oczywiście oznacza to, że zapytanie

może zostać wykonane w ciągu niecałej sekundy, podobnie jak zapytanie z wyrażeniem wbudowanym.

Możliwość odwołania się do tabelarycznej funkcji UDF w zapytaniu podrzędnym i przekazania atrybutów zewnętrznej tabeli w roli danych wejściowych przypomina niejawne zastosowanie operatora APPLY. Można również zastosować operator APPLY w sposób jawny:

```
SELECT Nums.n, A.val AS n_plus_one
FROM dbo.Nums
  CROSS APPLY dbo.AddOneInline(n) AS A
WHERE n <= 1000000;
```

Plan wykonania powyższego zapytania będzie taki sam jak plan z Rysunku 2-1. Teraz możemy już wyłączyć opcję Discard Results After Execution w narzędziu SSMS.

Można odnieść wrażenie, że niektóre rozwiązania realizowane zwykle za pomocą funkcji UDF nie mogą zostać przekonwertowane do postaci wyrażeń wbudowanych, ponieważ implementują logikę iteracyjną lub proceduralną przy użyciu wielu instrukcji. Jednak wystarczy wykazać się odrobiną kreatywności, aby znaleźć metodę bazującą na pojedynczym wyrażeniu, które może zostać wbudowane w zapytanie. Spróbujmy na przykład znaleźć sposób zaimplementowania licznika wystąpień jednego ciągu tekstowego w drugim. Przedstawiony fragment kodu ilustruje, w jaki sposób można osiągnąć ten cel przy użyciu wyrażenia wbudowanego. Zapytanie służy do określenia liczby wystąpień ciągu *@find* w kolumnie *companyname* dla każdego wiersza z tabeli Customers:

```
DECLARE @find AS NVARCHAR(40);
SET @find = N'n';

SELECT companyname,
  (LEN(companyname+'*') – LEN(REPLACE(companyname, @find, ")+'*'))
    / LEN(@find) AS cnt
FROM Sales.Customers;
```

Wykonanie zapytania spowoduje wyświetlenie następujących danych wynikowych, które zostały zaprezentowane poniżej w skróconej postaci:

```
companyname     cnt
--------------- -----------
Customer AHPOP  0
Customer AHXHT  0
Customer AZJED  0
Customer BSVAR  0
Customer CCFIZ  0
Customer CCKOT  0
Customer CQRAA  0
Customer CYZTN  1
Customer DTDMN  1
Customer DVFMB  0
Customer EEALV  0
Customer EFFTC  0
Customer ENQZT  1
Customer EYHKM  0
Customer FAPSM  0
```

```
Customer FEVNN  2
Customer FRXZL  0
Customer FVXPQ  0
...
```

Wyrażenie implementuje licznik z wykorzystaniem funkcji REPLACE. Bazuje ono na założeniu, że można wyznaczyć liczbę wystąpień ciągu *@find* w ciągu tekstowym, sprawdzając, o ile krótszy jest ciąg tekstowy po usunięciu wszystkich wystąpień (tzn. zastąpieniu ich symbolem „). Warto zauważyć, że przed pomiarem długości do obu ciągów dodane zostało wyrażenie '*', dzięki któremu długość jest określana prawidłowo nawet wtedy, gdy ciąg zakończony jest spacjami.

Funkcje UDF wykorzystywane w ograniczeniach

Skalarne funkcje UDF można także stosować w ograniczeniach. Kolejne podrozdziały poświęcone zostaną omówieniu metod wykorzystywania funkcji UDF w ograniczeniach DEFAULT, CHECK, PRIMARY KEY oraz UNIQUE.

Ograniczenia DEFAULT

Skalarne funkcje UDF można wykorzystywać w wyrażeniach DEFAULT. Jednak należy zdawać sobie sprawę z faktu, iż funkcja UDF zastosowana w ograniczeniu DEFAULT nie może akceptować danych wejściowych w postaci kolumn tabeli. Następujący przykładowy fragment kodu służy do stworzenia tabeli T1 oraz funkcji UDF *T1_getkey* zwracającej minimalną wartość klucza, która nie znajduje się w tabeli T1:

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
  keycol INT NOT NULL CONSTRAINT PK_T1 PRIMARY KEY CHECK (keycol > 0),
  datacol VARCHAR(10) NOT NULL
);

IF OBJECT_ID('dbo.T1_getkey', 'FN') IS NOT NULL
  DROP FUNCTION dbo.T1_getkey;
GO
CREATE FUNCTION dbo.T1_getkey() RETURNS INT
AS
BEGIN
  RETURN
    CASE
      WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
      ELSE (SELECT MIN(keycol + 1)
            FROM dbo.T1 AS A
            WHERE NOT EXISTS
              (SELECT *
               FROM dbo.T1 AS B
               WHERE B.keycol = A.keycol + 1))
    END;
END
GO
```

Poniższa instrukcja służy do dodania do kolumny *keycol* ograniczenia DEFAULT, które wywołuje funkcję *T1_getkey*:

```
ALTER TABLE dbo.T1 ADD DEFAULT(dbo.T1_getkey()) FOR keycol;
```

Uwaga Przedstawione ograniczenie DEFAULT ma sens jedynie w przypadku wstawiania pojedynczych wierszy, nie wielu wierszy naraz. Ponadto w rzeczywistych scenariuszach biznesowych z reguły odradza się ponowne wykorzystywanie tych samych wartości klucza. Przykład służy jedynie do celów demonstracyjnych.

Poniższy fragment kodu służy do wstawienia trzech wierszy, wygenerowania kluczy (1, 2 i 3), usunięcia wiersza z kluczem 2, a następnie wstawienia kolejnego wiersza z wykorzystaniem wygenerowanego klucza 2:

```
INSERT INTO dbo.T1(datacol) VALUES('a');
INSERT INTO dbo.T1(datacol) VALUES('b');
INSERT INTO dbo.T1(datacol) VALUES('c');
DELETE FROM dbo.T1 WHERE keycol = 2;
INSERT INTO dbo.T1(datacol) VALUES('d');
```

Wykonanie poniższego zapytania na tabeli:

```
SELECT * FROM dbo.T1;
```

Prowadzi do wygenerowania następującego wyniku:

```
keycol      datacol
----------- ----------
1           a
2           d
3           c
```

Jak można zauważyć, ostatni wstawiany wiersz zawiera klucz 2 (*datacol = 'd'*), ponieważ wiersz z kluczem 2 został uprzednio usunięty.

Ograniczenia CHECK

W odróżnieniu od funkcji UDF stosowanych w ograniczeniach DEFAULT, funkcje UDF stosowane w ograniczeniach CHECK mogą odwoływać się do kolumn tabeli jako danych wejściowych. Ograniczenia CHECK wykorzystujące funkcje UDF zapewniają szersze możliwości wymuszania więzów integralności, dzięki czemu w niektórych przypadkach nie trzeba stosować wyzwalaczy, które są z reguły bardziej obciążające. W dalszej części tego rozdziału zademonstrujemy przykłady zastosowania funkcji UDF w ograniczeniach CHECK, w których ciągi wejściowe będą analizowane z wykorzystaniem wyrażeń regularnych.

Ograniczenia PRIMARY KEY oraz UNIQUE

Istnieje możliwość stworzenia ograniczenia UNIQUE lub PRIMARY KEY dla kolumny obliczeniowej, która wywołuje funkcję UDF. Należy jednak pamiętać, że oba ograniczenia powodują w praktyce utworzenie unikalnego indeksu. To oznacza, że docelowa

kolumna obliczeniowa oraz wywoływana funkcja UDF muszą spełniać reguły związane z indeksowaniem. W związku z tym funkcja UDF musi być powiązana ze schematem (przy użyciu opcji SCHEMABINDING), a kolumna obliczeniowa musi być deterministyczna i precyzyjna lub deterministyczna i trwała itd. Szczegółowe informacje o ograniczeniach indeksowania dotyczących kolumn obliczeniowych oraz funkcji UDF znaleźć można w dokumentacji SQL Server Books Online.

Poniższy fragment kodu służy do dodania do tabeli T1 kolumny obliczeniowej *col1*, która zawiera wywołanie funkcji UDF *AddOne*, a także stworzenia ograniczenia UNIQUE dla tej kolumny:

```
ALTER TABLE dbo.T1
  ADD col1 AS dbo.AddOne(keycol) CONSTRAINT UQ_T1_col1 UNIQUE;
```

Próba wykonania powyższej instrukcji zakończy się wyświetleniem następującego komunikatu o błędzie:

```
Msg 2729, Level 16, State 1, Line 1
Column 'col1' in table 'dbo.T1' cannot be used in an index or statistics or as
a partition key because it is non-deterministic.
Msg 1750, Level 16, State 0, Line 1
Could not create constraint. See previous errors.
```

Wykonanie nie powiedzie się, ponieważ funkcja nie spełnia jednego z wymagań dotyczących indeksowania, zgodnie z którym funkcja musi być powiązana ze schematem. Jak można zauważyć, komunikat o błędzie nie jest zbyt pomocny, gdyż nie wskazuje rzeczywistej przyczyny problemu ani potencjalnego rozwiązania. Trzeba samodzielnie domyślić się, że aby naprawić problem, należy zmodyfikować funkcję, dodając do niej opcję SCHEMABINDING:

```
ALTER FUNCTION dbo.AddOne(@i AS INT) RETURNS INT
  WITH SCHEMABINDING
AS
BEGIN
  RETURN @i + 1;
END
GO
```

Gdy ponownie spróbujemy dodać kolumnę obliczeniową z ograniczeniem UNIQUE, kod zostanie pomyślnie wykonany:

```
ALTER TABLE dbo.T1
  ADD col1 AS dbo.AddOne(keycol) CONSTRAINT UQ_T1_col1 UNIQUE;
```

Trochę trudniej jest stworzyć ograniczenie PRIMARY KEY dla kolumny obliczeniowej. Aby przetestować ten mechanizm, usuwamy istniejące ograniczenie PRIMARY KEY z kolumny T1:

```
ALTER TABLE dbo.T1 DROP CONSTRAINT PK_T1;
```

Następnie próbujemy dodać kolejną kolumnę obliczeniową o nazwie *col2* z wykorzystaniem ograniczenia PRIMARY KEY:

```
ALTER TABLE dbo.T1
  ADD col2 AS dbo.AddOne(keycol)
    CONSTRAINT PK_T1 PRIMARY KEY;
```

Próba wykonania nie powiedzie się i wyświetlony zostanie następujący komunikat o błędzie:

```
Msg 1711, Level 16, State 1, Line 1
Cannot define PRIMARY KEY constraint on column 'col2' in table 'T1'. The computed
column has to be persisted and not nullable.
Msg 1750, Level 16, State 0, Line 1
Could not create constraint. See previous errors.
```

Musimy zagwarantować, że kolumna *col2* nie będzie zawierać wartości NULL. W tym celu definiujemy kolumnę jako PERSISTED oraz NOT NULL:

```
ALTER TABLE dbo.T1
  ADD col2 AS dbo.AddOne(keycol) PERSISTED NOT NULL
    CONSTRAINT PK_T1 PRIMARY KEY;
```

Na zakończenie uruchamiamy następujący kod służący do przywrócenia pierwotnego stanu:

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL
  DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.T1_getkey', 'FN') IS NOT NULL
  DROP FUNCTION dbo.T1_getkey;
IF OBJECT_ID('dbo.AddOne', 'FN') IS NOT NULL
  DROP FUNCTION dbo.AddOne;
IF OBJECT_ID('dbo.AddOneInline', 'IF') IS NOT NULL
  DROP FUNCTION dbo.AddOneInline;
```

Skalarne funkcje UDF CLR

W tej części rozdziału omówione zostaną skalarne funkcje UDF CLR i gdy to możliwe, zostaną one porównane do funkcji UDF T-SQL. Jak już wspomniano, Dodatek A zawiera instrukcje dotyczące rozwijania, budowania, instalowania i testowania procedur CLR. W tym rozdziale oraz pozostałych fragmentach książki poświęconych konstrukcjom CLR skoncentrujemy się na kodzie samych obiektów. W Dodatku A znaleźć można definicje przestrzeni nazw oraz klasy *CLRUtilities*, w której znajdują się procedury. Oto wersja C# definicji przestrzeni nazw oraz nagłówka klasy *CLRUtilities*:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
using System.Text.RegularExpressions;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Reflection;
```

```
public partial class CLRUtilities
{
  ... definicje procedury ...
}
```

A oto wersja Visual Basic:

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Text
Imports System.Text.RegularExpressions
Imports System.Collections
Imports System.Collections.Generic
Imports System.Diagnostics
Imports System.Reflection
Imports System.Runtime.InteropServices

Partial Public Class CLRUtilities
  ... definicje procedury ...
End Class
```

Nie będziemy powtarzać definicji przestrzeni nazw oraz klasy. Co więcej, nie będziemy prezentować tak szczegółowych instrukcji konfiguracji, jak te zaprezentowane w Dodatku A.

Obiekty CLR

Możliwość tworzenia obiektów CLR w SQL Server niesie za sobą ogromny potencjał, a jednocześnie duże ryzyko. Proceduralne języki .NET oferują bogatszy zestaw możliwości programistycznych, a także wyższą wydajność w przypadku realizacji zadań, które są niezgodne z przeznaczeniem języka T-SQL. Do zadań tych należą między innymi: złożone obliczenia, logika iteracyjna i proceduralna, przetwarzanie ciągów czy zewnętrzny dostęp do zasobów systemu operacyjnego. T-SQL stanowi język deklaracyjny i z reguły zapewnia szersze możliwości oraz wyższą wydajność w przypadku przetwarzania danych z wykorzystaniem zapytań bazujących na zbiorach. Istnieje ryzyko, że integracja .NET zachęci programistów, którzy jeszcze nie opanowali specyfiki języka SQL, do implementowania mało wydajnego kodu z wykorzystaniem nieodpowiednich technik. W tej książce zaprezentujemy przykładowe scenariusze, w których zalecane jest stosowanie procedur tworzonych z wykorzystaniem platformy .NET.

Dodatkowe informacje Dodatkowe informacje o zapytaniach bazujących na zbiorach oraz efektywnych rozwiązaniach bazujących na zbiorach znaleźć można w książce *Inside Zapytania T-SQL*, która zwiera szczegółowe omówienie tych zagadnień.

Przetwarzanie ciągów

Jeden z obszarów, w których języki proceduralne (.NET) zapewniają większe możliwości i wyższą wydajność niż język T-SQL, stanowi przetwarzanie ciągów. Następujący scenariusz ilustruje, w jaki sposób można realizować zadania związane z przetwarzaniem ciągów, takie jak dopasowywanie ciągów, zastępowanie ciągów czy formatowanie wartości daty i godziny w SQL Server.

Dopasowywanie w oparciu o wyrażenia regularne Wyrażenia regularne oferują szerokie możliwości dopasowywania wzorców tekstu z wykorzystaniem zwięzłej i elastycznej notacji. *Wyrażenie regularne* to standardowy i znaczący termin, który jest powszechnie stosowany już od dłuższego czasu. Język ANSI SQL definiuje predykat SIMILAR TO, który zapewnia wsparcie dla wyrażeń regularnych, jednak język T-SQL w SQL Server 2008 nie zawiera jeszcze implementacji tego predykatu. Na szczęście istnieje możliwość stosowania wyrażeń regularnych w kodzie .NET. Poniższy przykładowy fragment kodu C# definiuje funkcję o nazwie *RegexIsMatch*:

```
// Funkcja RegexIsMatch
// Waliduje ciąg wejściowy przy użyciu wyrażenia regularnego
[SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
public static SqlBoolean RegexIsMatch(SqlString input,
  SqlString pattern)
{
    if (input.IsNull || pattern.IsNull)
        return SqlBoolean.Null;
    else
        return (SqlBoolean)Regex.IsMatch(input.Value, pattern.Value,
          RegexOptions.CultureInvariant);
}
```

Atrybuty w nagłówku informują SQL Server, że funkcja jest deterministyczna i nie wymaga dostępu do danych. Warto zwrócić uwagę na zastosowanie opcji *RegexOptions.CultureInvariant* do uzyskania dopasowania niezależnego od kultury. W przeciwnym przypadku funkcja nie byłaby deterministyczna (szczegółowe informacje znaleźć można w artykule *http://msdn.microsoft.com/en-us/library/z0sbec17.aspx*).

Funkcja akceptuje dwa parametry: ciąg tekstowy (*input*) oraz wyrażenie regularne (*pattern*), a następnie zwraca wartość typu *SqlBoolean*, czyli jedną z trzech możliwych wartości: 0, 1 lub Null. Wartość Null zostaje zwrócona, gdy parametr *input* lub *pattern* ma wartość Null, wartość 1, gdy wzorzec *pattern* został znaleziony w ciągu *input*, a wartość 0 w przeciwnym przypadku. Jak widać, kod funkcji jest bardzo prosty. Na początku sprawdzana jest wartość parametrów wejściowych i jeśli którykolwiek z nich jest równy Null, zwracana jest wartość Null. W przeciwnym przypadku funkcja zwraca wynik wykonania metody *Regex.IsMatch*. Ta metoda sprawdza, czy określony w pierwszym parametrze ciąg zawiera wzorzec określony w drugim parametrze. Metoda *Regex.IsMatch* zwraca wartość .NET *System.Boolean*, która musi zostać bezpośrednio przekonwertowana do typu *SqlBoolean*.

Oto kod funkcji w języku Visual Basic dla czytelników, którzy preferują ten język:

```
' Funkcja RegexIsMatch
' Waliduje ciąg wejściowy przy użyciu wyrażenia regularnego
<SqlFunction(IsDeterministic:=True, DataAccess:=DataAccessKind.None)> _
Public Shared Function RegexIsMatch(ByVal input As SqlString, _
  ByVal pattern As SqlString) As SqlBoolean
    If (input.IsNull Or pattern.IsNull) Then
        Return SqlBoolean.Null
    Else
        Return CType(Regex.IsMatch(input.Value, pattern.Value, _
          RegexOptions.CultureInvariant), SqlBoolean)
    End If
End Function
```

Czytelnicy, którzy wykonali wszystkie instrukcje opisane w Dodatku A, mogą już teraz przetestować funkcję. Wspomniane instrukcje obejmują: włączenie obsługi CLR w SQL Server (domyślnie wyłączonej), stworzenie testowej bazy danych o nazwie CLRUtilities, przygotowanie kodu w Microsoft Visual Studio 2008, zbudowanie zestawu w postaci pliku .dll na dysku, załadowanie kodu Intermediate Language (IL) z zestawu do bazy danych SQL Server i zarejestrowanie obiektów z zestawu w bazie danych. Oto kod potrzebny do włączenia funkcji CLR w SQL Server i stworzenia testowej bazy danych CLRUtilities (prezentowany z myślą o czytelnikach, którzy nie zrealizowali instrukcji przedstawionych w Dodatku A):

```
SET NOCOUNT ON;
USE master;
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE;
GO
IF DB_ID('CLRUtilities') IS NOT NULL
  DROP DATABASE CLRUtilities;
GO
CREATE DATABASE CLRUtilities;
GO
USE CLRUtilities;
```

We wszystkich przykładach prezentowanych w dalszej części książki przyjęliśmy założenie, że baza danych CLRUtilities istnieje i zestaw został zbudowany w Visual Studio.

Teraz uruchamiamy poniższy fragment kodu, aby dodać zestaw do bazy danych (jeśli nie został on jeszcze załadowany):

```
USE CLRUtilities;

CREATE ASSEMBLY CLRUtilities
FROM 'C:\CLRUtilities\CLRUtilities\bin\Debug\CLRUtilities.dll'
WITH PERMISSION_SET = SAFE;
-- W przypadku braku folderu Debug należy użyć:
-- FROM 'C:\CLRUtilities\CLRUtilities\bin\CLRUtilities.dll'
```

Oczywiście, jeśli plik CLRUtilities.dll zawierający zestaw został stworzony w innym folderze, należy odpowiednio zmodyfikować ścieżkę. Polecenie CREATE ASSEMBLY służy do załadowania kodu IL z pliku .dll do bazy danych, dzięki czemu plik zewnętrzny przestaje być potrzebny. Po dodaniu nowego obiektu do zestawu i ponownym jego zbudowaniu (jeśli nie została zastosowana opcja automatycznej instalacji dostępna w edycji

Uwaga Włączenie opcji konfiguracyjnej serwera *'clr enabled'* (domyślnie wyłączonej) powoduje, że niestandardowe zestawy mogą być uruchamiane na danym wystąpieniu SQL Server. Ponieważ nie można kontrolować tej opcji na bardziej szczegółowym poziomie, zostaje ona skonfigurowana dla całego wystąpienia SQL Server. Włączenie wsparcia dla obsługi CLR może pociągać za sobą pewne zagrożenia. Stopień ryzyka zależy od uprawnień poszczególnych zestawów. Tworząc zestawy przy użyciu polecenia CREATE ASSEMBLY, można kontrolować uprawnienia dostępu do kodu, ustawiając wartość SAFE, EXTERNAL_ACCESS lub UNSAFE w opcji PERMISSION_SET. Dokumentacja SQL Books Online w następujący sposób opisuje te trzy poziomy zabezpieczeń:

SAFE *to zalecany poziom uprawnień dla zestawu, który realizuje zadania przetwarzania lub zarządzania danymi bez uzyskiwania dostępu do zasobów znajdujących się poza wystąpieniem SQL Server.*

Zaleca się stosowanie ustawienia EXTERNAL_ACCESS *dla zestawów, które uzyskują dostęp do zasobów znajdujących się poza wystąpieniem SQL Server. Zestawy* EXTERNAL_ACCESS *oferują taką niezawodność oraz skalowalność, jak zestawy* SAFE, *ale pod względem bezpieczeństwa przypominają zestawy* UNSAFE. *Wynika to z faktu, iż kod w zestawach* EXTERNAL_ACCESS *jest domyślnie uruchamiany w kontekście konta usługi SQL Server i uzyskuje dostęp do zewnętrznych zasobów przy użyciu tego konta* (*o ile nie zastosowano bezpośredniej personifikacji użytkownika wywołującego*). *W związku z tym uprawnienie do tworzenia zestawów* EXTERNAL_ACCESS *powinno być przyznawane jedynie użytkownikom, którym można powierzyć wywoływanie kodu w kontekście konta SQL Server. Dodatkowe informacje o personifikacji zawiera sekcja CLR Integration Security.*

Określenie wartości UNSAFE *powoduje, że kod zestawu ma pełną swobodę w zakresie realizowania operacji w przestrzeni procesu SQL Server, co może potencjalnie zagrażać stabilności serwera. Zestawy* UNSAFE *mogą również obniżać bezpieczeństwo systemu SQL Server lub wspólnego środowiska uruchomieniowego. Uprawnienia* UNSAFE *powinny być przyznawane jedynie najbardziej zaufanym zestawom. Tylko członkowie stałej roli serwerowej sysadmin mogą tworzyć i modyfikować zestawy* UNSAFE.

Żadna z funkcji omawianych w tym rozdziale nie wymaga dostępu do zasobów zewnętrznych, a zatem zestaw zostanie stworzony z wykorzystaniem uprawnienia SAFE. W kolejnych rozdziałach zademonstrujemy procedurę składowaną, która wymaga poziomu uprawnień EXTERNAL_ACCESS oraz wyzwalacz wymagający poziomu uprawnień UNSAFE, aczkolwiek obiekty te będą służyły jedynie do celów demonstracyjnych. Omawiając te przykłady, zmodyfikujemy zestaw, dodając wsparcie dla wymaganych uprawnień. Nie należy zapominać o zagrożeniach, jakie pociąga za sobą zezwolenie zestawowi na uzyskiwanie dostępu do zasobów zewnętrznych.

Visual Studio Professional) trzeba własnoręcznie wywołać polecenie ALTER ASSEMBLY lub polecenia DROP oraz CREATE ASSEMBLY, aby ponownie załadować kod IL do bazy danych. Jednak w wyniku przeprowadzenia operacji omówionych w Dodatku A wszystkie niezbędne obiekty zostaną dodane do bazy danych. W dalszej części książki powyższe instrukcje zostaną pominięte.

Omówieniu nowych obiektów CLR towarzyszyć będzie prezentacja kodu T-SQL koniecznego do zarejestrowania ich w bazie danych (polecenie CREATE FUNCTION | PROCEDURE | TRIGGER). Aczkolwiek w przypadku zrealizowania wszystkich instrukcji z Dodatku A, wykonanie tej procedury nie jest konieczne.

Oto kod potrzebny do zarejestrowania wersji C# funkcji *RegexIsMatch* w bazie danych CLRUtilities:

```
USE CLRUtilities;
IF OBJECT_ID('dbo.RegexIsMatch', 'FS') IS NOT NULL
  DROP FUNCTION dbo.RegexIsMatch;
GO
CREATE FUNCTION dbo.RegexIsMatch
  (@inpstr AS NVARCHAR(MAX), @regexstr AS NVARCHAR(MAX))
RETURNS BIT
EXTERNAL NAME CLRUtilities.CLRUtilities.RegexIsMatch;
GO
```

A oto kod rejestrujący wersję Visual Basic:

```
CREATE FUNCTION dbo.RegexIsMatch
  (@inpstr AS NVARCHAR(MAX), @regexstr AS NVARCHAR(MAX))
RETURNS BIT
EXTERNAL NAME CLRUtilities.[CLRUtilities.CLRUtilities].RegexIsMatch;
GO
```

Uwaga Warto zwrócić uwagę na różne nazwy zewnętrzne, które posłużyły do zarejestrowania funkcji stworzonych z wykorzystaniem języka C# (*CLRUtilities.CLRUtilities.RegexIsMatch*) oraz języka Visual Basic (*CLRUtilities.*[*CLRUtilities.CLRUtilities*].*RegexIsMatch*). Ta dość kłopotliwa rozbieżność wynika z faktu, iż język Visual Basic w odróżnieniu od języka C# tworzy główną przestrzeń nazw. Aby kod T-SQL działał prawidłowo niezależnie od wykorzystanego języka .NET, trzeba zapobiec tworzeniu głównej przestrzeni nazw w przypadku programowania w języku Visual Basic. W Visual Studio należy prawym przyciskiem myszy kliknąć projekt, wybrać opcję Properties, a następnie wybrać stronę Application i usunąć wartość z pola tekstowego Root Namespace. W tej książce przyjęliśmy założenie, że opcja ta nie została odznaczona. W związku z tym instrukcje rejestracji obiektów w poszczególnych wersjach językowych będą zawierać różne nazwy zewnętrzne.

Jak już wspominano, aby funkcja zwracała wartość NULL, jeśli którykolwiek z parametrów wejściowych ma wartość NULL, można, rejestrując funkcję, zastosować opcję RETURNS NULL ON NULL INPUT. W efekcie, gdy którykolwiek z parametrów wejściowych będzie miał wartość NULL, SQL Server w ogóle nie wywoła funkcji i zwróci wartość NULL. Oto kod służący do zarejestrowania funkcji z wykorzystaniem opcji RETURNS NULL ON NULL INPUT:

```
IF OBJECT_ID('dbo.RegexIsMatch', 'FS') IS NOT NULL
  DROP FUNCTION dbo.RegexIsMatch;
GO
CREATE FUNCTION dbo.RegexIsMatch
  (@inpstr AS NVARCHAR(MAX), @regexstr AS NVARCHAR(MAX))
RETURNS BIT
```

```
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME CLRUtilities.CLRUtilities.RegexIsMatch;
GO
```

Funkcja *RegexIsMatch* jest już gotowa do użycia.

Dodatkowe informacje Wiele przydatnych wyrażeń regularnych można znaleźć w Internecie np. w witrynie *http://www.regexlib.com*.

Aby poznać przykładowe zastosowanie nowej funkcji, załóżmy, że chcemy upewnić się, iż wprowadzony adres e-mail jest prawidłowy. Wykorzystamy do tego celu następujące wyrażenie regularne:

```
N'^([\w-]+\.)*?[\w-]+@[\w-]+\.([\w-]+\.)*?[\w]+$'
```

Wyrażenie regularne sprawdza, czy adres rozpoczyna się od słowa, zawiera symbol @ oraz przynajmniej dwa słowa rozdzielone znakiem kropki (.), występujące po symbolu @. Adres może zawierać dodatkowe, rozdzielone kropką słowa przed i po tym symbolu. Powyższe wyrażenie regularne stanowi uproszczoną wersję, która służy jedynie do celów demonstracyjnych. Bardziej niezawodne i kompleksowe wyrażenia regularne zostały zaprezentowane w witrynie: http://www.regularexpressions.info/.

Poniższy fragment kodu zwraca wartość 1, ponieważ dostarczony adres e-mail jest prawidłowy:

```
SELECT dbo.RegexIsMatch(
  N'dejan@solidq.com',
  N'^([\w-]+\.)*?[\w-]+@[\w-]+\.([\w-]+\.)*?[\w]+$');
```

Natomiast poniższy fragment kodu zwraca wartość 0, ponieważ adres jest nieprawidłowy:

```
SELECT dbo.RegexIsMatch(
  N'dejan#solidq.com',
  N'^([\w-]+\.)*?[\w-]+@[\w-]+\.([\w-]+\.)*?[\w]+$');
```

Wskazówka Jeśli pojawi się potrzeba udostępnienia tej samej funkcji (np. funkcji *RegexIsMatch*) w innej bazie danych, wygodniej jest odwoływać się do funkcji bez konieczności określania bazy danych w jej nazwie (np. CLRUtilities.dbo.RegexIsMatch). Do tego celu służy synonim funkcji dodawany do każdej bazy danych, w której funkcja ma być dostępna. Następujący przykładowy fragment kodu służy do udostępnienia funkcji w bazie danych o nazwie MyDB:

```
USE MyDB;

CREATE SYNONYM dbo.RegexIsMatch FOR CLRUtilities.dbo.RegexIsMatch;
```

Jeśli dodamy synonim do bazy danych *model*, zostanie on umieszczony w każdej nowo tworzonej bazie danych, ponieważ nowe bazy danych stanowią kopie bazy danych *model*. Reguła ta odnosi się także do bazy danych tempdb, która jest tworzona podczas każdego ponownego uruchomienia systemu SQL Server.

Można również użyć funkcji RegexIsMatch w ograniczeniu CHECK. Następujący przykładowy fragment kodu służy do stworzenia tabeli TestRegex z wykorzystaniem ograniczenia CHECK, które sprawia, że kolumna *jpgfilename* może zawierać jedynie nazwy plików z rozszerzeniem *jpg*:

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

CREATE TABLE dbo.TestRegex
(
  jpgfilename NVARCHAR(4000) NOT NULL
  CHECK(dbo.RegexIsMatch(jpgfilename,
    N'^(([a-zA-Z]:)|(\\{2}\w+)\$?)(\\(\w[\w]*.*))+\.(jpg|JPG)$')
      = CAST(1 As BIT))
);
```

Wartość kolumny *jpgfilename* musi być zgodna z następującym wzorcem: musi rozpoczynać się od litery z zakresu od A do Z, po której następuje dwukropek (litera dysku) lub od dwóch ukośników wstecznych oraz słowa (udział sieciowy). Następnie wartość musi zawierać przynajmniej jeden ukośnik wsteczny symbolizujący główny folder dysku lub udziału. Później mogą występować dodatkowe kombinacje ukośników wstecznych i słów określających foldery podrzędne. Po ostatnim słowie musi znajdować się znak kropki oraz litery *jpg* (małe bądź wielkie).

Poniższe instrukcje INSERT zawierające prawidłowe nazwy plików JPEG zostaną zaakceptowane:

```
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'C:\Temp\myFile.jpg');
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'\\MyShare\Temp\myFile.jpg');
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'\\MyShare\myFile.jpg');
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'C:\myFile.jpg');
```

Poniższe instrukcje INSERT, które nie zawierają prawidłowych nazw plików JPEG, zostaną odrzucone:

```
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'C:\Temp\myFile.txt');
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'\\MyShare\\Temp\myFile.jpg');
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'\\myFile.jpg');
INSERT INTO dbo.TestRegex(jpgfilename) VALUES(N'C:myFile.jpg');
```

Zastąpienie bazujące na wyrażeniach regularnych Wzorce umożliwiają nie tylko dopasowywanie ciągów, ale także ich zastępowanie. Niestety język T-SQL w niewielkim stopniu wspomaga realizację tego celu. Rozważmy na przykład proste zadanie usunięcia wszystkich znaków danego wzorca z ciągu wejściowego. Poniżej zaprezentowana została implementacja tego zadania przy użyciu funkcji UDF T-SQL o nazwie *RemoveChars*:

```
IF OBJECT_ID('dbo.RemoveChars', 'FN') IS NOT NULL
  DROP FUNCTION dbo.RemoveChars;
GO

CREATE FUNCTION dbo.RemoveChars
  (@string AS NVARCHAR(MAX), @pattern AS NVARCHAR(MAX))
  RETURNS NVARCHAR(MAX)
AS
BEGIN
```

```
  DECLARE @pos AS INT;
  SET @pos = PATINDEX(@pattern, @string);

  WHILE @pos > 0
  BEGIN
    SET @string = STUFF(@string, @pos, 1, N'');
    SET @pos = PATINDEX(@pattern, @string);
  END

  RETURN @string;
END
GO
```

Funkcja akceptuje dwa parametry wejściowe: przetwarzany ciąg (*@string*) oraz wzorzec znaków (*@pattern*). Funkcja PATINDEX służy do znalezienia pozycji pierwszego wystąpienia wzorca w ciągu wejściowym i umieszcza wynik w zmiennej *@pos*. Jeśli ciąg zawiera wzorzec, funkcja STUFF zostaje użyta do usunięcia znaków z pozycji wskazanej przez zmienną *@pos*. Później następuje kolejne wyszukanie pierwszej pozycji wzorca w ciągu wejściowym. Na zakończenie funkcja zwraca ciąg, z którego usunięte zostały wszystkie znaki pasujące do wzorca.

Następujący kod prezentuje przykładowe zastosowanie funkcji. Na tabeli Sales.Customers w bazie danych InsideTSQL2008 wykonane zostaje zapytanie, które wyświetla numery telefonów po usunięciu z nich znaków, które nie są cyfrą ani literą:

```
SELECT custid, phone,
  dbo.RemoveChars(phone, N'%[^0-9a-zA-Z]%') AS cleanphone
FROM InsideTSQL2008.Sales.Customers;
```

To powoduje wygenerowanie wyniku zaprezentowanego poniżej w skróconej postaci:

```
custid      phone           cleanphone
----------- --------------- -----------
1           030-3456789     0303456789
2           (5) 789-0123    57890123
3           (5) 123-4567    51234567
4           (171) 456-7890  1714567890
5           0921-67 89 01   0921678901
6           0621-67890      062167890
7           67.89.01.23     67890123
8           (91) 345 67 89  913456789
9           23.45.67.89     23456789
10          (604) 901-2345  6049012345
...
```

Ponieważ język T-SQL nie wspiera wyrażeń regularnych, musi odwoływać się do bardziej ograniczonych metod bazujących na wykorzystaniu funkcji PATINDEX. Sposoby stosowania funkcji PATINDEX przypominają techniki bazujące na predykacie LIKE. Ponadto, iteracja w języku T-SQL przebiega dużo wolniej niż iteracje realizowane przez platformę .NET.

Można zaimplementować funkcję .NET, która wykorzystuje wyrażenia regularne do zastępowania wzorca, czyli wstawiania innego wyrażenia w miejsce wszystkich wystąpień wzorca w danym ciągu. Oto definicja C# funkcji *RegexReplace*, która zawiera wywołanie metody *Replace* obiektu *Regex*:

```
// Funkcja RegexReplace
// Zastępowanie wzorca w ciągu przy użyciu wyrażenienia regularnego
[SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
public static SqlString RegexReplace(
    SqlString input, SqlString pattern, SqlString replacement)
{
  if (input.IsNull || pattern.IsNull || replacement.IsNull)
      return SqlString.Null;
  else
    return (SqlString)Regex.Replace(
      input.Value, pattern.Value, replacement.Value);
}
```

A oto definicja funkcji w wersji Visual Basic:

```
' Funkcja RegexReplace
' Zastępowanie wzorca w ciągu przy użyciu wyrażenienia regularnego
<SqlFunction(IsDeterministic:=True, DataAccess:=DataAccessKind.None)> _
Public Shared Function RegexReplace( _
  ByVal input As SqlString, ByVal pattern As SqlString, _
  ByVal replacement As SqlString) As SqlString

  If (input.IsNull Or pattern.IsNull Or replacement.IsNull) Then
    Return SqlString.Null
  Else
    Return CType(Regex.Replace( _
      input.Value, pattern.Value, replacement.Value), SqlString)
  End If
End Function
```

Funkcja akceptuje trzy parametry wejściowe: *input*, *pattern* oraz *replacement*. Po walidacji danych wejściowych funkcja wywołuje metodę *Regex.Replace*, która wstawia wzorzec *replacement* w miejsce każdego wystąpienia wzorca *pattern* w ciągu *input*. Aby zastąpić wszystkie wystąpienia wzorca, nie trzeba dokonywać bezpośredniej iteracji. Co więcej, funkcja obsługuje wzorce reprezentujące ciągi o dowolnej długości, podczas gdy wersja T-SQL wspierała jedynie wzorce reprezentujące pojedynczy znak.

Następujący fragment kodu służy do zarejestrowania funkcji *RegexReplace* w wersji C#:

```
IF OBJECT_ID('dbo.RegexReplace', 'SF') IS NOT NULL
  DROP FUNCTION dbo.RegexReplace;
GO
CREATE FUNCTION dbo.RegexReplace(
  @input       AS NVARCHAR(MAX),
  @pattern     AS NVARCHAR(MAX),
  @replacement AS NVARCHAR(MAX))
RETURNS NVARCHAR(MAX)
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME CLRUtilities.CLRUtilities.RegexReplace;
GO
```

Natomiast poniższy fragment kodu służy do zarejestrowania funkcji zaimplementowanej w języku Visual Basic:

```
CREATE FUNCTION dbo.RegexReplace(
  @input       AS NVARCHAR(MAX),
  @pattern     AS NVARCHAR(MAX),
  @replacement AS NVARCHAR(MAX))
RETURNS NVARCHAR(MAX)
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME CLRUtilities.[CLRUtilities.CLRUtilities].RegexReplace;
GO
```

Oto zmodyfikowane zapytanie, które wyświetla „oczyszczone" numery telefonu – tym razem z wykorzystaniem funkcji *RegexReplace*:

```
SELECT phone, dbo.RegexReplace(phone, N'[^0-9a-zA-Z]', N'') AS cleanphone
FROM InsideTSQL2008.Sales.Customers;
```

Zastosowany w instrukcji wzorzec przypomina wzorzec z przykładu, w którym użyliśmy funkcji T-SQL, jednak wersja CLR wykorzystuje wyrażenie regularne. Wyrażenia regularne oferują dużo szersze możliwości niż rozwiązania bazujące na funkcji T-SQL PATINDEX. Co więcej, wersja CLR funkcji jest znacznie szybsza.

W dalszej części tego rozdziału w części zatytułowanej „Podpis SQL" zaprezentowane zostaną bardziej zaawansowane metody stosowania funkcji *RegexReplace*.

Formatowanie wartości daty i czasu Kolejny typ operacji przetwarzania ciągów, których realizacja w języku T-SQL bywa żmudna i powolna, stanowi formatowanie wartości daty i godziny w oparciu o podany ciąg formatowania. Język T-SQL oferuje funkcję CONVERT, która umożliwia konwertowanie wartości daty i godziny do postaci tekstowej, jednak dostępny jest tylko niewielki zestaw predefiniowanych stylów. T-SQL nie pozwala na stosowanie ciągów formatowania, które są wspierane przez wiele proceduralnych języków programowania. Aby osiągnąć ten cel, trzeba zaimplementować własną funkcję. Czytelnicy, którzy kiedykolwiek próbowali zrealizować to zadanie w języku T-SQL, wiedzą, że jest to trudny i nieefektywny proces.

Implementacja tego wymagania z wykorzystaniem funkcji CLR jest banalnie prosta. Wystarczy napisać funkcję akceptującą parametry definiujące wartość daty i godziny (typu *SqlDateTime*) oraz ciąg formatowania (typu *SqlString*). Definicja funkcji bazuje na pojedynczej linii kodu, w której na wartości daty i godziny wykonana zostaje metoda *.Value.ToString* z ciągiem formatowania w roli parametru.

Uwaga Istotną różnicą między funkcją CONVERT a procedurą CLR jest to, że funkcja CONVERT będzie działała zgodnie z ustawieniem SET LANGUAGE sesji SQL Server, natomiast procedura CLR zgodnie z kulturą aktualnego wątku. W przypadku wykorzystywania procedury CLR ustawienie SET LANGUAGE nie zostaje uwzględnione (decyduje jedynie o sposobie konwertowania ciągów do przekazywanych parametrów przed wywołaniem procedury CLR).

Oto definicja funkcji w języku C#:

```
// Funkcja FormatDatetime
// Formatuje wartość DATETIME w oparciu o ciąg formatowania
[Microsoft.SqlServer.Server.SqlFunction]
```

```
  public static SqlString FormatDatetime(SqlDateTime dt, SqlString formatstring)
  {
    if (dt.IsNull || formatstring.IsNull)
      return SqlString.Null;
    else
      return (SqlString)dt.Value.ToString(formatstring.Value);
  }
```

A oto definicja Visual Basic:

```
  'Funkcja FormatDatetime
  ' Formatowanie wartości DATETIME w oparciu o ciąg formatowania
  <SqlFunction(IsDeterministic:=True, DataAccess:=DataAccessKind.None)> _
  Public Shared Function FormatDatetime( _
    ByVal dt As SqlDateTime, ByVal formatstring As SqlString) As SqlString

    If (dt.IsNull Or formatstring.IsNull) Then
      Return SqlString.Null
    Else
      Return CType(dt.Value.ToString(formatstring.Value), SqlString)
    End If
  End Function
```

Następujący fragment kodu służy do zarejestrowania funkcji w wersji C#:

```
IF OBJECT_ID('dbo.FormatDatetime', 'SF') IS NOT NULL
  DROP FUNCTION dbo.FormatDatetime;
GO
CREATE FUNCTION dbo.FormatDatetime
  (@dt AS DATETIME, @formatstring AS NVARCHAR(500))
RETURNS NVARCHAR(500)
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME CLRUtilities.CLRUtilities.FormatDatetime;
GO
```

Natomiast poniższy fragment kodu służy do zarejestrowania funkcji w wersji Visual Basic:

```
CREATE FUNCTION dbo.FormatDatetime
  (@dt AS DATETIME, @formatstring AS NVARCHAR(500))
RETURNS NVARCHAR(500)
WITH RETURNS NULL ON NULL INPUT
EXTERNAL NAME CLRUtilities.[CLRUtilities.CLRUtilities].FormatDatetime;
GO
```

Teraz można już zastosować funkcję *FormatDatetime*, określając wybrany ciąg formatowania np.:

```
SELECT dbo.FormatDatetime(GETDATE(), 'MM/dd/yyyy');
```

Szczegółowe informacje o ciągach formatowania daty i godziny znaleźć można w artykule http://msdn.microsoft.com/en-us/library/97x6twsz.aspx.

Na zakończenie uruchamiamy następujący fragment kodu służący do przywrócenia pierwotnego stanu:

```
IF OBJECT_ID('dbo.TestRegex','U') IS NOT NULL
  DROP TABLE dbo.TestRegex;
IF OBJECT_ID('dbo.RegexIsMatch', 'FS') IS NOT NULL
```

```
  DROP FUNCTION dbo.RegexIsMatch;
IF OBJECT_ID('dbo.RemoveChars', 'FN') IS NOT NULL
  DROP FUNCTION dbo.RemoveChars;
IF OBJECT_ID('dbo.RegexReplace', 'FS') IS NOT NULL
  DROP FUNCTION dbo.RegexReplace;
IF OBJECT_ID('dbo.FormatDatetime', 'SF') IS NOT NULL
  DROP FUNCTION dbo.FormatDatetime;
```

Konwersje jawne kontra niejawne

Rozwijając obiekty CLR w SQL Server 2008, można przypisywać zmiennym oraz parametrom wejściowym/wyjściowym macierzyste typy .NET lub typy .NET SQL. Typy .NET SQL mogą być dokładniej mapowane do typów SQL Server. Natomiast stosowanie macierzystych typów .NET w interfejsach obiektów wiąże się z niejawnym rzutowaniem wartości w momencie przekazywania ich do lub z systemu SQL Server. Niektórzy programiści stosują jedynie typy .NET SQL, gdyż uważają, że niejawne konwersje stanowią niepotrzebne obciążenie. Takie podejście może być w pewnych sytuacjach ograniczeniem, ponieważ typy .NET SQL nie oferują tak bogatej funkcjonalności jak macierzyste typy .NET. Na przykład macierzysty typ .NET *System.String* (*string* w C#, *String* w Visual Basic) udostępnia metodę *Substring*, której nie dostarcza typ .NET SQL *SqlString*.

W rzeczywistości wydajność nie jest najistotniejszym czynnikiem. Najważniejszym argumentem przemawiającym za stosowaniem typów SQL w kodzie .NET bazodanowych obiektów CLR jest fakt, iż macierzyste typy .NET nie wspierają wartości NULL. Typy SQL pozwalają na sprawdzenie przy użyciu metody *IsNull*, czy dane wejściowe mają wartość NULL, natomiast macierzyste typy .NET nie oferują tej metody. Próba wywołania procedury CLR z kodu T-SQL i przekazania wartości NULL do argumentu macierzystego typu .NET spowoduje wygenerowanie wyjątku (o ile funkcja nie została zarejestrowana z wykorzystaniem opcji RETURNS NULL ON NULL INPUT). Gdy potrzebna jest dodatkowa funkcjonalność oferowana przez macierzyste typy .NET, konieczne jest jawne rzutowanie. Można również pobrać wartość macierzystego typu .NET przy użyciu właściwości *Value* zmiennej typu SQL, przechować tę wartość w innej zmiennej macierzystego typu .NET, a następnie zastosować wybrane właściwości lub metody macierzystego typu. W funkcji *RegexIsMatch* parametrem wejściowym metody *Regex.IsMatch* jest ciąg macierzystego typu .NET, w związku z tym zastosowana została właściwość *Value* typów .NET SQL. Metoda zwraca wartość logiczną macierzystego typu .NET, a zatem kod rzutuje ją w sposób jawny do typu *SqlBoolean*.

W tym podrozdziale pokażemy, że nie istnieją duże różnice wydajności między rzutowaniem niejawnym a jawnym. Następujący fragment kodu C# służy do zdefiniowania funkcji *ImpCast* (wykorzystującej macierzyste typy.NET i niejawną konwersję) oraz funkcji *ExpCast* (wykorzystującej typy.NET SQL i jawną konwersję)

```
// Porównanie konwersji niejawnej z jawną
[SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
public static string ImpCast(string inpStr)
{
    return inpStr.Substring(2, 3);
}
```

```
    [SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
    public static SqlString ExpCast(SqlString inpStr)
    {
        return (SqlString)inpStr.ToString().Substring(2, 3);
    }
```

A oto definicja funkcji w języku Visual Basic:

```
    ' Porównanie konwersji niejawnej z jawną
    <SqlFunction(IsDeterministic:=True, DataAccess:=DataAccessKind.None)> _
    Public Shared Function ImpCast(ByVal inpStr As String) As String
        Return inpStr.Substring(2, 3)
    End Function

    <SqlFunction(IsDeterministic:=True, DataAccess:=DataAccessKind.None)> _
    Public Shared Function ExpCast(ByVal inpStr As SqlString) As SqlString
        Return CType(inpStr.ToString().Substring(2, 3), SqlString)
    End Function
```

Następujący fragment kodu służy do zarejestrowania funkcji C# w bazie danych:

```
IF OBJECT_ID('dbo.ImpCast', 'FS') IS NOT NULL
  DROP FUNCTION dbo.ImpCast;
IF OBJECT_ID('dbo.ExpCast', 'FS') IS NOT NULL
  DROP FUNCTION dbo.ExpCast;
GO
-- Stworzenie funkcji ImpCast
CREATE FUNCTION dbo.ImpCast(@inpstr AS NVARCHAR(4000))
RETURNS NVARCHAR(4000)
EXTERNAL NAME CLRUtilities.CLRUtilities.ImpCast;
GO
--Stworzenie funkcji ExpCast
CREATE FUNCTION dbo.ExpCast(@inpstr AS NVARCHAR(4000))
RETURNS NVARCHAR(4000)
EXTERNAL NAME CLRUtilities.CLRUtilities.ExpCast;
GO
```

Natomiast poniższy fragment kodu służy do zarejestrowania funkcji Visual Basic:

```
-- Stworzenie funkcji ImpCast
CREATE FUNCTION dbo.ImpCast(@inpstr AS NVARCHAR(4000))
RETURNS NVARCHAR(4000)
EXTERNAL NAME CLRUtilities.[CLRUtilities.CLRUtilities].ImpCast;
GO
-- Stworzenie funkcji ExpCast
CREATE FUNCTION dbo.ExpCast(@inpstr AS NVARCHAR(4000))
RETURNS NVARCHAR(4000)
EXTERNAL NAME CLRUtilities.[CLRUtilities.CLRUtilities].ExpCast;
GO
```

Poniższy fragment kodu zawiera milion wywołań funkcji *ImpCast* w pętli. Jego wykonanie w przykładowym systemie testowym zajęło 17 sekund:

```
SET NOCOUNT ON;

DECLARE @a AS NVARCHAR(4000);
DECLARE @i AS INT;
SET @i = 1;
```

```
WHILE @i <= 1000000
BEGIN
 SET @a = dbo.ImpCast(N'123456');
 SET @i = @i + 1;
END
```

Poniższy fragment kodu służy do wielokrotnego wywoływania funkcji *ExpCast* i był wykonywany przez 18 sekund:

```
DECLARE @a AS NVARCHAR(4000);
DECLARE @i AS INT;
SET @i = 1;
WHILE @i <= 1000000
BEGIN
 SET @a = dbo.ExpCast(N'123456');
 SET @i = @i + 1;
END
```

Jak widać, różnica nie jest znacząca. Co więcej, w tym teście metoda niejawnego rzutowania została zrealizowana trochę szybciej niż metoda jawnego rzutowania.

Na zakończenie uruchamiamy następujący fragment kodu służący do przywrócenia pierwotnego stanu:

```
IF OBJECT_ID('dbo.ImpCast', 'FS') IS NOT NULL
  DROP FUNCTION dbo.ImpCast;
IF OBJECT_ID('dbo.ExpCast', 'FS') IS NOT NULL
  DROP FUNCTION dbo.ExpCast;
```

Podpis SQL

Niniejszy podrozdział zawiera implementacje T-SQL oraz CLR funkcji, która zwraca podpis dla ciągu zapytania. Funkcja przyjmuje parametr wejściowy zawierający ciąg zapytania i zwraca ciąg reprezentujący podpis zapytania lub szablon. W podpisie wszystkie literały w wejściowym ciągu zapytania zostają zastąpione tym samym symbolem (w tym przypadku #). Na przykład dla następującego ciągu zapytania:

```
N'SELECT * FROM dbo.T1 WHERE col1 = 3 AND col2 > 78;'
```

Funkcja powinna zwrócić następujący ciąg:

```
N'SELECT * FROM dbo.T1 WHERE col1 = # AND col2 > #'
```

Tego typu funkcja może okazać się bardzo przydatna, gdy chcemy agregować dane o wydajności zapytań po wstawieniu zarejestrowanych informacji do tabeli. Gdybyśmy pogrupowali dane według oryginalnych ciągów zapytań, zapytania, które są w rzeczywistości logicznie równoważne, zostałyby umieszczone w osobnych grupach. Agregacja danych wydajności według podpisu zapytania oferuje bardziej użyteczne i wartościowe informacje.

Funkcja UDF T-SQL generująca podpis SQL

Poniższy fragment kodu zawiera implementację T-SQL funkcji zwracającej podpis SQL:

```
IF OBJECT_ID('dbo.SQLSigTSQL', 'FN') IS NOT NULL
  DROP FUNCTION dbo.SQLSigTSQL;
GO

CREATE FUNCTION dbo.SQLSigTSQL
  (@p1 NVARCHAR(MAX), @parselength INT = 4000)
RETURNS NVARCHAR(4000)

-- Funkcja została napisana przez programistów Microsoft
-- i umieszczona w tej książce za ich zgodą.
-- Funkcja nie jest objęta żadną gwarancją.
-- Zasady użycia przykładowych skryptów zostały zaprezentowane w umowie
-- http://www.microsoft.com/info/cpyright.htm

-- Analizuje ciągi zapytań
AS
BEGIN
  DECLARE @pos AS INT;
  DECLARE @mode AS CHAR(10);
  DECLARE @maxlength AS INT;
  DECLARE @p2 AS NCHAR(4000);
  DECLARE @currchar AS CHAR(1), @nextchar AS CHAR(1);
  DECLARE @p2len AS INT;

  SET @maxlength = LEN(RTRIM(SUBSTRING(@p1,1,4000)));
  SET @maxlength = CASE WHEN @maxlength > @parselength
                     THEN @parselength ELSE @maxlength END;
  SET @pos = 1;
  SET @p2 = '';
  SET @p2len = 0;
  SET @currchar = '';
  set @nextchar = '';
  SET @mode = 'command';

  WHILE (@pos <= @maxlength)
  BEGIN
    SET @currchar = SUBSTRING(@p1,@pos,1);
    SET @nextchar = SUBSTRING(@p1,@pos+1,1);
    IF @mode = 'command'
    BEGIN
      SET @p2 = LEFT(@p2,@p2len) + @currchar;
      SET @p2len = @p2len + 1 ;
      IF @currchar IN (',','(',' ','=','<','>','!')
        AND @nextchar BETWEEN '0' AND '9'
      BEGIN
        SET @mode = 'number';
        SET @p2 = LEFT(@p2,@p2len) + '#';
        SET @p2len = @p2len + 1;
      END
      IF @currchar = '"'
      BEGIN
        SET @mode = 'literal';
        SET @p2 = LEFT(@p2,@p2len) + '#"';
        SET @p2len = @p2len + 2;
      END
    END
    ELSE IF @mode = 'number' AND @nextchar IN (',',')',' ','=','<','>','!')
```

```
      SET @mode= 'command';
    ELSE IF @mode = 'literal' AND @currchar = """
      SET @mode= 'command';

    SET @pos = @pos + 1;
  END
  RETURN @p2;
END
GO
```

Uwaga Podziękowania dla twórcy funkcji Stuarta Ozera za zgodę na zaprezentowanie jej w tej książce. Stuart należy do zespołu Microsoft SQL Server Customer Advisory Team (SQL CAT).

Funkcja *SQLSigTSQL* akceptuje dwa parametry wejściowe: ciąg zapytania *@p1* oraz maksymalną liczbę analizowanych znaków *@parselength*. Jeśli wartość *@parselength* jest mniejsza niż długość ciągu zapytania przechowywana w parametrze *@p1*, funkcja analizuje jedynie *@parselength* pierwszych znaków. Funkcja dokonuje iteracji po kolejnych znakach ciągu. Zachowuje wartość stanu w zmiennej *@mode*, która może zawierać trzy wartości: *'command'*, *'number'* oraz *'literal'*.

Command to stan domyślny, który oznacza, że aktualny znak zostanie dołączony do ciągu wyjściowego w niezmienionej postaci. Wartość *Number* oznacza, że zidentyfikowany został literał liczbowy, w związku z czym dołączony zostanie symbol #. Literał *number* zostaje zidentyfikowany, gdy cyfra jest poprzedzona przecinkiem, otwierającym nawiasem, spacją lub operatorem. Stan zmienia się z *number* na *command*, gdy następny znak to przecinek, zamykający nawias lub operator. Wartość *Literal* oznacza, że zidentyfikowany został ciąg tekstowy, co powoduje dołączenie do ciągu symbolu #. Literał ciągu tekstowego zostaje zidentyfikowany przez wykrycie otwierającego cudzysłowu. Stan zmienia się z *literal* na *command* po wykryciu zamykającego cudzysłowu.

Aby przetestować funkcję *SQLSigTSQL*, uruchamiamy następującą instrukcję:

```
SELECT dbo.SQLSigTSQL
  (N'SELECT * FROM dbo.T1 WHERE col1 = 3 AND col2 > 78', 4000);
```

Zaprezentowany zostanie następujący wynik:

```
SELECT * FROM dbo.T1 WHERE col1 = # AND col2 > #
```

Funkcja UDF CLR generująca podpis SQL

Poniższy fragment kodu stanowi implementację funkcji generującej podpis SQL w języku C#:

```
// Funkcja SQLSigCLR
// Generuje podpis SQL dla ciągu zapytania wejściowego
[SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
public static SqlString SQLSigCLR(SqlString inpRawString,
  SqlInt32 inpParseLength)
{
    if (inpRawString.IsNull)
        return SqlString.Null;
```