

Benjamin Nevarez

Microsoft SQL Server 2014

Optymalizacja zapytań

**Wrzuć
piąty bieg!**

Helion 

Tytuł oryginału: Microsoft® SQL Server® 2014 Query Tuning & Optimization

Tłumaczenie: Jakub Hubisz

ISBN: 978-83-283-1165-7

Original edition copyright © 2015 by Benjamin Nevarez.
All rights reserved.

Polish edition copyright © 2015 by HELION SA.
All rights reserved.

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

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/sql14o.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/sql14o>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Książkę dedykuję moim rodzicom: Guadalupe Chavez i Humberto Nevarezowi

Spis treści

Podziękowania	11
Wprowadzenie	13
 Rozdział 1. Wprowadzenie do optymalizacji zapytań	17
Architektura	19
Parsowanie i przypisywanie	21
Optymalizacja zapytań	21
Generowanie możliwych planów zapytań	22
Określanie kosztu każdego z planów	23
Wykonywanie zapytań i przechowywanie planów	23
Plany wykonania	25
Plany graficzne	26
XML	32
Plany tekstowe	35
Dodatkowe właściwości planów	36
Ostrzeżenia w planach wykonania	39
Pobieranie planów za pomocą śledzenia lub z magazynu planów	44
Usuwanie planów z magazynu planów	49
SET STATISTICS TIME i SET STATISTICS IO	50
Podsumowanie	52
 Rozdział 2. Rozwiązywanie problemów w zapytaniach	53
DMV i DMF	55
sys.dm_exec_requests i sys.dm_exec_sessions	55
sys.dm_exec_query_stats	57
Wartości statement_start_offset i statement_end_offset	60
sql_handle i plan_handle	61
query_hash i plan_hash	62
Szukanie kosztownych zapytań	64
SQL Trace	65
Zdarzenia rozszerzone	69
Mapowanie zdarzeń SQL Trace na zdarzenia rozszerzone	71
Tworzenie sesji	73

6 Microsoft SQL Server 2014. Optymalizacja zapytań

Data Collector	82
Konfiguracja	83
Wykorzystanie Data Collectora	87
Zapytania na tabelach Data Collectora	88
Podsumowanie	90
 Rozdział 3. Optymalizator zapytań	91
Przegląd	92
sys.dm_exec_query_optimizer_info	94
Parsowanie i przypisywanie	101
Upraszczanie	104
Wykrywanie sprzeczności	105
Usuwanie złączeń z kluczem obcym	107
Plan trywialny	109
Reguły transformacji	112
Memo	122
Statystyki	127
Pełna optymalizacja	129
Search 0	131
Search 1	131
Search 2	133
Podsumowanie	135
 Rozdział 4. Operatory zapytań	137
Operatory dostępu do danych	138
Skanowanie	139
Przeszukiwanie	141
Wyszukiwanie zaznaczeń	143
Agregacje	147
Sortowanie i haszowanie	147
Stream Aggregate	147
Hash Aggregate	150
Distinct Sort	151
Złączenia	152
Nested Loops Join	153
Merge Join	155
Hash Join	157
Działania równoległe	158
Operator wymiany	160
Ograniczenia	166

Aktualizacje	167
Plany per wiersz i per indeks	169
Zabezpieczenie przed problemem Halloween	172
Podsumowanie	173
 Rozdział 5. Indeksy	175
Wprowadzenie	176
Tworzenie indeksów	177
Indeksy klastrowe a sterty	181
Klucz indeksu klastrowego	185
Indeksy pokrywające	186
Indeksy filtrowane	187
Operacje na indeksach	189
Database Engine Tuning Advisor	193
Optymalizacja zapytań i korzystanie z magazynu planów	196
Rozładowanie narzutu optymalizacji na serwer testowy	197
Brakujące indeksy	202
Fragmentacja indeksów	204
Nieużywane indeksy	206
Podsumowanie	208
 Rozdział 6. Statystyki	209
Statystyki	210
Tworzenie i aktualizacja statystyk	210
Sprawdzanie obiektów statystyk	213
Gęstość	216
Histogram	218
Nowy mechanizm szacowania kardynalności	222
Przykłady	223
Flaga 4137	227
Błędy szacunku kardynalności	227
Statystyki inkrementacyjne	229
Statystyki dla kolumn wyliczeniowych	232
Statystyki filtrowane	234
Statystyki dla kluczy rosnących	236
Flaga 2389	238
UPDATE STATISTICS z ROWCOUNT i PAGECOUNT	242
Statystyki na serwerach połączonych	245
Konserwacja statystyk	246
Szacowanie kosztów	249
Podsumowanie	251

8 Microsoft SQL Server 2014. Optymalizacja zapytań

Rozdział 7.	OLTP w pamięci — Hekaton	253
	Architektura	255
	Tabele i indeksy	257
	Tworzenie tabel Hekatona	258
	Indeksy haszowe	264
	Indeksy zakresowe	268
	Przykłady	269
	Natywnie kompilowane procedury przechowywane	273
	Tworzenie natywnie kompilowanych procedur przechowywanych	273
	DLL	276
	Ograniczenia	279
	Narzędzie AMR	280
	Podsumowanie	285
Rozdział 8.	Magazynowanie planów	287
	Kompilacja i rekompilacja zestawów zapytań	288
	Przeglądanie magazynu planów	292
	Jak usuwać plany?	294
	Parametryzacja	295
	Autoparametryzacja	296
	Opcja optymalizacji dla zapytań ad hoc	297
	Wymuszona parametryzacja	299
	Procedury przechowywane	300
	Podsluchiwanie parametrów	302
	Optymalizacja pod typowy parametr	304
	Optymalizacja przy każdym wykonaniu	305
	Zmienne lokalne i odpowiedź OPTIMIZE FOR UNKNOWN	306
	Wyłączanie podsłuchiwania parametrów	308
	Podsłuchiwanie parametrów i opcje SET wpływające na powtórne wykorzystanie planów	309
	Podsumowanie	315
Rozdział 9.	Hurtownie danych	317
	Hurtownie danych	318
	Optymalizacja złączenia gwiazdowego	321
	Indeksy magazynu kolumn	326
	Korzyści wydajnościowe	327
	Przetwarzanie partiami	329
	Tworzenie indeksów magazynu kolumn	330
	Podpowiedzi	335
	Podsumowanie	336

Rozdział 10.	Ograniczenia i podpowiedzi procesora zapytań	339
	Badania nad optymalizacją zapytań	341
	Kolejność złączeń	341
	Rozbijanie skomplikowanych zapytań	344
	Logika OR w klauzuli WHERE	345
	Złączenia i zagregowane zbiory danych	347
	Podpowiedzi	348
	Kiedy korzystać z podpowiedzi?	349
	Rodzaje podpowiedzi	351
	Złączenia	352
	Agregacje	355
	FORCE ORDER	356
	INDEX, FORCESCAN i FORCESEEK	359
	FAST N	361
	NOEXPAND i EXPAND VIEWS	363
	Wskazówki planów	364
	USE PLAN	366
	Podsumowanie	368
Dodatek	Źródła	369
	Opracowania techniczne	370
	Artykuły	371
	Prace naukowe	372
	Książki	374
	Skorowidz	375

O autorze

Benjamin Nevarez jest wybitnym specjalistą w dziedzinie platformy SQL Server i posiada tytuł MVP nadany przez Microsoft. Pracuje jako niezależny konsultant specjalizujący się w optymalizacji zapytań SQL Servera. Mieszka w Los Angeles w Stanach Zjednoczonych. Jest autorem książki *Inside the SQL Server Query Optimizer* i współtworzył inne pozycje dotyczące SQL Servera, między innymi *SQL Server 2012 Internals*. Ze względu na ponad 20 lat doświadczenia w pracy z relacyjnymi bazami danych Benjamin bywa również prelegentem na wielu konferencjach poświęconych tematyce SQL Servera, włączając w to PASS Summit, SQL Server Connections i SQLBits. Blog Benjaminą znajdziesz na stronie www.benjaminnevarez.com; możesz także skontaktować się z nim pod adresem admin@benjaminnevarez.com lub za pośrednictwem Twittera (@BenjaminNevarez).

O redaktorze merytorycznym

Dave Ballantyne pracuje jako architekt danych SQL Servera dla Win Technologies, firmy będącej częścią grupy BetWay, i mieszka niedaleko Londynu w Anglii. Jest regularnym mówcą na konferencjach i spotkaniach w Europie i Anglii, a aktualnie wspiera londyńską społeczność SQL Servera poprzez organizowanie spotkań grupy SQL Supper. Aktywnie interesuje się wszystkimi sprawami związanymi z SQL i bazami danych i jest najszczęśliwszy wtedy, gdy rozkłada na czynniki pierwsze niewydajne zapytania. Stworzył również dodatek open source do narzędzi SQL Server Data Tools (SSDT) o nazwie TSQL Smells, który wykrywa i raportuje „podejrzany” kod w projekcie. Narzędzie to znajdziesz pod adresem <http://tsqlsmellssdt.codeplex.com>. Poza pracą jest mężem, ojcem trojga dzieci i zapalonym łucznikiem.

Podziękowania

Wiele osób pomogło w stworzeniu tej książki. Najpierw chciałbym podziękować wszystkim z zespołu McGraw-Hill Education, z którymi pośrednio i bezpośrednio pracowałem nad książką, a są to: Wendy Rinaldi, Harleen Chopra, Bart Reed i Janet Walden. W szczególności dziękuję Wendy — za koordynację całego projektu i próby sprawienia, abym trzymał się harmonogramu. Specjalne podziękowania kieruję do mojego redaktora merytorycznego, Dave’a Balantyne’a — jego nieocenione opinie były kluczowe dla poprawy jakości tej książki.

Oprócz Dave’a i zespołu McGraw-Hill Education jest sporo osób, które pośrednio miały wpływ na książkę poprzez swe prace opisujące optymalizację zapytań w SQL Serverze, a są to Kalen Delaney, Cesar Galindo-Legaria, Craig Freedman, Conor Cunningham, Ken Henderson, Lubor Kollar, Eric Hanson, Goetz Graefe i inni. Podziękowania należą się również blogerom społeczności SQL Servera, od których także wiele się nauczyłem. Długa lista nazwisk byłaby niemożliwa do zamieszczenia tutaj.

Najbardziej osobiste podziękowania należą się mojej rodzinie: mojej żonie Rocio i moim trzem synom: Diego, Benjaminowi i Davidowi. Dziękuję Wam za bezwarunkowe wsparcie i za cierpliwość za każdym razem, kiedy musiałem pracować nad kolejną książką dotyczącą SQL Servera.

12 Microsoft SQL Server 2014. Optymalizacja zapytań

Wprowadzenie

Ta książka opisuje dostosowywanie i optymalizację zapytań SQL Servera i daje Ci narzędzia i wiedzę niezbędne, aby uzyskać najwyższą wydajność zapytań i aplikacji. Zazwyczaj kojarzymy optymalizację zapytań z pracą wykonywaną przez optymalizator zapytań, który tworzy wydajny plan wykonania, czasami jednak nie jesteśmy zadowoleni z wydajności wykonania i próbujemy ją poprawić, wprowadzając dodatkowe zmiany. Należy jednak zrozumieć, że rezultaty, które otrzymujemy z procesora zapytań, zależą od informacji, jakie mu zapewnimy — na przykład od projektu naszej bazy, zdefiniowanych indeksów, a nawet pewnych ustawień konfiguracyjnych. Jest wiele sposobów, by wpływać na pracę wykonywaną przez procesor zapytań, dlatego bardzo ważne jest zrozumienie tego, jak możemy pomóc w wykonywaniu dobrej jakości pracy. Zapewnienie wysokiej jakości informacji dla procesora zapytań najprawdopodobniej zaowocuje wysokiej jakości planami, które z kolei poprawią wydajność Twojej bazy. Żaden procesor zapytań nie jest jednak idealny i ważne jest również, aby rozumieć powody, dla których czasami faktycznie nie będziemy mogli otrzymać dobrego planu lub wysokiej wydajności zapytania, oraz znać możliwe rozwiązania tego problemu.

Kto powinien przeczytać tę książkę?

Ta książka przeznaczona jest dla wszystkich osób zawodowo zajmujących się bazami SQL Servera: programistów baz danych, administratorów, architektów danych — właściwie dla każdego, kto wykonuje bardziej skomplikowane zapytania na bazie SQL Servera. Zakładam, że masz pewne doświadczenie z pracą z SQL Serverem i znasz język SQL.

O czym jest ta książka?

W niniejszej książce omawiam tematykę uzyskania najwyższej wydajności zapytań i wykorzystania tej wiedzy do budowy wysoko wydajnych aplikacji. Pokazuję, jak lepsze zrozumienie tego, co robi procesor zapytań, może pomóc administratorom i programistom baz danych pisać lepsze zapytania i zapewniać procesorowi zapytań lepsze informacje potrzebne do tworzenia wydajnych planów zapytań. Książka pokazuje również, jak wykorzystać nowo zdobytą wiedzę na temat wewnętrznych operacji

procesora zapytań i narzędzi SQL Servera do szukania problemów w zapytaniach, które nie są odpowiednio wydajne.

Rozdział 1. rozpoczyna się od ogólnej architektury silnika relacyjnego SQL Servera, a następnie dokładnie opisuje wykorzystanie planów wykonywania, podstawowego narzędzia interakcji z procesorem zapytań.

Rozdział 2. to kontynuacja rozdziału 1. przedstawiająca dodatkowe narzędzia i techniki, takie jak śledzenie, zdarzenia rozszerzone czy widoki DMV, które pozwalają zbadać, jak Twoje zapytania wykorzystują zasoby systemowe, oraz zidentyfikować problemy związane z wydajnością. Rozdział kończy się wstępem do mechanizmu Data Collector, czyli funkcjonalności wprowadzonej w SQL Serverze 2008.

Rozdziały 3. i 4. stanowią zagłębienie się w wewnętrzne działanie optymalizatora zapytań i jego operatorów. W rozdziale 3. objaśniam, jak działa optymalizator, i pokazuję, dlaczego ta wiedza może dać Ci doskonale podwaliny do odszukiwania problemów oraz optymalizowania i dopracowywania zapytań. W rozdziale 4. poznasz najczęściej wykorzystywane w planach zapytań operatory.

Po dwóch rozdziałach dotyczących architektury i wewnętrznych mechanizmów procesora zapytań rozdział 5. pozwoli Ci powrócić do praktycznej wiedzy, omawiając indeksy. Indeksy to jedna z najważniejszych technik używanych podczas optymalizacji i mogą one znacznie podnieść wydajność Twoich zapytań i bazy danych.

Statystyki to kolejny ważny temat pomocny podczas optymalizowania i rozwiązywania problemów z zapytaniem. Statystyki są wykorzystywane przez optymalizator do podejmowania decyzji prowadzących do znalezienia wydajnego planu — informacje te są również dostępne dla Ciebie, abyś mógł wykorzystać je do rozwiązywania problemów z szacowaniem kardynalności. Statystyki zostały omówione w rozdziale 6.

Bazy OLTP w pamięci, znane pod nazwą Hekaton, są bez wątpienia najważniejszą technologią wprowadzoną w SQL Serverze 2014, a rozdział 7. pokazuje, jak ten nowy komponent może pomóc w tworzeniu wysoko wydajnych aplikacji. Hekaton to nowy silnik bazodanowy, którego główne funkcjonalności to tabele i indeksy zoptymalizowane do pracy w pamięci, procedury przechowywane kompilowane do kodu natywnego oraz eliminacja blokad logicznych i fizycznych.

Optymalizacja zapytań to relatywnie kosztowna operacja, dlatego przechowywanie i powtarzne wykorzystywanie planów pozwala uniknąć tego kosztu. To, jak działa magazynowanie planów i dlaczego jest tak ważne z punktu widzenia wydajności Twoich zapytań i całego SQL Servera, zostało opisane w rozdziale 8.

Rozdział 9. to wprowadzenie do hurtowni danych, wyjaśniające, jak optymalizator SQL Servera identyfikuje tabele faktów i wymiarów oraz jak optymalizuje zapytania ze złączeniami gwiazdzystymi. Rozdział ten zawiera również informacje na temat indeksów magazynu kolumn (funkcjonalności wprowadzonej w SQL Serverze 2012) oraz nowych algorytmów przetwarzania danych w partiach, które wielokrotnie poprawiają wydajność zapytań ze złączeniem gwiazdzystym.

W ostatnim rozdziale książki omawiam wyzwania stojące przed procesorem zapytań, które dzisiaj, po ponad czterech dekadach badań, wciąż są aktualne. Zawarte w nim zostały zalecenia i wskazówki dla skomplikowanych zapytań, dla których procesor może nie być w stanie znaleźć wydajnego planu. Wreszcie przedstawione są tu odpowiedzi, które należy wykorzystywać z rozwagą i tylko w ostateczności, są one bowiem sposobem na bezpośrednie wpływanie na decyzje podejmowane przez optymalizator zapytań.

16 Microsoft SQL Server 2014. Optymalizacja zapytań

Rozdział 1

Wprowadzenie do optymalizacji zapytań

W tym rozdziale:

- ▶ **Architektura**
- ▶ **Plany wykonania**
- ▶ SET STATISTICS TIME i SET STATISTICS IO
- ▶ **Podsumowanie**





Wszyscy to przeżyliśmy: nagle dostajesz telefon z informacją o awarii aplikacji i prośbą o pilne przyłączenie się do konferencji. Po połączeniu dowiadujesz się, że aplikacja jest tak wolna, iż firma nie jest w stanie spełniać celów biznesowych, traci pieniądze i być może także klientów. Zazwyczaj też nikt nie jest w stanie zapewnić żadnych dodatkowych informacji, które mogłyby pomóc w zlokalizowaniu problemu. Co zatem powinieneś zrobić? Gdzie zacząć? A po zlokalizowaniu i naprawieniu problemu co zrobić, aby taka sytuacja nie powtórzyła się w przyszłości?

Chociaż awaria może powstać z wielu różnych powodów, włączając w to problemy ze sprzętem i z systemem operacyjnym, jako specjalista baz danych powinieneś być w stanie odpowiednio dostosować i zoptymalizować swoje bazy tak, abyś mógł szybko zlokalizować ewentualne problemy. Ta książka zapewni Ci wiedzę i narzędzia do tego potrzebne. Skupiając się na wydajności bazy SQL Servera, a w szczególności na optymalizacji i dostosowaniu zapytań, książka ta pomoże Ci, po pierwsze, dzięki optymalizacji bazy danych uniknąć problemów z wydajnością, a po drugie, szybko znaleźć i naprawić problemy, które mimo wszystko mogą wystąpić.

Jednym z najlepszych sposobów na poprawienie wydajności baz danych jest nie tylko praca z technologią, ale również zrozumienie, jak działa technologia, co można dzięki niej uzyskać, jak najlepiej ją wykorzystać, a także jakie są jej ograniczenia. Najważniejszym składnikiem bazy SQL Servera wpływającym na wydajność zapytań jest procesor zapytań, który składa się z optymalizatora zapytań i silnika wykonującego. Mając idealny optymalizator zapytań, mógłbyś po prostu przesłać dowolne zapytanie i otrzymać za każdym razem idealny plan wykonania. Idealny silnik wykonujący pozwalałby natomiast wykonać każde zapytanie w ciągu kilku milisekund. W rzeczywistości jednak optymalizacja zapytań to bardzo złożony problem, a żaden optymalizator nie znajdzie idealnego planu za każdym razem — przynajmniej w rozsądnym czasie. W przypadku rozbudowanych zapytań optymalizator zapytań może przeanalizować tylko ograniczoną liczbę planów wykonania. Nawet gdyby optymalizator zapytań mógł przeanalizować wszystkie możliwe rozwiązania, kolejnym problemem byłoby podjęcie decyzji, który plan wybrać. Który będzie najbardziej wydajny? Wybór planu wiązałby się z oszacowaniem kosztu każdego z rozwiązań, co również jest bardzo skomplikowanym zadaniem.

Nie zrozum mnie źle: optymalizator zapytań SQL Servera sprawuje się naprawdę świetnie i prawie za każdym razem wybiera dobry plan wykonania. Musisz jednak zrozumieć, jakie informacje należy przekazać do optymalizatora zapytań, aby mógł dobrze wykonać swoją pracę — może to wiązać się z koniecznością zapewnienia odpowiednich indeksów lub statystyk, a także z koniecznością zdefiniowania odpowiednich więzów integralności i dobrego projektu bazy danych. SQL Server zawiera nawet narzędzia, które mogą pomóc Ci w niektórych z tych obszarów, m.in. Database Engine Tuning Advisor (DTA) czy funkcjonalności automatycznego tworzenia i aktu-

alizowania statystyk. Możesz jednak zrobić więcej, aby poprawić wydajność swoich baz, szczególnie jeżeli budujesz wysoko wydajne aplikacje. Musisz też zrozumieć przypadki, w których optymalizator zapytań może nie zwrócić dobrych wyników, i dowiedzieć się, co możesz wtedy zrobić.

Abyś więc mógł lepiej zrozumieć technologię, ten rozdział rozpoczyna się od opisu działania optymalizatora zapytań bazy SQL Servera i od wprowadzenia koncepcji dokładniej omawianych w dalszej części książki. Wyjaśniam cel istnienia zarówno optymalizatora zapytań, jak i silnika wykonującego i ich maksymalnej interakcji z pamięcią podręczną planów zapytania. W dalszej części pokazuję, jak pracować z planami wykonania, które są podstawowym narzędziem podczas pracy z procesorem zapytań.

Architektura

W sercu bazy danych SQL Servera znajdują się dwa główne komponenty: silnik przechowywania i silnik relacyjny, nazywany również procesorem zapytań. Silnik przechowywania jest odpowiedzialny za odczyt danych pomiędzy dyskiem a pamięcią w sposób optymalizujący współbieżność i pozwalający zachować integralność danych. Procesor zapytań, jak sugeruje nazwa, przyjmuje zapytania kierowane do serwera, buduje plan ich optymalnego wykonania, a następnie wykonuje go i dostarcza dane wynikowe.

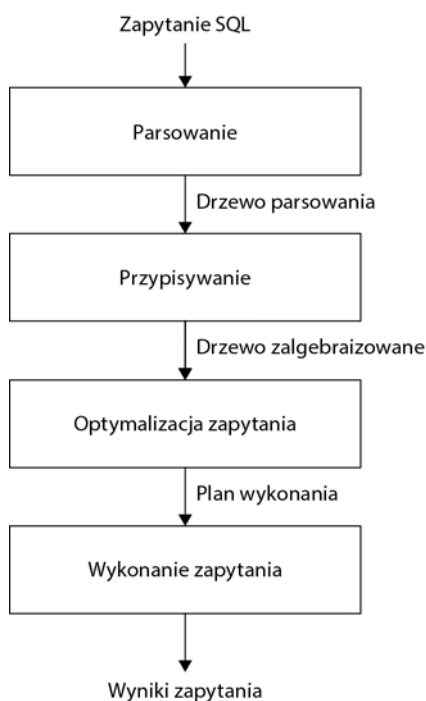
Zapytania są przekazywane do SQL Servera za pomocą języka SQL (lub T-SQL, który jest rozszerzeniem języka SQL wykorzystywanym w Microsoft SQL Serverze). Ponieważ SQL jest językiem deklaratywnym wysokiego poziomu, definiuje tylko, jakie dane pobrać z bazy danych, nie definiuje natomiast kroków potrzebnych do ich pobrania ani algorytmów przetwarzania żądania. Dlatego, dla każdego odebranego zapytania, pierwszym krokiem wykonywanym przez procesor zapytań jest jak najszybsze określenie planu zapytania, który opisuje najlepszy możliwy sposób (lub co najmniej wydajny sposób) na wykonanie danego zapytania. Jego drugim zadaniem jest wykonanie zapytania zgodnie z planem. Każde z tych zadań jest powierzane odrębnemu komponentowi wewnątrz procesora zapytań; optymalizator zapytań opracowuje plan i przekazuje go do silnika wykonującego, który wykona zapytanie i pobierze rezultaty z bazy danych.

Optymalizator zapytań bazy SQL Servera działa na podstawie kosztów. Dla danego zapytania analizuje kilka potencjalnych planów zapytania, szacuje koszt każdego z nich i wybiera plan o najniższym koszcie. W rzeczy samej, biorąc pod uwagę fakt, że optymalizator nie może przeanalizować wszystkich możliwych planów wykonania zapytania, musi znaleźć równowagę pomiędzy czasem optymalizacji i jakością wybranego planu.

W związku z tym jest to komponent serwera, który ma największy wpływ na wydajność bazy. W końcu wybór dobrego lub złego planu zapytania może stanowić różnicę

pomiędzy wykonaniem zapytania w ciągu milisekund a wykonaniem go w ciągu minut lub nawet godzin. Naturalnie, lepsze zrozumienie sposobu działania optymalizatora zapytań może pomóc zarówno administratorom baz, jak i programistom w pisaniu lepszych zapytań i w zapewnianiu optymalizatorowi informacji potrzebnych do jego jak najlepszego działania. Ta książka pokaże Ci, jak wykorzystać nowo poznaną wiedzę na temat architektury optymalizatora zapytań, a ponadto przekaze Ci wiedzę i narzędzia do radzenia sobie z sytuacjami, w których optymalizator nie produkuje dobrego planu.

Aby dotrzeć do optymalnego planu zapytania, procesor zapytań wykonuje kilka kroków. Cały proces przetwarzania zapytania został przedstawiony na rysunku 1.1.



Rysunek 1.1. Proces przetwarzania zapytania

Procesowi temu dokładniej przyjrzymy się w rozdziale 3., ale teraz omówię pokrótce poszczególne kroki:

- 1. Parsowanie i przypisywanie.** Zapytanie jest parsowane i przypisywane. Zakładając, że zapytanie jest poprawne, wynikiem tej fazy jest drzewo logiczne, którego każdy węzeł reprezentuje operację logiczną, jaką musi wykonać zapytanie, taką jak na przykład czytanie tabeli lub wykonanie złączenia.

2. **Optymalizacja zapytania.** Drzewo logiczne jest wykorzystywane w procesie optymalizacji zapytania, który, ogólnie rzecz ujmując, składa się z następujących kroków:
 - ▶ **Generowanie możliwych planów zapytania.** Korzystając z drzewa logicznego, optymalizator określa kilka możliwych sposobów wykonania zapytania (czyli możliwe plany wykonania). Plan wykonania to, ogólnie mówiąc, zestaw fizycznych operacji (takich jak *Index Seek* [przeszukanie indeksu] lub *Nested Loops Join* [złączenie z zagnieżdżoną pętlą]), które mogą zostać wykonane w celu osiągnięcia wymaganego rezultatu opisanego w drzewie logicznym.
 - ▶ **Określenie kosztu każdego z planów.** Chociaż optymalizator nie generuje wszystkich możliwych planów zapytania, określa koszt zasobów i czasu każdego z wygenerowanych planów. Plan, którego szacowany koszt będzie najniższy, zostanie wybrany i przekazany dalej do silnika wykonującego.
3. **Wykonywanie zapytań i przechowywanie planów.** Zapytanie jest wykonywane przez silnik wykonujący zgodnie z wybranym planem; plan może być przechowywany w pamięci podręcznej planów.

Parsowanie i przypisywanie

Parsowanie i przypisywanie to pierwsze operacje wykonywane po przesłaniu zapytania do instancji SQL Servera. Parsowanie pozwala upewnić się, czy zapytanie T-SQL ma poprawną składnię, i przekształca je w drzewo, a dokładniej w drzewo logicznych operatorów reprezentujących kroki wysokiego poziomu prowadzące do wykonania zapytania. Na początku te operatory będą blisko związane z pierwotną składnią zapytania i będą zawierały takie operacje jak „pobierz dane z tabeli Klient”, „pobierz dane z tabeli Kontakt”, „wykonaj złączenie wewnętrzne” i tak dalej. W trakcie procesu optymalizacji będą wykorzystywane różne drzewa reprezentujące zapytanie, a drzewo to będzie miało różne nazwy, dopóki nie zostanie wykorzystane do inicjalizacji struktury Memo.

Przypisywanie jest związane z rozwiązywaniem nazw. Podczas operacji przypisywania SQL Server upewnia się, czy wszystkie nazwy obiektów istnieją, i przypisuje każdą tabelę i kolumnę na drzewie parsowania do powiązanych obiektów w katalogu systemowym. Wynik tego procesu nazywany jest *drzewem zalgabraizowanym*. Drzewo to jest następnie przesyłane do optymalizatora zapytań.

Optymalizacja zapytań

Kolejny krok to proces optymalizacji, który jest w zasadzie generowaniem potencjalnych planów wykonywania i wyborem najlepszego z nich na podstawie szacowanych kosztów ich wykonania. Jak już wspomniałem, SQL Server wykorzystuje model szacowania kosztów do określenia kosztu wykonania każdego z planów.

Na proces optymalizacji zapytań można również patrzeć jak na proces mapowania logicznych operacji zapytania wyrażonych w pierwotnym drzewie na operacje fizyczne, które będą mogły zostać wykorzystane przez silnik wykonujący. W planach wykonania tworzonych przez optymalizator zapytań jest w zasadzie implementowana funkcjonalność silnika wykonywania; to znaczy silnik wykonywania implementuje pewną liczbę algorytmów, a optymalizator podczas tworzenia planu wykonania wybiera spośród nich. Robi to, tłumacząc pierwotne operacje logiczne na operacje fizyczne, które silnik wykonujący będzie w stanie wykonać. Plany wykonania pokazują zarówno operacje logiczne, jak i fizyczne dla każdego operatora. Te same logiczne operacje, takie jak sortowanie, przekładają się na te same operacje fizyczne, niektóre jednak mapują się na więcej niż jedną możliwą operację fizyczną. Na przykład logiczne złączenie może być mapowane na kilka fizycznych operatorów: *Nested Loops Join*, *Merge Join* (złączenie ze scalaniem) lub *Hash Join* (złączenie haszowe). Nie jest to więc proces mapowania jeden do jednego i wiąże się z bardziej skomplikowanymi działaniami, które dokładniej omówię w rozdziale 3.

Produktem końcowym procesu optymalizacji zapytania jest plan wykonania — drzewo składające się z fizycznych operatorów, które zawierają algorytmy wykonywane przez silnik wykonujący w celu pobrania żądanych wyników z bazy danych.

Generowanie możliwych planów zapytań

Podstawowym celem istnienia optymalizatora zapytań jest odnalezienie wydajnego planu wykonania zapytania. Nawet dla relatywnie prostych zapytań może istnieć ogromna liczba różnych sposobów na uzyskanie dostępu do danych w celu otrzymania tych samych rezultatów. W związku z tym optymalizator musi wybrać najlepszy plan z puli, która może być bardzo duża, a podjęcie dobrej decyzji jest ważne, ponieważ czas potrzebny na zwrócenie wyniku użytkownikowi może być bardzo różny w zależności od wybranego planu.

Zadaniem optymalizatora zapytań jest stworzenie i sprawdzenie największej możliwej liczby planów w ramach pewnych kryteriów, aby znaleźć wystarczająco dobry plan, który może, ale nie musi, być planem optymalnym. Definiujemy obszar poszukiwań dla danego zapytania jako zestaw wszystkich możliwych planów zapytania, a każdy możliwy plan zwraca te same wyniki. Teoretycznie, aby znaleźć optymalny plan wykonania dla zapytania, optymalizator kosztowy powinien znaleźć wszystkie możliwe plany istniejące w danym obszarze poszukiwań i poprawnie oszacować koszty ich wszystkich. Niestety niektóre skomplikowane zapytania mogą mieć tysiące, a nawet miliony możliwych planów, i chociaż optymalizator może zazwyczaj brać pod uwagę znaczą liczbę planów, nie jest w stanie sprawdzić ich wszystkich. Gdyby to zrobić, czas potrzebny na taką operację byłby nieakceptowalnie długi i mógłby mieć znaczący wpływ na całociowy czas wykonania zapytania.

Optymalizator zapytań musi utrzymywać równowagę pomiędzy czasem optymalizacji a jakością planu. Na przykład, jeżeli optymalizator potrzebuje 1 sekundy na znalezienie wystarczająco dobrego planu, który wykonany zostanie w minutę, nie ma sensu szukać idealnego lub najbardziej optymalnego planu, skoro zajęłoby to 5 minut plus czas wykonania zapytania. Dlatego SQL Server nie wykonuje wyczerpującego wyszukiwania, lecz próbuje znaleźć wystarczająco dobry plan najszybciej jak to możliwe. Ponieważ praca optymalizatora jest ograniczona czasem, istnieje szansa, że znaleziony plan będzie planem optymalnym, ale może też być czymś tylko zbliżonym do planu optymalnego.

Aby zbadać obszar poszukiwań, optymalizator wykorzystuje reguły transformacji i heurystykę. Generowanie potencjalnych planów zapytania jest wykonywane wewnątrz optymalizatora z wykorzystaniem reguł transformacji, a wykorzystanie heurystyki ogranicza liczbę branych pod uwagę planów, tak aby czas wykonywania optymalizacji był akceptowalny. Zestaw alternatywnych planów rozpatrywanych przez optymalizator nazywany jest obszarem planów, a same plany podczas optymalizacji przechowywane są w pamięci w komponencie o nazwie Memo. Reguły transformacji, heurystykę i strukturę Memo omówię dokładnie w rozdziale 3.

Określanie kosztu każdego z planów

Wyszukanie lub ponumerowanie potencjalnych planów to tylko jedna część procesu optymalizacji. Optymalizator zapytań musi jeszcze oszacować koszt tych planów i wybrać najmniej kosztowny z nich. Aby oszacować koszt planu, musi oszacować koszt każdego fizycznego operatora w planie, korzystając z formuł kosztowych, które uwzględniają wykorzystanie zasobów takich jak I/O, CPU i pamięć. Szacunek kosztów zależny jest w dużej mierze od algorytmu wykorzystywanego przez fizyczny operator i szacowanej liczby rekordów, które będą musiały zostać przetworzone. Szacunek liczby rekordów do przetworzenia nazywa się *szacunkiem kardynalności*.

Aby pomóc w szacunku kardynalności, SQL Server wykorzystuje i przechowuje statystyki, które zawierają informacje opisujące rozkład wartości w jednej lub większej liczbie kolumn tabeli. Kiedy koszt każdego z operatorów zostanie oszacowany z wykorzystaniem szacunku kardynalności i wymagań dotyczących zasobów, optymalizator doda do siebie wszystkie te koszty, w wyniku czego otrzyma szacowany koszt planu. Nie będę tutaj zagłębiał się w szczegóły, statystyki omówię dokładniej w rozdziale 6.

Wykonywanie zapytań i przechowywanie planów

Kiedy zapytanie zostało zoptymalizowane, plan wynikowy jest wykorzystywany przez silnik wykonujący do pobrania żądanych danych. Wygenerowany plan zapytania może być przechowywany w pamięci w magazynie planów, dzięki czemu będzie mógł zostać powtórnie wykorzystany, jeżeli to samo zapytanie zostanie wykonane powtórnie.

SQL Server ma pulę pamięci, która jest wykorzystywana do przechowywania zarówno stron danych, jak i planów zapytań. Większość tej pamięci wykorzystywana jest do przechowywania stron danych i nazywa się *pulą bufora*. Część tej pamięci zawiera plany wykonywania zapytań, które zostały zoptymalizowane przez optymalizator, i nazywa się *magazynem planów* (wcześniej nazywana była *magazynem procedur*). Procentowa ilość miejsca w pamięci przypisana do magazynu planów lub puli bufora jest dynamiczna i zależy od stanu systemu.

Przed optymalizowaniem zapytania SQL Server sprawdza, czy w magazynie planów nie został zapisany plan dla wykonanego zestawu zapytań. Optymalizacja zapytań to relatywnie kosztowna operacja, jeżeli więc w magazynie planów dostępny jest odpowiedni plan, proces optymalizacji może zostać pominięty, dzięki czemu system unika kosztów w postaci czasu optymalizacji, zasobów procesora i tak dalej. Jeżeli plan dla zestawu zapytań nie zostanie odnaleziony, zestaw jest kompilowany w celu wygenerowania planów dla wszystkich zapytań w procedurze przechowywanej, wyzwalaczu lub dynamicznym kodzie SQL. Optymalizacja rozpoczyna się od załadowania wszystkich interesujących statystyk. Następnie optymalizator zapytań sprawdza, czy statystyki są aktualne. Dla nieaktualnych statystyk, w przypadku domyślnych ustawień statystyk, przed przejściem do dalszego etapu optymalizacji statystyki zostaną zaktualizowane.

Kiedy plan zostanie znaleziony w magazynie planów lub zostanie stworzony nowy plan, nastąpi sprawdzenie, czy nie pojawiły się zmiany w strukturze lub statystykach. Zmiany w strukturze są weryfikowane pod kątem poprawności planu. Statystyki również są weryfikowane: optymalizator sprawdza, czy nie występują nowsze statystyki lub czy statystyki się nie przedawniły. Jeżeli z któregoś z tych powodów plan nie jest poprawny, wówczas jest odrzucany, a zestaw lub pojedyncze zapytanie jest kompilowane jeszcze raz. Takie kompilacje nazywane są *rekompilacjami*. Proces ten został pokazany na rysunku 1.2.

Plany mogą być usuwane z magazynu planów, jeżeli serwer wymaga zwolnienia pamięci lub kiedy zostały wykonane pewne instrukcje. Zmiana niektórych opcji konfiguracji (np. maksymalny poziom współbieżności) wyczyści cały magazyn planów. Podobnie niektóre instrukcje, takie jak operacje na bazie z wykorzystaniem pewnych opcji ALTER DATABASE, spowodują wyczyszczenie planów związanych z tą bazą.

Warto również zauważyć, że powtórne wykorzystanie istniejącego planu nie zawsze musi być dobrym rozwiązaniem dla danego zapytania oraz że w takim przypadku mogą pojawić się pewne problemy. Na przykład, w zależności od rozkładu danych w tabeli, optymalny plan zapytania może w dużej mierze zależeć od wykorzystanych parametrów. Więcej na temat tego typu problemów i samego magazynu planów dowiesz się z rozdziału 8.



Rysunek 1.2. Proces kompilacji i rekompilacji

Plany wykonania

Teraz, kiedy znamy podstawy działania procesora zapytań, czas dowiedzieć się, jak możemy wpływać na jego działanie. Podstawowym sposobem interakcji z procesorem zapytań są plany wykonania, które, jak wcześniej wspomniałem, są drzewami składającymi się z operatorów fizycznych, które z kolei zawierają algorytmy prowadzące do wygenerowania żądanych wyników z bazy danych. Biorąc pod uwagę, że w całej książce będziemy często korzystać z planów wykonania, w tym podrozdziale pokażę, jak je wyświetlać i czytać.

Możesz zażądać właściwego lub szacowanego planu zapytania dla danego zapytania, a oba typy mogą zostać wyświetlone jako grafika, tekst lub XML. Każdy z tych formatów pokazuje ten sam plan wykonania, a jedyna różnica tkwi w sposobie przedstawienia i w stopniu szczegółowości informacji w nich zawartych.

Kiedy wykonujemy żądanie wyświetlenia szacowanego planu wykonywania, zapytanie nie jest wykonywane; wyświetlany plan jest tym, który najprawdopodobniej zostałby wykorzystany, gdyby zapytanie zostało wykonane (należy jednak pamiętać, że rekompilacja może spowodować wygenerowanie innego planu zapytania — dokładniej omówię tę kwestię w dalszej części). Gdy jednak zażądamy właściwego planu, zapytanie musi zostać wykonane, a plan zostanie wyświetlony wraz z wynikami zapytania. Mimo wszystko wykorzystanie planu szacowanego ma kilka zalet, możemy bowiem na przykład przejrzeć plan dla zapytania, które wykonuje się bardzo długo bez konieczności jego wykonywania, lub wyświetlić plan zapytania dla operacji aktualizacji bez konieczności zmieniania bazy danych.

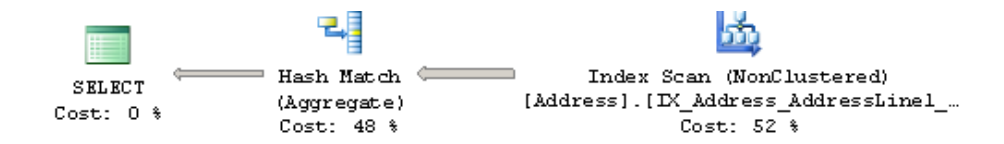
Plany graficzne

W programie SQL Server Management Studio możesz wyświetlać plany graficzne, klikając przycisk *Display Estimated Execution Plan* (wyświetlenie szacowanego planu zapytania) lub *Include Actual Execution Plan* (uwzględnienie właściwego planu zapytania) na pasku narzędzi. Kliknięcie przycisku *Display Estimated Execution Plan* wyświetli plan zapytania od razu, bez wykonywania zapytania. Aby uzyskać właściwy plan, musisz kliknąć przycisk *Include Actual Execution Plan*, a następnie wykonać zapytanie i przejść do zakładki *Execution plan* (plan wykonania).

Aby uzyskać przykład, przekopiuj poniższe zapytanie do edytora zapytań programu SQL Server Management Studio, wybierz bazę AdventureWorks2012, kliknij przycisk *Include Actual Execution Plan*, a następnie wykonaj zapytanie:

```
SELECT DISTINCT(City) FROM Person.Address
```

W panelu rezultatów wybierz zakładkę *Execution plan*. Wyświetlony zostanie plan przedstawiony na rysunku 1.3.



Rysunek 1.3. Graficzny plan wykonania

Każdy węzeł w drzewie jest reprezentowany przez ikonę przedstawiającą logiczny i fizyczny operator, taki jak *Index Scan* czy *Hash Aggregate*, zgodnie z rysunkiem 1.3. Pierwsza ikona to element języka o nazwie *Result operator* (operator rezultatu) i reprezentuje polecenie SELECT, a zazwyczaj jest też głównym elementem planu.

Operatory implementują podstawową funkcję lub operację silnika wykonującego; na przykład logiczna operacja złączenia może być implementowana przez jedną z trzech fizycznych operacji złączenia (*Nested Loops Join*, *Merge Join*, *Hash Join*). Oczywiście,



UWAGA

W tej książce znajdziesz ogromną liczbę przykładowych zapytań SQL — wszystkie zapytania tworzone były dla bazy AdventureWorks2012, chociaż rozdział 9. wykorzystuje również bazę AdventureWorksDW2012. Kod był testowany na bazie *SQL Server 2014 RTM*. Te bazy danych nie są dołączane do standardowej instalacji MS SQL Servera, ale możesz je pobrać ze strony CodePlex. Musisz pobrać rodzinę baz przykładowych dla SQL Servera 2012 (w momencie pisania tej książki przykładowe bazy dla MS SQL Servera 2014 nie istniały). Podczas instalacji możesz zainstalować wszystkie bazy lub tylko AdventureWorks2012 i AdventureWorksDW2012.

w silniku wykonującym jest zaimplementowanych znacznie więcej operatorów, a całą ich listę znajdziesz pod adresem [http://msdn.microsoft.com/en-us/library/ms191158\(v=sql.110\).aspx](http://msdn.microsoft.com/en-us/library/ms191158(v=sql.110).aspx). Ikony operatorów logicznych i fizycznych wyświetlane są w kolorze niebieskim, z wyjątkiem operatorów kursora, które są żółte, i elementów języka, które są zielone:



Operator logiczny/fizyczny



Element języka

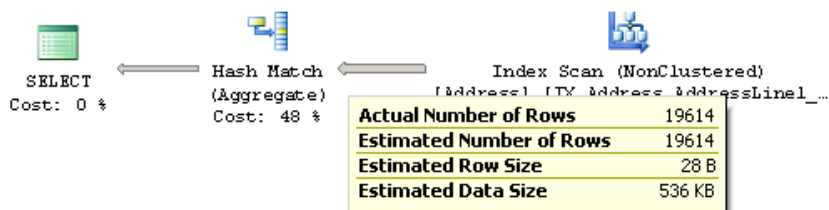


Kursor

Optymalizator zapytań buduje plan wykonania, wybierając spośród operatorów, które mogą odczytywać rekordy z bazy, jak na przykład operator *Index Scan* przedstawiony na poprzednim planie; mogą też odczytywać rekordy z innego operatora, jak na przykład operator *Hash Aggregate* odczytujący rekordy z operatora *Index Scan*.

Każdy węzeł jest związany z węzłem nadrzędnym, z którym jest połączony strzałką, a dane płyną od operatora potomnego do nadrzędnego, przy czym szerokość strzałki jest proporcjonalna do liczby rekordów. Kiedy operator wykonuje pewne funkcje na przeczytanych rekordach, rezultaty zaś są przekazywane do węzła nadrzędnego. Możesz najechać kursorem myszy, aby uzyskać więcej informacji na temat ilości danych — zostaną zaprezentowane w oknie odpowiedzi. Jeżeli, na przykład, najedziesz na strzałkę pomiędzy operacjami *Index Scan* i *Hash Aggregate* przedstawioną na rysunku 1.3, otrzymasz informacje o danych przepływających pomiędzy tymi operacjami (zobacz rysunek 1.4).

Operator *Index Scan* odczytuje 19 614 rekordów i przesyła je do operatora *Hash Aggregate*. Z kolei operator *Hash Aggregate* wykonuje pewne operacje na danych i do swojego węzła nadrzędnego przesyła 575 rekordów, co również możesz zobaczyć, najedząc kursorem myszy na strzałkę pomiędzy tymi operacjami.



Rysunek 1.4. Przepływ danych pomiędzy operatorami Index Scan i Hash Aggregate

W tym planie operacja *Index Scan* czyta wszystkie 19 614 wierszy indeksu, a operacja *Hash Aggregate* wykonuje operacje mające wybrać niepowtarzające się nazwy miast, których jest 575 — te nazwy zostaną wyświetlone w oknie rezultatów programu Management Studio. Zauważ także, że oprócz rzeczywistej liczby rekordów otrzymujesz też informacje o szacowanej liczbie wierszy, która jest szacunkiem kardynalności wykonywanym przez optymalizator dla tego operatora. Porównanie rzeczywistej i szacowanej liczby rekordów może pomóc w wykryciu błędów szacunku kardynalności, które mogą wpłynąć na jakość planów wykonywania (więcej na ten temat dowiesz się z rozdziału 6.).

Aby były w stanie wykonywać swoje zadanie, operatory fizyczne muszą implementować przynajmniej trzy poniższe metody:

- ▶ **Open()** — inicjalizuje operator, może zawierać zadania przygotowujące wymagane struktury danych.
- ▶ **GetRow()** — wykonuje żądanie rekordu z operatora.
- ▶ **Close()** — wykonuje operacje sprzątające i zamyka operator po zakończeniu jego roli.

Operator pobiera rekordy z innych operatorów za pomocą metody *GetRow()*, co oznacza również, że wykonanie planu rozpoczyna się od lewej do prawej. Ponieważ *GetRow()* tworzy tylko jeden wiersz w danym momencie, liczba rekordów wyświetlona w planie zapytania jest również liczbą wywołań metody dla danego operatora; do oznaczenia końca zestawu danych wynikowych wykorzystywane jest jedno dodatkowe wywołanie metody *GetRow()*. W poprzednim przykładzie operator *Hash Aggregate* wywołuje na operatorze *Index Scan* metodę *Open()* raz, metodę *GetRow()* 19 615 razy i raz metodę *Close()*.



UWAGA

Na razie wyjaśnimy tradycyjny tryb przetwarzania zapytań, w którym operatory przetwarzają jeden wiersz w danym momencie. Ten tryb przetwarzania był wykorzystywany we wszystkich wersjach SQL Servera od wersji 7.0. W rozdziale 9. wspomnę o nowym trybie przetwarzania partiami, wprowadzonym w SQL Serverze 2012 i wykorzystywanym przez operatory związane z indeksami magazynów kolumn.

Możesz również najechać kursorem myszy na operator, aby uzyskać o nim więcej informacji. Na przykład rysunek 1.5 przedstawia informacje o operatorze *Index Scan*; zauważ, że zawiera między innymi opis operatora i dane na temat szacowanych kosztów, takich jak koszt I/O, CPU, operatora i poddrzewa (*subtree*).

Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	19614
Actual Number of Batches	0
Estimated I/O Cost	0.158681
Estimated Operator Cost	0.180413 (52%)
Estimated Subtree Cost	0.180413
Estimated CPU Cost	0.0217324
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	19614
Estimated Row Size	28 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1
Object	
[AdventureWorks2012].[Person].[Address]. [IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode]	
Output List	
[AdventureWorks2012].[Person].[Address].City	

Rysunek 1.5. Okno z informacjami o operatorze Index Scan

Niektóre z tych właściwości omawiam w tabeli 1.1, inne wyjaśnię w dalszej części książki.



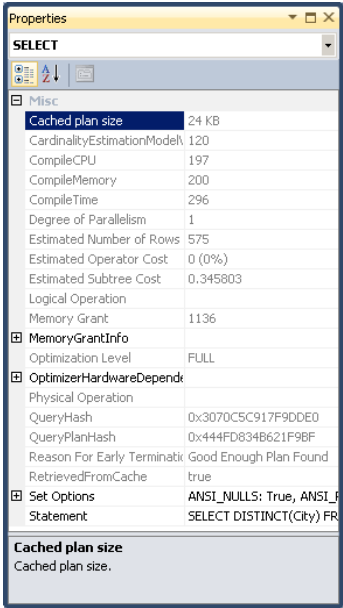
UWAGA

Warto nadmienić, że koszt wyrażony jest w wewnętrznych jednostkach, które nie powinny być utożsamiane z sekundami czy milisekundami.

Koszt każdego z operatorów jest również przedstawiany relatywnie, jako wartość procentowa całego planu (zobacz rysunek 1.3). Na przykład koszt operacji *Index Scan* wynosi 52% kosztu całego planu. Dodatkowe informacje z operatora lub całego zapytania można pozyskać za pomocą okna *Properties* (właściwości). Wybór ikony **SELECT** i otwarcie okna *Properties* z menu *View* (widok) lub naciśnięcie klawisza *F4* pokaże właściwości całego zapytania (zobacz rysunek 1.6).

Tabela 1.1. Właściwości operatorów

Właściwość	Opis
Physical Operation	Algorytm fizyczny dla węzła.
Logical Operation	Operator algebry relacyjnej reprezentowany przez węzeł.
Actual Number of Rows	Liczba rekordów wytworzonych przez operator.
Estimated I/O Cost	Szacunkowy koszt operacji I/O. Nie wszystkie operacje związane są z kosztem I/O.
Estimated Operator Cost	Koszt operacji oszacowany przez optymalizator zapytań. Jest to szacowany koszt I/O i CPU. Zawiera także koszt operacji jako procentową wartość całego kosztu zapytania wyświetlony w nawiasach.
Estimated Subtree Cost	Szacowany narastający koszt wykonania tej operacji i wszystkich operacji poprzedzających ją w tym samym poddrzewie.
Estimated CPU Cost	Szacowany koszt procesora dla tej operacji.
Estimated Number of Executions	Szacowana liczba wywołań tego operatora podczas wykonywania bieżącego zapytania.
Number of Executions	Liczba wywołań tego operatora po wykonaniu zapytania.
Estimated Number of Rows	Szacowana liczba rekordów wytworzona przez operator (szacunek kardynalności).
Estimated Row Size	Szacowany średni rozmiar rekordu przetwarzanego przez ten operator.



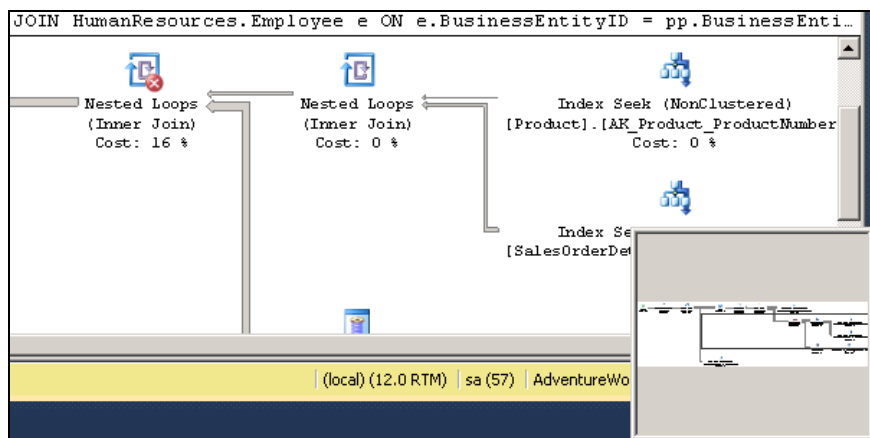
Rysunek 1.6. Okno właściwości zapytania

Tabela 1.2 zawiera większość właściwości z rysunku 1.6. W zależności od zapytania mogą pojawić się inne, opcjonalne właściwości (na przykład *Parameter List* lub *Warnings*).

Tabela 1.2. Właściwości zapytań

Właściwość	Opis
<i>Cached plan size</i>	Ilość pamięci w kilobajtach w magazynie planów wykorzystana przez plan zapytania.
<i>CompileCPU</i>	Czas procesora w milisekundach wykorzystany do skompilowania zapytania.
<i>CompileMemory</i>	Pamięć w kilobajtach wykorzystana do skompilowania zapytania.
<i>CompileTime</i>	Czas w milisekundach wykorzystany do skompilowania zapytania.
<i>Degree of Parallelism</i>	Liczba wątków, które mogą zostać wykorzystane do wykonania zapytania, jeżeli procesor zapytań wybierze plan równoległy.
<i>Memory Grant</i>	Ilość pamięci w kilobajtach udzielonej do uruchomienia tego zapytania.
<i>MemoryGrantInfo</i>	Informacje dotyczące szacunku na temat udzielonej pamięci i informacje o rzeczywistej ilości udzielonej pamięci.
<i>Optimization Level</i>	Poziom optymalizacji wykorzystany do skompilowania tego zapytania. Wyświetlany jako <i>StatementOptmLevel</i> na planie w formacie XML. Zostanie dokładniej omówiony w dalszej części tego podrozdziału.
<i>OptimizerHardwareDependentProperties</i>	Właściwości uzależnione od platformy sprzętowej, które mają wpływ na szacunek kosztów (a co za tym idzie, wybór planu) z punktu widzenia optymalizatora zapytań.
<i>QueryHash</i>	Wartość hasza binarnego obliczona na zapytaniu i wykorzystywana do identyfikacji zapytań o podobnej logice.
<i>QueryPlanHash</i>	Wartość hasza binarnego obliczona na planie zapytania i wykorzystywana do identyfikacji podobnych planów.
<i>Reason For Early Termination Of Statement Optimization</i>	Na planie w formacie XML wyświetlana jako <i>StatementOptmEarlyAbortReason</i> . Zostanie dokładniej omówiona w dalszej części tego podrozdziału.
<i>RetrievedFromCache</i>	Wskazuje, czy plan został pobrany z magazynu planów.
<i>Set Options</i>	Status opcji SET mających wpływ na koszt zapytania. Na planie w formacie XML wyświetlana jako <i>StatementSetOptions</i> . Te opcje to: <code>NSI_NULLS</code> , <code>ANSI_PADDING</code> , <code>ANSI_WARNINGS</code> , <code>ARITHABORT</code> , <code>CONCAT_NULL_YIELDS_NULL</code> , <code>NUMERIC_ROUNDABORT</code> i <code>QUOTED_IDENTIFIER</code> .
<i>Statement</i>	Tekst zapytania SQL.

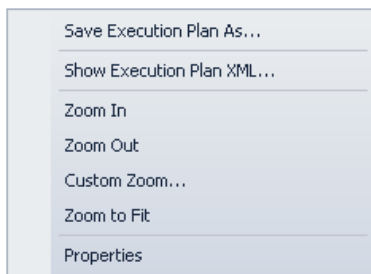
SQL Server udostępnia też funkcjonalność oddalania planu, którą możesz wykorzystać podczas nawigacji po dużych graficznych planach, które mogą nie mieścić się na ekranie. Możesz uzyskać dostęp do tego narzędzia, klikając ikonę plusa zlokalizowaną w prawym dolnym rogu zakładki planu wykonania. Przykład został przedstawiony na rysunku 1.7. Popularnym narzędziem do pracy z planami zapytań jest też SQL Sentry Plan Explorer. Możesz go pobrać za darmo pod adresem <http://sqlsentry.net/plan-explorer>.



Rysunek 1.7. Funkcjonalność oddalania planu zapytania

XML

Kiedy już wyświetliłeś plan graficzny, możesz również z łatwością wyświetlić go w formacie XML. Po prostu kliknij prawym przyciskiem myszy w dowolnym miejscu okna planu wykonywania i w menu kontekstowym wybierz opcję *Show Execution Plan XML...* (pokaż plan wykonania w formacie XML), zgodnie z rysunkiem 1.8. Wówczas zostanie otwarty edytor XML i wyświetli się plan XML. Jak widać, możesz łatwo przełączać się pomiędzy wersjami graficzną i XML.



Rysunek 1.8. Menu kontekstowe dla planu wykonania

Jeżeli będzie to potrzebne, możesz zapisywać plany graficzne w pliku, wybierając opcję *Save Execution Plan As...* (zapisz plan wykonania jako) z menu kontekstowego przedstawionego na rysunku 1.8. Plan, zapisywany zazwyczaj z rozszerzeniem *.sqlplan*, to tak naprawdę dokument XML zawierający plan w formacie XML, który może zostać odczytany przez program Management Studio i przekształcony w plan graficzny. Możesz powtórnie załadować ten plik, wybierając z menu *File* (plik) opcję *Open* (otwórz), a wówczas plan zostanie otwarty w nowym oknie i będzie się zachowywał dokładnie tak jak poprzednio. Plany zapytania w formacie XML mogą również być wykorzystywane w zapytaniu z odpowiednią USEPLAN, co omówię w rozdziale 10.

Tabela 1.3 zawiera różne polecenia, jakie możesz wykorzystać do uzyskania szacowanego albo rzeczywistego planu w formacie tekstowym, graficznym lub XML.

Tabela 1.3. Polecenia pozwalające wyświetlać plany zapytań

	Szacowany plan wykonania	Rzeczywisty plan wykonania
<i>Text Plan</i>	SET SHOWPLAN_TEXT SET SHOWPLAN_ALL	SET STATISTICS PROFILE
<i>Graphic Plan</i>	Management Studio	Management Studio
<i>XML Plan</i>	SET SHOWPLAN_XML	SET STATISTICS XML



UWAGA

Jeżeli uruchomisz którąkolwiek z opcji z tabeli 1.3, korzystając z klauzuli ON, opcja ta będzie stosowana do wszystkich kolejnych instrukcji, do czasu ręcznego wyłączenia jej za pomocą klauzuli OFF.

Aby wyświetlić plan XML, możesz skorzystać z poniższych poleceń:

```
SET SHOWPLAN_XML ON
GO
SELECT DISTINCT(City) FROM Person.Address
GO
SET SHOWPLAN_XML OFF
```

To zapytanie pozwala wyświetlić wynik z pojedynczym wierszem z jedną kolumną o nazwie *Microsoft SQL Server 2005 XML Showplan* zawierający dane w formacie XML zaczynające się od:

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004 ...
```

Kliknięcie linku spowoduje wyświetlenie planu graficznego, a plan XML będzie można wyświetlić, stosując tę samą metodę co poprzednio.

Możesz przejrzeć podstawową strukturę planu XML dzięki poniższemu ćwiczeniu. Bardzo proste zapytanie stworzy podstawową strukturę XML, tutaj jednak przedstawiam zapytanie, które pozwoli wygenerować dwie dodatkowe części: brakujące indeksy i listę parametrów. Wykonaj poniższe zapytanie i otwórz plan XML:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE OrderQty = 1
```

Zwiń elementy <MissingIndexes>, <RelOp> i <ParameterList>, klikając znak minusa (–) po ich lewej stronie, dzięki czemu będziesz mógł zobaczyć całą strukturę. Powinieneś zobaczyć coś podobnego jak na rysunku 1.9.



```
<?xml version="1.0" encoding="utf-16"?>
<ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementCompId="1" StatementEstRows="68089" StatementId="1" StatementOptmLevel="FULL" Cardi
          <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI_WARNINGS="true" ARITHABORT="true" CONC
            <QueryPlan DegreeOfParallelism="1" CachedPlanSize="32" CompileTime="863" CompileCPU="862" CompileMemor
              <MissingIndexes>...</MissingIndexes>
              <MemoryGrantInfo SerialRequiredMemory="0" SerialDesiredMemory="0" />
              <OptimizerHardwareDependentProperties EstimatedAvailableMemoryGrant="101808" EstimatedPagesCached="2
                <RelOp AvgRowSize="112" EstimateCPU="0.0582322" EstimateIO="0" EstimateRebinds="0" EstimateRewinds="
                  <ParameterList>...</ParameterList>
            </QueryPlan>
          </StmtSimple>
        </Statements>
      </Batch>
    </BatchSequence>
  </ShowPlanXML>
```

Rysunek 1.9. Plan wykonania w formacie XML

Jak widzisz, główne komponenty planu XML to elementy <StmtSimple>, <StatementSetOptions> i <QueryPlan>. Te trzy elementy zawierają po kilka atrybutów, niektóre z nich już objaśniłem, kiedy omawiałem plany graficzne. Ponadto element <QueryPlan> zawiera również inne elementy, takie jak <MissingIndexes>, <MemoryGrantInfo>, <OptimizerHardwareDependentProperties>, <RelOp>, <ParameterList>, a także takie, które nie zostały pokazane na rysunku 1.9 (na przykład <Warnings>), a które omówię w dalszej części tego podrozdziału. Na przykład element <StmtSimple> wygląda tak:

```
<StmtSimple StatementCompId="1" StatementEstRows="68089" StatementId="1"
  <StatementOptmLevel="FULL" CardinalityEstimationModelVersion="70"
  <StatementSubTreeCost="1.13478" StatementText="SELECT * FROM [Sales].[SalesOrderDetail]
  <WHERE [OrderQty]=@1" StatementType="SELECT" QueryHash="0x42CFD97ABC9592DD"
  <QueryPlanHash="0xC5F6C30459CD7C41" RetrievedFromCache="false">
```

A element <QueryPlan> tak:

```
<QueryPlan DegreeOfParallelism="1" CachedPlanSize="32" CompileTime="3" CompileCPU="3"
  <CompileMemory="264">
```

Jak wspomniałem, atrybuty tych i innych elementów omówiłem już w podrozdziale dotyczącym planów graficznych. Inne wyjaśnię w dalszej części tego podrozdziału lub w innych podrozdziałach tej książki.

Plany tekstowe

Jak widzisz w tabeli 1.3, dwa polecenia pozwalają otrzymać szacowany plan tekstowy: SET SHOWPLAN_TEXT i SET SHOWPLAN_ALL. Oba polecenia powodują wyświetlenie szacowanego planu tekstowego, ale SET SHOWPLAN_ALL pokazuje pewne dodatkowe informacje, zawierające szacowaną liczbę wierszy, szacowany koszt CPU, szacowany koszt I/O i szacowany koszt operatora. Najnowsze wersje dokumentacji Books Online, włączając w to te dotyczące SQL Servera 2014, wskazują jednak, że wszystkie formy planu tekstowego zostaną wycofane w przyszłych wersjach SQL Servera i dlatego zalecanym sposobem wyświetlania planu jest XML.

Do wyświetlenia planu tekstowego możesz wykorzystać poniższy kod:

```
SET SHOWPLAN_TEXT ON
GO
SELECT DISTINCT(City) FROM Person.Address
GO
SET SHOWPLAN_TEXT OFF
GO
```

Kod ten spowoduje wyświetlenie dwóch zestawów wyników: pierwszy będzie zawierał tekst zapytania T-SQL, drugi natomiast będzie zawierał poniższy plan tekstowy (wyedytowany tak, aby zmieścił się na stronie), pokazujący te same operatory *Hash Aggregate* i *Index Scan*, które widniały na planie graficznym z rysunku 1.3:

```
--Hash Match(Aggregate, HASH:([Person].[Address].[City]), RESIDUAL ...
--Index Scan(OBJECT:([AdventureWorks].[Person].[Address]). [IX_Address ...
```

SET SHOWPLAN_ALL i SET STATISTICS PROFILE mogą zwracać więcej informacji niż SET SHOWPLAN_TEXT. Możesz także, zgodnie z tabelą 1.3, skorzystać z polecenia SET SHOWPLAN_ALL do wyświetlenia samego planu i SET STATISTICS PROFILE do wykonania zapytania. Uruchom poniższy przykład:

```
SET SHOWPLAN_ALL ON
GO
SELECT DISTINCT(City) FROM Person.Address
GO
SET SHOWPLAN_ALL OFF
GO
```

Wynik został pokazany na rysunku 1.10.

	StmtText	StmtId	NodeId	Parent	PhysicalOp	LogicalOp
1	SELECT DISTINCT(City) FROM Person.Address	1	1	0	NULL	NULL
2	I-Hash Match(Aggregate, HASH:([AdventureWorks...	1	2	1	Hash Match	Aggregate
3	I-Index Scan(OBJECT:([AdventureWorks2012][...	1	3	2	Index Scan	Index Scan

Rysunek 1.10. Wynik operacji SET SHOWPLAN_ALL

Ponieważ SET STATISTICS PROFILE powoduje wykonanie zapytania, pozwala w łatwy sposób wyszukać problemy w szacowaniu kardynalności. Porównanie wielu operatorów jednocześnie jest bardzo proste, natomiast na planie graficznym lub XML może to być bardziej skomplikowane. Teraz uruchom poniższy kod:

```
SET STATISTICS PROFILE ON
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE OrderQty * UnitPrice > 25000
GO
SET STATISTICS PROFILE OFF
GO
```

Wynik został pokazany na rysunku 1.11.

	Rows	EstimateRows	Executes	StmtText
1	5	36395.1	1	SELECT * FROM [Sales].[SalesOrderDetail] WHERE [OrderQty]*[UnitPrice]>@1
2	5	36395.1	1	I-Filter(WHERE:([Expr1003]>(\$25000.0000)))
3	0	121317	0	I-Compute Scalar(DEFINE:([AdventureWorks2012].[Sales].[SalesOrderDetail].[Li...
4	0	121317	0	I-Compute Scalar(DEFINE:([AdventureWorks2012].[Sales].[SalesOrderDetail]....
5	121317	121317	1	I-Clustered Index Scan(OBJECT:([AdventureWorks2012].[Sales].[SalesOr...

Rysunek 1.11. Wynik operacji SET STATISTICS PROFILE

Zauważ, że kolumna *EstimateRows* została ręcznie przeniesiona w Management Studio obok kolumny *Rows*, co pozwala na ich łatwe porównanie. W tym konkretnym przykładzie możesz zobaczyć dużą różnicę w szacunku kardynalności dla operacji *Filter* (filtr) — oszacowane zostało 36 395,1, podczas gdy rzeczywiście w operacji uczestniczyło tylko 5 wierszy.

Dodatkowe właściwości planów

Interesującym sposobem na naukę komponentów planów wykonania, także tych z przyszłych wersji SQL Servera, jest oglądanie schematu showplan. Plany XML muszą stosować się do opublikowanego schematu XSD. Aktualna i starsze wersje są dostępne pod adresem <http://schemas.microsoft.com/sqlserver/2004/07/showplan/>, który znajdziesz także na początku każdego planu w formacie XML. Aktualnie uruchomienie tego adresu w przeglądarce wyświetli linki do schematów dla wersji SQL Server 2014 RTM (jako *Current version* — aktualna wersja), SQL Server 2012 RTM, SQL Server 2008 RTM, SQL Server 2005 SP2 i SQL Server 2005 RTM.

Omówienie wszystkich elementów i atrybutów planu wykonywania zajęłoby wiele stron, dlatego omówię tylko kilka bardziej interesujących. Operatory wykorzystywane w planach wykonywania omówię dokładniej w rozdziale 4. Rozpocznijmy od atrybutów StatementOptmLevel, StatementOptmEarlyAbortReason i CardinalityEstimationModelVersion elementu <StmtSimple>.

Chociaż atrybuty te dotyczą zagadnień omawianych dokładniej w dalszej części książki, warto przedstawić je już teraz. `StatementOptmLevel` dotyczy poziomu optymalizacji zapytania i może przyjmować wartości `TRIVIAL` (trywialna) lub `FULL` (pełna). Proces optymalizacji dla prostych zapytań niewymagających szacowania kosztów może być kosztowny, aby więc uniknąć tych kosztów dla prostych zapytań, SQL Server wykorzystuje optymalizację trywialną. Jeżeli zapytanie nie kwalifikuje się do optymalizacji trywialnej, wykonana zostanie pełna optymalizacja. Na przykład w SQL Serverze 2014 następujące zapytanie będzie podległo optymalizacji trywialnej:

```
SELECT * FROM Sales.SalesOrderHeader  
WHERE SalesOrderID = 43666
```

Możesz wykorzystać nieudokumentowaną (a tym samym niewspieraną) flagę 8757 do przetestowania zachowania z wyłączoną optymalizacją trywialną:

```
SELECT * FROM Sales.SalesOrderHeader  
WHERE SalesOrderID = 43666  
OPTION (QUERYTRACEON 8757)
```

Podpowiedź zapytania `QUERYTRACEON` jest wykorzystywana do przypisywania flag do zapytań. Po uruchomieniu poprzedniego zapytania SQL Server wykona pełną optymalizację, co możesz potwierdzić, sprawdzając wartość właściwości `StatementOptmLevel` w planie zapytania. Zauważ, że chociaż podpowiedź zapytania `QUERYTRACEON` jest dobrze znana, aktualnie wspierana jest tylko w niewielkim stopniu. W chwili pisania tej książki `QUERYTRACEON` jest wspierana tylko podczas stosowania flag udokumentowanych w artykule dostępnym pod adresem <http://support.microsoft.com/kb/2801413>.



UWAGA

W tej książce znajdziesz wiele nieudokumentowanych i niewspieranych funkcjonalności. Możesz korzystać z nich w środowisku testowym, do szukania problemów lub do nauki nowej technologii. Nie są jednak przeznaczone do wykorzystania w środowiskach produkcyjnych i nie są wspierane przez Microsoft. Będę wskazywał, które polecenia lub flagi są nieudokumentowane lub niewspierane.

Z drugiej strony, atrybut `StatementOptmEarlyAbortReason` (powód wczesnego zakończenia optymalizacji) może przyjmować wartości `GoodEnoughPlanFound` (znaleziono wystarczająco dobry plan), `Timeout` (przekroczenie dozwolonego czasu) i `MemoryLimitExceeded` (przekroczenie limitu pamięci) i pojawia się tylko wtedy, kiedy optymalizacja została zakończona przedwcześnie (w starszych wersjach SQL, aby zobaczyć tę informację, trzeba było skorzystać z nieudokumentowanej flagi 8675). Ponieważ celem optymalizatora jest stworzenie wystarczająco dobrego planu najszybciej jak to możliwe, optymalizator na początku działania wylicza dwie wartości. Pierwsza z nich to koszt wystarczająco dobrego zapytania, a druga to maksymalny czas, który może

być wykorzystany do stworzenia planu. Jeżeli podczas procesu optymalizacji zostanie znaleziony plan o koszcie wykonania niższym niż wcześniej obliczona wartość graniczna, optymalizacja jest przerywana, a znaleziony plan zostanie zwrócony z wartością `GoodEnoughPlanFound`. Jeśli jednak proces optymalizacji przekroczy pierwotnie założony czas, optymalizacja również zostanie przerwana i zwrócony zostanie najlepszy do tej pory plan z właściwością `StatementOptmEarlyAbortReason` o wartości `TimeOut`. Wartości `GoodEnoughPlanFound` i `TimeOut` nie oznaczają problemu — we wszystkich trzech (włączając w to `MemoryLimitExceeded`) przypadkach plan zapytania będzie poprawny. Jednak w przypadku `MemoryLimitExceeded` plan może nie być optymalny. Wówczas być może będziesz musiał uprościć zapytanie lub zwiększyć ilość dostępnej pamięci. Te i inne szczegóły procesu optymalizacji zapytania omówię w rozdziale 3.

Na przykład, nawet jeżeli poniższe zapytanie wykonuje złączenie czterech tabel i wymaga sortowania, i tak kończy się przedwcześnie z komunikatem o odnalezieniu wystarczająco dobrego planu:

```
SELECT pm.ProductModelID, pm.Name, Description, pl.CultureID, cl.Name AS Language
FROM Production.ProductModel AS pm
    JOIN Production.ProductModelProductDescriptionCulture AS pl
        ON pm.ProductModelID = pl.ProductModelID
    JOIN Production.Culture AS cl
        ON cl.CultureID = pl.CultureID
    JOIN Production.ProductDescription AS pd
        ON pd.ProductDescriptionID = pl.ProductDescriptionID
ORDER BY pm.ProductModelID
```

Atrybut `CardinalityEstimationModelVersion` (wersja modelu szacowania kardynalności) odnosi się do wersji modelu szacowania kardynalności wykorzystanej w optymalizatorze zapytań. SQL Server 2014 wykorzystuje nowy sposób szacowania kardynalności, masz jednak możliwość korzystania również ze starego sposobu, poprzez zmianę trybu kompatybilności bazy danych lub dzięki flagom 2312 i 9481. Więcej szczegółów dotyczących obu modeli szacowania kardynalności znajdziesz w rozdziale 6.

Opcjonalny atrybut `NonParallelPlanReason` (powód wykorzystania planu nierównoległego) elementu `QueryPlan`, który został wprowadzony w SQL Serverze 2012, zawiera informację, dlaczego plan równoległy nie może zostać wybrany dla zapytania. Chociaż lista możliwych wartości nie jest udokumentowana, najczęściej spotykane są poniższe:

```
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43666
OPTION (MAXDOP 1)
```

Ponieważ skorzystaliśmy z `MAXDOP 1`, właściwość przyjmie wartość:

```
NonParallelPlanReason="MaxDOPSetToOne"
```

Wykorzystanie tej funkcji:

```
SELECT CustomerID, ('AW' + dbo.ufnLeadingZeros(CustomerID))
AS GenerateAccountNumber
```

```
FROM Sales.Customer
ORDER BY CustomerID;
```

wygeneruje następującą wartość:

```
NonParallelPlanReason="CouldNotGenerateValidParallelPlan"
```

Jeżeli w systemie z jednym procesorem spróbujesz wykonać poniższe zapytanie:

```
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43666
OPTION (MAXDOP 8)
```

otrzymasz taki wynik:

```
NonParallelPlanReason="EstimatedDOPIsOne"
```

W schemacie XSD znajduje się wprowadzony również w SQL Serverze 2012 element `OptimizerHardwareDependentProperties` (właściwości optymalizatora uzależnione od sprzętu), zawierający właściwości uzależnione od sprzętu, które mogą mieć wpływ na wybór planu. Element ten ma poniższe udokumentowane właściwości:

- ▶ **EstimatedAvailableMemoryGrant** — szacowana ilość pamięci (w kB), która będzie dostępna w momencie wykonania zapytania.
- ▶ **EstimatedPagesCached** — szacowana liczba stron danych, które pozostaną w puli bufora, jeżeli zapytanie będzie musiało powtórnie czytać.
- ▶ **EstimatedAvailableDegreeOfParallelism** — szacowana liczba procesorów, które będą mogły być wykorzystane do wykonania zapytania, gdyby optymalizator wybrał plan równoległy.

Na przykład zapytanie:

```
SELECT DISTINCT(CustomerID)
FROM Sales.SalesOrderHeader
```

zwróci poniższą wartość:

```
<OptimizerHardwareDependentProperties EstimatedAvailableMemoryGrant="101808"
EstimatedPagesCached="8877" EstimatedAvailableDegreeOfParallelism="2" />
```

Ostrzeżenia w planach wykonania

Plany wykonania mogą również zawierać komunikaty ostrzeżeń. Plany zawierające ostrzeżenia powinny być szczegółowo analizowane, ponieważ ich wystąpienie może wskazywać na to, że optymalizator wybiera mniej optymalne plany. Przed SQL Serverem 2012 występowały tylko ostrzeżenia `ColumnsWithNoStatistics` (kolumny bez statystyk) i `NoJoinPredicate` (brak predykatów złączenia). Schemat XSD z wersji SQL Server 2012 dodaje sześć nowych ostrzeżeń iteratora i zapytania:

- ▶ SpillToTempDb (przekazywanie danych do bazy tymczasowej),
- ▶ Wait (oczekiwanie),
- ▶ PlanAffectingConvert (konwersje wpływające na plan),
- ▶ SpatialGuess (przypuszczenie przestrzenne),
- ▶ UnmatchedIndexes (niepasujące indeksy),
- ▶ FullUpdateForOnlineIndexBuild (pełna aktualizacja dla indeksu online).

Omówmy kilka z nich.

ColumnsWithNoStatistics

To ostrzeżenie oznacza, że optymalizator próbował wykorzystać statystyki, ale nie były one dostępne. Jak tłumaczyłem we wcześniejszej części rozdziału, optymalizator polega na statystykach w celu stworzenia optymalnego planu. Aby zasymulować wystąpienie ostrzeżenia, wykonaj poniższe polecenia.

Wykonaj następujące polecenie, aby usunąć (jeżeli istnieją) statystyki dla kolumny VacationHours:

```
DROP STATISTICS HumanResources.Employee._WA_Sys_0000000C_49C3F6B7
```

Następnie tymczasowo wyłącz automatyczne tworzenie statystyk na poziomie bazy danych:

```
ALTER DATABASE AdventureWorks2012 SET AUTO_CREATE_STATISTICS OFF
```

Wreszcie uruchom następujące zapytanie:

```
SELECT * FROM HumanResources.Employee
WHERE VacationHours = 48
```

Otrzymasz plan pokazany na rysunku 1.12.



Rysunek 1.12. Plan z ostrzeżeniem ColumnsWithNoStatistics

Zwróć uwagę na ostrzeżenie (symbol ze znakiem wykrzyknika) dla operatora *Clustered Index Scan*. Jeżeli spojrzysz na właściwości tego operatora, zobaczysz wartość: *Columns With No Statistics: [AdventureWorks2012].[HumanResources].[Employee].VacationHours*.

Nie zapomnij o powtórnym włączeniu automatycznego tworzenia statystyk (uruchom poniższe zapytanie). Nie trzeba tworzyć usuniętego wcześniej obiektu statystyk, ponieważ jeżeli będzie potrzebny, może zostać utworzony automatycznie.

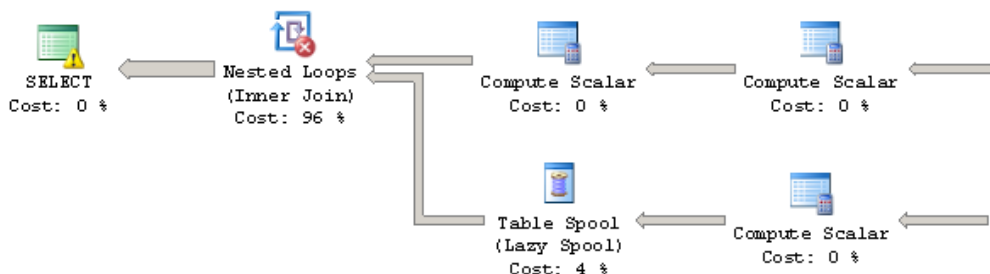
```
ALTER DATABASE AdventureWorks2012 SET AUTO_CREATE_STATISTICS ON
```


NoJoinPredicate

Wykorzystanie składni połączeń ze specyfikacji ANSI SQL-89 związane jest z ryzykiem nieumyślnego pominięcia predykatów złączenia, co z kolei wiąże się z wystąpieniem ostrzeżenia NoJoinPredicate. Załóżmy, że chcesz uruchomić poniższe zapytanie, ale zapomnisz o klauzuli WHERE:

```
SELECT * FROM Sales.SalesOrderHeader soh, Sales.SalesOrderDetail sod
WHERE soh.SalesOrderID = sod.SalesOrderID
```

Pierwszą wskazówką, że wystąpił problem, może być długi czas wykonywania zapytania, nawet dla małych tabel. Później zobaczysz także, że wynik zwracany przez zapytanie jest bardzo obszerny. Czasami dobrym sposobem na szukanie problemów z długo wykonującymi się zapytaniami jest zatrzymanie ich i zażądanie szacowanego planu zapytania. Jeżeli nie dołączysz predykatu złączenia (w klauzuli WHERE), otrzymasz plan zgodny z rysunkiem 1.13.



Rysunek 1.13. Plan z ostrzeżeniem NoJoinPredicate

Tym razem ostrzeżenie NoJoinPredicate znajduje się na operacji *Nested Loops Join* i ma inną ikonę. Zauważ, że jeżeli korzystasz ze składni złączeń ze specyfikacji ANSI SQL-92, nie możesz ominąć predykatów złączenia, ponieważ otrzymasz błąd, dlatego właśnie ta składnia jest zalecana. Na przykład ominięcie predykatu złączenia z poniższego zapytania zwróci błąd składni:

```
SELECT * FROM Sales.SalesOrderHeader soh JOIN Sales.SalesOrderDetail sod
-- ON soh.SalesOrderID = sod.SalesOrderID
```



UWAGA

Jeżeli zajdzie taka potrzeba, możesz otrzymać po jednym wierszu dla wszystkich możliwych par wierszy z dwóch tabel — taki wynik nazywa się również *wynikiem kartezjańskim* i możesz go uzyskać, stosując składnię *CROSS JOIN*.

PlanAffectingConvert

To ostrzeżenie pokazuje, że wykonane były konwersje typów, które mogły mieć wpływ na wydajność wynikowego planu zapytania. Wykonaj poniższy przykład, w którym najpierw deklarowana jest zmienna typu nvarchar, a następnie jest ona porównywana do kolumny typu varchar, CreditCardApprovalCode:

```
DECLARE @code nvarchar(15)
SET @code = '95555Vi4081'
SELECT * FROM Sales.SalesOrderHeader
WHERE CreditCardApprovalCode = @code
```

Dla zapytania stworzony zostanie plan zgodny z rysunkiem 1.14.



Rysunek 1.14. Plan z ostrzeżeniem PlanAffectingConvert

Ikona ostrzeżenia na operacji SELECT dotyczy dwóch poniższych ostrzeżeń:

Type conversion in expression
 (CONVERT_IMPLICIT(nvarchar(15),[AdventureWorks2012].[Sales].[SalesOrderHeader].
 ↳[CreditCardApprovalCode],0)) may affect "CardinalityEstimate" in query plan choice,
 Type conversion in expression
 (CONVERT_IMPLICIT(nvarchar(15),[AdventureWorks2012].[Sales].[SalesOrderHeader].
 ↳[CreditCardApprovalCode],0)=[@code]) may affect "SeekPlan" in query plan choice

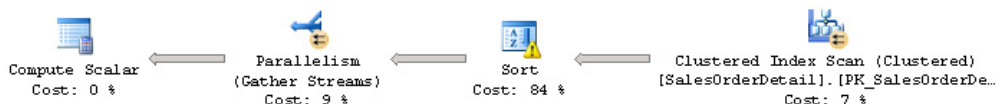
Oczywiście zalecane jest wykorzystywanie podobnych typów danych podczas porównań.

SpillToTempDb

To ostrzeżenie wskazuje, że operacja nie miała wystarczająco dużo pamięci i musiała podczas wykonywania zapytania przenieść dane na dysk, co może być problemem ze względu na dodatkowe operacje I/O. Aby zasymulować ten problem, wykonaj poniższe zapytanie:

```
SELECT * FROM Sales.SalesOrderDetail
ORDER BY UnitPrice
```

Jest to bardzo proste zapytanie, a uzyskanie ostrzeżenia jest uzależnione od ilości pamięci w systemie, być może więc będziesz musiał spróbować z większą tabelą. Wygenerowany zostanie plan jak na rysunku 1.15.



Rysunek 1.15. Plan z ostrzeżeniem SpillToTempDb

Ostrzeżenie jest pokazywane na operatorze *Sort* i zawiera komunikat *Operator used tempdb to spill data during execution with spill level 1* (operator podczas wykonywania zapytania przekazał dane do bazy tymczasowej z poziomem przekazania 1). Plan XML zawiera również to:

```
<SpillToTempDb SpillLevel="1" />
```

UnmatchedIndexes

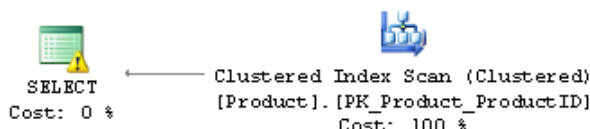
Ostrzeżenie *UnmatchedIndexes* informuje, że optymalizator nie był w stanie dopasować indeksu filtrowanego do danego zapytania (na przykład kiedy nie może zobaczyć wartości parametru). Załóżmy, że stworzyłeś poniższy indeks filtrowany:

```
CREATE INDEX IX_Color ON Production.Product(Name, ProductNumber)
WHERE Color = 'White'
```

a następnie wykonałeś następujące zapytanie:

```
DECLARE @color nvarchar(15)
SET @color = 'White'
SELECT Name, ProductNumber FROM Production.Product
WHERE Color = @color
```

Indeks *IX_Color* nie zostanie wykorzystany, a plan będzie zawierał ostrzeżenie (zobacz rysunek 1.16).



Rysunek 1.16. Plan z ostrzeżeniem *UnmatchedIndexes*

Na planie XML będziesz mógł zobaczyć następujące informacje (lub zaglądając do właściwości *UnmatchedIndexes* we właściwościach operatora *SELECT*):

```
<UnmatchedIndexes>
  <Parameterization>
    <Object Database="[AdventureWorks2012]" Schema="[Production]"
      Table="[Product]" Index="[IX_Color]" />
  </Parameterization>
</UnmatchedIndexes>
<Warnings UnmatchedIndexes="true" />
```

Natomiast dla poniższego zapytania indeks zostanie wykorzystany:

```
SELECT Name, ProductNumber FROM Production.Product
WHERE Color = 'White'
```

Indeksy filtrowane i element *UnmatchedIndexes* omówię dokładnie w rozdziale 5. Na razie usuń indeks, który utworzyliśmy wcześniej:

```
DROP INDEX Production.Product.IX_Color
```

**UWAGA**

Wiele ćwiczeń w tej książce będzie wymagało wprowadzania zmian w bazie AdventureWorks2012. Chociaż baza jest przywracana do stanu pierwotnego, możesz również rozważyć przywrócenie świeżej kopii bazy po wykonaniu serii ćwiczeń.

Pobieranie planów za pomocą śledzenia lub z magazynu planów

Do tej pory testowaliśmy pobieranie planów wykonania poprzez bezpośrednie wykorzystanie kodu zapytania w Management Studio. Jednakże pobieranie planu w ten sposób nie zawsze jest możliwe, a czasami będziesz musiał przechwycić plan wykonywania z innych lokalizacji (na przykład z magazynu danych lub z puli aktualnie wykonywanych zapytań). W takich przypadkach może będziesz musiał pobrać plan za pośrednictwem śledzenia, na przykład korzystając ze zdarzeń śledzenia lub z rozszerzonych zdarzeń, albo z magazynu planów za pomocą dynamicznej funkcji zarządzania (*Dynamic Management Function* — DMF) `sys.dm_exec_query_plan` lub być może z wykorzystaniem danych zebranych przez komponent SQL Server Data Collector. Przyjrzyjmy się niektórym z tych opcji.

`sys.dm_exec_query_plan`

Jak wspomniałem wcześniej, kiedy zapytanie jest optymalizowane, jego plan wykonywania może być przechowywany w magazynie planów, a funkcja `sys.dm_exec_query_plan` może zostać wykorzystana do zwrócenia zachowanego planu, a także dowolnego planu, który jest aktualnie wykonywany. Jeżeli jednak plan zostanie usunięty z magazynu, przestanie być dostępny, a kolumna `query_plan` zwracanej tabeli będzie miała wartość `null`.

Na przykład poniższe zapytanie pokaże plany wykonania dla wszystkich zapytań aktualnie wykonywanych w systemie. Dynamiczny widok zarządzania (*Dynamic Management View* — DMV) `sys.dm_exec_requests`, który zwraca informacje o wszystkich aktualnie wykonywanych żądaniach, jest potrzebny do uzyskania wartości `plan_handle` (uchwyt planu), która z kolei jest niezbędna do pobrania planu poprzez wykorzystanie funkcji `sys.dm_exec_query_plan`. Wartość `plan_handle` to unikalny w obrębie systemu hasz reprezentujący konkretny plan wykonania.

```
SELECT * FROM sys.dm_exec_requests
CROSS APPLY
sys.dm_exec_query_plan(plan_handle)
```

W wyniku tego zapytania otrzymujemy rezultat zawierający kolumnę `query_plan`, która zawiera linki podobne do tych z podrozdziału dotyczącego planów XML. Jak już tłumaczyłem, kliknięcie linku spowoduje wyświetlenie planu graficznego.

W ten sam sposób poniższy przykład wyświetla wszystkie plany przechowywane w magazynie planów. Widok `sys.dm_exec_query_stats` zawiera jeden wiersz dla każdego zapytania i udostępnia wartość `plan_handle` potrzebną dla funkcji `sys.dm_exec_query_plan`.

```
SELECT * FROM sys.dm_exec_query_stats
CROSS APPLY
sys.dm_exec_query_plan(plan_handle)
```

Założmy teraz, że chcesz znaleźć 10 zapytań najbardziej kosztownych pod względem wykorzystania czasu procesora. Do uzyskania takiej informacji możesz wykorzystać poniższe zapytanie, które zwróci średni czas procesora w mikrosekundach dla wykonania:

```
SELECT TOP 10 total_worker_time/execution_count AS avg_cpu_time,
plan_handle, query_plan
FROM sys.dm_exec_query_stats
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
ORDER BY avg_cpu_time DESC
```

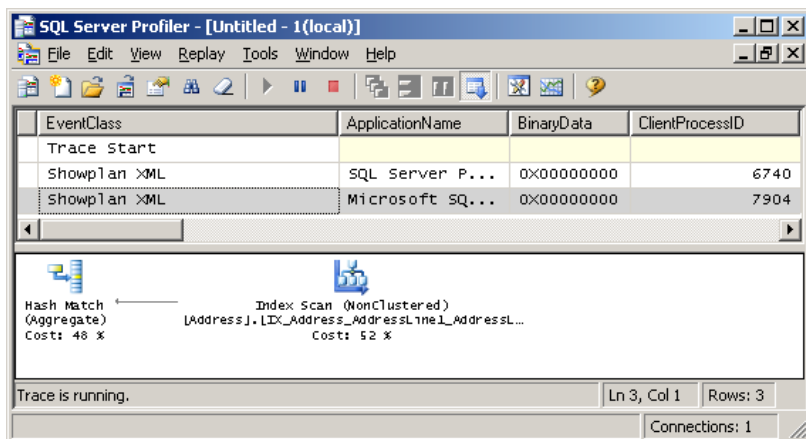
SQL Trace/Profiler

Możesz również wykorzystać program SQL Profiler do przechwytywania planów wykonania aktualnie wykonywanych zapytań. Możesz skorzystać z kategorii zdarzenia *Performance* (wydajność), która zawiera następujące zdarzenia:

- ▶ *Performance Statistics* (statystyki wydajności),
- ▶ *Showplan All* (pokaż plan — wszystko),
- ▶ *Showplan All For Query Compile* (pokaż plan — wszystko dla kompilacji zapytania),
- ▶ *Showplan Statistics Profile* (pokaż plan — profil statystyk),
- ▶ *Showplan Text* (pokaż plan — tekst),
- ▶ *Showplan Text (Unencoded)* (pokaż plan — tekst bez kodowania),
- ▶ *Showplan XML* (pokaż plan — XML),
- ▶ *Showplan XML For Query Compile* (pokaż plan — XML dla kompilacji zapytania),
- ▶ *Showplan XML Statistics Profile* (pokaż plan — XML dla profilu statystyk).

Aby prześledzić któreś z tych zdarzeń, uruchom program Profiler, połącz się ze swoją instancją SQL Servera, kliknij zakładkę *Events Selection* (wybór zdarzeń), rozwiń kategorię *Performance* i wybierz te zdarzenia, które są dla Ciebie interesujące. Możesz wybrać wszystkie kolumny lub tylko ich część, wybrać filtr kolumn i tak dalej. Kliknij przycisk *Run* (uruchom), aby rozpocząć śledzenie. Rysunek 1.17 przedstawia przykład śledzenia dla zdarzenia *Showplan XML*.

Możesz również utworzyć proces śledzenia, korzystając ze skryptu, a nawet wykorzystać program Profiler jako narzędzie do tworzenia skryptów. Aby to zrobić, zdefiniuj zdarzenia do śledzenia, uruchom i zatrzymaj proces śledzenia, a następnie wybierz *File/Export/Script Trace Definition/For SQL Server 2005 – 2014...* (plik/eksport/



Rysunek 1.17. Śledzenie w programie Profiler dla zdarzenia Showplan XML

definicja skryptu śledzenia/dla SQL Server 2005-2014...). Dzięki temu uzyskasz kod pozwalający uruchomić proces śledzenia, który będzie wymagał tylko podania nazwy pliku, w którym mają być zapisywane przechwycone dane. Część wygenerowanego kodu została pokazana na listingu poniżej:

```

/*****
/* Created by: SQL Server 2014 Profiler */
/* Date: 12/18/2013 08:37:22 AM */
*****/

-- Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

-- Please replace the text InsertFileNameHere, with an appropriate
-- filename prefixed by a path, e.g., c:\MyFolder\MyTrace. The .trc extension
-- will be appended to the filename automatically. If you are writing from
-- remote server to local drive, please use UNC path and make sure server has
-- write access to your network share
exec @rc = sp_trace_create @TraceID output, 0, N'InsertFileNameHere', @maxfilesize,
NULL
if (@rc != 0) goto error

-- Client side File and Table cannot be scripted

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 10, 1, @on
exec sp_trace_setevent @TraceID, 10, 9, @on
exec sp_trace_setevent @TraceID, 10, 2, @on
exec sp_trace_setevent @TraceID, 10, 66, @on
exec sp_trace_setevent @TraceID, 10, 10, @on
exec sp_trace_setevent @TraceID, 10, 3, @on
exec sp_trace_setevent @TraceID, 10, 4, @on

```

```

exec sp_trace_setevent @TraceID, 10, 6, @on
exec sp_trace_setevent @TraceID, 10, 7, @on
exec sp_trace_setevent @TraceID, 10, 8, @on
exec sp_trace_setevent @TraceID, 10, 11, @on
exec sp_trace_setevent @TraceID, 10, 12, @on
exec sp_trace_setevent @TraceID, 10, 13, @on

```



UWAGA

W wersji 2008 wszystkie zdarzenia niezwiązane z XML-em, takie jak *Showplan All* czy *Showplan Text*, zostały zdeprecjonowane. Począwszy od tej wersji, Microsoft zaleca korzystanie ze zdarzeń XML. Także SQL Trace zostało zdeprecjonowane, w wersji 2012, i zalecane jest korzystanie ze zdarzeń rozszerzonych.

Więcej szczegółów na temat korzystania z Profiler'a i SQL Trace znajdziesz w dokumentacji SQL Server Books Online.

Zdarzenia rozszerzone

Do przechwytywania planów zapytania możesz również wykorzystać zdarzenia rozszerzone. Choć Microsoft zaleca korzystanie ze zdarzeń rozszerzonych zamiast SQL Trace, jak wcześniej wspomniałem, w aktualnych wersjach SQL Servera zdarzenia przechwytyjące plany zapytania są kosztowne. Dokumentacja zawiera następujące ostrzeżenie dla wszystkich trzech zdarzeń rozszerzonych dotyczących przechwytywania planów wykonania: „Wykorzystanie tego zdarzenia może mieć znaczący wpływ na spadek wydajności, więc powinno być wykorzystywane tylko do szukania problemów lub monitorowania konkretnych problemów przez krótkie okresy czasu”.

Możesz stworzyć i uruchomić sesję zdarzeń rozszerzonych za pomocą polecenia `CREATE EVENT SESSION` i `ALTER EVENT SESSION`. Możesz także skorzystać z nowego interfejsu graficznego wprowadzonego w SQL Serverze 2012. Oto zdarzenia związane z planami wykonania:

- ▶ **query_post_compilation_showplan** — występuje po skompilowaniu polecenia SQL. To zdarzenie zwraca reprezentację szacowanego planu wykonania w formacie XML, który generowany jest po skompilowaniu zapytania.
- ▶ **query_post_execution_showplan** — występuje po wykonaniu polecenia SQL. To zdarzenie zwraca reprezentację właściwego planu zapytania w formacie XML.
- ▶ **query_pre_execution_showplan** — występuje po skompilowaniu polecenia SQL. To zdarzenie zwraca reprezentację szacowanego planu wykonania w formacie XML, który generowany jest po optymalizacji zapytania.

Załóżmy na przykład, że chcesz rozpocząć sesję śledzenia zdarzenia `query_post_execution_showplan`. Do stworzenia takiej sesji możesz wykorzystać poniższe zapytanie:

```

CREATE EVENT SESSION [test] ON SERVER
ADD EVENT sqlserver.query_post_execution_showplan(
    ACTION(sqlserver.plan_handle)
    WHERE ([sqlserver].[database_name]=N'AdventureWorks2012'))
ADD TARGET package0.ring_buffer
WITH (STARTUP_STATE=OFF)
GO

```

Zdarzenia rozszerzone omówię dokładniej w rozdziale 2. Na razie możesz zwrócić uwagę na to, że argument `ADD EVENT` zawiera nazwę zdarzenia (w tym przypadku `query_post_execution_showplan`), `ACTION` odnosi się do pól globalnych, które mają zostać przechwycone w sesji zdarzenia (w tym wypadku `plan_handle`), a `WHERE` jest wykorzystywane do zdefiniowania filtra do ograniczenia danych, które mają zostać przechwycone. Predykat `[sqlserver].[database_name]=N'AdventureWorks2012'` wskazuje, że chcemy przechwytywać zdarzenia wyłącznie dla bazy AdventureWorks2012. `TARGET` to konsument zdarzenia i możemy wykorzystać ten parametr do zbierania danych do analizy. W tym przypadku celem jest `ring_buffer`. Wreszcie `STARTUP_STATE` to jedna z opcji zdarzenia rozszerzonego i wykorzystywana jest do określenia, czy sesja powinna być uruchamiana automatycznie po starcie SQL Servera.

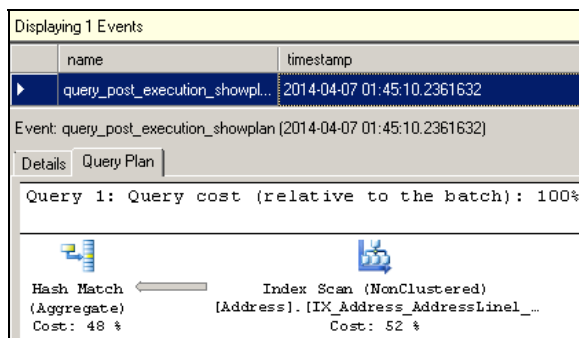
Po stworzeniu sesji możesz ją uruchomić, korzystając z polecenia `ALTER EVENT SESSION`:

```

ALTER EVENT SESSION [test]
ON SERVER
STATE=START

```

Do podglądu danych przechwyconych przez sesję zdarzenia rozszerzonego możesz skorzystać z funkcjonalności *Watch Live Data* (podgląd danych na żywo), wprowadzonej w SQL Serverze 2012. Aby to zrobić, w oknie *Object Explorer* (eksplorator obiektów) rozwiń katalog *Management/Extended Events/Sessions*, a następnie kliknij prawym przyciskiem na sesji i wybierz *Watch Live Data*. Rysunek 1.18 pokazuje przykład przechwycenia planu wykonania.



Rysunek 1.18. Funkcjonalność podglądania danych na żywo

Aby zobaczyć te dane, możesz również wykorzystać poniższy kod:

```
SELECT
    event_data.value('(event/@name)[1]', 'varchar(50)') AS event_name,
    event_data.value('(event/action[@name="plan_handle"]/value)[1]',
        'varchar(max)') as plan_handle,
    event_data.query('event/data[@name="showplan_xml"]/value/*') as showplan_xml,
    event_data.value('(event/action[@name="sql_text"]/value)[1]',
        'varchar(max)') AS sql_text
FROM( SELECT evnt.query('.') AS event_data
FROM
    ( SELECT CAST(target_data AS xml) AS target_data
    FROM sys.dm_xe_sessions AS s
    JOIN sys.dm_xe_session_targets AS t
    ON s.address = t.event_session_address
    WHERE s.name = 'test' AND t.target_name = 'ring_buffer'
    ) AS data
CROSS APPLY target_data.nodes('RingBufferTarget/event') AS xevent(evnt)
) AS xevent(event_data)
```

Kiedy skończysz testy, musisz zatrzymać lub usunąć sesję. Uruchom następujące polecenia:

```
ALTER EVENT SESSION [test]
ON SERVER
STATE=STOP
GO
DROP EVENT SESSION [test] ON SERVER
```

Jest jeszcze kilka innych narzędzi SQL Servera pozwalających przeglądać plany, w tym narzędzie Data Collector, wprowadzone w SQL Serverze 2008. Omówię je w rozdziale 2.

Usuwanie planów z magazynu planów

Możesz wykorzystać kilka różnych poleceń do usuwania planów z magazynu planów. Te polecenia, dokładniej omówione w rozdziale 8., mogą być użyteczne podczas testowania i nie powinny być uruchamiane w środowisku produkcyjnym, jeżeli nie jesteśmy pewni, jaki efekt chcemy osiągnąć. Polecenie `DBCC FREEPROCCACHE` jest wykorzystywane do usuwania wszystkich wpisów z magazynu. Może również przyjmować uchwyt do planu lub uchwyt SQL w celu usuwania tylko pojedynczych planów, może też przyjmować nazwę puli *Resource Governor* (zarządca zasobów) i usuwać plany z nią związane. Polecenia `DBCC FREESYSTEMCACHE` można użyć do usunięcia wszystkich elementów z magazynu planów lub tylko elementów skojarzonych z nazwą puli *Resource Governor*. `DBCC FLUSHPROCINDB` można wykorzystać do usunięcia wszystkich planów dla konkretnej bazy danych.

Chociaż nie jest ono związane z magazynem planów, można też wykorzystać polecenie `DBCC DROPCLEANBUFFERS` do usunięcia wszystkich buforów z puli buforów. Możesz skorzystać z tego polecenia, gdy zechcesz zasymulować uruchomienie zapytania z pustą pamięcią podręczną, tak jak zrobimy to w kolejnym podrozdziale.

SET STATISTICS TIME i SET STATISTICS IO

Zamknijemy ten rozdział dwoma poleceniami, które mogą udzielić Ci dodatkowych informacji o Twoich zapytaniach i które pozwolą Ci skorzystać z dodatkowych technik poprawiania wydajności zapytań. Mogą być doskonałym uzupełnieniem planów wykonania, jeżeli chodzi o informacje na temat wykonania i optymalizacji zapytań. Często zdarza się, że programiści próbują porównywać koszt zapytania z wydajnością zapytania. Nie powinieneś zakładać bezpośredniej korelacji pomiędzy szacowanym kosztem zapytania a wydajnością planu. Koszt to wewnętrzna jednostka wykorzystywana przez optymalizator zapytań i nie powinna być stosowana do porównywania wydajności planu; zamiast tego możesz użyć SET STATISTICS TIME i SET STATISTICS IO. Ten podrozdział opisuje oba polecenia.

Polecenie SET STATISTICS TIME możesz wykorzystać do sprawdzania, jak długo, w milisekundach, trwa parsowanie, kompilacja i wykonanie zapytania. Na przykład uruchom poniższe polecenie:

```
SET STATISTICS TIME ON
```

a następnie poniższe zapytanie:

```
SELECT DISTINCT(CustomerID)
FROM Sales.SalesOrderHeader
```

Aby zobaczyć wynik, będziesz musiał przełączyć się do zakładki *Messages* (komunikaty) okna wyników; zobaczysz wynik podobny do tego:

```
SQL Server parse and compile time:
  CPU time = 16 ms, elapsed time = 226 ms.
SQL Server Execution Times:
  CPU time = 16 ms, elapsed time = 148 ms.
```

Fragment „parse and compile” odnosi się do czasu potrzebnego na optymalizację zapytania. Polecenie SET STATISTICS TIME będzie włączone dla wszystkich wykonywanych zapytań. Możesz wyłączyć śledzenie czasu następującym poleceniem:

```
SET STATISTICS TIME OFF
```

Jak już wspomniałem, informacje dotyczące parsowania i kompilacji można także zobaczyć na planie wykonania:

```
<QueryPlan DegreeOfParallelism="1" CachedPlanSize="16" CompileTime="226" CompileCPU="9"
  ↳CompileMemory="232">
```

Oczywiście, jeżeli potrzebujesz tylko informacji o czasie wykonywania każdego zapytania, możesz otrzymać tę informację bezpośrednio w edytorze zapytań Management Studio Query Editor.

Polecenie SET STATISTICS IO pozwala uzyskać informację o ilości operacji dyskowych generowanych przez zapytanie. Aby je uruchomić, wykonaj następujące polecenie:

```
SET STATISTICS IO ON
```

Wykonaj poniższe polecenie, aby opróżnić bufor z puli buforów i zyskać dzięki temu pewność, że dla tej tabeli żadne strony nie są załadowane do pamięci:

```
DBCC DROPCLEANBUFFERS
```

Następnie wykonaj poniższe zapytanie:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 870
```

Otrzymasz wynik podobny do tego:

```
Table 'SalesOrderDetail'. Scan count 1, logical reads 1246, physical reads 3,
↳read-ahead reads 1277, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Oto definicje tych pozycji (wszystkie korzystają ze stron o rozmiarze 8 kB):

- ▶ **Logical reads** — liczba stron przeczytanych z puli bufora.
- ▶ **Physical reads** — liczba stron przeczytanych z dysku.
- ▶ **Read-ahead reads** — *Read-ahead* to mechanizm optymalizacji wydajności, który przewiduje, jakie strony będą potrzebne, i odczytuje je z dysku. Może przeczytać do 64 stron z jednego pliku danych.
- ▶ **Lob logical reads** — liczba stron LOB (ang. *large object* — duże obiekty) przeczytanych z puli bufora.
- ▶ **Lob physical reads** — liczba stron LOB przeczytanych z dysku.
- ▶ **Lob read-ahead reads** — liczba stron LOB przeczytanych z dysku za pomocą mechanizmu *Read-ahead*.

Jeżeli teraz uruchomisz to samo zapytanie, nie będzie żadnych fizycznych odczytów mechanizmu *Read-ahead*, a wynik będzie podobny do poniższego:

```
Table 'SalesOrderDetail'. Scan count 1, logical reads 1246, physical reads 0,
↳read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Atrybut *scan count* jest definiowany jako liczba wyszukikań lub skanów rozpoczętych po osiągnięciu poziomu liścia (czyli najniższego poziomu indeksu). Jedyny przypadek, w którym wartość ta będzie równa 0, to taki, kiedy szukasz jednej wartości dla indeksu unikalnego, jak w poniższym przykładzie:

```
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 51119
```

Jeżeli wypróbujesz poniższe zapytanie, w którym *SalesOrderID* będzie zdefiniowane jako indeks nieunikalny mogący zwracać więcej niż jeden wiersz, zobaczysz, że atrybut *scan count* przyjmie wartość 1.

```
SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 51119
```

Wreszcie dla poniższego przykładu atrybut *scan count* przyjmie wartość 4, ponieważ SQL Server będzie musiał uruchomić cztery wyszukiwania:

```
SELECT * FROM Sales.SalesOrderHeader  
WHERE SalesOrderID IN (51119, 43664, 63371, 75119)
```

Podsumowanie

W tym rozdziale pokazałem, w jaki sposób lepsze zrozumienie tego, co robi procesor zapytań, może pomóc zarówno administratorom baz, jak i programistom pisać lepsze zapytania i zapewniać optymalizatorowi zapytań informacje, których potrzebuje do tworzenia wydajnych planów zapytania. Pokazałem też, jak możesz wykorzystać nową wiedzę dotyczącą działania procesora zapytań i narzędzi wbudowanych w SQL Server do analizowania przypadków, w których zapytania nie zachowują się zgodnie z oczekiwaniami. Na tej podstawie przedstawiłem najważniejsze informacje dotyczące optymalizatora zapytań, silnika wykonywania i magazynu planów, a komponenty te omówię dokładniej w dalszej części książki.

Ponieważ plany zapytań będziemy wykorzystywać w całej książce, pokazałem również, jak je czytać, jakie są ich najważniejsze właściwości i jak pobierać je ze źródeł takich jak magazyn planów czy za pomocą śledzenia aktywności serwera. Ta część powinna dać Ci wystarczającą wiedzę do zrozumienia reszty książki. Wprowadziłem także pojęcie operatorów zapytań, ale omówię je dokładniej w rozdziale 4. i w dalszej części książki.

W następnym rozdziale pokażę Ci dodatkowe narzędzia pozwalające poprawić wydajność zapytań oraz techniki takie jak SQL Trace, zdarzenia rozszerzone i widoki DMV, które pozwolą ocenić, jakie zapytania zużywają najwięcej zasobów, i pomogą odnaleźć inne problemy związane z wydajnością.

Rozdział 2

Rozwiązywanie problemów w zapytaniach

W tym rozdziale:

- ▶ DMV i DMF
- ▶ SQL Trace
- ▶ Zdarzenia rozszerzone
- ▶ Data Collector
- ▶ Podsumowanie





rozdziale 1. przedstawiłem, jak czytać plany zapytań, które będą głównym narzędziem, z jakiego będziemy korzystać podczas interakcji z procesorem zapytań. Pokazałem również polecenia `SET STATISTICS TIME` i `SET STATISTICS IO`, które mogą udzielić dodatkowych informacji dotyczących wydajności zapytań. W tym rozdziale będziemy kontynuować ten temat. Pokażę Ci dodatkowe narzędzia i techniki, które będziesz mógł wykorzystać do sprawdzenia, ile zasobów serwera zużywają Twoje zapytania i jak znaleźć w systemie najbardziej kosztowne zapytania.

Dynamiczne widoki zarządzania (DMV) zostały wprowadzone w SQL Serverze 2005 jako świetne narzędzie do diagnozowania problemów, poprawiania wydajności i monitorowania kondycji instancji serwera. Dostępnych jest wiele widoków DMV, a pierwszy podrozdział tego rozdziału skupia się na `sys.dm_exec_requests`, `sys.dm_exec_sessions` i `sys.dm_exec_query_stats`, których możesz użyć do sprawdzenia zasobów serwera takich jak CPU i I/O, wykorzystywanych przez zapytania działające w systemie. W dalszej części tego rozdziału, kiedy będę omawiać zdarzenia rozszerzone, a także w pozostałych rozdziałach książki przedstawię jeszcze wiele widoków DMV.

Chociaż śledzenie aktywności serwera za pomocą SQL Trace zostało zdeprecjonowane w wersji 2012, wciąż jest szeroko stosowane i będzie dostępne w kolejnych wersjach. SQL Trace zazwyczaj łączy się z programem SQL Server Profiler, ponieważ korzystanie z tego narzędzia to najprostszy sposób na definiowanie i uruchamianie procesów śledzenia, jest także doskonałym narzędziem do tworzenia skryptów i procesów śledzenia na serwerze, co jest wykorzystywane w niektórych scenariuszach, w których uruchamianie Profiler'a może być kosztowne. W tym rozdziale omówię niektóre ze zdarzeń śledzenia, które mogą być użyteczne podczas szukania problemów z wydajnością zapytań.

W kolejnym podrozdziale, kontynuując podobną koncepcję jak w przypadku SQL Trace, przedstawię zdarzenia rozszerzone. Najpierw omówię wszystkie podstawowe koncepcje i definicje, włączając w to zdarzenia, predykaty, akcje, cele i sesje, a następnie stworzymy kilka sesji, które pozwolą uzyskać informacje dotyczące wydajności zapytań. Ponieważ większość osób pracujących z bazą SQL Servera zna już SQL Trace i SQL Profiler, pokażę również jak zmapować stare zdarzenia śledzenia na zdarzenia rozszerzone.

Na koniec pokażę wprowadzoną w wersji 2008 funkcjonalność o nazwie Data Collector. Pomoże ona aktywnie zbierać dane dotyczące wydajności, które będą mogły zostać wykorzystane w razie pojawienia się problemów z wydajnością.

DMV i DMF

W tym podrozdziale pokażę kilka dynamicznych widoków zarządzania (DMV) i dynamicznych funkcji zarządzania (DMF), które pomogą Ci w sprawdzeniu ilości zasobów serwera wykorzystywanych przez Twoje zapytania oraz znalezieniu najbardziej kosztownych zapytań.

sys.dm_exec_requests i sys.dm_exec_sessions

Widok `sys.dm_exec_requests` może posłużyć do wyświetlenia żądań aktualnie wykonywanych przez serwer, natomiast `sys.dm_exec_sessions` zawiera uwierzytelnione sesje na instancji. Widoki zawierają wiele kolumn, ale w tym podrozdziale skupimy się na kolumnach związanych z wykorzystaniem zasobów i wydajnością zapytań. Definicje pozostałych kolumn znajdziesz w dokumentacji Books Online.

Oba widoki zawierają kilka wspólnych kolumn, które przedstawia poniższa tabela:

Kolumna	Definicja
<code>cpu_time</code>	Czas procesora w milisekundach wykorzystany przez żądanie lub przez żądania w sesji.
<code>total_elapsed_time</code>	Całkowity czas w milisekundach od odebrania żądania lub od początku sesji.
<code>reads</code>	Liczba odczytów wykonanych przez żądanie lub żądania w sesji.
<code>writes</code>	Liczba zapisów wykonanych przez żądanie lub żądania w sesji.
<code>logical_reads</code>	Liczba odczytów logicznych wykonanych przez żądanie lub żądania w sesji.
<code>row_count</code>	Liczba wierszy zwróconych do klienta na podstawie tego żądania.

Widok `sys.dm_exec_requests` pokazuje zasoby wykorzystane przez konkretne, aktualnie wykonywane żądanie, podczas gdy `sys.dm_exec_sessions` pokaże sumę zasobów wszystkich żądań danej sesji. Aby zrozumieć, jak widoki zbierają informacje o wykorzystaniu zasobów, możemy wykonać ćwiczenie wykorzystujące zapytanie, które trwa przynajmniej kilka sekund. Otwórz nowe okno Management Studio i odczytaj jego identyfikator sesji (na przykład za pomocą `SELECT @@SPID`), ale jeszcze nic nie uruchamiaj, ponieważ wykorzystanie zasobów w widoku `sys.dm_exec_sessions` jest sumowane. Skopiuj i przygotuj do wykonania następujące zapytanie:

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
GO
SELECT * FROM Production.Product p1 CROSS JOIN
Production.Product p2
```

Do drugiego okna skopiuj poniższe zapytanie, zamieniając wartość `session_id` na wartość otrzymaną w pierwszym oknie:

```

SELECT cpu_time, reads, total_elapsed_time, logical_reads, row_count
FROM sys.dm_exec_requests
WHERE session_id = 56
GO
SELECT cpu_time, reads, total_elapsed_time, logical_reads, row_count
FROM sys.dm_exec_sessions
WHERE session_id = 56

```

Uruchom zapytanie w pierwszej sesji, a następnie w tym samym czasie kilkakrotnie uruchom zapytania z drugiej sesji, aby zaobserwować zużycie zasobów. Poniższe dane pokazują przykładowe wykonanie w trakcie działania zapytania. Zauważ, że widok `sys.dm_exec_requests` pokazuje częściowo wykorzystane zasoby, które w widoku `sys.dm_exec_sessions` przedstawiane są jako jeszcze nie wykorzystane. Najprawdopodobniej oba widoki nie zwrócą takich samych danych.

cpu_time	reads	total_elapsed_time	logical_reads	row_count
468	62	4767	5868	1

cpu_time	reads	total_elapsed_time	logical_reads	row_count
0	0	5	0	1

Kiedy wykonanie zapytania się zakończy, żądanie przestaje istnieć, a widok `sys.dm_exec_sessions` rejestruje już zasoby wykorzystane przez pierwsze zapytanie.

cpu_time	reads	total_elapsed_time	logical_reads	row_count
671	62	6996	8192	254016

Jeżeli jeszcze raz uruchomisz zapytanie z pierwszej sesji, widok `sys.dm_exec_sessions` zsumuje zasoby z obu uruchomień, więc wartości będą ponad dwa razy większe od poprzednich:

cpu_time	reads	total_elapsed_time	logical_reads	row_count
1295	124	14062	16384	254016

Pamiętaj, że czas procesora i czas trwania mogą i najprawdopodobniej będą się nieznacznie różnić między wykonaniami. Liczba odczytów logicznych to 8192,

a wartość sumaryczna dla dwóch wykonań wynosi 16 384. Widać też, że widok `sys.dm_exec_requests` pokazuje tylko dane dotyczące aktualnie wykonywanych zapytań, możesz nie zobaczyć interesujących cię danych, jeżeli nie zdążysz odpytać widoku przed zakończeniem wykonywania zapytania.

Podsumowując: widoki `sys.dm_exec_requests` i `sys.dm_exec_sessions` mogą być użyteczne do kontrolowania zasobów aktualnie wykorzystywanych przez zapytania lub sumarycznych zasobów wykorzystywanych przez zapytania danej sesji od jej rozpoczęcia.

sys.dm_exec_query_stats

Jeżeli pracowałeś z wersją SQL Servera starszą niż 2005, być może pamiętasz, jak trudno było znaleźć najbardziej kosztowne zapytania. Wykonanie takiej analizy wymagało zazwyczaj uruchomienia na pewien czas śledzenia na serwerze dla Twojej instancji, a następnie przejrzenia zebranych danych, przeważnie zajmujących wiele gigabajtów, za pomocą zewnętrznych narzędzi lub opracowanych przez siebie metod (bardzo czasochłonny proces). Nie wspominając już o fakcie, że uruchomienie takiego śledzenia także mogło mieć wpływ na wydajność systemu, który i tak prawdopodobnie już miał problemy z wydajnością.

Widoki DMV zostały wprowadzone w wersji 2005 i są doskonałym narzędziem pozwalającym diagnozować problemy, poprawiać wydajność i monitorować stan systemu. Szczególnie `sys.dm_exec_query_stats` udostępnia bogate informacje, niedostępne we wcześniejszych wersjach SQL Servera, dotyczące zagregowanych danych wydajnościowych lub przechowywanych planów zapytań. Te informacje pozwalają w większości przypadków uniknąć wspomnianej wcześniej konieczności uruchamiania śledzenia. Ten widok zwraca po jednym wierszu dla każdego planu przechowywanego w magazynie planów, a w SQL Serverze 2008 dodano jeszcze hasze planu zapytania i samego zapytania, które to wartości omówię wkrótce.

Przyjrzyjmy się, jak działa `sys.dm_exec_query_stats` i jakie udostępnia informacje. Utwórz poniższą procedurę przechowywaną zawierającą trzy proste zapytania:

```
CREATE PROC test
AS
SELECT * FROM Sales.SalesOrderDetail WHERE SalesOrderID = 60677
SELECT * FROM Person.Address WHERE AddressID = 21
SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
```

Wykonaj poniższy kod czyszczący magazyn planów (aby łatwiej było przeglądać informacje z nim związane), który usuwa wszystkie czyste bufor z puli buforów, uruchamia stworzoną wcześniej procedurę przechowywaną i podgląda magazyn planów. Zauważ, że w kodzie wykorzystywana jest funkcja DMF `sys.dm_exec_sql_text`, która wymaga podania uchwytu do planu (`plan_handle`) lub do zapytania (`sql_handle`) i zwraca tekst zestawu zapytań SQL.

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
GO
EXEC test
GO
SELECT * FROM sys.dm_exec_query_stats
CROSS APPLY sys.dm_exec_sql_text(sql_handle)
WHERE objectid = OBJECT_ID('dbo.test')
```

Przjrzyj się wynikowi. Ponieważ liczba kolumn jest zbyt duża, aby pokazać je wszystkie w książce, poniższa tabela zawiera tylko niektóre z nich:

statement_start_offset	statement_end_offset	execution_count	total_worker_time	last_worker_time	min_worker_time	max_worker_time	text
44	168	1	532	532	532	532	CREATE PROC test AS ...
174	270	1	622	622	622	622	CREATE PROC test AS ...
276	406	1	667	667	667	667	CREATE PROC test AS ...

Jak widać po kolumnie text, wszystkie trzy zapytania zostały skompilowane jako część tego samego zestawu, co potwierdza fakt, że mają ten sam uchwyt do planu i uchwyt do zapytania. Kolumny statement_start_offset (offset początku polecenia) i statement_end_offset (offset końca polecenia) mogą zostać wykorzystane do zidentyfikowania konkretnych zapytań, a proces ten wyjaśnię w dalszej części niniejszego podrozdziału. W wyniku znajduje się również informacja o tym, ile razy zostało wykonane zapytanie, oraz kilka kolumn zawierających czas procesora wykorzystany przez każde z zapytań — total_worker_time, last_worker_time, min_worker_time i max_worker_time. Jeżeli zapytanie zostanie wykonane więcej niż raz, sumaryczny czas procesora zostanie pokazany w kolumnie total_worker_time. W tabeli powyżej nie zostały pokazane statystyki wydajnościowe dotyczące odczytów fizycznych, zapisów logicznych, odczytów logicznych, czasu CLR i czasu wykonywania. Tabela 2.1 przedstawia listę kolumn, włącznie ze statystykami wydajnościowymi i ich opisem z dokumentacji.

Pamiętaj, że ten widok zawiera tylko informacje dla zakończonych zapytań. Aby uzyskać informacje na temat aktualnie wykonywanych zapytań skorzystaj z opisywanego wcześniej widoku sys.dm_exec_requests. Jak już tłumaczyłem w rozdziale 1., niektóre rodzaje planów nie będą przechowywane w magazynie, a niektóre będą z niego usuwane z różnych powodów, na przykład z powodu wewnętrznych lub zewnętrznych ograniczeń pamięci.

Tabela 2.1. Kolumny widoku sys.dm_exec_query_stats związane z wydajnością

Kolumna	Opis
total_worker_time	Całkowity czas procesora w mikrosekundach (z dokładnością do milisekund) dla wszystkich wykonań planu od jego kompilacji.
last_worker_time	Czas procesora w mikrosekundach (z dokładnością do milisekund) dla ostatniego wykonania planu.
min_worker_time	Najkrótszy czas procesora w mikrosekundach (z dokładnością do milisekund) spośród wszystkich wykonań planu.
max_worker_time	Maksymalny czas procesora w mikrosekundach (z dokładnością do milisekund) spośród wszystkich wykonań planu.
total_physical_reads	Sumaryczna liczba odczytów fizycznych dla planu od czasu jego skompilowania.
last_physical_reads	Liczba odczytów fizycznych podczas ostatniego wykonania planu.
min_physical_reads	Minimalna liczba odczytów fizycznych spośród wszystkich wykonań planu.
max_physical_reads	Maksymalna liczba odczytów fizycznych spośród wszystkich wykonań planu.
total_logical_writes	Sumaryczna liczba zapisów logicznych dla planu od czasu jego skompilowania.
last_logical_writes	Liczba zapisów logicznych podczas ostatniego wykonania planu.
min_logical_writes	Minimalna liczba zapisów logicznych spośród wszystkich wykonań planu.
max_logical_writes	Maksymalna liczba zapisów logicznych spośród wszystkich wykonań planu.
total_logical_reads	Sumaryczna liczba odczytów logicznych dla planu od czasu jego skompilowania.
last_logical_reads	Liczba odczytów logicznych podczas ostatniego wykonania planu.
min_logical_reads	Minimalna liczba odczytów logicznych spośród wszystkich wykonań planu.
max_logical_reads	Maksymalna liczba odczytów logicznych spośród wszystkich wykonań planu.
total_clr_time	Czas w mikrosekundach (z dokładnością do milisekund) spędzony wewnątrz obiektów wspólnego środowiska uruchomieniowego (CLR) .NET Framework podczas wszystkich uruchomień tego planu od czasu jego kompilacji. Obiekty CLR mogą być procedurami przechowywanymi, funkcjami, wyzwalaczami, typami i agregacjami.
last_clr_time	Czas w mikrosekundach (z dokładnością do milisekund) spędzony wewnątrz obiektów CLR .NET Framework podczas ostatniego uruchomienia planu. Obiekty CLR mogą być procedurami przechowywanymi, funkcjami, wyzwalaczami, typami i agregacjami.
min_clr_time	Minimalny czas w mikrosekundach (z dokładnością do milisekund) spędzony wewnątrz obiektów CLR .NET Framework spośród wszystkich uruchomień planu. Obiekty CLR mogą być procedurami przechowywanymi, funkcjami, wyzwalaczami, typami i agregacjami.
max_clr_time	Maksymalny czas w mikrosekundach (z dokładnością do milisekund) spędzony wewnątrz obiektów CLR .NET Framework spośród wszystkich uruchomień planu. Obiekty CLR mogą być procedurami przechowywanymi, funkcjami, wyzwalaczami, typami i agregacjami.

Tabela 2.1. Kolumny widoku sys.dm_exec_query_stats związane z wydajnością — *ciąg dalszy*

Kolumna	Opis
total_elapsed_time	Sumaryczny czas w mikrosekundach (z dokładnością do milisekund) dla wszystkich zakończonych uruchomień planu.
last_elapsed_time	Czas w mikrosekundach (z dokładnością do milisekund) dla ostatniego uruchomienia planu.
min_elapsed_time	Najkrótszy czas w mikrosekundach (z dokładnością do milisekunda) spośród wszystkich ukończonych uruchomień planu.
max_elapsed_time	Najdłuższy czas w mikrosekundach (z dokładnością do milisekunda) spośród wszystkich ukończonych uruchomień planu.

Przyjrzyjmy się teraz wartościom `statement_start_offset` i `statement_end_offset`.

Wartości `statement_start_offset` i `statement_end_offset`

Jak widać na podstawie poprzedniego wyniku z widoku `sys.dm_exec_query_stats`, kolumny `sql_handle`, `plan_handle` i `text` pokazująca kod dla procedury przechowywanej są dokładnie takie same dla wszystkich trzech wierszy. Dla całego zestawu wykorzystywane są ten sam plan i to samo zapytanie. Jak więc zidentyfikować poszczególne zapytania, zakładając, na przykład, że tylko jedno z nich jest naprawdę kosztowne? Musimy skorzystać z kolumn `statement_start_offset` i `statement_end_offset`. Kolumna `statement_start_offset` jest definiowana jako pozycja startowa zapytania, które opisuje dany wiersz, wewnątrz zestawu zapytań, natomiast `statement_end_offset` to punkt końcowy zapytania, które opisuje dany wiersz, wewnątrz zestawu zapytań. Obie wartości wyrażone są w bajtach, zaczynając od 0, a wartość `-1` identyfikuje koniec zestawu zapytań.

Możemy z łatwością rozszerzyć nasze poprzednie zapytanie, aby wykorzystywało `statement_start_offset` i `statement_end_offset`. Otrzymamy zapytanie podobne do poniższego:

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
GO
EXEC test
GO
SELECT SUBSTRING(text, (statement_start_offset/2) + 1,
((CASE statement_end_offset
WHEN -1
THEN DATALENGTH(text)
ELSE
statement_end_offset
END
```



```
SELECT * FROM sys.dm_os_memory_objects
WHERE type = 'MEMOBJ_SQLMGR'
```

Ponieważ `sql_handle` połączone jest z `plan_handle` relacją 1:N (czyli dla danego zapytania może istnieć więcej niż jeden wygenerowany plan), tekst zapytania pozostanie w magazynie `SQLMGR`, dopóki ostatni z dotyczących go planów nie zostanie usunięty.

Wartość `plan_handle` to hasz odnoszący się do planu zapytania, którego częścią jest zapytanie, i może zostać wykorzystana w funkcji `DMF sys.dm_exec_query_plan` do pobrania tego planu. Jest unikalna dla każdego zestawu i pozostanie niezmienniona, nawet jeżeli jedno lub więcej zapytań z zestawu zostanie przekompilowanych. Oto przykład:

```
SELECT * FROM sys.dm_exec_query_plan  
↳(0x050000500996DB224B0C9B8F801000000010000000000000000000000000000000000000000000000000000)
```

Uruchomienie tego zapytania zwróci poniższe wyniki, a kliknięcie linku z kolumny query plan spowoduje wyświetlenie planu graficznego:

dbid	objectid	number	encrypted	query_plan
5	615673241	1	0	<ShowPlanXML xmlns="http://schemas.microsoft.com/SQLServer/2004/07/showplan" ...

Plany wykonywania są przechowywane w magazynach SQLCP i OBJCP: plany dla obiektów, czyli procedur przechowywanych, wyzwalaczy i funkcji, są przechowywane w magazynie OBJCP, natomiast plany dla zapytań *ad hoc*, zapytań autoparametryzowanych i zapytań przygotowywanych są przechowywane w magazynie SQLCP.

query_hash i plan_hash

Chociaż widok `sys.dm_exec_query_stats`, kiedy został wprowadzony w wersji 2005, był doskonałym źródłem informacji dotyczących wydajności, jednym z jego ograniczeń była trudność agregowania informacji dla tego samego zapytania, gdy zapytanie nie było parametryzowane. Kolumny `query_hash` i `plan_hash`, wprowadzone w SQL Serverze 2008, stanowią rozwiązanie tego problemu. Aby zrozumieć problem, przyjrzyjmy się, jak zachowuje się `sys.dm_exec_query_stats` dla zapytań autoparametryzowanych:

```
DBCC FREEPROCCACHE
DBCC DROPCCLEANBUFFERS
GO
SELECT * FROM Person.Address
WHERE AddressID = 12
GO
SELECT * FROM Person.Address
WHERE AddressID = 37
GO
SELECT * FROM sys.dm_exec_query_stats
```


pośrednio lub bezpośrednio sparametryzowane. Zarówno `query_hash`, jak i `plan_hash` dostępne są w widokach `sys.dm_exec_query_stats` i `sys.dm_exec_requests`.

Wartość `query_hash` jest wyliczana na podstawie drzewa operatorów logicznych stworzonego po sparsowaniu, a tuż przed optymalizacją zapytania. Drzewo logiczne jest używane jako dane wsadowe dla optymalizatora zapytań. Dlatego dwa lub więcej zapytań nie musi mieć dokładnie takiego samego tekstu, aby powstała ta sama wartość `query_hash`, ponieważ parametry, komentarze i kilka innych pomniejszych różnic nie są brane pod uwagę. Jak widać w pierwszym przykładzie, dwa zapytania z taką samą wartością `query_hash` mogą mieć różne plany zapytania (czyli różne wartości `query_plan_hash`). Z drugiej strony, wartość `query_plan_hash` obliczana jest na podstawie drzewa operatorów fizycznych składających się na plan zapytania. Jeżeli plany są bardzo podobne (bardzo niewielkie różnice są ignorowane), wartość `query_plan_hash` będzie dla nich taka sama.

Ograniczeniem algorytmów haszujących jest to, że mogą powodować kolizje, prawdopodobieństwo takiej kolizji jest jednak bardzo małe. Oznacza to, że dwa podobne zapytania mogą wygenerować dwie różne wartości `query_hash`, a dwa różne zapytania mogą wygenerować taką samą wartość `query_hash`, ale powtórzę jeszcze raz: prawdopodobieństwo jest bardzo niskie i nie powinno być brane pod uwagę.

Szukanie kosztownych zapytań

Wypróbujmy w praktyce koncepcje przedstawione w tym podrozdziale i skorzystajmy z widoku `sys.dm_exec_query_stats` do wyszukania najbardziej kosztownych zapytań w Twoim systemie. Typowe zapytanie do szukania najbardziej kosztownych pod względem wykorzystania procesora zapytań na podstawie magazynu planów zapytań pokazuję poniżej. Zauważ, że zapytanie grupuje na podstawie wartości `query_hash`, aby zagregować podobne zapytania, niezależnie od tego, czy są sparametryzowane.

```
SELECT TOP 20 query_stats.query_hash,
    SUM(query_stats.total_worker_time) / SUM(query_stats.execution_count)
    AS avg_cpu_time,
    MIN(query_stats.statement_text) AS statement_text
FROM
    (SELECT qs.*,
    SUBSTRING(st.text, (qs.statement_start_offset/2) + 1,
    ((CASE statement_end_offset
    WHEN -1 THEN DATALength(ST.text)
    ELSE qs.statement_end_offset END
    - qs.statement_start_offset)/2) + 1) AS statement_text
    FROM sys.dm_exec_query_stats qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st) AS query_stats
GROUP BY query_stats.query_hash
ORDER BY avg_cpu_time DESC
```

Możesz również zauważyć, że każdy zwracany wiersz reprezentuje jedno zapytanie w zestawie (np. zestaw pięciu zapytań w widoku `sys.dm_exec_query_stats` będzie repre-

zentowany przez pięć wierszy — zgodnie z wcześniejszymi wyjaśnieniami). Możemy zmodyfikować poprzednie zapytanie do poniższej postaci, aby skupić się na poziomie zestawu i planu. Zauważ, że nie ma potrzeby korzystania z kolumn `statement_start_offset` i `statement_end_offset` do oddzielenia poszczególnych zapytań i że tym razem grupujemy na podstawie wartości `query_plan_hash`.

```
SELECT TOP 20 query_plan_hash,
    SUM(total_worker_time) / SUM(execution_count) AS avg_cpu_time,
    MIN(plan_handle) AS plan_handle, MIN(text) AS query_text
FROM sys.dm_exec_query_stats qs
    CROSS APPLY sys.dm_exec_sql_text(qs.plan_handle) AS st
GROUP BY query_plan_hash
ORDER BY avg_cpu_time DESC
```

Te przykłady bazują na czasie procesora. Zatem w ten sam sposób możesz zaktualizować te zapytania, aby wyszukiwały inne zasoby dostępne w widoku `sys.dm_exec_query_stats`, takie jak liczba odczytów fizycznych, liczba zapisów logicznych, liczba odczytów logicznych, czas CLR i czas wykonywania.

Możemy również zaaplikować tę samą koncepcję, aby znaleźć najbardziej kosztowne spośród aktualnie wykonywanych zapytań, korzystając z widoku `sys.dm_exec_requests`, zgodnie z poniższym zapytaniem:

```
SELECT TOP 20 SUBSTRING(st.text, (er.statement_start_offset/2) + 1,
    ((CASE statement_end_offset
    WHEN -1
    THEN DATALength(st.text)
    ELSE
    er.statement_end_offset
    END
    - er.statement_start_offset)/2) + 1) AS statement_text
, *
FROM sys.dm_exec_requests er
    CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) st
ORDER BY total_elapsed_time DESC
```

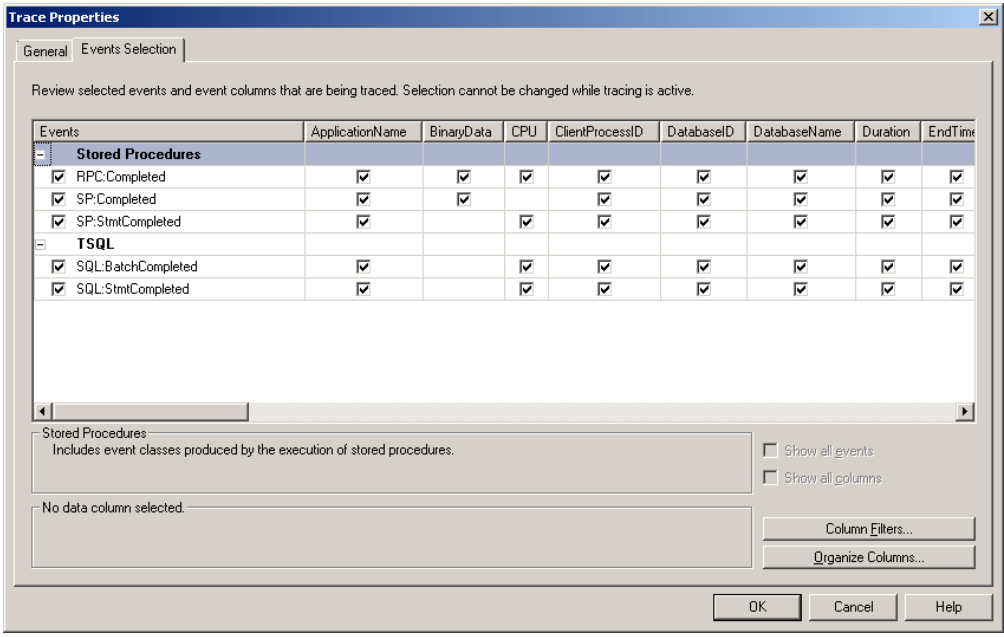
SQL Trace

SQL Trace jest funkcjonalnością, której możesz użyć do wyszukiwania problemów z wydajnością. Jest dostępna od dawna, więc jest dobrze znana wśród programistów i administratorów. Jak jednak wspomniałem w rozdziale 1., w wersji 2012 została ona zdeprecjonowana i zamiast niej Microsoft zaleca korzystanie ze zdarzeń rozszerzonych.

Chociaż za pomocą SQL Trace można śledzić bardzo wiele zdarzeń, w tym podrozdziale skupię się na tych, które pozwalają mierzyć wykorzystanie zasobów przez zapytania. Ponieważ uruchomienie śledzenia również zużywa zasoby, zazwyczaj należy uruchamiać je tylko wtedy, kiedy szuka się problemów z zapytaniem, czyli śledzenie nie powinno być włączone przez cały czas. Oto główne zdarzenia związane z wykorzystaniem zasobów:

Procedury przechowywane	RPC:Completed	Występuje w momencie zakończenia wywołania procedury zdalnej.
	SP:Completed	Występuje w momencie zakończenia działania procedury przechowywanej.
	SP:StmtCompleted	Wskazuje, że zapytanie wewnątrz procedury przechowywanej zostało zakończone.
T-SQL	SQL:BatchCompleted	Występuje, kiedy zakończony zostanie zestaw zapytań SQL.
	SQL:StmtCompleted	Występuje, kiedy zakończone zostanie polecenie SQL.

Rysunek 2.1 przedstawia przykład konfiguracji śledzenia w programie SQL Server Profiler. Zazwyczaj Profiler powinien być wykorzystywany do uruchamiania śledzenia na bardzo krótki okres. Jeżeli musisz uruchomić śledzenie na dłużej, na przykład na kilka godzin lub dni, śledzenie po stronie serwera byłoby lepsze, ponieważ wykorzystuje mniej zasobów. W rozdziale 1. pokazałem, jak w programie Profiler stworzyć skrypt i uruchomić go po stronie serwera.



Rysunek 2.1. Konfiguracja śledzenia z użyciem aplikacji SQL Server Profiler

Spójrzmy teraz, jak to działa. Uruchom program Profiler i wybierz poprzednie pięć zdarzeń. Uruchom śledzenie, a następnie w Management Studio uruchom poniższe zapytanie *ad hoc*:

```
SELECT * FROM Sales.SalesOrderDetail WHERE SalesOrderID = 60677
```

Uruchomienie tego zapytania wyzwoli następujące zdarzenia:

- ▶ SQL:StmtCompleted. SELECT * FROM Sales.SalesOrderDetail WHERE SalesOrderID = 60677
- ▶ SQL:BatchCompleted. SELECT * FROM Sales.SalesOrderDetail WHERE SalesOrderID = 60677

Jeżeli zobaczysz więcej zdarzeń, szukaj nazwy aplikacji „Microsoft SQL Server Management Studio — Query”. SQL Profiler udostępnia również możliwość filtrowania po wartości SPID.

Jeżeli natomiast stworzymy to samo zapytanie w ramach prostej procedury przechowywanej:

```
CREATE PROC test
AS
SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
```

a następnie je uruchomimy:

```
EXEC test
```

zobaczymy poniższe zdarzenia:

- ▶ SP:StmtCompleted. SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
- ▶ SP:Completed. SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
- ▶ SP:Completed. EXEC test
- ▶ SQL:StmtCompleted. EXEC test
- ▶ SQL:BatchCompleted. EXEC test

Tylko trzy pierwsze zdarzenia są związane z wykonaniem samej procedury przechowywanej. Ostatnie dwa związane są z wywołaniem zestawu zapytań z uruchomieniem procedury (EXEC).

Kiedy zatem możemy zobaczyć zdarzenie RPC:Completed? Do tego celu potrzebujemy wywołania procedury zdalnej (na przykład z użyciem aplikacji .NET). Do tego testu wykorzystamy kod z ramki *Kod C# do testu RPC*. Skompiluj ten kod i uruchom wynikowy plik wykonywalny. Ponieważ wywołujemy procedurę przechowywaną wewnątrz kodu C#, otrzymamy poniższe zdarzenia:

- ▶ SP:Completed. SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
- ▶ SP:StmtCompleted. SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
- ▶ SP:Completed. exec dbo.test
- ▶ RPC:Completed. exec dbo.test

Tym razem, jeżeli zobaczysz również inne zdarzenia, szukaj nazwy aplikacji „Net SqlClient Data Provider”.

KOD C# DO TESTU RPC

Chociaż omawianie kodu C# jest poza zakresem tej książki, na potrzeby testu możesz wykorzystać poniższy kod:

```
using System;
using System.Data;
using System.Data.SqlClient;
class Test
{
    static void Main()
    {
        SqlConnection cnn = null;
        SqlDataReader reader = null;
        try
        {
            cnn = new SqlConnection("Data Source=(SQLSERVER);Initial Catalog=
↳AdventureWorks2014;Integrated Security=SSPI");
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = cnn;
            cmd.CommandText = "dbo.test";
            cmd.CommandType = CommandType.StoredProcedure;
            cnn.Open();
            reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                Console.WriteLine(reader[0]);
            }
            return;
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            if (cnn != null)
            {
                if (cnn.State != ConnectionState.Closed)
                    cnn.Close();
            }
        }
    }
}
```

Aby skompilować powyższy kod, musisz uruchomić w konsoli następujące zapytanie:

```
csc test.cs
```

Nie potrzebujesz mieć zainstalowanego Visual Studio, lecz tylko bibliotekę .NET Framework, która jest konieczna do zainstalowania SQL Servera, więc będzie już w Twoim systemie. Być może będziesz musiał znaleźć program CSC, jeżeli nie

została dodana do zmiennej systemowej PATH — zazwyczaj znajduje się w katalogu *C:\Windows\Microsoft.NET*. Być może będziesz także musiał zmienić parametry połączenia, ponieważ skrypt zakłada, że łączysz się z domyślną instancją SQL Servera z użyciem uwierzytelnienia systemu Windows.

Zdarzenia rozszerzone

W rozdziale 1. pokrótce omówiłem zdarzenia rozszerzone, pokazując, jak przechwycić plany wykonania. W tym podrozdziale zawarłem więcej informacji o tej funkcjonalności wprowadzonej w SQL Serverze 2008. Jest ku temu jeszcze jeden ważny powód: od wersji 2012 SQL Trace jest zdeprecjonowane, przez co zdarzenia rozszerzone są teraz domyślnym narzędziem do diagnostyki w SQL Serverze. Chociaż dokładne omówienie zdarzeń rozszerzonych wykracza poza zakres niniejszej książki, ten podrozdział powinien dać Ci wystarczającą wiedzę, aby rozpocząć pracę z tą funkcjonalnością w celu wyszukiwania problemów w działaniu zapytań.

Wprowadzam tu podstawowe koncepcje dotyczące zdarzeń rozszerzonych, włączając w to zdarzenia, predykaty, akcje, cele i sesje. Chociaż przykłady w tej książce pokazują głównie za pomocą kodu, uważam, że warto również pokazać nowy interfejs dla zdarzeń rozszerzonych, który będzie użyteczny, jeżeli dopiero zaczynasz przygodę z tą technologią, i który umożliwia tworzenie skryptów w podobny sposób, w jaki SQL Profiler pozwalał tworzyć skrypty dla śledzenia po stronie serwera. Interfejs ten został wprowadzony w SQL Serverze 2012.

Jedną z moich ulubionych czynności z SQL Trace — a teraz także ze zdarzeniami rozszerzonymi — jest nauka tego, jak działają narzędzia SQL Servera. Możesz uruchomić śledzenie na swojej instancji i uruchomić dowolne z narzędzi, aby przechwycić wszystkie polecenie T-SQL przesyłane z narzędzia do serwera. Co ciekawe, interfejs dla zdarzeń rozszerzonych nie jest wyjątkiem, a śledząc jego zapytania, możesz się dowiedzieć, skąd pochodzą informacje. Oczywiście na razie nie musisz się tym martwić, ponieważ zaraz pokażę Ci kilka widoków DMV dla zdarzeń rozszerzonych.

Zdarzenia rozszerzone zaprojektowano tak, aby miały niewielki wpływ na wydajność serwera. Odnoszą się do dobrze znanych punktów w kodzie serwera SQL, kiedy więc wykonywane jest jakieś zadanie, SQL Server dokona szybkiego sprawdzenia, czy nie ma sesji skonfigurowanych na nasłuchiwanie tego zdarzenia. Jeżeli nie ma aktywnych sesji, zdarzenie nie zostanie odpalone, a zadanie będzie kontynuowane bez żadnego narzutu w wydajności. Jeżeli jednak są aktywne sesje z włączonym zdarzeniem, SQL Server rozpocznie zbieranie danych z nim związanych. Jeśli predykat zostanie rozwiązany negatywnie, zadanie będzie kontynuowane z minimalnym narzutem. Jeśli predykat zostanie spełniony, akcje zdefiniowane w sesji zostaną wykonane. Wszystkie dane będą zbierane przez zdefiniowane cele do późniejszej analizy.

Poniższe zapytanie możesz wykorzystać do znalezienia listy zdarzeń dostępnych w aktualnej wersji SQL Servera. Możesz tworzyć zdarzenia rozszerzone, wybierając jedno lub więcej z spośród poniższych zdarzeń:

```
SELECT name, description
FROM sys.dm_xe_objects
WHERE object_type = 'event' AND
(capabilities & 1 = 0 OR capabilities IS NULL)
ORDER BY name
```

Dla tej wersji SQL Servera zwracane jest 870 zdarzeń, włączając w to `sp_statement_completed`, `sql_batch_completed` i `sql_statement_completed`, które omówimy w dalszej części.

Każde zdarzenie ma zestaw kolumn, które możesz wyświetlić, używając widoku `sys.dm_xe_object_columns`, zgodnie z poniższym kodem:

```
SELECT o.name, c.name as column_name, c.description
FROM sys.dm_xe_objects o
JOIN sys.dm_xe_object_columns c
ON o.name = c.object_name
WHERE object_type = 'event' AND
c.column_type <> 'readonly' AND
(o.capabilities & 1 = 0 OR o.capabilities IS NULL)
ORDER BY o.name, c.name
```

Akcja to programistyczna odpowiedź na zdarzenie, która udostępnia możliwość uruchomienia dodatkowego kodu. Chociaż możesz korzystać z akcji do wykonywania operacji takich jak przechwycenie rzutu stosu lub wstawienie pułapki debugera w kodzie SQL Servera, najprawdopodobniej będą one wykorzystywane do przechwytywania pól globalnych, które są wspólne dla wszystkich zdarzeń, takich jak `plan_handle` i `database_name`. Akcje są również uruchamiane synchronicznie. Możesz uruchomić poniższy kod do wyświetlenia listy dostępnych akcji:

```
SELECT name, description
FROM sys.dm_xe_objects
WHERE object_type = 'action' AND
(capabilities & 1 = 0 OR capabilities IS NULL)
ORDER BY name
```

Predykaty są wykorzystywane do ograniczenia danych, które chcesz przechwycić, a filtrować możesz na podstawie kolumn z danymi lub dowolnych danych globalnych zwracanych przez poniższe zapytanie:

```
SELECT name, description
FROM sys.dm_xe_objects
WHERE object_type = 'pred_source' AND
(capabilities & 1 = 0 OR capabilities IS NULL)
ORDER BY name
```

Zapytanie zwraca 44 rekordy, włączając w to `database_id`, `session_id` i `query_hash`. Predykaty to wyrażenia boolowskie, które mogą przyjmować wartość prawda lub fałsz, wspierają również szybkie określanie, gdzie całe wyrażenie będzie fałszywe, gdy któryś z jego predykatów będzie fałszywy.

Wreszcie możesz skorzystać z *celów* do określenia, w jaki sposób chcesz zbierać dane do analizy; na przykład, możesz przechowywać dane zdarzeń w pliku lub w buforze pierścieniowym (bufor pierścieniowy to struktura danych przez krótki czas przechowująca dane zdarzeń w pamięci w sposób zapętłony). Te cele nazywają się odpowiednio `event_file` i `ring_buffer`. Mogą odbierać dane ze zdarzeń zarówno synchronicznie, jak i asynchronicznie, a każdy cel może odbierać każde zdarzenie. Możesz wylistować sześć dostępnych celów za pomocą poniższego zapytania:

```
SELECT name, description
FROM sys.dm_xe_objects
WHERE object_type = 'target' AND
(capabilities & 1 = 0 OR capabilities IS NULL)
ORDER BY name
```

Jak korzystać z tych elementów, omówię w dalszej części tego rozdziału, ale najpierw pokażę, jak znaleźć nazwy zdarzeń, które możesz już znać z mechanizmu SQL Trace.

Mapowanie zdarzeń SQL Trace na zdarzenia rozszerzone

Prawdopodobnie znasz już niektóre zdarzenia SQL Trace lub nawet masz w swoim systemie skonfigurowane śledzenia. Możesz skorzystać z tabeli systemowej zdarzeń rozszerzonych `sys.trace_xe_event_map`, aby uzyskać pomoc w mapowaniu klas zdarzeń SQL Trace na zdarzenia rozszerzone. Tabela `sys.trace_xe_event_map` zawiera jeden wiersz dla każdego zdarzenia rozszerzonego zmapowanego na klasę zdarzeń SQL Trace. Aby sprawdzić, jak to działa, uruchom poniższe zapytanie:

```
SELECT te.trace_event_id, name, package_name, xe_event_name
FROM sys.trace_events te
JOIN sys.trace_xe_event_map txe ON te.trace_event_id = txe.trace_event_id
WHERE te.trace_event_id IS NOT NULL
ORDER BY name
```

Zapytanie zwraca 138 rekordów — niektóre z nich znajdują się w tabeli na kolejnej stronie.

Ponadto możesz wykorzystać tabelę systemową `sys.trace_xe_event_map` w połączeniu z funkcją `sys.fn_trace_geteventinfo` do mapowania zdarzeń skonfigurowanych w istniejącym śledzeniu na zdarzenia rozszerzone. Funkcja `sys.fn_trace_geteventinfo` zwraca informacje o aktualnie działającym śledzeniu i wymaga podania identyfikatora śledzenia. Aby przetestować jej działanie, uruchom śledzenie (wyjaśnione wcześniej)

trace_event_id	name	package_name	xe_event_name
196	Assembly Load	sqlserver	assembly_load
16	Attention	sqlserver	attention
14	Audit Login	sqlserver	login
15	Audit Logout	sqlserver	logout
18	Audit Server Starts And Stops	sqlserver	server_start_stop
58	Auto Stats	sqlserver	auto_stats
193	Background Job Error	sqlserver	background_job_error
212	Bitmap Warning	sqlserver	bitmap_disabled_warning
137	Blocked Process Report	sqlserver	blocked_process_report

i uruchom poniższe zapytanie, by uzyskać identyfikator śledzenia. Śledzenie z `trace_id = 1` to zazwyczaj domyślne śledzenie. Możesz z łatwością zidentyfikować swój proces, patrząc na kolumnę `path` — jeżeli uruchomiłeś śledzenie z programu SQL Profiler, w kolumnie tej będzie wartość `NULL`.

```
SELECT * FROM sys.traces
```

Kiedy masz już identyfikator, możesz uruchomić poniższy kod, który w moim przypadku wykorzystuje identyfikator śledzenia równy 2 (w funkcji `sys.fn_trace_geteventinfo`):

```
SELECT te.trace_event_id, name, package_name, xe_event_name
FROM sys.trace_events te
JOIN sys.trace_xe_event_map txe ON te.trace_event_id = txe.trace_event_id
WHERE te.trace_event_id IN (
  SELECT DISTINCT(eventid) FROM sys.fn_trace_geteventinfo(2))
ORDER BY name
```

Jeżeli uruchomimy śledzenie, które skonfigurowaliśmy wcześniej, otrzymamy następujący wynik:

trace_event_id	name	package_name	xe_event_name
10	RPC:Completed	sqlserver	rpc_completed
43	SP:Completed	sqlserver	module_end
42	SP:StmtCompleted	sqlserver	sp_statement_completed
12	SQL:BatchCompleted	sqlserver	sql_batch_completed
41	SQL:StmtCompleted	sqlserver	sql_statement_completed

Jak widzisz, nazwy zdarzeń naszego śledzenia SQL Trace są bardzo podobne do odpowiedników w zdarzeniach rozszerzonych, poza `SP:Completed`, dla którego odpowiednik ma nazwę `module_end`. Oto definicje tych zdarzeń:

rpc_completed	Występuje w momencie zakończenia wywołania procedury zdalnej.
module_end	Występuje w momencie zakończenia działania procedury przechowywanej.
sp_statement_completed	Wskazuje, że zapytanie wewnątrz procedury przechowywanej zostało zakończone.
sql_batch_completed	Występuje, kiedy zakończony zostanie zestaw zapytań SQL.
sql_statement_completed	Występuje, kiedy zakończone zostanie polecenie SQL.



UWAGA

SQL Server 2014 zawiera kilka szablonów zdarzeń rozszerzonych. Jeden z nich, *Query Detail Sampling* (samplowanie szczegółów zapytania), zbiera dokładne informacje dotyczące polecenia i błędów, które zawierają pięć powyższych zdarzeń i zdarzenie `error_reported`. Ma również kilka predefiniowanych akcji i predykatów, a jako cel wykorzystuje `ring_buffer`. Domyślne predykaty powodują zbieranie tylko 20% aktywnych sesji na serwerze w dowolnym momencie, więc być może jest to coś, co będziesz chciał zmienić.

Tworzenie sesji

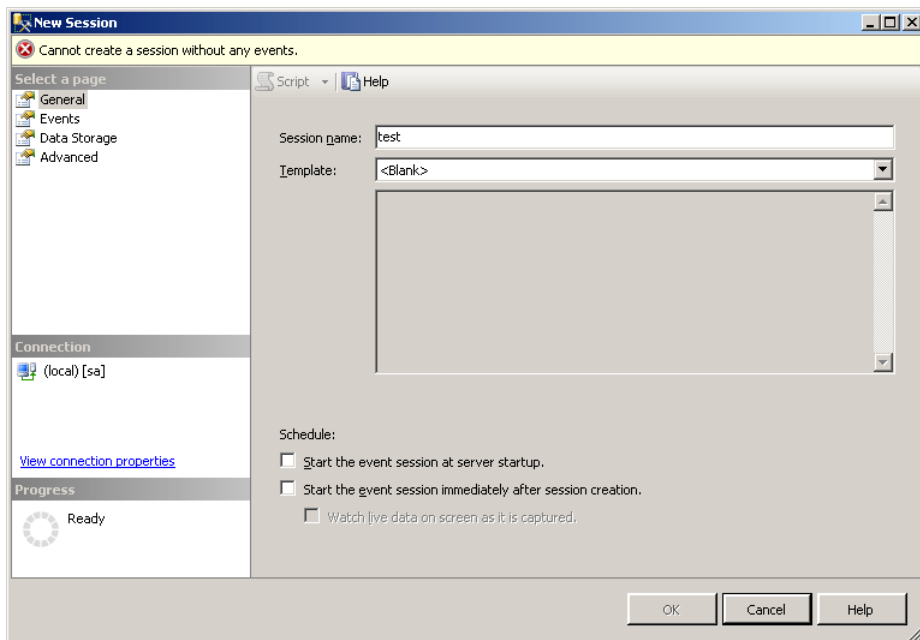
Mamy już dość informacji, aby stworzyć sesję zdarzeń rozszerzonych. Zdarzenia rozszerzone udostępniają wiele poleceń DDL pozwalających pracować z sesjami, takich jak `CREATE EVENT SESSION`, `ALTER EVENT SESSION` i `DROP EVENT SESSION`. Najpierw jednak pokażę, jak stworzyć sesję z użyciem interfejsu graficznego, za pomocą którego możesz z łatwością tworzyć sesje i nimi zarządzać, ale także generować skrypty. Aby rozpocząć, w Management Studio rozwiń węzły *Management* (zarządzanie) i *Extended Events* (zdarzenia rozszerzone), a następnie kliknij prawym przyciskiem węzeł *Sessions* (sesje).



UWAGA

Rozwijając węzeł *Sessions*, zobaczysz dwie sesje zdarzeń rozszerzonych domyślnie zdefiniowanych jako `AlwaysOn_health` i `system_health`. Sesja `system_health` jest uruchamiana domyślnie przy starcie SQL Studio i wykorzystywana do zbierania kilku predefiniowanych zdarzeń pozwalających szukać problemów z wydajnością. `AlwaysOn_health`, domyślnie wyłączona, to sesja zaprojektowana do monitorowania grup dostępności (*Availability Groups*), która to funkcjonalność została wprowadzona w SQL Serverze 2012. Możesz podglądać zdarzenia, akcje, predykaty, cele i konfiguracje zdefiniowane w tych sesjach, zaglądając do ich właściwości lub poprzez stworzenie ich skryptu. Aby w Management Studio stworzyć skrypt dla sesji, kliknij na niej prawym przyciskiem myszy i wybierz *Script Session As CREATE To*.

Możesz wybrać opcje *New Session Wizard* (konfigurator nowej sesji) i *New Session* (nowa sesja). W tym podrozdziale z grubsza omawiam opcję nowej sesji. Kiedy ją wybierzesz, zobaczysz okno podobne do tego z rysunku 2.2, z czterema różnymi stronami: *General* (ogólne), *Events* (zdarzenia), *Data Storage* (magazyn danych) i *Advanced* (zaawansowane).

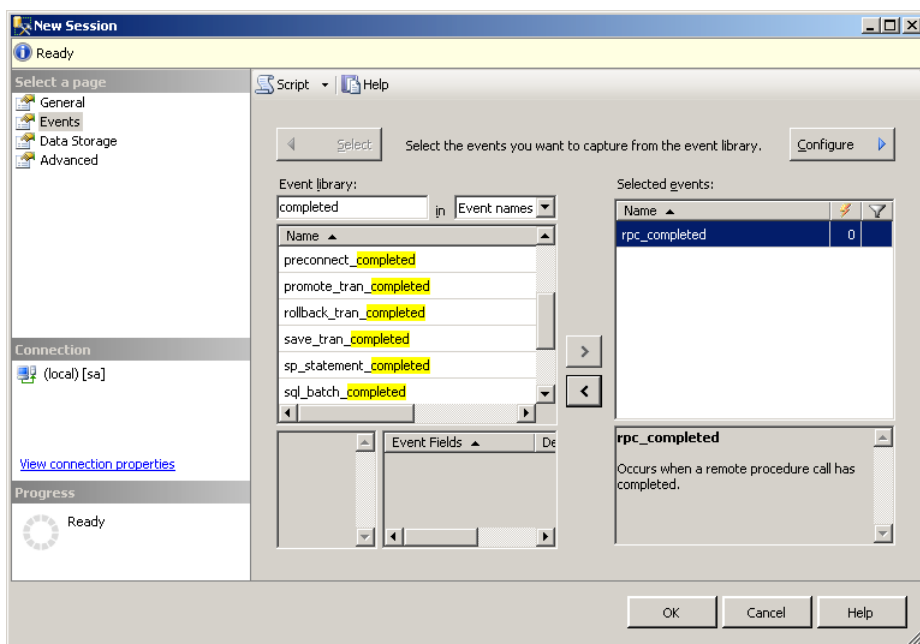


Rysunek 2.2. Strona General okna nowej sesji

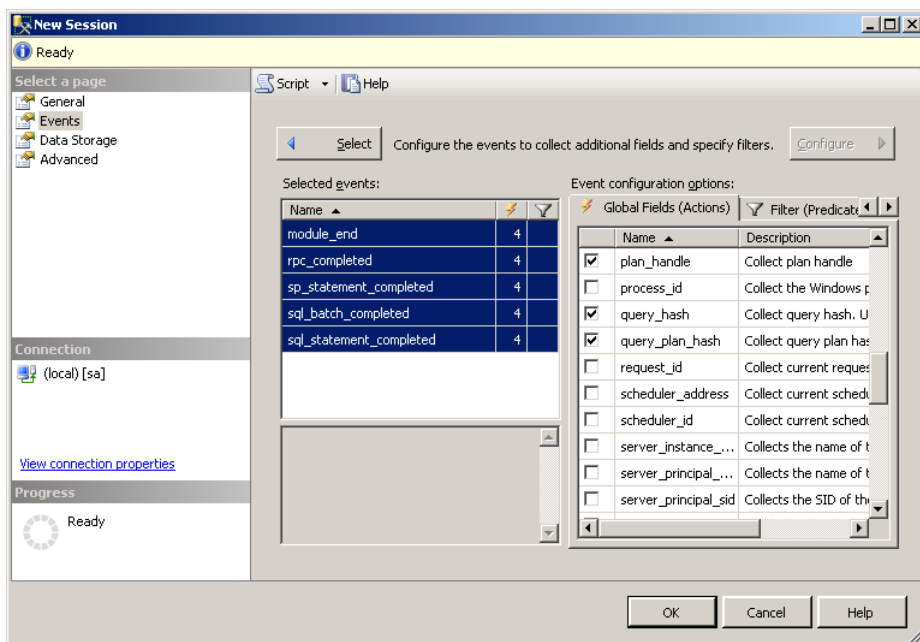
Nazwij sesję *Test* i kliknij stronę *Events* po lewej.

Strona *Events* pozwala wybrać zdarzenia dla sesji. Ponieważ ta strona może zawierać bardzo wiele informacji, możesz zechcieć zmaksymalizować okno. Szukanie zdarzeń w bibliotece zdarzeń jest możliwe; na przykład, ponieważ cztery z pięciu zdarzeń, których szukamy, zawierają słowo *completed*, możesz wpisać to słowo, aby wyszukać te zdarzenia (zobacz rysunek 2.3).

Kliknij przycisk ➤, aby wybrać zdarzenia *rpc_completed*, *sp_statement_completed*, *sql_batch_completed* i *sql_statement_completed*. Wykonaj podobne wyszukiwanie i dodaj zdarzenie *module_end*. Strona *Events* pozwala również zdefiniować akcje i predykaty (filtry) po kliknięciu przycisku *Configure* (konfiguracja), co spowoduje pokazanie opcji konfiguracji zdarzeń. Kliknij przycisk *Configure* i zaznacz pola dla następujących akcji w zakładce *Global Fields (Actions)* (pola globalne — akcje): *plan_handle*, *query_hash*, *query_plan_hash* i *sql_text*. Po tych czynnościach okno powinno wyglądać jak na rysunku 2.4.

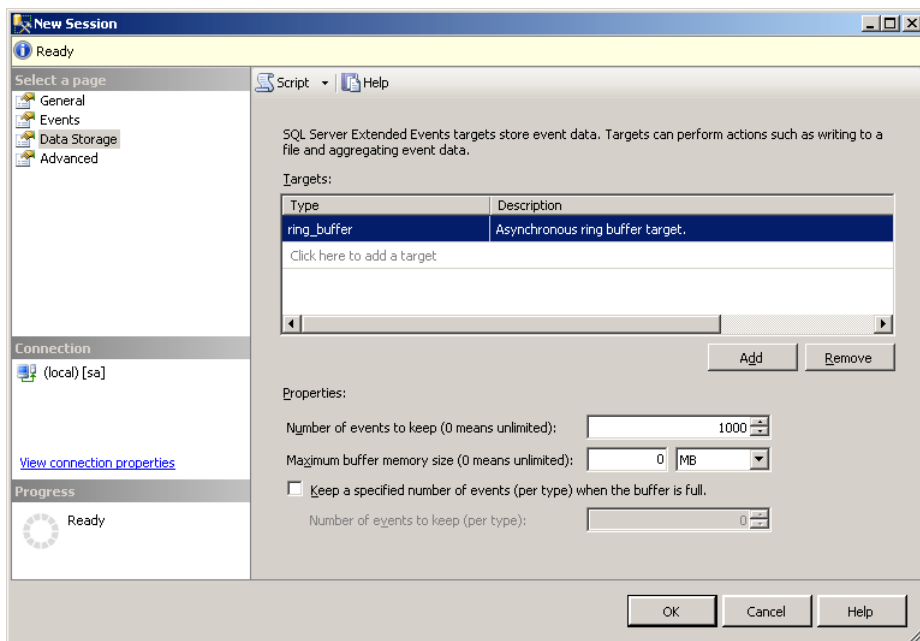


Rysunek 2.3. Strona Events okna nowej sesji



Rysunek 2.4. Opcje konfiguracji zdarzenia na stronie Events

Kliknięcie strony *Data Storage* pozwoli wybrać jeden lub więcej celów do zbierania danych. Wybierz cel *ring_buffer*, zgodnie z rysunkiem 2.5.

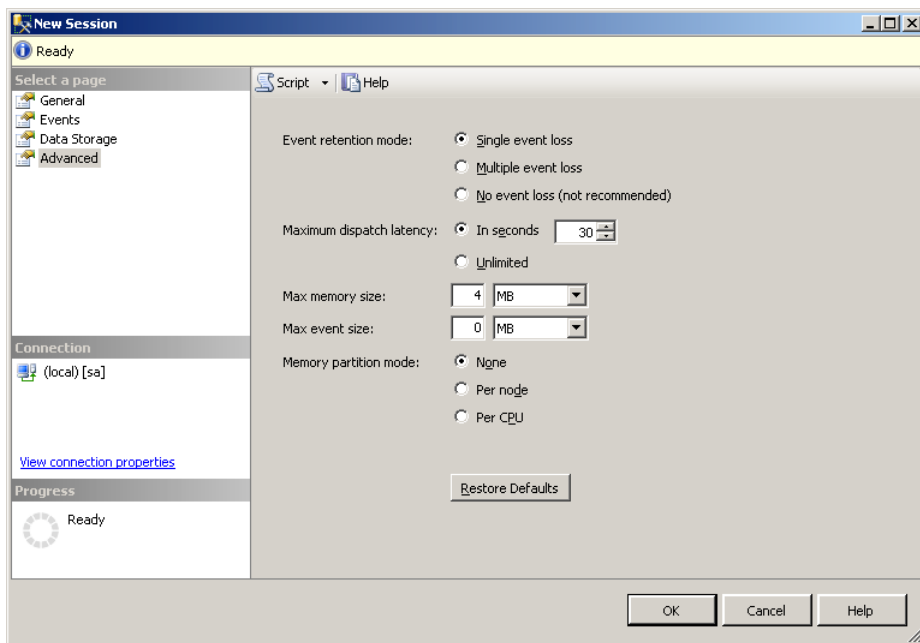


Rysunek 2.5. Strona Data Storage okna nowej sesji

Strona *Advanced* została pokazana na rysunku 2.6. Pozwoli Ci wyszczególnić dodatkowe opcje do wykorzystania w sesji zdarzeń rozszerzonych.

W końcu, jak zazwyczaj w Management Studio, możesz wygenerować skrypt na podstawie wybranych opcji, klikając ikonę *Script* (skrypt) na górze okna nowej sesji. Kontynuując nasz przykład, wykorzystaj poniższy skrypt do utworzenia sesji zdarzeń rozszerzonych:

```
CREATE EVENT SESSION test ON SERVER
ADD EVENT sqlserver.module_end(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.rpc_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.sp_statement_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.sql_batch_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
```



Rysunek 2.6. Strona Advanced okna nowej sesji

```
ADD EVENT sqlserver.sql_statement_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text))
ADD TARGET package0.ring_buffer
WITH (STARTUP_STATE=OFF)
```

Jak pokazałem w rozdziale 1., musimy również uruchomić sesję za pomocą poniższego polecenia:

```
ALTER EVENT SESSION [test]
ON SERVER
STATE=START
```

Od tego momentu nasza sesja jest aktywna, a my musimy tylko poczekać na pojawienie się zdarzeń. Aby przetestować sesję, wykonaj poniższe polecenia:

```
SELECT * FROM Sales.SalesOrderDetail WHERE SalesOrderID = 60677
GO
SELECT * FROM Person.Address WHERE AddressID = 21
GO
SELECT * FROM HumanResources.Employee WHERE BusinessEntityID = 229
GO
```

Po przechwyceniu zdarzeń możesz chcieć przeczytać i przeanalizować te dane. Do przeglądania i analizy danych przechwyconych przez sesję zdarzeń rozszerzonych możesz użyć funkcjonalności Watch Live Data wprowadzoną w SQL Serverze 2012

(mechanizm ten przybliżyłem w rozdziale 1.). Możesz również skorzystać z XQuery do odczytu danych z dowolnego celu. Aby odczytać przechwycone dane, zastosuj następujące zapytanie:

```
SELECT name, target_name, execution_count, CAST(target_data AS xml)
AS target_data
FROM sys.dm_xe_sessions s
JOIN sys.dm_xe_session_targets t
ON s.address = t.event_session_address
WHERE s.name = 'test'
```

Otrzymasz wynik podobny do poniższego:

name	target_name	execution_count	target_data
test	ring_buffer	68	<RingBufferTarget truncated="1" ↳processingTime="267"...

Po kliknięciu linku zobaczysz przechwycone dane w formacie XML. Ponieważ plik XML byłby zbyt duży, aby pokazać go w książce, poniżej zamieściłem tylko mały fragment pierwszego przechwyconego zdarzenia pokazujący wartości `cpu_time`, `duration`, `physical_reads`, `logical_reads` i `writes`.

```
<RingBufferTarget truncated="0" processingTime="0" totalEventsProcessed="12" eventCount="12"
↳droppedCount="0" memoryUsed="5810">
  <event name="sql_batch_completed" package="sqlserver" timestamp="2013-05-04T20:08:42.240Z">
    <data name="cpu_time">
      <type name="uint64" package="package0" />
      <value>0</value>
    </data>
    <data name="duration">
      <type name="uint64" package="package0" />
      <value>1731</value>
    </data>
    <data name="physical_reads">
      <type name="uint64" package="package0" />
      <value>0</value>
    </data>
    <data name="logical_reads">
      <type name="uint64" package="package0" />
      <value>4</value>
    </data>
    <data name="writes">
      <type name="uint64" package="package0" />
      <value>0</value>
    </data>
```

Ponieważ jednak bezpośrednie czytanie dokumentów XML nie jest zbyt wygodne, do wyodrębnienia danych z dokumentu XML możemy wykorzystać XQuery:

```
SELECT
  event_data.value('(event/@name)[1]', 'varchar(50)') AS event_name,
  event_data.value('(event/action[@name="query_hash"]/value)[1]', 'varchar(max)')
  AS query_hash,
```

```

event_data.value('(event/data[@name="cpu_time"]/value)[1]', 'int')
    AS cpu_time,
event_data.value('(event/data[@name="duration"]/value)[1]', 'int')
    AS duration,
event_data.value('(event/data[@name="logical_reads"]/value)[1]', 'int')
    AS logical_reads,
event_data.value('(event/data[@name="physical_reads"]/value)[1]', 'int')
    AS physical_reads,
event_data.value('(event/data[@name="writes"]/value)[1]', 'int') AS writes,
event_data.value('(event/data[@name="statement"]/value)[1]', 'varchar(max)')
    AS statement
FROM(SELECT evnt.query('.') AS event_data
    FROM
        (SELECT CAST(target_data AS xml) AS target_data
        FROM sys.dm_xe_sessions s
        JOIN sys.dm_xe_session_targets t
        ON s.address = t.event_session_address
        WHERE s.name = 'test'
        AND t.target_name = 'ring_buffer'
        ) AS data
    CROSS APPLY target_data.nodes('RingBufferTarget/event') AS xevent(evnt)
    ) AS xevent(event_data)

```

Otrzymasz wynik podobny do poniższego:

event_name	cpu_time	duration	logical_reads	physical_reads	writes	statement
sql_statement_completed	0	42522	3	24	0	SELECT * FROM Sales. ↳SalesOrderDetail WHERE SalesOrderID = 60677
sql_batch_completed	0	80265	13	48	0	NULL
sql_statement_completed	0	23970	2	16	0	SELECT * FROM Person. ↳Address WHERE AddressID = 21
sql_batch_completed	0	53971	4	24	0	NULL
sql_statement_completed	0	29976	2	16	0	SELECT * FROM HumanResources. ↳Employee WHERE BusinessEntityID = 229
sql_batch_completed	0	64189	20	31	0	NULL

Korzystając z tego zapytania, otrzymujesz wszystkie dane, ale czasami możesz chcieć je zagregować. Na przykład sortowanie danych w poszukiwaniu najwyższego wykorzystania procesora może nie być wystarczające. Jak zobaczysz również w innych fragmentach tej książki, niekiedy powodem problemu wydajnościowego może być

zapytanie, które nawet jeżeli nie pokazuje się pośród 10 zapytań najbardziej obciążających procesor, jest wykonywane tak wiele razy, że sumaryczne wykorzystanie procesora byłoby jednym z najwyższych. Możesz zmodyfikować poprzednie zapytanie, aby bezpośrednio agregować dane, możesz też zmienić je tak, aby dane zapisywane były w tabeli tymczasowej (na przykład za pomocą polecenia `SELECT ... INTO`), a następnie grupować po kolumnie `query_hash` i być może sortować, tak jak poniżej:

```
SELECT query_hash, SUM(cpu_time) AS cpu_time, SUM(duration) AS duration,
       SUM(logical_reads) AS logical_reads, SUM(physical_reads) AS physical_reads,
       SUM(writes) AS writes, MAX(statement) AS statement
FROM #eventdata
GROUP BY query_hash
```

Jeszcze raz, jak wspomniałem w rozdziale 1., po zakończonych testach musisz zatrzymać i usunąć sesję zdarzeń. Uruchom poniższe zapytania:

```
ALTER EVENT SESSION [test]
ON SERVER
STATE=STOP
GO
DROP EVENT SESSION [test] ON SERVER
```

W przypadku zbierania dużych ilości danych lub jeżeli sesja ma działać przez dłuższy czas, możesz również jako cel wybrać plik. Poniższy przykład jest identyczny jak poprzednio, ale jako cel wykorzystany został plik (zwróć uwagę na definicję pliku w katalogu `C:\Data`):

```
CREATE EVENT SESSION test ON SERVER
ADD EVENT sqlserver.module_end(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.rpc_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.sp_statement_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.sql_batch_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text)),
ADD EVENT sqlserver.sql_statement_completed(
    ACTION(sqlserver.plan_handle,sqlserver.query_hash,sqlserver.query_plan_hash,
    sqlserver.sql_text))
ADD TARGET package0.event_file(SET filename=N'C:\Data\test.xel')
WITH (STARTUP_STATE=OFF)
```

Po uruchomieniu sesji i przechwyceniu kilku zdarzeń możesz przejrzeć dane za pomocą poniższego zapytania i funkcji `sys.fn_xe_file_target_read_file`:

```
SELECT
    event_data.value('(event/@name)[1]', 'varchar(50)') AS event_name,
    event_data.value('(event/action[@name="query_hash"]/value)[1]',
    'varchar(max)') AS query_hash,
    event_data.value('(event/data[@name="cpu_time"]/value)[1]', 'int')
```



```

    AS cpu_time,
event_data.value('(event/data[@name="duration"]/value)[1]', 'int')
    AS duration,
event_data.value('(event/data[@name="logical_reads"]/value)[1]', 'int')
    AS logical_reads,
event_data.value('(event/data[@name="physical_reads"]/value)[1]', 'int')
    AS physical_reads,
event_data.value('(event/data[@name="writes"]/value)[1]', 'int') AS writes,
event_data.value('(event/data[@name="statement"]/value)[1]', 'varchar(max)')
    AS statement
FROM
(
    SELECT CAST(event_data AS xml)
    FROM sys.fn_xe_file_target_read_file
    (
        'C:\Data\test*.xel',
        NULL,
        NULL,
        NULL
    )
) AS xevent(event_data)

```

Jeżeli zajrzysz do katalogu *C:\Data*, znajdziesz tam plik o nazwie zbliżonej do *test_0_130133932321310000.xel*. SQL Server dodaje do podanej nazwy ciąg „_0_” i liczbę całkowitą reprezentującą liczbę milisekund od 1 stycznia 1600 roku. Możesz przejrzeć zawartość danego pliku poprzez podanie jego nazwy lub wykorzystać maskę (np. znak * pokazany w kodzie) do przeglądania zawartości wszystkich pasujących plików. Jeżeli interesują Cię dodatkowe informacje na temat wykorzystania plików jako celów sesji zdarzeń rozszerzonych, znajdziesz je w dokumentacji. Nie zapomnij też o zatrzymaniu i usunięciu sesji po zakończeniu testów.

Na koniec pokazuję, jak za pomocą zdarzeń rozszerzonych uzyskać czasy oczekiwania dla konkretnego zapytania, a jest to coś, co przed pojawieniem się zdarzeń rozszerzonych nie było w ogóle możliwe. Musisz wykorzystać zdarzenie *wait_info* i wybrać jedno z wielu dostępnych pól (takich jak *username* (nazwa użytkownika) bądź *query_hash*) lub wybranych akcji w celu ustawienia filtra (lub predykatu). W tym przykładzie wykorzystałem pole *session_id* (identyfikator sesji). Jeżeli będziesz uruchamiać testowy kod, pamiętaj o zmianie wartości *session_id* na odpowiednią dla siebie.

```

CREATE EVENT SESSION [test] ON SERVER
ADD EVENT sqllos.wait_info(
WHERE ([sqlserver].[session_id]=(61)))
ADD TARGET package0.ring_buffer
WITH (STARTUP_STATE=OFF)
GO

```

Uruchomienie zdarzenia:

```

ALTER EVENT SESSION [test]
ON SERVER
STATE=START

```

Wykonaj kilka transakcji, ale pamiętaj, że muszą być wykonywane w ramach sesji o podanym identyfikatorze (i muszą generować czas oczekiwania). Na przykład uruchom poniższe zapytanie:

```
SELECT * FROM Production.Product p1 CROSS JOIN
Production.Product p2
```

Możesz teraz odczytać przechwycone dane:

```
SELECT
    event_data.value(' (event/@name)[1]', 'varchar(50)') AS event_name,
    event_data.value(' (event/data[@name="wait_type"]/text)[1]', 'varchar(40)')
        AS wait_type,
    event_data.value(' (event/data[@name="duration"]/value)[1]', 'int')
        AS duration,
    event_data.value(' (event/data[@name="opcode"]/text)[1]', 'varchar(40)')
        AS opcode,
    event_data.value(' (event/data[@name="signal_duration"]/value)[1]', 'int')
        AS signal_duration
FROM (SELECT evnt.query('.') AS event_data
      FROM
        (SELECT CAST(target_data AS xml) AS target_data
         FROM sys.dm_xe_sessions s
         JOIN sys.dm_xe_session_targets t
           ON s.address = t.event_session_address
         WHERE s.name = 'test'
              AND t.target_name = 'ring_buffer'
        ) AS data
      CROSS APPLY target_data.nodes('RingBufferTarget/event') AS xevent(evnt)
      ) AS xevent(event_data)
```

Oto wynik, jaki otrzymałem w swoim systemie:

event_name	wait_type	wait_type	opcode	signal_duration
wait_info	NETWORK_IO	0	Begin	0
wait_info	NETWORK_IO	0	End	0
wait_info	NETWORK_IO	0	Begin	0
wait_info	NETWORK_IO	0	End	0

Jest to kolejny przykład, w którym agregacja danych jest przydatna. Tak jak poprzednio, nie zapomnij o zatrzymaniu i usunięciu sesji.

Data Collector

Mogą zdarzyć się przypadki, w których występuje problem z wydajnością i brak jest informacji pozwalających zdiagnozować problem. Na przykład możesz otrzymać informację, że procesor jest przez kilka minut wykorzystywany w 100%, co powoduje spowolnienie Twojej aplikacji, ale zanim połączysz się z systemem w celu znale-

zienia problemu, problem może już zniknąć. Często konkretne problemy są trudne do odtworzenia, a jedyną opcją jest uruchomienie śledzenia lub jakiegoś innego sposobu zbierania danych i czekanie, aż problem pojawi się ponownie. W takim przypadku proaktywne zbieranie danych związanych z wydajnością jest bardzo ważne, a Data Collector, funkcjonalność wprowadzona w SQL Serverze 2008, może ci w tym bardzo pomóc. Data Collector pozwala zbierać dane związane z wydajnością, z których możesz skorzystać natychmiast po tym, jak wystąpi problem. Musisz znać tylko czas wystąpienia problemu i przejrzeć dane zebrane w okolicach tego czasu.

Omówienie funkcjonalności Data Collector zajęłoby cały rozdział, a może nawet całą książkę. Dlatego w tym rozdziale pokażę tylko, jak zacząć z nim pracę. Więcej informacji znajdziesz w dokumentacji Books Online lub w artykule *Using Management Data Warehouse for Performance Monitoring* autorstwa Kena Lassena.

Konfiguracja

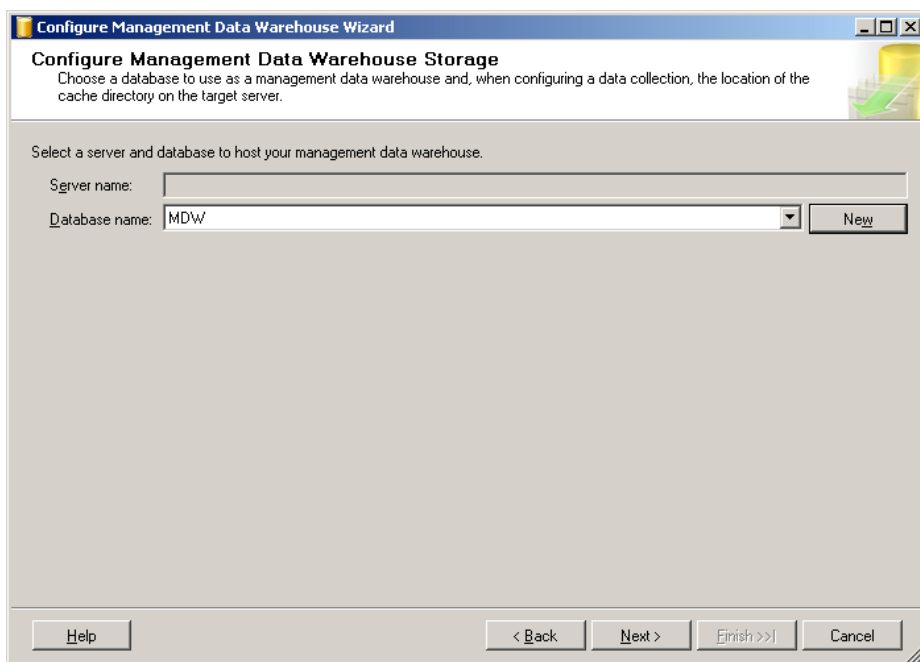
Domyślnie Data Collector jest po instalacji nowego serwera wyłączony. Aby go skonfigurować, musisz przejść proces składający się z dwóch kroków. Aby skonfigurować pierwszą część, w Management Studio rozwiń katalog *Management*, kliknij prawym przyciskiem węzeł *Data Collection* (zbieranie danych) i wybierz *Tasks* (zadania), a następnie *Configure Management Data Warehouse* (konfiguruj magazyn danych zarządzania). Zostaniesz przeniesiony do okna *Configure Management Data Warehouse Storage* (zobacz rysunek 2.7). Na tym ekranie możesz wybrać bazę, która zostanie wykorzystana do zbierania danych. Opcjonalnie, klikając przycisk *New*, możesz stworzyć nową bazę.

Wybierz istniejącą bazę lub utwórz nową i kliknij przycisk *Next* (następny). Kolejny ekran, *Map Logins and Users* (mapowanie loginów i użytkowników), pokazany na rysunku 2.8, pozwala mapować loginy i użytkowników na role magazynu danych zarządzania.

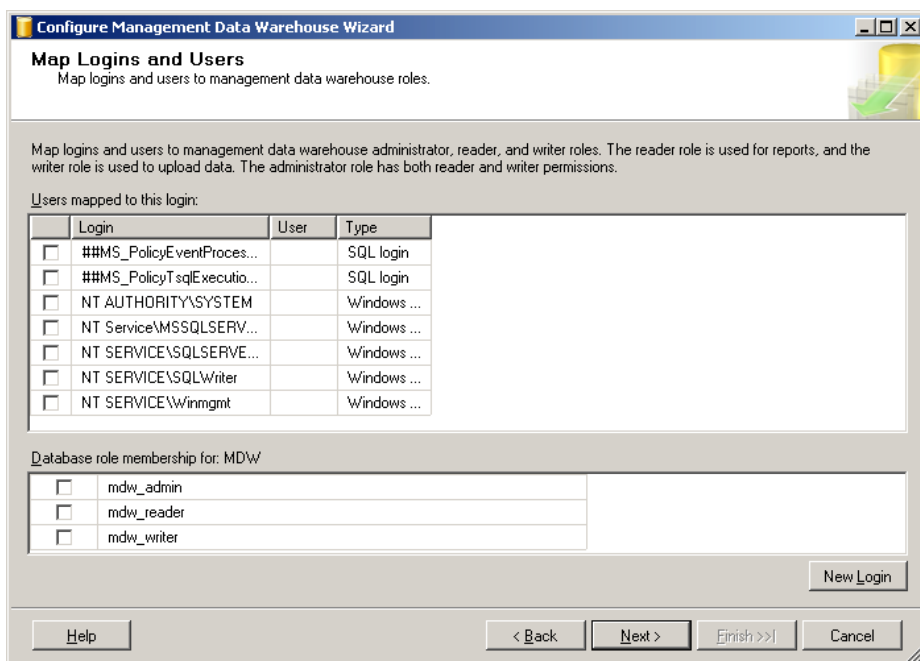
Kliknij przycisk *Next*. Zobacysz wówczas ekran *Complete the Wizard* (zakończ kreator), przedstawiony na rysunku 2.9.

Na ostatnim ekranie kliknij przycisk *Finish* (zakończ). Zobacysz okno *Configure Data Collection Wizard Progress* (postęp konfiguracji kreatora kolekcji danych). Upewnij się, że wszystkie pokazane kroki zostały zakończone pomyślnie, a następnie kliknij przycisk *Close* (zamknij). Ten krok skonfigurował bazę danych dla magazynu danych zarządzania i oprócz innych obiektów stworzył kolekcję tabel, które będziemy odpytywać w dalszej części.

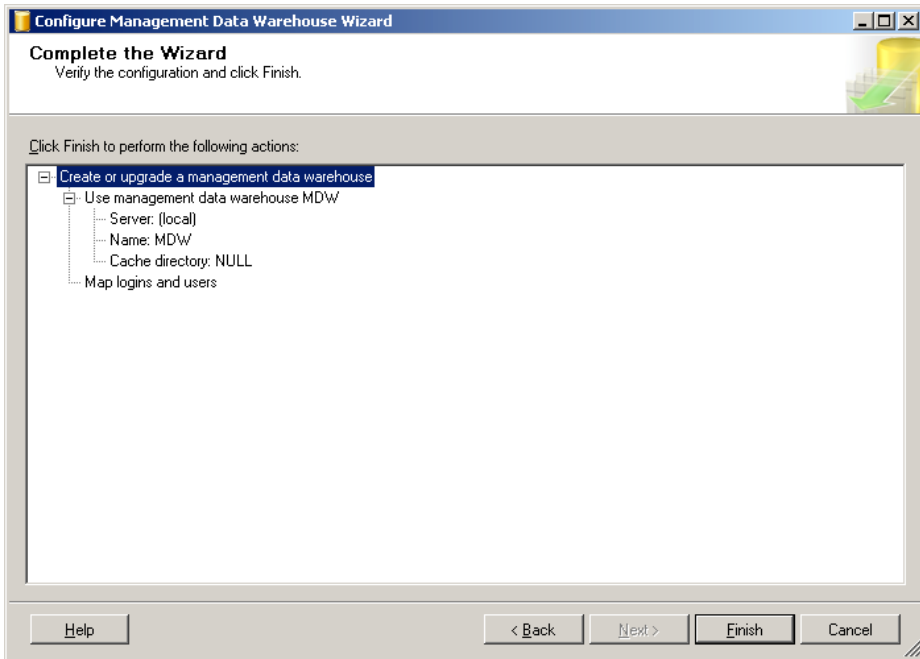
Aby skonfigurować drugi krok, kliknij prawym przyciskiem myszy węzeł *Data Collection* i wybierz *Tasks*, a następnie *Configure Data Collection* (konfiguruj zbieranie danych). Uruchomiony zostanie kolejny kreator. Kliknij przycisk *Next*. Zobacysz okno *Setup Data Collection Sets* (konfiguruj zestawy danych), pokazane na rysunku 2.10.



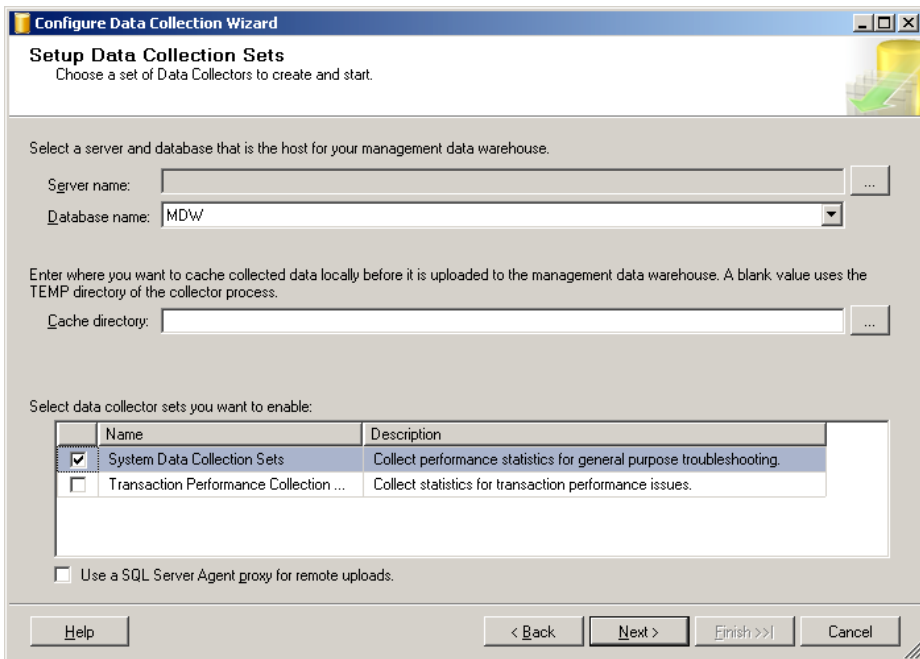
Rysunek 2.7. Okno Configure Management Data Warehouse Storage



Rysunek 2.8. Ekran Map Logins and Users



Rysunek 2.9. Ekran Complete the Wizard



Rysunek 2.10. Ekran Setup Data Collection Sets

W tym miejscu musisz wybrać bazę, która zostanie wykorzystana do zbierania danych, czyli bazę skonfigurowaną w kroku pierwszym. Możesz również wybrać katalog pośredni, który będzie wykorzystywany do przechowywania danych, zanim zostaną przekazane do bazy.

W przeciwieństwie do poprzednich wersji, w SQL Serverze 2014 musisz zdefiniować zestawy, które chcesz włączyć, co w naszym przypadku wiąże się z wybraniem *Setup Data Collection Sets* (kolekcja zestawów zbierania danych systemowych). Druga kolekcja, o nazwie *Transaction Performance Collection Sets* (kolekcja zestawów zbierania danych wydajnościowych), jest wykorzystywana w narzędziu Hekaton AMR (*Analysis, Migration and Reporting* — analiza, migracja i raportowanie), które dokładnie omówię w rozdziale 7. Kliknij przycisk *Next*, a następnie, na ostatnim ekranie, *Finish*.

Po ukończeniu konfiguracji zobaczysz, oprócz innych rzeczy, trzy uruchomione zestawy zbierania danych: *Disk Usage* (wykorzystanie dysku), *Query Statistics* (statystyki zapytań) i *Server Activity* (aktywność serwera). Zestaw *Utility Information* (informacje o narzędziach) jest domyślnie wyłączony i nie będę omawiał go w tej książce. Oto dane zbierane przez zestawy:

<i>Disk Usage</i>	Zbiera dane o wykorzystaniu dysku i logów dla wszystkich baz zainstalowanych na tej instancji SQL Servera.
<i>Query Statistics</i>	Zbiera statystyki, teksty zapytań, plany zapytań i konkretne zapytania.
<i>Server Activity</i>	Zbiera statystyki dotyczące wykorzystania zasobów i wydajności dla serwera i instancji SQL Servera.

Ponadto gorąco polecam zainstalowanie dodatkowej kolekcji *Query Hash Statistics* (statystyki haszów zapytań), którą możesz pobrać pod adresem <http://blogs.msdn.com/b/bartd/archive/2010/11/03/query-hash-statistics-a-query-cost-analysis-tool-now-available-for-download.aspx>. Ta kolekcja, która niestety nie jest częścią SQL Servera 2014, oparta jest na wartościach `query_hash` i `plan_hash`, zgodnie z wcześniejszymi wyjaśnieniami. Zbiera statystyki historyczne o zapytaniach i planach wykonania, dzięki czemu będziesz mógł z łatwością sprawdzić sumaryczny koszt swoich zapytań w poszczególnych bazach. Po zainstalowaniu kolekcji *Query Hash Statistics* będziesz musiał dezaktywować kolekcję *Query Statistics*, ponieważ obie zbierają te same informacje.

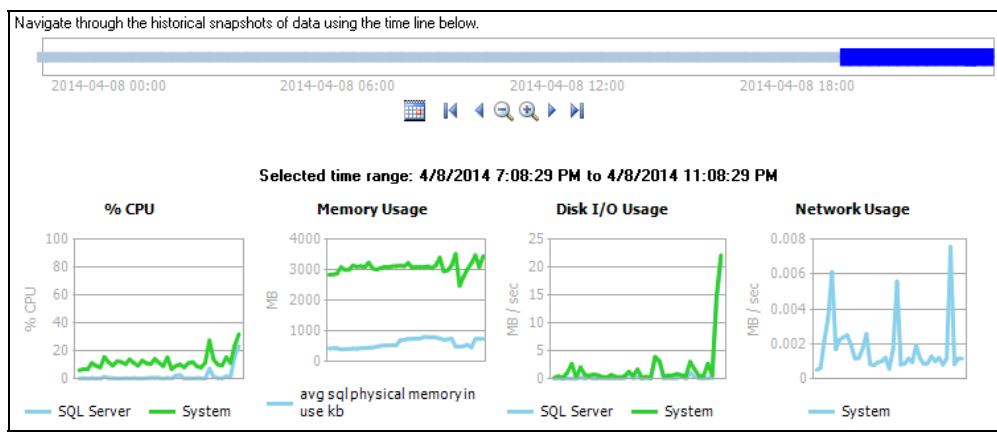
Powinieneś również wiedzieć, jakie zostały stworzone zadania komponentu SQL Server Agent:

- ▶ *collection_set_1_noncached_collect_and_upload*
- ▶ *collection_set_2_collection*
- ▶ *collection_set_2_upload*
- ▶ *collection_set_3_collection*

- ▶ *collection_set_3_upload*
- ▶ *mdw_purge_data_[MDW]*
- ▶ *syspolicy_purge_history*

Wykorzystanie Data Collectora

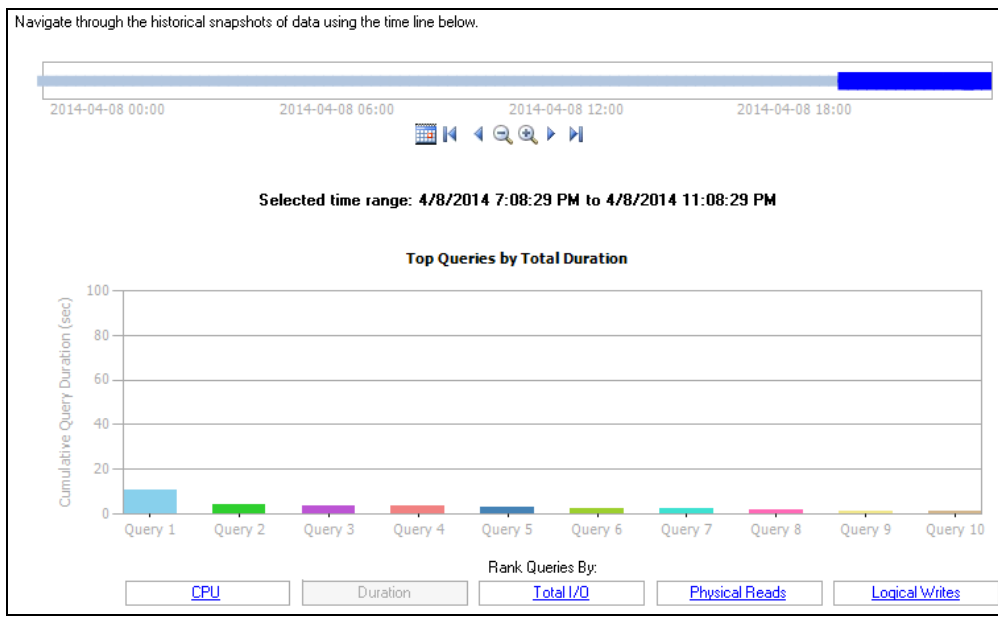
Teraz powinieneś zapoznać się z działaniem mechanizmu Data Collector — w szczególności z dostępnymi raportami i informacjami zebranymi w poszczególnych tabelach. Aby zacząć oglądać raporty, kliknij prawym przyciskiem myszy węzeł *Data Collection* i wybierz *Reports/Management Data Warehouse/Server Activity History* (raporty/magazyn danych zarządzania/historia aktywności serwera). Jeśli zebrane zostało wystarczająco dużo danych, zobaczysz raport podobny do tego z rysunku 2.11 (widoczna jest tylko część).



Rysunek 2.11. Raport Server Activity History

Kliknięcie sekcji SQL Server grafu *% CPU* przeniesie Cię do raportu *Query Statistics History* (historyczne statystyki zapytań). Do tego raportu możesz również dotrzeć, klikając prawym przyciskiem myszy węzeł *Data Collection*, a następnie *Reports/Management Data Warehouse/Query Statistics History*. W obu przypadkach zobaczysz raport podobny do tego z rysunku 2.12 (widoczna jest tylko część).

Uruchomienie tego raportu pozwoli Ci zobaczyć 10 najbardziej kosztownych pod względem wykorzystania procesora zapytań, ale możesz również wybrać koszt pod względem czasu trwania, ilości operacji I/O, odczytów fizycznych i zapisów logicznych. Data Collector zawiera inne raporty i, jak już widziałeś, niektóre raporty zawierają linki do innych raportów, zawierających bardziej szczegółowe informacje.



Rysunek 2.12. Raport Query Statistics History

Zapytania na tabelach Data Collector

Bardziej zaawansowani użytkownicy mogą chcieć samodzielnie wykonywać zapytania do tabel Data Collector, aby tworzyć własne raporty lub uzyskać głębszy wgląd w zebrane dane. Tworzenie własnych kolekcji jest również możliwe i może być konieczne, jeżeli będziesz potrzebować danych, które nie są domyślnie przechowywane. Posiadanie własnych kolekcji zmusza także do tworzenia własnych zapytań i raportów.

Na przykład Data Collector zbiera wiele liczników wydajności, które możesz zobaczyć, przeglądając właściwości kolekcji *Server Activity*. Aby to zrobić, rozwiń węzły *Data Collection* i *System Data Collection Sets*, kliknij prawym przyciskiem myszy kolekcję *Server Activity* i wybierz opcję *Properties*. W oknie właściwości, na liście *Collection Items* (elementy kolekcji), wybierz *Server Activity — Performance Counters* (aktywność serwera — liczniki wydajności) i spójrz na okno *Input Parameters* (parametry wejściowe). Oto mała próbka tych liczników wydajności:

- ▶ *\Memory \% Committed Bytes In Use*
- ▶ *\Memory \Available Bytes*
- ▶ *\Memory \Cache Bytes*
- ▶ *\Memory \Cache Faults/sec*
- ▶ *\Memory \Committed Bytes*
- ▶ *\Memory \Free & Zero Page List Bytes*

- ▶ *\Memory \Modified Page List Bytes*
- ▶ *\Memory \Pages/sec*
- ▶ *\Memory \Page Reads/sec*
- ▶ *\Memory \Page Write/sec*
- ▶ *\Memory \Page Faults/sec*
- ▶ *\Memory \Pool Nonpaged Bytes*
- ▶ *\Memory \Pool Paged Bytes*

Możesz również uzyskać bardzo dokładne definicje liczników wydajności ze swojej instancji, zaglądając do tabeli `snapshots.performance_counter_instances`. Dane liczników wydajności są przechowywane w tabeli `snapshots.performance_counter_values`. Jednym z najczęściej wykorzystywanych przez administratorów baz danych licznikiem jest *\Processor(_Total)\% Processor Time*, który pozwala zbierać informacje dotyczące procentowego wykorzystania procesora. Aby pobrać zebrane dane, możemy użyć następującego zapytania:

```
SELECT sii.instance_name, collection_time, [path] AS counter_name,
       formatted_value AS counter_value_percent
FROM snapshots.performance_counter_values pcv
JOIN snapshots.performance_counter_instances pci
  ON pcv.performance_counter_instance_id = pci.performance_counter_id
JOIN core.snapshots_internal si ON pcv.snapshot_id = si.snapshot_id
JOIN core.source_info_internal sii ON sii.source_id = si.source_id
WHERE pci.[path] = '\Processor(_Total)\% Processor Time'
ORDER BY pcv.collection_time desc
```

Zostanie pokazany wynik podobny do poniższego:

collection_time	counter_name	counter_value_percent
2013-05-12 17:14:34.0000000 -07:00	\Processor(_Total)\% Processor Time	19.156125208
2013-05-12 17:13:34.0000000 -07:00	\Processor(_Total)\% Processor Time	18.674633138
2013-05-12 17:12:34.0000000 -07:00	\Processor(_Total)\% Processor Time	18.695325723
2013-05-12 17:11:33.0000000 -07:00	\Processor(_Total)\% Processor Time	17.121825074
2013-05-12 17:10:33.0000000 -07:00	\Processor(_Total)\% Processor Time	21.20307118
2013-05-12 17:09:33.0000000 -07:00	\Processor(_Total)\% Processor Time	22.30240009
2013-05-12 17:08:33.0000000 -07:00	\Processor(_Total)\% Processor Time	21.74221733
2013-05-12 17:07:33.0000000 -07:00	\Processor(_Total)\% Processor Time	21.22852141

Jest więcej interesujących tabel, przynajmniej z punktu widzenia zbierania danych dotyczących zapytań, które będziesz musiał odpytywać bezpośrednio. Kolekcja Query Statistics wykorzystuje zapytania zdefiniowane w pakietach SSIS *QueryActivityCollect.dtsx*

i *QueryActivityUpload.dtsx*, a zebrane dane są ładowane do tabel `snapshots.query_stats`, `snapshots.notable_query_text` i `snapshots.notable_query_plan`. Tabele te przechowują odpowiednio statystyki zapytań, teksty zapytań i plany zapytań. Jeżeli zainstalowałeś kolekcję *Query Hash Statistics*, zamiast tego zostaną wykorzystane pakiety `QueryHashStats` ➔ `PlanCollect` i `QueryHashStatsPlanUpload`. Kolejną interesującą tabelą może być `snapshots` ➔ `.active_sessions_and_request`, która zbiera informacje o sesjach i żądaniach SQL Servera.

Podsumowanie

Ten rozdział zapoznał Cię z kilkoma technikami, których możesz użyć do sprawdzenia, jak Twoje zapytania wykorzystują zasoby systemowe takie jak dysk czy procesor. Najpierw objaśniłem kilka niezbędnych widoków (DMV) i funkcji (DMF) systemowych, które są bardzo przydatne podczas szukania kosztownych zapytań. Omówiłem również dwie funkcje wprowadzone w SQL Serverze 2008, a mianowicie zdarzenia rozszerzone i Data Collector, oraz sposób przechwytywania za ich pomocą zdarzeń i danych dotyczących wydajności. Przybliżyłem także SQL Trace, funkcjonalność, która jest częścią SQL Servera, odkąd pamiętam.

Mamy już podstawy wyszukiwania kosztownych zapytań, więc co teraz? Naszym ostatecznym celem jest poprawienie wydajności zapytań, a w kolejnych rozdziałach omówię różne podejścia pozwalające to osiągnąć. Czy potrzebny jest lepszy indeks? Może Twoje zapytanie jest spowalniane przez różne parametry, a może optymalizator nie wybiera dobrego planu, ponieważ wykorzystana funkcjonalność nie jest odpowiednio wspierana przez statystyki. W kolejnych rozdziałach omówimy te i inne problemy.

Kiedy więc znajdziemy zapytanie, które może powodować problemy, musisz jeszcze się zorientować, co konkretnie jest problemem, i jakoś temu zaradzić. Zazwyczaj będziemy w stanie znaleźć problem tylko analizując bogate informacje dostępne w planie zapytania. Aby to zrobić, musimy poznać dokładniej, jak działa optymalizator zapytań i co robią poszczególne operatory. Omawiam to szczegółowo w następnym rozdziale.

Rozdział 3

Optymalizator zapytań

W tym rozdziale:

- ▶ Przegląd
- ▶ `sys.dm_exec_query_optimizer_info`
- ▶ Parsowanie i przypisywanie
- ▶ Uproszczenie
- ▶ Plany trywialne
- ▶ Reguły transformacji
- ▶ Struktura Memo
- ▶ Statystyki
- ▶ Pełna optymalizacja
- ▶ Podsumowanie





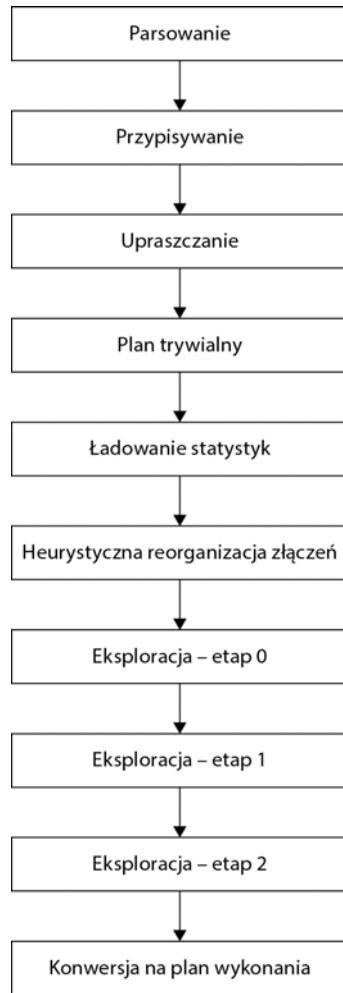
W tym rozdziale omawiam sposób działania optymalizatora zapytań i pokazuję kroki, które wykonywane są w tle i których nie widzimy. Dotyczy to wszystkich aspektów — od momentu przekazania zapytania do SQL Servera do wygenerowania planu wykonania gotowego do wykonania — i obejmuje kroki takie jak parsowanie, przypisywanie, upraszczanie, optymalizacja planu trywialnego i pełna optymalizacja. Omówione zostaną również ważne komponenty będące częścią architektury optymalizatora zapytań, takie jak reguły transformacji i struktura Memo.

Celem optymalizatora zapytań jest zapewnienie optymalnego — lub przynajmniej wystarczająco dobrego — planu zapytania. Aby tego dokonać, za pomocą reguł transformacji generuje on wiele możliwych planów. Te alternatywne plany są przechowywane przez czas trwania optymalizacji w strukturze o nazwie Memo. Niestety minusem optymalizacji opartej na koszcie jest koszt samej optymalizacji. Ponieważ znalezienie optymalnego planu dla niektórych zapytań trwałoby nieakceptowalnie długo, do ograniczenia liczby alternatywnych planów wykorzystywana jest heurystyka — pamiętaj, że celem jest znalezienie wystarczająco dobrego planu jak najszybciej. Heurystyka pomaga optymalizatorowi zapytań w poradeniu sobie z rosnącą liczbą możliwych kombinacji wraz ze wzrostem poziomu skomplikowania zapytań. Jednak wykorzystanie reguł transformacji i heurystyki niekoniecznie obniża koszt dostępnych alternatyw, dlatego determinowany jest również koszt planów alternatywnych, a na podstawie tych kosztów wybierany jest najlepszy plan.

Przegląd

Procesy optymalizacji i wykonywania zapytań zostały wprowadzone w rozdziale 1., a w tym rozdziale omówię je dokładniej. Zanim jednak zaczniemy, pokrótce objaśnię wewnętrzne działanie procesu optymalizacji, które obejmuje zarówno działania przed optymalizatorem zapytań, jak i po nim. Diagram na rysunku 3.1 pokazuje główne fazy procesu przetwarzania zapytań; każdą z nich dokładniej omówię w kolejnych podrozdziałach.

Parsowanie i przypisywanie to pierwsze operacje wykonywane po przesłaniu zapytania do instancji serwera. W ich efekcie powstaje drzewo reprezentujące zapytanie, które jest następnie przesyłane do optymalizatora i na którego podstawie przeprowadzana jest optymalizacja. Na początku procesu to drzewo logiczne zostanie uproszczone, a optymalizator sprawdzi, czy drzewo kwalifikuje się do wykorzystania planu trywialnego. Jeżeli tak, zwracany jest plan trywialny i proces optymalizacji zostaje natychmiast zakończony. Procesy parsowania, przypisywania, upraszczania i generowania planu trywialnego nie są zależne od zawartości bazy (samych danych), ale



Rysunek 3.1. Proces przetwarzania zapytań

tylko od schematu bazy i definicji zapytania. Z tego powodu te procesy nie wykorzystują statystyk czy szacowania kosztów ani nie podejmują decyzji w oparciu o koszty — te elementy są wykorzystywane tylko podczas pełnej optymalizacji.

Jeżeli zapytanie nie kwalifikuje się do wykorzystania planu trywialnego, optymalizator wykona pełny proces optymalizacji, który składa się z trzech etapów, a zakończenie każdego z nich może owocować powstaniem ostatecznego planu zapytania. Dodatkowo, aby rozpatrywane były wszystkie informacje zebrane w poprzednich fazach, takie jak definicja zapytania i schemat bazy, pełny proces optymalizacji wykorzystuje statystyki i szacowanie kosztów, a najlepszy plan zapytania zostanie wybrany wyłącznie na podstawie kosztów poszczególnych planów (w ramach dostępnego czasu).

Optymalizator zapytań wykonuje kilka faz, zaprojektowanych tak, aby jak najszybciej optymalizować zapytania i aby bardziej kosztowne i zaawansowane opcje były wykorzystywane, tylko jeżeli będzie to naprawdę konieczne. Te fazy to upraszczanie, optymalizacja planu trywialnego i pełna optymalizacja. Podobnie pełna optymalizacja składa się z trzech faz: wyszukiwanie 0, wyszukiwanie 1 i wyszukiwanie 2 (nazywane są one również odpowiednio przetwarzaniem transakcji, szybkim planem i pełnym procesem optymalizacji). Plan może zostać stworzony w którejkolwiek z tych faz oprócz fazy upraszczania, którą omówię za chwilę.



OSTRZEŻENIE

Ten rozdział zawiera wiele nieudokumentowanych i niewspieranych funkcjonalności, które zostaną odpowiednio oznaczone. Zamieściłem je tutaj w celach edukacyjnych, aby czytelnik mógł lepiej zrozumieć, jak działa optymalizator zapytań, i nie należy ich stosować w środowiskach produkcyjnych. Korzystaj z nich uważnie w ramach własnego, odizolowanego środowiska.

sys.dm_exec_query_optimizer_info

Możesz wykorzystać widok DMV `sys.dm_exec_query_optimizer_info` do uzyskania dodatkowych informacji o pracy wykonywanej przez optymalizator zapytań. Ten widok, który jest tylko częściowo udokumentowany, zawiera informacje dotyczące optymalizacji wykonywanych na instancji SQL Servera. Chociaż zawiera sumaryczne statystyki od uruchomienia danej instancji, jak się zaraz przekonamy, można z niego skorzystać również do pozyskania informacji na temat konkretnego zapytania lub zadania.

Jak wspomniałem, możesz wykorzystać ten widok do pozyskania statystyk na temat operacji wykonywanych przez optymalizator zapytań, takich jak informacje o tym, jak zostały zoptymalizowane zapytania czy ile zapytań zostało zoptymalizowanych od uruchomienia instancji. Widok zwraca trzy kolumny:

- ▶ **Counter (licznik)** — nazwa zdarzenia optymalizatora.
- ▶ **Occurrence (wystąpienia)** — liczba wystąpień zdarzenia optymalizatora dla danego pola *Counter*.
- ▶ **Value (wartość)** — średnia wartość dla wystąpienia licznika.

W SQL Serverze 2005 zostało zdefiniowane 38 liczników, a w SQL Serverze 2008 został dodany nowy, o nazwie *merge stmt*, co daje w sumie 39 i taka liczba jest dostępna w SQL Serverze 2014. Aby zobaczyć statystyki dla wszystkich optymalizacji od uruchomienia instancji SQL Servera, możemy wykorzystać poniższe zapytanie:

```
SELECT * FROM sys.dm_exec_query_optimizer_info
```

Oto częściowy wynik dla jednej z instancji produkcyjnych:

counter	occurrence	value
optimizations	691473	1
elapsed time	691465	0.007806012
final cost	691465	1.398120739
trivial plan	29476	1
tasks	661989	332.5988816
no plan	0	NULL
search 0	26724	1
search 0 time	31420	0.01646922
search 0 tasks	31420	1198.811617

Wynik pokazuje, że od uruchomienia instancji miało miejsce 691 473 optymalizacji, że średni czas optymalizacji wynosił 0,0078 sekundy i że średni szacowany koszt każdej optymalizacji, w wewnętrznych jednostkach, wynosił około 1,398. Ten konkretny przykład pokazuje optymalizacje mało kosztownych zapytań, typowych dla systemu OLTP.

Chociaż widok `sys.dm_exec_query_optimizer_info` był całkowicie udokumentowany w Books Online dla wersji 2005, nowsze wersje nie zawierają prawie połowy (18 z 39) opisów liczników, a brakujące liczniki oznaczają jako „Tylko wewnętrzne”. Dlatego w tabeli 3.1 zamieściłem opisy z aktualnej wersji oraz opisy 18 nieudokumentowanych liczników z wersji 2005 (opisy te są nadal aktualne dla SQL Servera 2014). Dodatkowe opisy zostały oznaczone kursywą.

Tabela 3.1. Liczniki udokumentowane i nieudokumentowane widoku `sys.dm_exec_query_optimizer_info` DMV

Licznik	Wystąpienia	Wartość
optimizations	Całkowita liczba optymalizacji.	Nie dotyczy.
elapsed time	Całkowita liczba optymalizacji.	Średni czas wykonywania dla optymalizacji poszczególnych zapytań, w sekundach.
final cost	Całkowita liczba optymalizacji.	Średni szacowany koszt zoptymalizowanego planu, w wewnętrznych jednostkach kosztu.
trivial plan	<i>Całkowita liczba planów trywialnych (wykorzystanych jako plany ostateczne).</i>	<i>Nie dotyczy.</i>

Tabela 3.1. Liczniki udokumentowane i nieudokumentowane widoku sys.dm_exec_query_optimizer_info DMV — *ciąg dalszy*

Licznik	Wystąpienia	Wartość
tasks	Liczba optymalizacji, które wykorzystały zadania (eksploracja, implementacja, derywacja właściwości).	Średnia liczba wykonanych zadań.
no plan	Liczba optymalizacji, dla których po pełnym procesie optymalizacji nie został znaleziony plan i dla których nie zostały zgłoszone błędy.	Nie dotyczy.
search 0	Całkowita liczba planów znalezionych w etapie search 0.	Nie dotyczy.
search 0 time	Liczba optymalizacji, które weszły do etapu search 0.	Średni czas spędzony w etapie search 0, w sekundach.
search 0 tasks	Liczba optymalizacji, które weszły do etapu search 0.	Średnia liczba zadań wykonanych w etapie search 0.
search 1	Całkowita liczba planów znalezionych w etapie search 1.	Nie dotyczy.
search 1 time	Liczba optymalizacji, które weszły do etapu search 1.	Średni czas spędzony w etapie search 1, w sekundach.
search 1 tasks	Liczba optymalizacji, które weszły do etapu search 1.	Średnia liczba zadań wykonanych w etapie search 1.
search 2	Całkowita liczba planów znalezionych w etapie search 2.	Nie dotyczy.
search 2 time	Liczba optymalizacji, które weszły do etapu search 2.	Średni czas spędzony w etapie search 2, w sekundach.
search 2 tasks	Liczba optymalizacji, które weszły do etapu search 2.	Średnia liczba zadań wykonanych w etapie search 2.
gain stage 0 to stage 1	Liczba uruchomień search 1 po etapie search 0.	Średni wzrost między etapem 0 a etapem 1 liczony jako: (MinimalnyKosztPlanu(search 0) – MinimalnyKosztPlanu(search 1)) / MinimalnyKosztPlanu(search 0).
gain stage 1 to stage 2	Liczba uruchomień search 2 po etapie search 1.	Średni wzrost między etapem 1 a etapem 2 liczony jako: (MinimalnyKosztPlanu(search 1) – MinimalnyKosztPlanu(search 2)) / MinimalnyKosztPlanu(search 1).

Tabela 3.1. Liczniki udokumentowane i nieudokumentowane widoku
sys.dm_exec_query_optimizer_info DMV — *ciąg dalszy*

Licznik	Wystąpienia	Wartość
timeout	<i>Liczba optymalizacji, dla których wystąpiło przekroczenie dozwolonego czasu.</i>	<i>Nie dotyczy.</i>
memory limit exceeded	<i>Liczba optymalizacji, dla których został przekroczony wewnętrzny limit pamięci.</i>	<i>Nie dotyczy.</i>
insert stmt	Liczba optymalizacji dla polecenia INSERT.	Nie dotyczy.
delete stmt	Liczba optymalizacji dla polecenia DELETE.	Nie dotyczy.
update stmt	Liczba optymalizacji dla polecenia UPDATE.	Nie dotyczy.
merge stmt	Liczba optymalizacji dla polecenia MERGE.	Nie dotyczy.
contains subquery	Liczba optymalizacji dla zapytań zawierających przynajmniej jedno podzapytanie.	Nie dotyczy.
unnest failed	<i>Liczba razy, kiedy usuwanie zagnieżdżenia nie mogło usunąć podzapytania.</i>	<i>Nie dotyczy.</i>
tables	Całkowita liczba optymalizacji.	Średnia liczba tabel, do których odnosiło się optymalizowane zapytanie.
hints	Liczba razy, kiedy wyszczególniona została jakaś odpowiedź. Wliczane są wskazówki zapytań JOIN, GROUP, UNION i FORCE ORDER, opcja FORCE PLAN i wskazówki złączeń.	Nie dotyczy.
order hint	Liczba razy, kiedy wyspecyfikowana została wskazówka FORCE ORDER.	Nie dotyczy.
join hint	Liczba razy, kiedy algorytm złączenia był wymuszony przez wskazówkę złączenia.	Nie dotyczy.
view reference	Liczba razy, kiedy w zapytaniu brał udział widok.	Nie dotyczy.

Tabela 3.1. Liczniki udokumentowane i nieudokumentowane widoku sys.dm_exec_query_optimizer_info DMV — *ciąg dalszy*

Licznik	Wystąpienia	Wartość
remote query	Liczba optymalizacji, gdzie zapytanie odnosiło się do przynajmniej jednego zdalnego źródła danych, takiego jak tabela z czterocłonową nazwą lub wynik OPENROWSET.	Nie dotyczy.
maximum DOP	Całkowita liczba optymalizacji.	Średnia efektywna liczba MAXDOP dla zoptymalizowanego planu. Domyślnie MAXDOP jest determinowane przez maksymalny poziom współbieżności wynikającej z konfiguracji serwera i może zostać wymuszona dla zapytania za pomocą wskazówki MAXDOP.
maximum recursion level	Liczba optymalizacji, w których poziom MAXRECURSION wyższy niż 0 został wyspecyfikowany za pomocą wskazówki zapytania.	Średni poziom MAXRECURSION w optymalizacjach, w których maksymalny poziom rekurencji został wyspecyfikowany za pomocą wskazówki zapytania.
indexed views loaded	<i>Liczba zapytań, dla których na potrzeby dopasowywania został załadowany jeden lub więcej widoków indeksowanych.</i>	<i>Średnia liczba załadowanych widoków.</i>
indexed views matched	Liczba optymalizacji, dla których dopasowany został jeden lub więcej widoków indeksowanych.	Średnia liczba dopasowanych widoków.
indexed views used	Liczba optymalizacji, w których jeden lub więcej widoków indeksowanych jest wykorzystywanych w planach wynikowych po dopasowaniu.	Średnia liczba wykorzystanych widoków.
indexed views updated	Liczba optymalizacji poleceń DML, które wyprodukowały plan utrzymujący jeden lub więcej widoków indeksowanych.	Średnia liczba utrzymywanych widoków.
dynamic cursor request	Liczba optymalizacji, w których wyspecyfikowane zostało żądanie dynamicznego kursora.	Nie dotyczy.
fast forward cursor request	Liczba optymalizacji, w których wyspecyfikowane zostało żądanie szybkiego kursora.	Nie dotyczy.

Liczniki mogą być wykorzystywane na kilka sposobów, aby pokazać ważne informacje wewnętrzne na temat optymalizacji na danej instancji. Na przykład poniższe zapytanie wyświetla procentową liczbę optymalizacji z wykorzystaniem wskazówek. Ta informacja może być użyteczna, aby pokazać, jak powszechne jest wykorzystanie wskazówek, co z kolei może wskazywać, że Twój kod jest mniej elastyczny, niż byś tego chciał, i może wymagać prac konserwacyjnych. Wskazówki omówię w rozdziale 10.

```
SELECT (SELECT occurrence FROM sys.dm_exec_query_optimizer_info WHERE counter =
'hints' ) * 100.0 / ( SELECT occurrence FROM sys.dm_exec_query_optimizer_info WHERE
↳counter = 'optimizations' )
```

Jak już wspomniałem, możesz wykorzystać widok DMV na dwa różne sposoby: możesz pobierać informacje dotyczące akumulowanej historii optymalizacji od momentu uruchomienia instancji lub możesz użyć go do pobrania informacji dotyczących konkretnego zapytania lub grupy zapytań. Aby otrzymać informacje o zapytaniu lub o grupie zapytań, musisz wykonać dwie migawki widoku — jedną przed optymalizacją zapytania, a drugą zaraz po aktualizacji — i ręcznie znaleźć różnice między nimi. Niestety nie ma możliwości zainicjalizowania wartości tego widoku.

Podczas przechwytywania tych informacji musisz brać pod uwagę kilka problemów. Po pierwsze, musisz wyeliminować efekty zapytań generowanych przez system i zapytań wykonywanych przez innych użytkowników, które mogłyby być wykonywane w tym samym czasie co Twoje testy. Spróbuj wyizolować zapytanie lub grupę zapytań na swojej instancji i upewnij się, korzystając z licznika `optimizations`, czy raportowana liczba optymalizacji jest taka sama jak liczba żądanych optymalizacji. Jeżeli wartość w liczniku jest większa, dane będą prawdopodobnie zawierały zapytania wykonywane przez system lub innych użytkowników. Oczywiście możliwe jest również, że Twoje własne zapytanie do widoku `sys.dm_exec_query_optimizer_info` może zostać policzone w poczet tych optymalizacji.

Po drugie, musisz się upewnić, że optymalizacja zostanie wykonana. Na przykład, jeżeli uruchomisz to samo zapytanie więcej niż raz, SQL Server może wykorzystać plan przechowywany w magazynie planów, bez uruchamiania jakichkolwiek mechanizmów związanych z optymalizacją. Możesz wymusić optymalizację, korzystając z wypowiedzi `RECOMPILE` (pokazana w dalszej części), za pomocą `sp_recompile` lub ręcznie usuwając plany z magazynu planów. Od wersji 2008 polecenie `DBCC FREEPROCCACHE` może być stosowane do usuwania konkretnych planów, do usuwania wszystkich planów związanych z konkretną pulą lub do czyszczenia całego magazynu. Chociaż najprawdopodobniej pracujesz na serwerze testowym, warto zaznaczyć, że nie powinieneś uruchamiać poleceń czyszczących magazyn planów na serwerach produkcyjnych.

Zgodnie z powyższym skrypt z listingu 3.1 wyświetli informacje dotyczące optymalizacji dla konkretnego zapytania, unikając jednocześnie wszystkich wyszczególnionych wyżej problemów. Skrypt bazuje na pomysłe Lubora Kollara z zespołu SQL Servera w Microsoftzie, a w skrypcie znajduje się miejsce na wstawienie zapytania, dla którego chcemy uzyskać informacje o optymalizacji.

Listing 3.1. Kod wykorzystujący widok sys.dm_exec_query_optimizer_info

```

-- optymalizacja tych zapytań teraz,
-- aby nie zaburzały zebranych danych
GO
SELECT *
INTO after_query_optimizer_info
FROM sys.dm_exec_query_optimizer_info
GO
SELECT *
INTO before_query_optimizer_info
FROM sys.dm_exec_query_optimizer_info
GO
DROP TABLE before_query_optimizer_info
DROP TABLE after_query_optimizer_info
GO
-- rozpoczęcie właściwego wykonywania
GO
SELECT *
INTO before_query_optimizer_info
FROM sys.dm_exec_query_optimizer_info
GO
-- tutaj wstaw swoje zapytanie
SELECT *
FROM Person.Address
-- zachowaj poniższe, aby wymusić nową optymalizację
OPTION (RECOMPILE)
GO
SELECT *
INTO after_query_optimizer_info
FROM sys.dm_exec_query_optimizer_info
GO
SELECT a.counter,
(a.occurrence - b.occurrence) AS occurrence,
(a.occurrence * a.value - b.occurrence *
b.value) AS value
FROM before_query_optimizer_info b
JOIN after_query_optimizer_info a
ON b.counter = a.counter
WHERE b.occurrence <> a.occurrence
DROP TABLE before_query_optimizer_info
DROP TABLE after_query_optimizer_info

```

Zauważ, że niektóre zapytania powtarzają się w kodzie dwukrotnie. Ma to na celu zoptymalizowanie ich za pierwszym razem, kiedy będą wykonywane, aby ich plan został zapisany w magazynie planów i był wykorzystywany podczas późniejszych wywołań. W ten sposób najlepiej jak możemy izolujemy nasze zapytanie od zapytań analizujących. Należy zwrócić uwagę, aby oba zapytania były dokładnie takie same, włączając w to wielkość liter, komentarze i tak dalej. Muszą być również odseparowane od innych zapytań poleceniami GO.

Jeżeli uruchomisz ten skrypt na bazie AdventureWorks2012, wyniki (po przeszkrołowaniu danych z adresami) powinny wyglądać zgodnie z poniższymi. Pokazywane czasy

mogą oczywiście być różne zarówno dla tego, jak i dla innych przykładów z tego rozdziału. Wynik pokazuje między innymi, że wykonana została jedna optymalizacja, odnosząca się do jednej tabeli, a jej koszt wyniósł 0,278931474.

counter	occurrence	value
elapsed time	1	0
final cost	1	0.278931474
maximum DOP	1	0
optimizations	1	1
tables	1	1
trivial plan	1	1

Oczywiście dla tego prostego zapytania moglibyśmy znaleźć te same informacje w kilku innych miejscach, na przykład w samym planie wykonania. Jak jednak pokażę w dalszej części rozdziału, widok ten może dostarczyć informacji, których nie można znaleźć w żadnym innym miejscu. Z tego widoku będę korzystał w dalszej części rozdziału i zauważysz, dlaczego jest bardzo użyteczny w pozyskiwaniu dodatkowych informacji na temat pracy wykonywanej przez optymalizator.

Parsowanie i przypisywanie

Parsowanie i przypisywanie to pierwsze operacje wykonywane przez SQL Server po zatwierdzeniu zapytania; są wykonywane przez komponent o nazwie Algebrizer. Parsowanie sprawdza najpierw, czy zapytanie T-SQL jest poprawne składniowo, a następnie wykorzystuje informacje o zapytaniu do zbudowania drzewa operatorów relacyjnych. Oznacza to, że parser tłumaczy zapytanie SQL na jego reprezentację algebraiczną w formie drzewa operatorów logicznych, nazywanego *drzewem parsowania*. Parsowanie sprawdza tylko poprawność składni T-SQL, nie sprawdza poprawności nazw tabel czy kolumn, które weryfikowane są w kolejnej fazie — przypisywania.

Parsowanie jest podobne do funkcjonalności parsowania w Management Studio (dostępnej po kliknięciu przycisku *Parse* na domyślnym pasku zadań) lub polecenia SET PARSEONLY. Na przykład poniższe zapytanie zostanie pomyślnie sparsowane dla bazy AdventureWorks2012, mimo że wymienione kolumny i tabela nie istnieją w tej bazie:

```
SELECT lname, fname FROM authors
```

Jeżeli jednak niewłaściwie zapiszesz słowo kluczowe SELECT lub FROM, SQL Server zwróci komunikat błędu informujący o niepoprawnej składni.

Kiedy drzewo parsowania zostanie skonstruowane, Algebrizer wykona operację przypisywania, która związana jest głównie z przypisywaniem nazw. Podczas tej operacji Algebrizer upewnia się, czy wszystkie obiekty wymienione w zapytaniu istnieją,

potwierdza, czy żądane operacje pomiędzy nimi są dozwolone, i sprawdza, czy obiekty są widoczne (czyli mają odpowiednie ustawienia bezpieczeństwa) dla użytkownika uruchamiającego zapytanie. Łączy również każdą tabelę i kolumnę z odpowiadającym jej obiektem w katalogu systemowym. Przydzielanie nazw dla widoków obejmuje proces zastępowania widoków, w którym referencja do widoku jest rozszerzana o jego definicję — na przykład, aby bezpośrednio dołączyć tabele wykorzystywane w widoku. Wynik operacji przypisywania, nazywany *drzewem algebrizera*, jest następnie przekazywany do optymalizatora w celu (jak zapewne się domyślasz) optymalizacji.

Pierwotnie drzewo to będzie reprezentowane jako seria logicznych operacji blisko skorelowanych ze składnią zapytania. Składają się na nie takie logiczne operacje jak „pobierz dane z tabeli Customer”, „pobierz dane z tabeli Contact”, „wykonaj złączenie wewnętrzne” i tak dalej. Podczas procesu optymalizacji będą wykorzystywane różne reprezentacje drzewa, a drzewo logiczne będzie otrzymywało różne nazwy, dopóki nie zostanie wykorzystane do zainicjowania struktury Memo, którą omówię w dalszej części.

Nie ma żadnej udokumentowanej informacji na temat tych drzew logicznych w SQL Serverze, ale, co ciekawe, poniższe zapytanie zwraca nazwy wykorzystywane przez te drzewa:

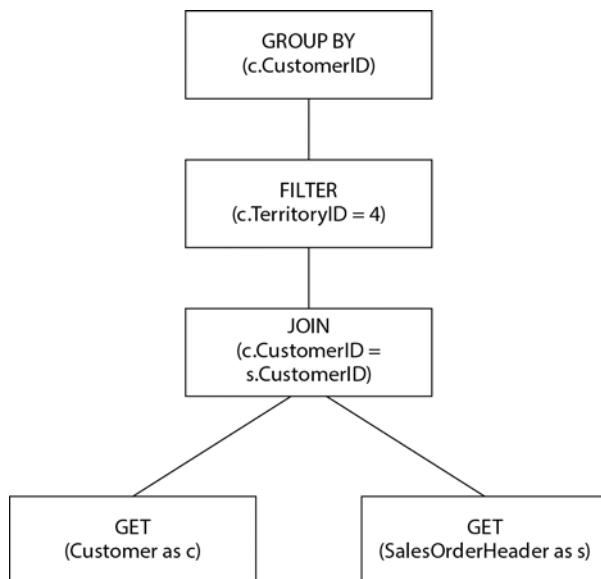
```
SELECT * FROM sys.dm_xe_map_values WHERE name = 'query_optimizer_tree_id'
```

Zapytanie zwraca następujące wyniki:

map_key	map_value
0	CONVERTED_TREE
1	INPUT_TREE
2	SIMPLIFIED_TREE
3	JOIN_COLLAPSED_TREE
4	TREE_BEFORE_PROJECT_NORM
5	TREE_AFTER_PROJECT_NORM
6	OUTPUT_TREE
7	TREE_COPIED_OUT

W dalszej części rozdziału przyjrzymy się niektórym z tych drzew logicznych. Na przykład poniższe zapytanie będzie reprezentowane przez drzewo przedstawione na rysunku 3.2:

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c JOIN Sales.SalesOrderHeader s
ON c.CustomerID = s.CustomerID
WHERE c.TerritoryID = 4
GROUP BY c.CustomerID
```



Rysunek 3.2. Reprezentacja zapytania w formie drzewa

Jest również kilka nieudokumentowanych flag śledzenia, które pozwalają zobaczyć różne drzewa logiczne. Na przykład możesz wykorzystać poniższe zapytanie z nieudokumentowaną flagą 8605. Najpierw jednak włącz flagę 3604, zgodnie z poniższym poleceniem:

```
DBCC TRACEON(3604)
```

Flaga 3604 pozwala przekierować wynik śledzenia do klienta wykonującego polecenie, w tym przypadku do SQL Server Management Studio. Wynik ten będziesz mógł zobaczyć w zakładce *Messages*.

```
SELECT ProductID, name FROM Production.Product
WHERE ProductID = 877
OPTION (RECOMPILE, QUERYTRACEON 8605)
```



UWAGA

QUERYTRACEON to podpowiedź zapytania pozwalająca dołączać flagi śledzenia na poziomie zapytania — została ona przedstawiona w rozdziale 1.

Otrzymasz poniższy wynik:

*** Drzewo skonwertowane: ***

```
LogOp_Project QCOL: [AdventureWorks2012].[Production].[Product].ProductID
QCOL: [AdventureWorks2012].[Production].[Product].Name
LogOp_Select
LogOp_Get TBL: Production.Product Production.Product
```

```
TableID=1973582069 TableReferenceID=0 IsRow: COL: IsBaseRow1001
ScaOp_Comp x_cmpEq
ScaOp_Identifier QCOL:
[AdventureWorks2012].[Production].[Product].ProductID
ScaOp_Const TI(int,ML=4) XVAR(int,Not Owned,Value=877)
AncOp_PrjList
```

Wynik staje się szybko bardzo obszerny, nawet dla prostych zapytań; niestety nie ma żadnych udokumentowanych informacji, które mogą pomóc zrozumieć te drzewa i ich operacje. Te operacje to tak naprawdę operacje algebry relacyjnej. Na przykład operacja `Select`, pokazywana jako `LogOp_Select`, wybiera rekordy na podstawie danego predykatu i nie powinna być mylona z poleceniem `SELECT` języka `SQL`. Operacja `Project`, pokazywana jako `LogOp_Project`, jest wykorzystywana do specyfikowania kolumn mających wchodzić w skład wyniku. W zapytaniu żądamy jedynie kolumn `ProductID` i `name`, co znajduje odzwierciedlenie w operacji `LogOp_Project`. Więcej informacji na temat operacji algebry relacyjnej znajdziesz w książce Abrahama Silberschatza, Henry'ego F. Kortha i S. Sudarshana pod tytułem *Database System Concepts* (McGraw-Hill, 2010).

Teraz, kiedy znamy podstawowe informacje na temat drzew logicznych, w następnym podrozdziale pokażę, w jaki sposób `SQL Server` próbuje uprościć aktualne drzewo.

Upraszczenie

W tej fazie wykonywane są zmiany zapytania, albo może dokładniej: zmiany drzewa, które mają zredukować je do prostszej formy, dzięki czemu proces optymalizacji będzie łatwiejszy. Niektóre z tych uproszczeń obejmują następujące działania:

- ▶ Podzapytania są konwertowane na złączenia, ale ponieważ podzapytanie nie zawsze można bezpośrednio przetłumaczyć na złączenie wewnętrzne, mogą być wykorzystywane złączenia zewnętrzne i grupowanie.
- ▶ Nadmiarowe złączenia wewnętrzne i zewnętrzne mogą być usuwane. Typowym przykładem jest eliminacja złączenia z kluczem obcym, która ma miejsce, jeżeli `SQL Server` wykryje, że niektóre złączenia nie są potrzebne, ponieważ istnieją więzy integralności z kluczem obcym, a żądane są tylko kolumny tabeli odwodzącej się. Przykład eliminacji złączenia z kluczem obcym pokażę w dalszej części.
- ▶ Filtry w klauzuli `WHERE` są przesuwane w dół drzewa zapytania, aby wykonać filtrowanie możliwie najwcześniej oraz potencjalnie wykorzystać lepsze dopasowanie indeksów i kolumn kalkulowanych w dalszej części procesu optymalizacji (to uproszczenie jest znane jako *przesunięcie predykatów*).
- ▶ Sprzeczności są wykrywane i usuwane. Ponieważ te części zapytań nie są w ogóle wykonywane, `SQL Server` oszczędza zasoby takie jak I/O, blokady, pamięć i CPU, a co za tym idzie, przyspiesza samo zapytanie. Na przykład optymalizator

może wiedzieć, że żadne rekordy nie będą pasowały do predykatu, nawet bez dotykania jakichkolwiek danych. Warunek może być związany z więzami integralności semantycznej lub ze sposobem, w jaki jest napisane zapytanie. Oba scenariusze pokażę w dalszej części.

Wynikiem tego etapu jest uproszczone drzewo operatorów logicznych.

Wykrywanie sprzeczności

Przjrzyjmy się kilku przykładom procesu upraszczania, zaczynając od wykrywania sprzeczności. Na początek potrzebujemy tabeli z założonymi więzami integralności semantycznej — na szczęście tabela `Employee` ma takie ograniczenie (nie uruchamiaj kodu, więzy integralności są już założone):

```
ALTER TABLE HumanResources.Employee WITH CHECK ADD CONSTRAINT
CK_Employee_VacationHours CHECK (VacationHours>=-40 AND VacationHours<=240)
```

To ograniczenie sprawdza, czy liczba godzin urlopu jest liczbą pomiędzy -40 a 240, dlatego jeżeli wykonam zapytanie:

```
SELECT * FROM HumanResources.Employee WHERE VacationHours > 80
```

SQL Server wykorzysta operator *Clustered Index Scan*, zgodnie z rysunkiem 3.3.

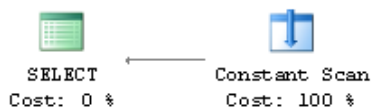


Rysunek 3.3. Plan bez wykrywania sprzeczności

Jeżeli jednak będę chciał wybrać wszystkich pracowników z liczbą godzin urlopowych większą niż 300, ze względu na ograniczenie semantyczne optymalizator będzie natychmiast wiedział, że żaden z rekordów nie będzie spełniał warunku. Wykonaj poniższe zapytanie:

```
SELECT * FROM HumanResources.Employee WHERE VacationHours > 300
```

Zgodnie z oczekiwaniami zapytanie nie zwróci żadnych rekordów, ale tym razem pokazany zostanie plan wykonywania analogiczny jak na rysunku 3.4.



Rysunek 3.4. Przykład wykrywania sprzeczności

Zauważ, że tym razem, zamiast operatora *Clustered Index Scan*, SQL Server wyko-

wierszy stałych i jest praktycznie bezkosztowy. Ponieważ nie ma potrzeby uzyskiwania dostępu do tabeli, SQL Server oszczędza zasoby takie jak I/O, blokady, pamięć i CPU, a tym samym poprawia wydajność zapytania.

Zobaczmy teraz, co się stanie, jeżeli wyłączę sprawdzanie tego ograniczenia:

```
ALTER TABLE HumanResources.Employee NOCHECK CONSTRAINT CK_Employee_VacationHours
```

Tym razem uruchomienie poprzedniego zapytania spowoduje wykorzystanie operatora *Clustered Index Scan*, ponieważ optymalizator nie może już korzystać z ograniczenia podczas podejmowania decyzji. Nie zapomnij powtórnie włączyć ograniczenia poprzez uruchomienie poniższego zapytania:

```
ALTER TABLE HumanResources.Employee WITH CHECK CHECK CONSTRAINT CK_Employee_VacationHours
```

Drugi przypadek zaprzeczenia występuje, jeżeli samo zapytanie zawiera zaprzeczenie. Spójrzmy na poniższe zapytanie:

```
SELECT * FROM HumanResources.Employee WHERE VacationHours > 10 AND VacationHours < 5
```

W tym przypadku żadne więzy integralności semantycznej nie są brane pod uwagę; oba predykaty są poprawne, a każdy z nich osobno spowodowałby zwrócenie rekordów, razem jednak się wykluczają. W rezultacie zapytanie nie zwraca rekordów, a plan pokazuje operator *Constant Scan*, podobnie jak w przypadku planu z rysunku 3.4. Może to wyglądać na źle napisane zapytanie, pamiętaj jednak, że niektóre predykaty mogą być zawarte na przykład w definicji widoku, a twórca zapytania może o nich nie wiedzieć. Dla poprzedniego zapytania widok może zawierać predykat `VacationHours > 10`, a autor zapytania mógł wywołać widok z predykatem `VacationHours < 5`. Ponieważ predykaty wzajemnie się wykluczają, znów zostanie wykorzystany operator *Constant Scan*. W niektórych skomplikowanych zapytaniach zaprzeczenie może być trudne do wykrycia.



UWAGA

Możesz wypróbować kilka podobnych prostych zapytań i nie uzyskać tego samego efektu. W niektórych przypadkach wykorzystanie planu trywialnego może zablokować powyższe zachowanie.

Spójrzmy teraz na drzewa logiczne tworzone podczas wykrywania zaprzeczeń — wykorzystamy do tego nieudokumentowaną flagę 8606:

```
SELECT * FROM HumanResources.Employee WHERE VacationHours > 300
OPTION (RECOMPILE, QUERYTRACEON 8606)
```

Otrzymamy poniższy wynik (dostosowany do strony książki):

*** Drzewo pierwotne: ***

```
LogOp_Project QCOL:[HumanResources].[Employee].BusinessEntityID
LogOp_Select
```

```

LogOp_Project
LogOp_Get TBL: HumanResources.Employee TableID=1237579447
TableReferenceID=0 IsRow: COL: IsBaseRow1001
AncOp_PrjList
  AncOp_PrjEl QCOL:
    [HumanResources].[Employee].OrganizationLevel
    ScaOp_UdtFunction EClrFunctionType_UdtMethodGetLevel
    IsDet NoDataAccess TI(smallint,Null,ML=2)
    ScaOp_Identifier QCOL:
      [HumanResources].[Employee].OrganizationNode
    ScaOp_Comp x_cmpGt
    ScaOp_Identifier QCOL:
      [HumanResources].[Employee].VacationHours
    ScaOp_Const TI(smallint,ML=2)
  AncOp_PrjList
*** Drzewo uproszczone: ***
LogOp_ConstTableGet (0) COL: Chk1000 COL: IsBaseRow1001 QCOL:
[HumanResources].[Employee].BusinessEntityID

```

Jak widzisz, wszystkie operatory logiczne są zastąpione przez drzewo uproszczone, składające się jedynie z operatora `LogOp_ConstTableGet`, który zostanie potem przetłumaczony na logiczny i fizyczny operator *Constant Scan*.

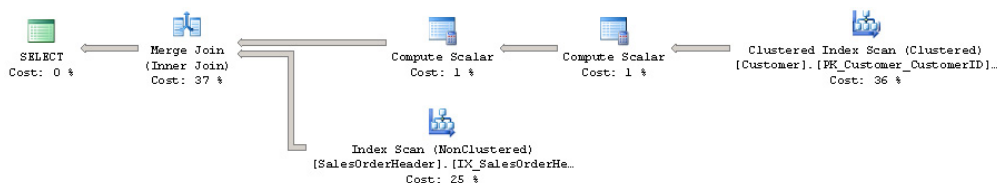
Usuwanie złączeń z kluczem obcym

Pokażę teraz przykład eliminacji złączenia z kluczem obcym. Poniższe zapytanie łączy dwie tabele i pokazuje plan wykonania z rysunku 3.5:

```

SELECT soh.SalesOrderID, c.AccountNumber
FROM Sales.SalesOrderHeader soh
JOIN Sales.Customer c ON soh.CustomerID = c.CustomerID

```



Rysunek 3.5. Pierwotny plan ze złączeniem dwóch tabel

Zobaczmy, co się stanie, jeżeli wykomentuję kolumnę `AccountNumber`:

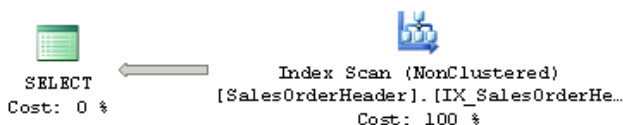
```

SELECT soh.SalesOrderID --, c.AccountNumber
FROM Sales.SalesOrderHeader soh
JOIN Sales.Customer c ON soh.CustomerID = c.CustomerID

```

Jeżeli jeszcze raz uruchomisz zapytanie, tabela `Customer` i operacja złączenia są eliminowane — widać to na planie z rysunku 3.6.

Zmiana ta ma dwa powody. Przede wszystkim, ponieważ kolumna `AccountNumber` nie jest już wymagana, nie są pobierane żadne kolumny z tabeli `Customer`. Wydaje się



Rysunek 3.6. Przykład eliminacji złączenia z kluczem obcym

jednak, że tabela Customer nadal jest potrzebna, gdyż jest częścią operatora równania w warunku złączenia. Oznacza to, że SQL Server musi mieć pewność, że dla każdego rekordu w tabeli SalesOrderHeader istnieje rekord w tabeli Customer.

W rzeczywistości ta walidacja jest wykonywana przez istniejące więzy klucza obcego, dlatego optymalizator może uznać złączenie za zbędne. Oto zdefiniowany klucz obcy (tym razem także uruchamianie tego kodu jest zbędne):

```
ALTER TABLE Sales.SalesOrderHeader WITH CHECK ADD CONSTRAINT
FK_SalesOrderHeader_Customer_CustomerID FOREIGN KEY(CustomerID)
REFERENCES Sales.Customer(CustomerID)
```

W ramach testu tymczasowo wyłącz klucz obcy za pomocą poniższego polecenia:

```
ALTER TABLE Sales.SalesOrderHeader NOCHECK CONSTRAINT
FK_SalesOrderHeader_Customer_CustomerID
```

Teraz powtórnie uruchom poprzednie zapytanie. Bez więzów klucza obcego SQL Server musi wykonać operację złączenia w celu sprawdzenia warunku złączenia. W rezultacie wykorzystany zostanie plan ze złączeniem obu tabel, podobny do tego z rysunku 3.5. Nie zapomnij powtórnie włączyć klucz obcy za pomocą poniższego kodu:

```
ALTER TABLE Sales.SalesOrderHeader WITH CHECK CHECK CONSTRAINT
FK_SalesOrderHeader_Customer_CustomerID
```

Możesz również zaobserwować to zachowanie na podstawie tworzonych drzew logicznych. Aby to zrobić, wykorzystaj powtórnie nieudokumentowaną flagę 8606, tak jak poniżej:

```
SELECT soh.SalesOrderID --, c.AccountNumber
FROM Sales.SalesOrderHeader soh
JOIN Sales.Customer c ON soh.CustomerID = c.CustomerID
OPTION (RECOMPILE, QUERYTRACEON 8606)
```

Zobaczysz wynik podobny do tego (dostosowanego do strony książki):

```
*** Drzewo pierwotne: ***
LogOp_Project QCOL: [soh].SalesOrderID
  LogOp_Select
    LogOp_Join
      LogOp_Project
        LogOp_Get TBL: Sales.SalesOrderHeader(alias TBL: soh)
        Sales.SalesOrderHeader TableID=1266103551
        TableReferenceID=0 IsRow: COL: IsBaseRow1001
```

```

AncOp_PrjList
  AncOp_PrjEl QCOL: [soh].SalesOrderNumber
    ScaOp_Intrinsic isnull
      ScaOp_Arithmetic x_aopAdd
...
LogOp_Project
LogOp_Get TBL: Sales.Customer(alias TBL: c)
Sales.Customer TableID=997578592 TableReferenceID=0
IsRow: COL: IsBaseRow1003
AncOp_PrjList
  AncOp_PrjEl QCOL: [c].AccountNumber
    ScaOp_Intrinsic isnull
      ScaOp_Arithmetic x_aopAdd
        ScaOp_Const TI(varchar collate 872468488,Var,Trim,ML=2)
        ScaOp_Udf dbo.ufnLeadingZeros IsDet
        ScaOp_Identifier QCOL: [c].CustomerID
...
*** Drzewo uproszczone: ***
LogOp_Join
  LogOp_Get TBL: Sales.SalesOrderHeader(alias TBL: soh)
  Sales.SalesOrderHeader TableID=1266103551 TableReferenceID=0
  IsRow: COL: IsBaseRow1001
  LogOp_Get TBL: Sales.Customer(alias TBL: c) Sales.Customer
  TableID=997578592 TableReferenceID=0 IsRow: COL: IsBaseRow1003
  ScaOp_Comp x_cmpEq
    ScaOp_Identifier QCOL: [c].CustomerID
    ScaOp_Identifier QCOL: [soh].CustomerID
*** Drzewo z wyeliminowanym złączeniem: ***
LogOp_Get TBL: Sales.SalesOrderHeader(alias TBL: soh)
Sales.SalesOrderHeader TableID=1266103551 TableReferenceID=0 IsRow: COL:
IsBaseRow1001

```

W wyniku możesz zaobserwować, że chociaż drugie drzewo jest odrobinę uproszczone (drzewo pierwotne zostało okrojone ze względu na jego rozmiary), to i tak wykorzystuje operatory logiczne LogOp_Get dla tabel Sales.SalesOrderHeader i Sales.↪Customer. Zauważ, że pierwotne drzewo zostało uproszczone, a dopiero później wyeliminowane zostało złączenie.

Plan trywialny

Inicjalizacja i wykonanie procesu optymalizacji dla prostych zapytań, które nie wymagają szacowania kosztów, może być kosztowne. Aby uniknąć tych operacji dla prostych zapytań, SQL Server wykorzystuje optymalizację planu trywialnego. W skrócie: jeżeli istnieje tylko jeden sposób — lub jeden oczywisty najlepszy sposób — na wykonanie zapytania, w zależności od definicji zapytania i dostępnych metadanych, można uniknąć wiele pracy. Na przykład następujące zapytanie wykonane na bazie Adventure↪Works2012 wykorzysta plan trywialny:

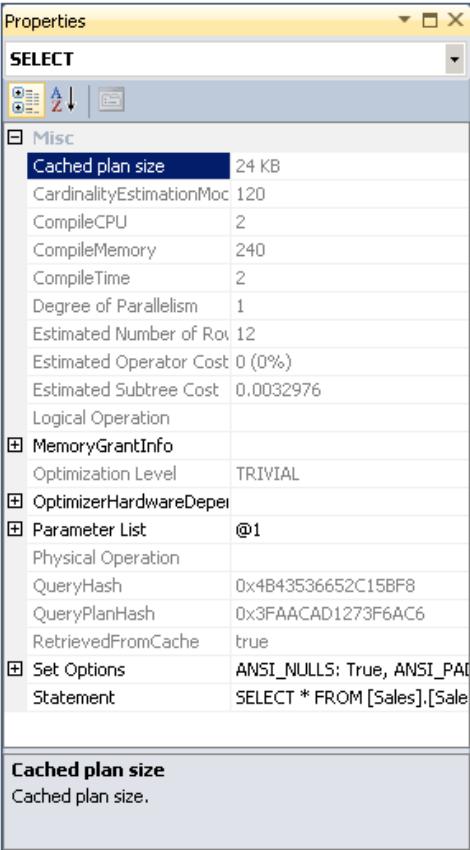
```

SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659

```

Plan wykonania pokaże, czy wykonana została optymalizacja planu trywialnego. Wpis `Optimization Level` (poziom optymalizacji) w oknie właściwości będzie zawierał słowo `TRIVIAL`, zgodnie z rysunkiem 3.7. Plan XML będzie natomiast zawierał atrybut `StatementOptmLevel` z wartością `TRIVIAL` — fragment takiego planu znajduje się poniżej:

```
<StmtSimple StatementCompId="1" StatementEstRows="12" StatementId="1"
StatementOptmLevel="TRIVIAL" StatementSubTreeCost="0.0032976"
StatementText="SELECT * FROM [Sales].[SalesOrderDetail] WHERE [SalesOrderID]=@1"
StatementType="SELECT" QueryHash="0x801851E3A6490741"
QueryPlanHash="0x3E34C903A0998272" RetrievedFromCache="true">
```



Rysunek 3.7. Właściwości planu trywialnego

Jak wspomniałem na początku tego rozdziału, dodatkowe informacje dotyczące procesu optymalizacji można uzyskać za pośrednictwem widoku DMV `sys.dm_exec_query_optimizer_info`, który dla tego zapytania zwróci wynik podobny do poniższego:

counter	occurrence	value
elapsed time	1	0.076
final cost	1	0.0032976
maximum DOP	1	0
optimizations	1	1
tables	1	1
trivial plan	1	1

Wynik pokazuje, że jest to optymalizacja planu trywialnego, wykorzystująca jedną tabelę i maksymalny DOP równy 0. Wyświetlany jest również czas i całkowity koszt. Jeżeli jednak delikatnie zmienimy zapytanie i będziemy pobierać ProductID zamiast SalesOrderID, otrzymamy pełną optymalizację:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 870
```

W tym przypadku wartość właściwości Optimization Level czy StatementOptLevel będzie równa FULL, co oczywiście oznacza, że zapytanie nie kwalifikowało się do wykorzystania planu trywialnego i musiała zostać wykonana pełna optymalizacja. Pełna optymalizacja jest używana dla bardziej skomplikowanych zapytań lub zapytań wykorzystujących bardziej złożone funkcjonalności, które do podjęcia decyzji odnośnie wyboru planu będą wymagały porównywania kosztów potencjalnych planów (wyjaśnię to w następnym podrozdziale). W tym konkretnym przykładzie, ponieważ predykat ProductID = 870 może spowodować zwrócenie zera, jednego lub więcej rekordów, mogą zostać stworzone różne plany w zależności od szacowanej kardynalności i dostępnych struktur negocjacyjnych. Inaczej było w poprzednim zapytaniu wykorzystującym kolumnę SalesOrderID, która jest częścią indeksu unikalnego, dlatego może zwracać zero lub jeden rekord.

Możesz też użyć nieudokumentowanej flagi do wyłączenia optymalizacji planu trywialnego, co można wykorzystać na potrzeby testów. Przedstawia to poniższy przykład:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
OPTION (RECOMPILE, QUERYTRACEON 8757)
```

Jeżeli po raz kolejny wykorzystamy widok DMV sys.dm_exec_query_optimizer_info, zobaczymy, że teraz, zamiast planu trywialnego, mamy odpowiedź, a optymalizacja dochodzi do fazy search 1, którą objaśnię w następnym rozdziale.

counter	occurrence	value
elapsed time	1	0.001
final cost	1	0.0032976
Hints	1	1
maximum DOP	1	0
optimizations	1	1
search 1	1	1
search 1 tasks	1	131
search 1 time	1	0
tables	1	1
tasks	1	131

Reguły transformacji

Jak już wcześniej wspomniałem, optymalizator zapytań SQL Servera wykorzystuje reguły transformacji do eksplorowania obszaru poszukiwań — czyli do odszukiwania możliwych planów dla danego zapytania. Reguły transformacji oparte są na algebrze relacyjnej; na podstawie drzewa operatorów logicznych generowane są równoważne drzewa alternatywne w postaci drzew operatorów relacyjnych. Na najbardziej podstawowym poziomie zapytanie składa się z wyrażeń logicznych, a zastosowanie reguł transformacji spowoduje wygenerowanie logicznych i fizycznych alternatyw, które są przechowywane w pamięci (w strukturze o nazwie Memo) w trakcie całego procesu optymalizacji. Jak wyjaśniłem wcześniej w tym rozdziale, optymalizator wykorzystuje maksymalnie trzy fazy optymalizacji, a podczas każdej z nich stosowane są inne reguły transformacji.

Każda z reguł ma wzorzec i substytut. Wzorzec to wyrażenie, które ma zostać przeanalizowane i dopasowane, a substytut to równoważne wyrażenie, które jest generowane jako wynik. Na przykład dla reguły komutatywnej, którą omówię poniżej, reguła transformacji może być zdefiniowana w następujący sposób:

Wyrażenie1 join Wyrażenie2 -> Wyrażenie2 join Wyrażenie1

Oznacza to, że SQL Server dopasuje wzorzec *Wyrażenie1 join Wyrażenie2*, czyli np. *Individual join Customer*, i zwróci wyrażenie równoznaczne *Wyrażenie2 join Wyrażenie1*, czyli w naszym przypadku *Customer join Individual*. Oba wyrażenie są logicznie równoważne, ponieważ oba zwracają dokładnie te same wyniki.

Na początku drzewo zapytań zawiera tylko wyrażenia logiczne, a reguły transformacji są stosowane do logicznych wyrażeń w celu wygenerowania logicznych lub fizycznych wyrażeń. Na przykład wyrażenie logiczne może być definicją logicznego

złączenia, podczas gdy wyrażenie fizyczne może być faktyczną implementacją złączenia, taką jak Merge Join lub Hash Join. Pamiętaj, że reguły transformacji nie mogą być stosowane do wyrażeń fizycznych.

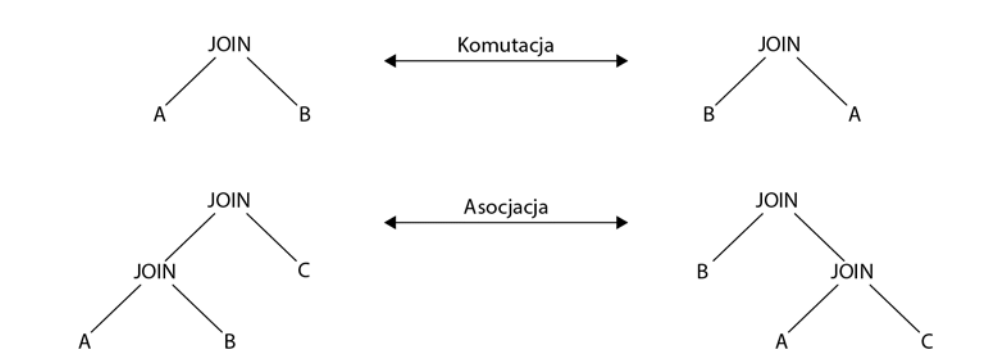
Główne typy reguł transformacji to reguły uproszczenia, eksploracji i implementacji. Reguły uproszczenia jako wynik produkują prostsze drzewa logiczne i są głównie wykorzystywane podczas fazy upraszczania, jeszcze przed pełną optymalizacją, zgodnie z wcześniejszymi wyjaśnieniami. Reguły eksploracji, nazywane również regułami transformacji logicznej, generują alternatywy logiczne. Reguły implementacji, lub reguły transformacji fizycznej, wykorzystywane są do tworzenia alternatyw fizycznych. Zarówno reguły eksploracji, jak i reguły implementacji są wywoływane podczas fazy pełnej optymalizacji.

Przykłady reguł eksploracji zawierają reguły komutatywne i asocjacyjne, które są wykorzystywane podczas operacji złączenia i zostały pokazane na rysunku 3.8. Reguły te są definiowane odpowiednio jako:

$$A \text{ join } B \rightarrow B \text{ join } A$$

oraz:

$$(A \text{ join } B) \text{ join } C \rightarrow B \text{ join } (A \text{ join } C)$$



Rysunek 3.8. Reguły komutacji i asocjacji

Reguła komutatywna ($A \text{ join } B \rightarrow B \text{ join } A$) oznacza, że $A \text{ join } B$ jest równoznaczne z $B \text{ join } A$, ałączenie tabel A i B w dowolnej kolejności zwróci te same wyniki. Zauważ również, że dwukrotne zastosowanie reguły komutacyjnej spowoduje powtórne wygenerowanie pierwotnego wyrażenia, czyli jeżeli najpierw wykonasz transformację w celu uzyskania $B \text{ join } A$, a następnie wykonasz tę samą transformację, powtórnie uzyskasz $A \text{ join } B$. Optymalizator potrafi jednak radzić sobie z tym problemem, aby uniknąć zduplikowanych wyrażeń. W ten sam sposób reguła asocjacyjna pokazuje, że $(A \text{ join } B) \text{ join } C$ jest równoznaczne z $B \text{ join } (A \text{ join } C)$, ponieważ oba te wyrażenia także generują ten sam wynik. Przykładem reguły implementacji jest wybór fizycznego algorytmu, takiego jak Merge Join lub Hash Join, dla logicznego złączenia.

Tak więc optymalizator zapytań wykorzystuje reguły transformacji do generowania i analizowania alternatywnych planów wykonania. Należy jednak pamiętać, że stosowanie reguł niekoniecznie zmniejsza koszt generowanych alternatyw, a komponent kosztów i tak musi oszacować koszty. Chociaż zarówno alternatywy logiczne, jak i fizyczne są przechowywane w strukturze Memo, tylko alternatywy fizyczne będą miały oszacowane koszty. Należy zatem pamiętać, że choć te alternatywy są równoznaczne i zwracają te same wyniki, koszty ich fizycznych reprezentacji mogą być bardzo różne. Ostatnia wybrana zostanie, co mam nadzieję jest już jasne, najlepsza (lub, jeśli wolisz, „najtańsza”) fizyczna alternatywa przechowywana w Memo.

Na przykład implementacja `A join B` może mieć różny koszt w zależności od tego, czy wybrany zostanie algorytm `Nested Loops Join` czy `Hash Join`. Dodatkowo dla niektórych fizycznych implementacji złączenie `A join B` może mieć inną wydajność niż `B join A`. Jak wyjaśniam w rozdziale 4., wydajność złączenia jest inna w zależności od tego, która tabela będzie wybrana jako wewnętrzna lub zewnętrzna w złączeniu `Nested Loops Join`, lub od tabel wsadowych w `Hash Join`. Jeżeli chcesz się dowiedzieć, dlaczego optymalizator nie wybiera danego algorytmu złączenia, możesz wykorzystać odpowiedź do wymuszenia konkretnego fizycznego złączenia i porównania, jaki będzie koszt planów z odpowiedzią i oryginalnego.

To są podstawowe zasady reguł transformacji, a widok DMV `sys.dm_exec_query_transformation_stats` udostępnia informacje o istniejących regułach transformacji i sposobie ich wykorzystania przez optymalizator. Zgodnie z tym widokiem SQL Server zawiera aktualnie 395 reguł transformacji, a w przyszłych wersjach produktu mogą znaleźć się kolejne.

Podobnie jak w przypadku `sys.dm_exec_query_optimizer_info`, ten widok DMV zawiera informacje o optymalizacjach wykonanych od czasu uruchomienia danej instancji SQL Servera, ale może również zostać wykorzystany do pobrania informacji na temat konkretnego zapytania lub zadania poprzez zrobienie dwóch migawek widoku (przed optymalizacją zapytania i po niej) i ręcznego porównania różnic.

Aby zacząć pracę z tym widokiem, wykonaj zapytanie:

```
SELECT * FROM sys.dm_exec_query_transformation_stats
```

Oto przykładowy wynik z mojego środowiska testowego w SQL Serverze 2014, pokazujący pierwsze kilka z 395 rekordów (wynik wyedytowany, aby zmieścił się na stronie):

name	promise_total	promise_avg	promised	build_substitute	succeeded
JNtoNL	203352155	92.96051768	2187511	350792	285532
LOJNtoNL	35938188	449.0701754	80028	80028	79504
LSJNtoNL	40614706	450.6936171	90116	90116	90116
LASJNtoNL	4029039	451.6353548	8921	8921	8921
JNtoSM	366499276	418.7495941	875223	814754	441413

name	promise_total	promise_avg	promised	build_substitute	succeeded
F0JNtoSM	6356	454	14	14	4
L0JNtoSM	11180944	443.3891422	25217	24996	17098
R0JNtoSM	11179128	443.3874589	25213	24992	17094
LSJNtoSM	4263232	443.2094812	9619	9453	1495
RSJNtoSM	4263232	443.2094812	9619	9453	6922

Widok `sys.dm_exec_query_transformation_stats` zawiera coś, co nazywane jest informacją o promesie, która informuje optymalizator, jak użyteczna może być dana transformacja. Pierwsze pole tabeli wynikowej zawiera nazwę transformacji. Na przykład pierwsze trzy reguły to `JNtoNL` (*Join to Nested Loops Join*), `L0JNtoNL` (*Left Outer Join to Nested Loops Join*) i `JNtoSM` (*Join to Sort Merge Join* — gdzie *Sort Merge Join* to naukowa nazwa operatora *Merge Join*).

Problemy dotyczące zbierania danych za pomocą widoku `sys.dm_exec_query_optimizer_info` dotyczą także widoku `dm_exec_query_transformation_stats`, dlatego zapytanie z listingu 3.2 pomoże Ci wyodrębnić informacje dotyczące optymalizacji konkretnego zapytania z pominięciem danych z zapytań powiązanych. Zapytanie bazuje na kolumnie `succeeded`, która śledzi, ile razy dana reguła została wykorzystana i pomyślnie stworzyła wynik.

Listing 3.2. Kod wykorzystujący widok `sys.dm_exec_query_transformation_stats`

```
-- optymalizacja tych zapytań teraz,
-- aby nie zaburzały zebranych danych
GO
SELECT *
INTO before_query_transformation_stats
FROM sys.dm_exec_query_transformation_stats
GO
SELECT *
INTO after_query_transformation_stats
FROM sys.dm_exec_query_transformation_stats
GO
DROP TABLE after_query_transformation_stats
DROP TABLE before_query_transformation_stats
-- rozpoczęcie właściwego wykonywania
GO
SELECT *
INTO before_query_transformation_stats
FROM sys.dm_exec_query_transformation_stats
GO
-- tutaj wstaw swoje zapytanie
SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
-- zachowaj poniższe, aby wymusić nową optymalizację
OPTION (RECOMPILE)
GO
SELECT *
```

```
INTO after_query_transformation_stats
FROM sys.dm_exec_query_transformation_stats
GO
SELECT a.name, (a.promised - b.promised) as promised
FROM before_query_transformation_stats b
JOIN after_query_transformation_stats a
ON b.name = a.name
WHERE b.succeeded <> a.succeeded
DROP TABLE before_query_transformation_stats
DROP TABLE after_query_transformation_stats
```

Na przykład test z bardzo prostym zapytaniem, takim jak:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
```

które zostało już zawarte na wcześniejszym listingu, pokaże, że wykorzystane zostały następujące reguły transformacji:

name	promised
ProjectToComputeScalar	1
SelIdxToRng	1
SelPrjGetToTrivialScan	1
SelToTrivialFilter	1

Wynikowy plan to plan trywialny. Dodajmy nieudokumentowaną flagę 8757, aby uniknąć generowania planu trywialnego (tylko na potrzeby testów):

```
SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 43659
OPTION (RECOMPILE, QUERYTRACEON 8757)
```

Otrzymalibyśmy wtedy następujący wynik:

name	promised
AddCCPrjToGet	2
GetIdxToRng	1
GetToIdxScan	2
GetToScan	2
ProjectToComputeScalar	2
SelectToFilter	1
SelIdxToRng	1
SelToIdxStrategy	1

Przetestujmy teraz bardziej skomplikowane zapytanie. Do kodu z listingu 3.2 wstaw poniższe zapytanie, aby zobaczyć, jakie reguły transformacji zostaną dla niego wykorzystane:

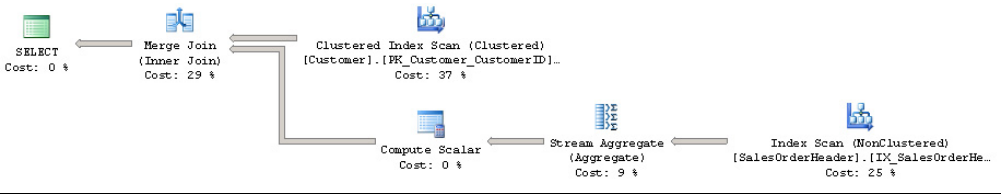
```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
```

Jak pokazują poniższe wyniki, podczas procesu optymalizacji wykorzystanych zostało 17 reguł transformacji:

name	promised
AppIdxToApp	0
EnforceSort	23
GbAggBeforeJoin	4
GbAggToHS	8
GbAggToStrm	8
GenLGAgg	2
GetIdxToRng	0
GetToIdxScan	4
GetToScan	4
ImplRestrRemap	3
JNtoHS	6
JNtoIdxLookup	6
JNtoSM	6
JoinCommute	2
ProjectToComputeScalar	2
SelIdxToRng	6
SELonJN	1

Podpowiedzi mogą wyłączyć niektóre z tych transformacji w celu uzyskania określonego zachowania (ten temat omówię dokładniej w rozdziale 10.). Do eksperymentów z efektami reguł możesz również wykorzystać nieudokumentowane polecenia DBCC RULEON i DBCC RULEOFF, pozwalające odpowiednio włączać i wyłączać reguły transformacji, a tym samym zdobywać dodatkowe informacje dotyczące działania optymalizatora zapytań. Zanim jednak to zrobisz, muszę Cię ostrzec: ponieważ polecenia te wpływają na cały proces optymalizacji wykonywany przez optymalizator zapytań, powinny być wykorzystywane tylko w środowisku testowym.

Aby zademonstrować efekty działania tych poleceń, poprzednie zapytanie zwróciłoby plan przedstawiony na rysunku 3.9.

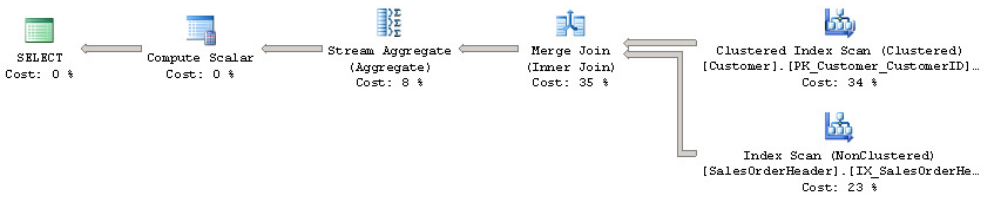


Rysunek 3.9. Oryginalny plan wykonania

Tutaj możesz zaobserwować między innymi, że SQL Server przenosi agregację poniżej złączenia (*Stream Aggregate* przed *Merge Join*). Optymalizator może przemieszczać agregacje i znacznie zmniejszać szacowanie kardynalności w planie najwcześniej jak to możliwe. Czynność ta wykonywana jest przez regułę *GbAggBeforeJoin* (czyli *Group By Aggregate Before Join*), która znajduje się w poprzednio pokazanych wynikach. Ta konkretna reguła transformacji jest wykorzystywana, tylko jeżeli spełnione są konkretne warunki — na przykład kiedy klauzula *GROUP BY* zawiera kolumny złączenia, co ma miejsce w naszym przykładzie. Aby tymczasowo wyłączyć regułę *GbAggBeforeJoin*, wykonaj następujące polecenie:

```
DBCC RULEOFF('GbAggBeforeJoin')
```

Po wyłączeniu reguły i powtórnym uruchomieniu zapytania, plan, przedstawiony na rysunku 3.10, pokaże teraz agregację po złączeniu, co według optymalizatora zapytań jest bardziej kosztownym planem. Możesz to zweryfikować, spoglądając na ich szacowany koszt, który wynosi odpowiednio 0,285331 i 0,312394 (liczby te nie są pokazane na rysunkach, ale możesz je zobaczyć, zgodnie z wcześniejszymi wyjaśnieniami, najeżdżając myszą na ikonę *SELECT* pod pozycją *Estimated Subtree Cost*). Zauważ, że na potrzeby tego ćwiczenia, aby zobaczyć nowy plan, być może będziesz musiał wymusić optymalizację — możesz skorzystać z podpowiedzi *OPTION (RECOMPILE)* lub którejś z wcześniej omówionych metod usuwania planów z magazynu planów, np. *DBCC FREEPROCCACHE*.



Rysunek 3.10. Plan z wyłączoną regułą *GbAggBeforeJoin*

Co więcej, jest kilka innych nieudokumentowanych poleceń pokazujących, które transformacje są włączone, a które wyłączone — są to odpowiednio *DBCC SHOWONRULES*

i DBCC SHOWOFFRULES. Domyślnie DBCC SHOWONRULES wyświetli 395 reguł transformacji zawartych w widoku sys.dm_exec_query_transformation_stats. Aby to przetestować, uruchom poniższy kod:

```
DBCC TRACEON (3604)
DBCC SHOWONRULES
```

Zaczynamy to ćwiczenie od włączenia flagi 3604, która, jak wcześniej wyjaśniałem, sprawia, że SQL Server przesyła wyniki do klienta (w tym przypadku do sesji Management Studio). Po tej operacji wynik DBCC SHOWONRULES, a później także DBCC SHOWOFFRULES, DBCC RULEON i DBCC RULEOFF będą wyświetlane. Poniżej pokazany został wynik (tylko kilka reguł ze względu na oszczędność miejsca). Poprzednio wyłączona reguła nie zostanie uwzględniona w wynikach.

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
Rules that are on globally:

```
JNtoNL
LOJNtoNL
LSJNtoNL
LASJNtoNL
JNtoSM
FOJNtoSM
LOJNtoSM
ROJNtoSM
LSJNtoSM
RSJNtoSM
LASJNtoSM
RASJNtoSM
```

Poniższy kod pokaże reguły, które są wyłączone:

```
DBCC SHOWOFFRULES
```

W naszym przypadku tylko jedna reguła została wyłączona:

Rules that are off globally:

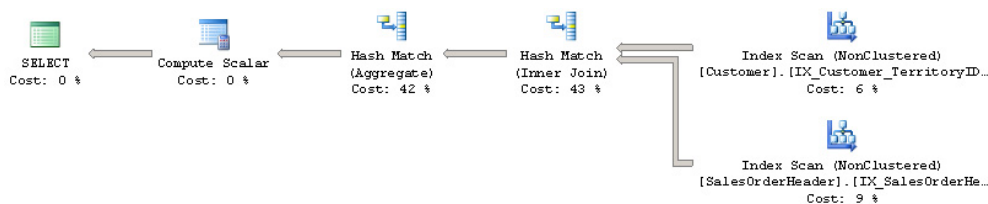
```
GbAggBeforeJoin
```

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Kontynuując nasz przykład, możemy wyłączyć wykorzystanie *Merge Join*, wyłączając regułę JNtoSM (*Join to Sort Merge Join*) za pomocą następującego kodu:

```
DBCC RULEOFF('JNtoSM')
```

Jeżeli podałeś za przykładami, tym razem polecenie DBCC RULEOFF pokaże wynik wskazujący, że reguła została wyłączona dla pewnego identyfikatora SPID. Oznacza to również, że polecenia DBCC RULEON i DBCC RULEOFF działają na poziomie sesji, ale nawet w tym przypadku Twoje polecenia mogą mieć wpływ na cały serwer, ponieważ stworzone przez Ciebie plany mogą być przechowywane w magazynie planów i potencjalnie wykorzystane przez inne sesje. Uruchomienie tego samego zapytania po raz kolejny da nam zupełnie nowy plan, wykorzystujący zarówno operacje *Hash Join*, jak i *Hash Aggregate*, zgodnie z rysunkiem 3.11.



Rysunek 3.11. Plan z wyłączoną regułą JNtoSM

Z rozdziału 10. dowiesz się, jak to samo zachowanie wymusić za pomocą podpowiedzi.

Zanim skończymy, nie zapomnij powtórnie włączyć reguł GbAggBeforeJoin i JNtoSM za pomocą poniższych poleceń:

```
DBCC RULEON('JNtoSM')
DBCC RULEON('GbAggBeforeJoin')
```

Następnie, korzystając z poniższego polecenia, sprawdź, czy nie zostały jakieś wyłączone reguły:

```
DBCC SHOWOFFRULES
```

Możesz również po prostu zamknąć swoją aktualną sesję, ponieważ polecenia DBCC RULEON i DBCC RULEOFF działają na poziomie sesji. Możesz też chcieć wyczyścić magazyn planów (aby upewnić się, że żaden z testowych planów nie pozostał w pamięci) poprzez uruchomienie polecenia:

```
DBCC FREEPROCCACHE
```

Drugim sposobem na osiągnięcie tego efektu jest użycie (także nieudokumentowanej) podpowiedzi QUERYRULEOFF. Na przykład poniższy kod wykorzystujący podpowiedź QUERYRULEOFF wyłączy regułę GbAggBeforeJoin tylko dla aktualnej optymalizacji:

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
OPTION (RECOMPILE, QUERYRULEOFF GbAggBeforeJoin)
```

Możesz dodać więcej niż jedną taką podpowiedź, jak w poniższym przykładzie, w którym wyłączone zostaną reguły GbAggBeforeJoin i JNtoSM:

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
OPTION (RECOMPILE, QUERYRULEOFF GbAggBeforeJoin, QUERYRULEOFF JNtoSM)
```

Optymalizator zapytań będzie posłuszny wyłączeniu reguł zarówno na poziomie sesji, jak i na poziomie zapytania. Nie ma podpowiedzi QUERYRULEON pozwalającej powtórnie włączyć regułę wyłączoną na poziomie sesji. Ponieważ QUERYRULEOFF to podpowiedź dla zapytania, dotyczy wyłącznie aktualnej optymalizacji.

W obu przypadkach, DBCCRULEOFF i QUERYRULEOFF, możesz znaleźć się w sytuacji, w której wyłączone zostało tyle reguł, że optymalizator nie jest w stanie zbudować planu zapytania. Jeżeli na przykład uruchomisz zapytanie:

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
OPTION (RECOMPILE, QUERYRULEOFF GbAggToStrm, QUERYRULEOFF GbAggToHS)
```

otrzymasz następujący błąd:

```
Msg 8622, Level 16, State 1, Line 1
Query processor could not produce a query plan because of the hints defined in this query.
↳Resubmit the query without specifying any hints and without using SET FORCEPLAN.
```

W tym konkretnym przykładzie wyłączam reguły GbAggToStrm (*Group by Aggregate to Stream*) i GbAggToHS (*Group by Aggregate to Hash*). Przynajmniej jedna z tych reguł jest wymagana, aby wykonać agregację; GbAggToStrm pozwala na wykorzystanie agregacji *Stream Aggregate*, a GbAggToHS — agregacji *Hash Aggregate*.

Ponieważ optymalizator wie, że zapytanie wykorzystuje podpowiedzi, prosi o powtórne uruchomienie zapytania bez podpowiedzi. Wyłączenie reguł GbAggToStrm i GbAggToHS za pomocą omówionego wcześniej polecenia DBCC RULEOFF i uruchomienie zapytania bez podpowiedzi zaowocuje poważniejszym komunikatem błędu:

```
Msg 8624, Level 16, State 1, Line 1
Internal Query Processor Error: The query processor could not produce a query plan.
↳For more information, contact Customer Support Services.
```

W końcu możesz również uzyskać informacje o regułach dzięki nieudokumentowanej fładze 2373. Choćby wynik jest obszerny i zawiera także informacje o pamięci, można go użyć do sprawdzenia, jakie reguły zostały wykorzystane podczas optymalizacji danego zapytania. Na przykład:

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
OPTION (RECOMPILE, QUERYTRACEON 2373)
```

spowoduje pokazanie następującego wyniku (wyedytowanego w celu dostosowania do strony):

```
Memory before rule NormalizeGbAgg: 27
Memory after rule NormalizeGbAgg: 27
Memory before rule IJtoIJSEL: 27
Memory after rule IJtoIJSEL: 28
Memory before rule MatchGet: 28
Memory after rule MatchGet: 28
Memory before rule MatchGet: 28
Memory after rule MatchGet: 28
Memory before rule JoinToIndexOnTheFly: 28
Memory after rule JoinToIndexOnTheFly: 28
```

```

Memory before rule JoinCommute: 28
Memory after rule JoinCommute: 28
Memory before rule JoinToIndexOnTheFly: 28
Memory after rule JoinToIndexOnTheFly: 28
Memory before rule GbAggBeforeJoin: 28
Memory after rule GbAggBeforeJoin: 28
Memory before rule GbAggBeforeJoin: 28
Memory after rule GbAggBeforeJoin: 30
Memory before rule IJtoIJSEL: 30
Memory after rule IJtoIJSEL: 30
Memory before rule NormalizeGbAgg: 30
Memory after rule NormalizeGbAgg: 30
Memory before rule GenLGAgg: 30

```

Memo

Struktura Memo została pierwotnie zdefiniowana w 1993 roku jako *The Volcano Optimizer Generator* (generator optymalizacji Volcano) przez Goetza Graefego i Williama McKennę. Analogicznie optymalizator zapytań w SQL Serverze bazuje na frameworku Cascades, który tak naprawdę jest potomkiem optymalizatora Volcano.

Memo to struktura danych wykorzystywana do przechowywania alternatyw wygenerowanych i analizowanych przez optymalizator zapytań. Te alternatywy mogą być operatorami logicznymi lub fizycznymi i są zorganizowane w grupy równoważnych alternatyw, takich że każda alternatywa w tej samej grupie daje takie same wyniki. Alternatywy z tej samej grupy dzielą również te same logiczne właściwości i w ten sam sposób, w jaki operatory mogą odnosić się do innych operatorów w drzewie relacyjnym, drzewa mogą odnosić się do innych drzew w strukturze Memo.

Dla każdej optymalizacji tworzona jest nowa struktura Memo. Optymalizator najpierw kopiuje wyrażenia logiczne pierwotnego wyrażenia do struktury Memo, umieszczając każdy operator drzewa zapytania w osobnej grupie, a następnie uruchamia cały proces optymalizacji. Podczas tego procesu do wygenerowania alternatyw wykorzystywane są reguły transformacji, zaczynając od tych początkowych wyrażen.

Kiedy reguły transformacji tworzą nowe alternatywy, są one dodawane do równoważnych grup. Reguły transformacji mogą również stworzyć nowe wyrażenie, które nie jest równoznaczne z żadną istniejącą grupą, a wtedy tworzona jest nowa grupa. Jak już wspomniałem, każda alternatywa w grupie to proste wyrażenie logiczne lub fizyczne, takie jak złączenie lub skan, a plan budowany jest jako kombinacja tych alternatyw. Liczba tych alternatyw (a nawet grup) w strukturze Memo może być ogromna.

Chociaż istnieje możliwość, że różne kombinacje reguł transformacji mogą tworzyć te same efekty, jak wcześniej wspomniałem, struktura Memo jest zaprojektowana tak, aby unikać zarówno duplikacji alternatyw, jak i nadmiarowych optymalizacji. Takie podejście oszczędza pamięć i jest bardziej efektywne, ponieważ nie trzeba wyszukiwać tych samych alternatyw więcej niż raz.

Choć zarówno alternatywy logiczne, jak i fizyczne są przechowywane w strukturze Memo, tylko dla alternatyw fizycznych określany jest koszt. Dlatego na końcu procesu optymalizacji Memo zawiera wszystkie alternatywy rozważane przez optymalizator, ale, w oparciu o koszty operacji, wybierany jest tylko jeden plan.

Wykorzystam teraz kilka nieudokumentowanych flag, aby pokazać, jak zasilana jest struktura Memo i jak wygląda po zakończeniu procesu optymalizacji. Do pokazania początkowej zawartości struktury Memo możemy wykorzystać nieudokumentowaną flagę 8608. Zauważ również, że bardzo proste zapytanie może nie pokazać niczego, jak w poniższym przykładzie (pamiętaj, aby wcześniej włączyć flagę 3604):

```
SELECT ProductID, name FROM Production.Product
OPTION (RECOMPILE, QUERYTRACEON 8608)
```

To zapytanie wykorzysta optymalizację trywialną, czyli nie wymaga pełnej optymalizacji i struktury Memo, a stworzony plan będzie planem trywialnym. Możesz wymusić pełną optymalizację za pośrednictwem nieudokumentowanej flagi 8757. Uruchom poniższe zapytanie:

```
SELECT ProductID, name FROM Production.Product
OPTION (RECOMPILE, QUERYTRACEON 8608, QUERYTRACEON 8757)
```

W tym przypadku zobaczymy prostą strukturę Memo:

```
--- Początkowa struktura Memo ---
Root Group 0: Card=504 (Max=10000, Min=0)
0 LogOp_Get
```

Wypróbujmy proste zapytanie, które nie będzie kwalifikowało się do planu trywialnego, i spójrzmy na jego ostateczne drzewo logiczne dzięki wykorzystaniu nieudokumentowanej flagi 8606:

```
SELECT ProductID, ListPrice FROM Production.Product
WHERE ListPrice > 90
OPTION (RECOMPILE, QUERYTRACEON 8606)
```

Ostateczne drzewo będzie następujące:

```
*** Drzewo po normalizacji ***
LogOp_Select
  LogOp_Get TBL: Production.Product Production.Product
  TableID=1973582069 TableReferenceID=0 IsRow: COL: IsBaseRow1001
  ScaOp_Comp x_cmpGt
    ScaOp_Identifier QCOL: Production].[Product].ListPrice
    ScaOp_Const TI(money,ML=8)
    XVAR(money,Not Owned,Value=(10000units)=(900000))
```

Teraz, za pomocą nieudokumentowanej flagi 8606, przyjrzymy się, jak wygląda początkowa struktura Memo:

```
SELECT ProductID, ListPrice FROM Production.Product
WHERE ListPrice > 90
OPTION (RECOMPILE, QUERYTRACEON 8608)
```

Otrzymamy wynik zbliżony do poniższego:

```
--- Początkowa struktura Memo ---
Root Group 4: Card=216 (Max=10000, Min=0)
  0 LogOp_Select 0 3
Group 3:
  0 ScaOp_Comp 1 2
Group 2:
  0 ScaOp_Const
Group 1:
  0 ScaOp_Identifier
Group 0: Card=504 (Max=10000, Min=0)
  0 LogOp_Get
```

Jak widzisz, operatory na drzewie logicznym są kopiowane do struktury Memo, a każdy operator umieszczany jest w swojej własnej grupie. Grupę 4 nazywamy grupą korzenia, ponieważ znajduje się w niej operator z korzenia drzewa logicznego planu pierwotnego (czyli z korzenia pierwotnego drzewa zapytania).

Możemy też skorzystać z nieudokumentowanej flagi 8615, aby zobaczyć strukturę Memo na końcu procesu optymalizacji:

```
SELECT ProductID, ListPrice FROM Production.Product
WHERE ListPrice > 90
OPTION (RECOMPILE, QUERYTRACEON 8615)
```

Otrzymamy strukturę Memo z następującą zawartością:

```
--- Końcowa struktura Memo ---
Root Group 4: Card=216 (Max=10000, Min=0)
  1 PhyOp_Filter 0.2 3.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 0.0129672
  0 LogOp_Select 0 3
Group 3:
  0 ScaOp_Comp 1.0 2.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 3
Group 2:
  0 ScaOp_Const Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 1
Group 1:
  0 ScaOp_Identifier Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 1
Group 0: Card=504 (Max=10000, Min=0)
  2 PhyOp_Range 1 ASC Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 0.0127253
  0 LogOp_Get
```

Spójrzmy teraz na pełen przykład. Uruchom poniższe zapytanie:

```
SELECT ProductID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
OPTION (RECOMPILE, QUERYTRACEON 8608)
```

Dla tego zapytania zostanie stworzona następująca początkowa struktura Memo:

```
--- Początkowa struktura Memo ---
Root Group 18: Card=266 (Max=133449, Min=0)
  0 LogOp_GbAgg 13 17
Group 17:
  0 AncOp_PrjList 16
Group 16:
```

```

    0 AncOp_PrjEl 15
Group 15:
    0 ScaOp_AggFunc 14
Group 14:
    0 ScaOp_Const
Group 13: Card=121317 (Max=133449, Min=0)
    0 LogOp_Get
Group 12:
    0 AncOp_PrjEl 11
Group 11:
    0 ScaOp_Intrinsic 9 10
Group 10:
    0 ScaOp_Const
Group 9:
    0 ScaOp_Arithmetic 6 8
Group 8:
    0 ScaOp_Convert 7
Group 7:
    0 ScaOp_Identifier
Group 6:
    0 ScaOp_Arithmetic 1 5
Group 5:
    0 ScaOp_Arithmetic 2 4
Group 4:
    0 ScaOp_Convert 3
Group 3:
    0 ScaOp_Identifier
Group 2:
    0 ScaOp_Const
Group 1:
    0 ScaOp_Convert 0
Group 0:
    0 ScaOp_Identifier

```

Teraz uruchom poniższe zapytanie, aby wyświetlić końcową strukturę Memo:

```

SELECT ProductID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
OPTION (RECOMPILE, QUERYTRACEON 8615)

```

Wynik będzie następujący:

```

--- Końcowa struktura Memo ---
Group 32: Card=266 (Max=133449, Min=0)
    0 LogOp_Project 30 31
Group 31:
    0 AncOp_PrjList 21
Group 30: Card=266 (Max=133449, Min=0)
    0 LogOp_GbAgg 25 29
Group 29:
    0 AncOp_PrjList 28
Group 28:
    0 AncOp_PrjEl 27
Group 27:
    0 ScaOp_AggFunc 26
Group 26:
    0 ScaOp_Identifier

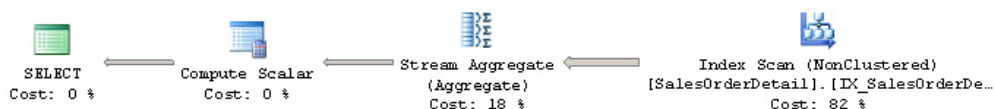
```

126 Microsoft SQL Server 2014. Optymalizacja zapytań

```
Group 25: Card=532 (Max=133449, Min=0)
  0 LogOp_GbAgg 13 24
Group 24:
  0 AncOp_PrjList 23
Group 23:
  0 AncOp_PrjEl 22
Group 22:
  0 ScaOp_AggFunc 14
Group 21:
  0 AncOp_PrjEl 20
Group 20:
  0 ScaOp_Convert 19
Group 19:
  0 ScaOp_Identifier
Root Group 18: Card=266 (Max=133449, Min=0)
  4 PhyOp_StreamGbAgg 13.2 17.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)=0.411876
  1 LogOp_RestrRemap 32
  0 LogOp_GbAgg 13 17
Group 17:
  0 AncOp_PrjList 16.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 0
Group 16:
  0 AncOp_PrjEl 15.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 0
Group 15:
  0 ScaOp_AggFunc 14.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 2
Group 14:
  0 ScaOp_Const Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 1
Group 13: Card=121317 (Max=133449, Min=0)
  2 PhyOp_Range 3 ASC Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 0.338953
  0 LogOp_Get
Group 12:
  0 AncOp_PrjEl 11
Group 11:
  0 ScaOp_Intrinsic 9 10
Group 10:
  0 ScaOp_Const
Group 9:
  0 ScaOp_Arithmetic 6 8
Group 8:
  0 ScaOp_Convert 7
Group 7:
  0 ScaOp_Identifier
Group 6:
  0 ScaOp_Arithmetic 1 5
Group 5:
  0 ScaOp_Arithmetic 2 4
Group 4:
  0 ScaOp_Convert 3
Group 3:
  0 ScaOp_Identifier
Group 2:
  0 ScaOp_Const
Group 1:
  0 ScaOp_Convert 0
Group 0:
  0 ScaOp_Identifier
```

Początkowa struktura Memo miała 18 grup, a końcowa już 32, co oznacza, że podczas procesu optymalizacji zostało dodane 14 grup. Jak wcześniej wyjaśniałem, podczas procesu optymalizacji wykonywane są reguły transformacji, które tworzą nowe alternatywy. Jeżeli alternatywy są równoznaczne z istniejącym operatorem, są wstawiane do tej samej grupy co on. Jeżeli nie, tworzona jest dodatkowa grupa. Możesz również zobaczyć, że grupa 18 jest grupą korzenia.

Pod koniec procesu, po zastosowaniu reguł implementacji, do dostępnych grup Memo zostaną dodane równoznaczne operatory fizyczne. Po oszacowaniu kosztu każdego z operatorów fizycznych optymalizator wyszuka najtańszego sposobu złożenia planu z dostępnych alternatyw. W tym przykładzie wybrany plan został pokazany na rysunku 3.12.



Rysunek 3.12. Plan pokazujący wybrane operatory

Jak widzieliśmy wcześniej, agregacja z klauzulą `GROUP BY` wymaga reguły transformacji `GbAggToStrm` (*Group by Aggregate to Stream*) lub `GbAggToHS` (*Group by Aggregate to Hash*). Stosując pokazane techniki, możesz zaobserwować, że w procesie optymalizacji wykorzystane zostały obie te reguły, ale tylko reguła `GbAggToStrm` wyprodukowała alternatywę w grupie 18, dodając operator `PhyOp_StreamGbAgg` jako ekwiwalent pierwotnego operatora logicznego `LogOp_GbAgg`. Operator `PhyOp_StreamGbAgg` to operator `Stream Aggregate` (agregacja strumieniowa), który znalazł się w ostatecznie wybranym planie pokazanym na rysunku 3.12. Kardynalność grupy 18 została oszacowana na 266, co zostało pokazane jako `Card=266`, a jej całkowity koszt na 0,411876, co możesz również potwierdzić w planie zapytania. W planie pokazany jest też operator `Index Scan`, który związany jest z operatorami `PhyOp_Range` i `LogOp_Get` z grupy 13.

Jako dodatkowy test możesz zobaczyć zawartość struktury Memo poprzez wymuszenie wykorzystania operatora `Hash Aggregate`, uruchamiając poniższe zapytanie z podpowiedzią:

```

SELECT ProductID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
OPTION (RECOMPILE, HASH GROUP, QUERYTRACEON 8615)
  
```

Statystyki

Do oszacowania kosztu planu zapytania optymalizator zapytań musi wiedzieć, najdokładniej jak to możliwe, jaka jest liczba rekordów zwracana dla danego zapytania, aby więc pomóc w szacowaniu kardynalności, SQL Server wykorzystuje i przechowuje

statystyki optymalizatora. Statystyki zawierają informacje statystyczne opisujące rozkład wartości w jednej lub więcej kolumn tabeli. Statystyki omówię dokładniej w rozdziale 6.

Możesz wykorzystać nieudokumentowane flagi 9292 i 9204 do wyświetlenia informacji o statystykach załadowanych podczas procesu optymalizacji. Uruchom zapytanie:

```
SELECT ProductID, name FROM Production.Product
WHERE ProductID = 877
OPTION (RECOMPILE, QUERYTRACEON 9292, QUERYTRACEON 9204)
```

Wyświetlony zostanie wynik podobny do poniższego:

```
Stats header loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 1, ColumnName: ProductID, EmptyTable: FALSE
```

```
Stats loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 1, ColumnName: ProductID, EmptyTable: FALSE
```

Aby lepiej zrozumieć, jak działa ten mechanizm, stwórzmy dodatkowe obiekty statystyk:

```
CREATE STATISTICS stat1 ON Production.Product(ProductID)
CREATE STATISTICS stat2 ON Production.Product(ProductID)
CREATE STATISTICS stat3 ON Production.Product(ProductID)
CREATE STATISTICS stat4 ON Production.Product(ProductID)
```

Flagi 9292 można użyć do wyświetlenia obiektów statystyk uważanych za interesujące. Uruchom poniższe zapytanie:

```
SELECT ProductID, name FROM Production.Product
WHERE ProductID = 877
OPTION (RECOMPILE, QUERYTRACEON 9292)
```

Oto wynik:

```
Stats header loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 1, ColumnName: ProductID, EmptyTable: FALSE
```

```
Stats header loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 10, ColumnName: ProductID, EmptyTable: FALSE
```

```
Stats header loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 11, ColumnName: ProductID, EmptyTable: FALSE
```

```
Stats header loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 12, ColumnName: ProductID, EmptyTable: FALSE
```

```
Stats header loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 13, ColumnName: ProductID, EmptyTable: FALSE
```

Flagi 9204 można użyć do wyświetlenia obiektów statystyk, które zostały wykorzystane podczas szacowania kardynalności. Wypróbuj poniższy kod:


```
SELECT ProductID, name FROM Production.Product
WHERE ProductID = 877
OPTION (RECOMPILE, QUERYTRACEON 9204)
```

Oto wynik:

```
Stats loaded: DbName: AdventureWorks2012, ObjName: Production.Product,
IndexId: 1, ColumnName: ProductID, EmptyTable: FALSE
```

Aby posprzątać obiekty, które stworzyliśmy, wykonaj poniższe polecenia:

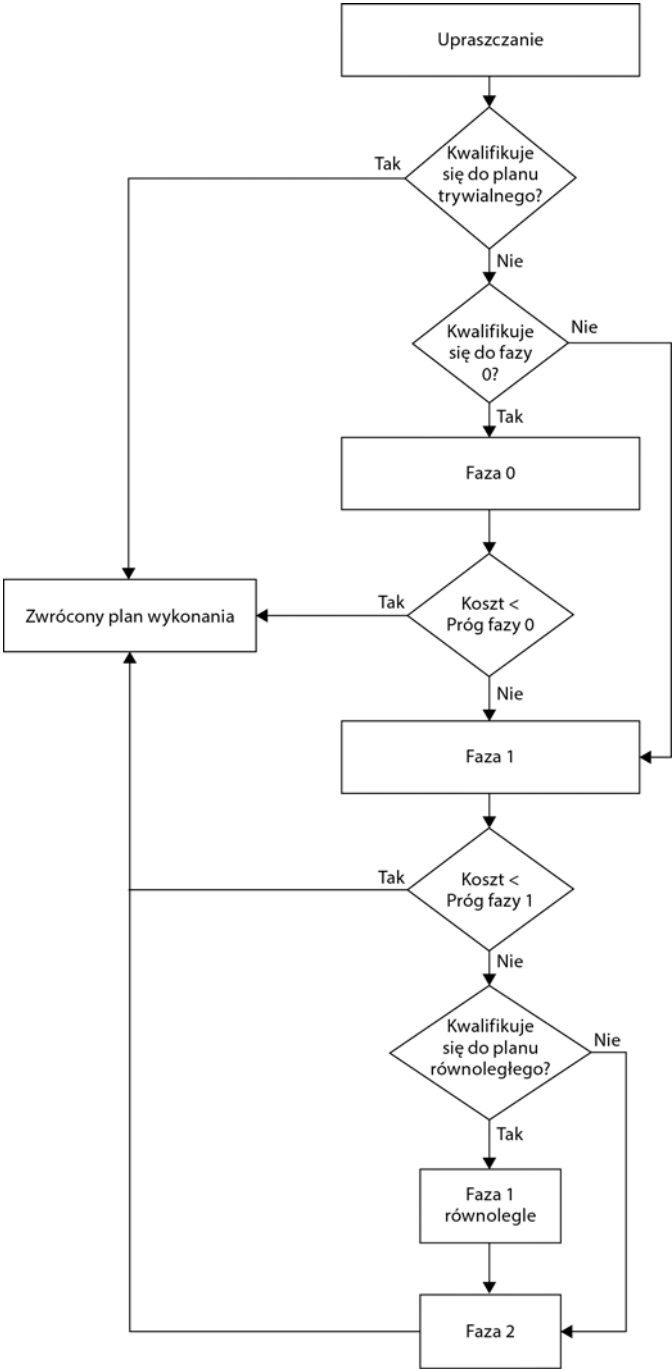
```
DROP STATISTICS Production.Product.stat1
DROP STATISTICS Production.Product.stat2
DROP STATISTICS Production.Product.stat3
DROP STATISTICS Production.Product.stat4
```

Pełna optymalizacja

Zgodnie z rysunkiem 3.13, jeżeli zapytanie nie kwalifikuje się do wykorzystania planu trywialnego, SQL Server uruchomi proces optymalizacji opartej na szacowanym koszcie, która stosuje reguły transformacji do generowania planów alternatywnych, przechowuje te alternatywy w strukturze Memo i wykorzystuje szacowanie kosztów do wybrania najlepszego planu. Ta optymalizacja może być wykonywana w trzech etapach, a podczas każdej fazy stosowane będą inne reguły transformacji.

Ponieważ niektóre zapytania mogą mieć ogromną liczbę możliwych planów, czasami niemożliwe jest przeszukanie całego obszaru poszukiwań — optymalizacja zapytania trwałaby zbyt długo. Dlatego oprócz reguł transformacji do kontrolowania strategii poszukiwania i ograniczenia liczby generowanych alternatyw wykorzystywana jest również heurystyka, dzięki czemu dobry plan może zostać znaleziony szybko. Optymalizator musi znaleźć równowagę między czasem optymalizacji i jakością wybranego planu. Na przykład, jak wyjaśnię w rozdziale 10., optymalizowanie kolejności złączeń może stworzyć ogromną liczbę możliwych alternatyw. Dlatego SQL Server za pomocą heurystyki tworzy początkowy zestaw kolejności złączeń na podstawie ich selektywności (proces ten pokażę w dalszej części).

Dodatkowo, zgodnie z informacjami z rozdziału 1., proces optymalizacji może zakończyć się natychmiast, kiedy tylko na końcu którejkolwiek z faz znaleziony zostanie wystarczająco dobry plan (relatywnie do wewnętrznych progów optymalizatora zapytań). Jeżeli jednak na końcu którejś z faz plan jest nadal bardzo kosztowny, optymalizator uruchomi kolejną fazę, która wykorzysta dodatkowy zestaw (zazwyczaj bardziej skomplikowanych) reguł transformacji. Fazy te, zgodnie z widokiem DMV `sys.dm_exec_query_optimizer_info`, nazywają się *search 0*, *search 1* i *search 2* (zobacz rysunek 3.13).



Rysunek 3.13. Proces optymalizacji

Search 0

Podobnie jak w przypadku planu trywialnego, pierwsza faza, *search 0*, będzie próbowała znaleźć plan najszybciej jak to możliwe bez wykorzystania wyszukanych transformacji. Faza *search 0*, nazywana fazą przetwarzania transakcji, jest idealna dla małych zapytań występujących zazwyczaj w systemach przetwarzających transakcje i wykorzystywana jest dla zapytań z przynajmniej trzema tabelami. Zanim zacznie się proces pełnej optymalizacji, optymalizator generuje początkowy zestaw kolejności złączeń z zastosowaniem heurystyki. Heurystyka rozpoczynana jest od połączenia najmniejszych tabel lub tabel, które osiągną największe filtrowanie na podstawie ich selektywności. Jest to jedyna kolejność złączeń rozpatrywana w tej fazie. Na końcu fazy optymalizator porównuje koszt najlepszego z planów do wewnętrznego progu i jeżeli plan zostanie uznany za zbyt kosztowny, uruchomiona zostanie następna faza.

Search 1

Następna faza, *search 1* (zwana także szybkim planem), wykorzystuje dodatkowe reguły transformacji, ograniczone zmiany kolejności złączeń i jest przystosowana do bardziej złożonych zapytań. Na końcu tej fazy SQL Server porównuje koszt najtańszego planu do drugiego wewnętrznego progu. Jeżeli plan będzie wystarczająco tani, to zostanie wybrany. Jeżeli zapytanie nadal jest kosztowne, a system jest w stanie uruchamiać zapytania równolegle, faza ta jest uruchamiana ponownie w celu znalezienia dobrego planu równoległego, jednak w tym miejscu nie jest wybierany żaden plan. Na końcu fazy są porównywane koszty najlepszego seryjnego i równoległego planu, a najtańszy z nich wykorzystywany jest w kolejnej fazie, *search 2*, którą omówimy już za chwilę.

Na przykład poniższe zapytanie nie kwalifikuje się do fazy *search 0* i przejdzie bezpośrednio do fazy *search 1*:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 870
```

Korzystając z widoku DMV `sys.dm_exec_query_optimizer_info`, zgodnie z wcześniejszymi wyjaśnieniami z tego rozdziału, możesz wyświetlić informacje dotyczące optymalizacji — przekonasz się, że wykonana została tylko faza *search 1*:

counter	occurrence	value
elapsed time	1	0.102
final cost	1	1.134781816
maximum DOP	1	0
optimizations	1	1

counter	occurrence	value
search 1	1	1
search 1 tasks	1	241
search 1 time	1	0.061
tables	1	1
tasks	1	241

Widok DMV `sys.dm_exec_query_optimizer_info` zawiera licznik o nazwie *gain stage 0 to stage 1* (zysk z etapu 0 do etapu 1), który pokazuje, ile razy *search 1* zostało wywołane po *search 0*, i zawiera średni spadek kosztu przy przejściu z jednego etapu do drugiego, zgodnie z poniższą formułą:

$$(\text{MinimalnyKosztPlanu (search 0)} - \text{MinimalnyKosztPlanu (search 1)}) / \text{MinimalnyKosztPlanu (search 0)}$$

Na przykład zapytanie:

```
SELECT soh.SalesOrderID, sod.SalesOrderDetailID, SalesReasonID
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
  ON soh.SalesOrderID = soh.SalesOrderID
JOIN Sales.SalesOrderHeaderSalesReason sohsr
  ON sohsr.SalesOrderID = soh.SalesOrderID
WHERE soh.SalesOrderID = 43697
```

spowoduje wyświetlenie poniższych informacji o optymalizacji:

counter	occurrence	value
elapsed time	1	0.002
final cost	1	1.413612922
gain stage 0 to stage 1	1	0.05339122
hints	1	1
maximum DOP	1	0
optimizations	1	1
search 0 tasks	1	158
search 0 time	1	0
search 1	1	1
search 1 tasks	1	101
search 1 time	1	0
tables	1	3
tasks	1	259

Wynik pokazuje, że proces optymalizacji przeszedł przez fazy *search 0* i *search 1*, a plan został znaleziony podczas tej drugiej. Pokazuje również, że zysk kosztów przy przejściu z etapu *search 0* do *search 1* wynosił 5%.

Search 2

Ostatnia faza, *search 2*, nazywana jest pełną optymalizacją i stosowana jest do skomplikowanych i bardzo skomplikowanych zapytań. W tej fazie wykorzystywane jest znacznie więcej reguł transformacji, operatory równoległe i inne zaawansowane strategie optymalizacji. Ponieważ jest to ostatnia faza, plan wykonywania musi zostać znaleziony (z wyjątkiem sytuacji przekroczenia czasu, co wyjaśnię w dalszej części).

Widok `sys.dm_exec_query_optimizer_info` zawiera jeszcze jeden użyteczny licznik, o nazwie *gain stage 1 to stage 2* (zysk z etapu 1 do etapu 2), pokazujący, ile razy etap *search 2* był uruchomiony po etapie *search 1* oraz jaki był średni spadek kosztu przy przejściu z jednego etapu do drugiego, zgodnie z poniższą formułą:

$$\frac{(\text{MinimalnyKosztPlanu}(\text{search } 1) - \text{MinimalnyKosztPlanu}(\text{search } 2))}{\text{MinimalnyKosztPlanu}(\text{search } 1)}$$

Moglibyśmy również zastosować nieudokumentowane flagi 8675 i 2372 do uzyskania dodatkowych informacji o tych fazach optymalizacji. Na przykład uruchom poniższe zapytanie:

```
SELECT DISTINCT pp.LastName, pp.FirstName
FROM Person.Person pp JOIN HumanResources.Employee e
  ON e.BusinessEntityID = pp.BusinessEntityID
JOIN Sales.SalesOrderHeader soh
  ON pp.BusinessEntityID = soh.SalesPersonID
JOIN Sales.SalesOrderDetail sod
  ON soh.SalesOrderID = soh.SalesOrderID
JOIN Production.Product p
  ON sod.ProductID = p.ProductID
WHERE ProductNumber = 'BK-M18B-44'
OPTION (RECOMPILE, QUERYTRACEON 8675)
```

Wyświetlone zostaną następujące informacje o optymalizacji:

```
End of simplification, time: 0.003 net: 0.003 total: 0.003 net: 0.003
end exploration, tasks: 149 no total cost time: 0.005 net: 0.005 total: 0.009
end search(0), cost: 13.884 tasks: 332 time: 0.002 net: 0.002 total: 0.011
end exploration, tasks: 926 Cost = 13.884 time: 0.01 net: 0.01 total: 0.021
end search(1), cost: 3.46578 tasks: 1906 time: 0.009 net: 0.009 total: 0.031
end exploration, tasks: 3301 Cost = 3.46578 time: 0.008 net: 0.008 total: 0.04
*** Optimizer time out abort at task 4248 ***
end search(2), cost: 0.832242 tasks: 4248 time: 0.013 net: 0.013 total: 0.053
*** Optimizer time out abort at task 4248 ***
End of post optimization rewrite, time: 0 net: 0 total: 0.053 net: 0.053
End of query plan compilation, time: 0 net: 0 total: 0.054 net: 0.054
```

Informacje te pokazują, że zapytanie przeszło przez wszystkie trzy etapy optymalizacji, a w etapie *search 2* wystąpiło przekroczenie dozwolonego czasu. O zjawisku przekroczenia czasu wspomniałem w rozdziale 1., a jego dokładniejszy opis przedstawię poniżej.

Uruchomienie tego samego zapytania z flagą 2372 pokaże poniższy wynik, w którym fazy *search 0*, *1* i *2* nazywają się odpowiednio *TP*, *QuickPlan* i *Full*:

```
Memory before NNFCConvert: 25
Memory after NNFCConvert: 25
Memory before project removal: 27
Memory after project removal: 29
Memory before simplification: 29
Memory after simplification: 58
Memory before heuristic join reordering: 58
Memory after heuristic join reordering: 65
Memory before project normalization: 65
Memory after project normalization: 65
Memory before stage TP: 68
Memory after stage TP: 84
Memory before stage QuickPlan: 84
Memory after stage QuickPlan: 126
Memory before stage Full: 126
Memory after stage Full: 172
Memory before copy out: 172
Memory after copy out: 173
```

Jak już zaznaczałem, optymalizator musi znaleźć najlepszy możliwy plan w jak najkrótszym czasie. Co więcej, musi w rezultacie wykonać ten plan, nawet jeżeli nie będzie on tak efektywny, jak by chciał. Proces optymalizacji zawiera również pojęcie budżetu kosztu optymalizacji. Jeżeli budżet ten zostanie przekroczony, poszukiwania planu są kończone, a optymalizator zgłosi przekroczenie dozwolonego czasu optymalizacji. Czas ten nie jest z góry ustalony, ale jest obliczany na podstawie liczby wykorzystanych transformacji i sumarycznego czasu pracy optymalizatora.

Jeżeli optymalizacja zakończy się przekroczeniem czasu, optymalizator zwraca najmniej kosztowny plan znaleziony do tej pory. Najlepszy plan może być planem znalezionym podczas aktualnego etapu, ale najprawdopodobniej będzie to plan znaleziony podczas poprzedniego etapu. Oznacza to, że przekroczenie czasu najprawdopodobniej będzie miało miejsce w etapie *search 1* lub *search 2*. Zdarzenie to jest pokazywane we właściwościach planu graficznego w polu *Reason For Early Termination Of Statement Optimization* lub we właściwości *StatementOptmEarlyAbortReason* planu XML, a także w widoku DMV `sys.dm_exec_query_optimizer_info` w liczniku `timeout`. Przykład, uzyskany po uruchomieniu poprzedniego zapytania z tego podrozdziału, przedstawiam na rysunku 3.14.

Na końcu procesu optymalizacji wybrany plan zostanie przesłany do silnika wykonującego w celu wykonania, a wyniki zostaną przesłane do klienta.

W tym rozdziale pokazałem również, że zrozumienie sposobu działania optymalizatora zapytań może dać Ci świetne podstawy do szukania problemów, optymalizacji i poprawiania zapytań. Musisz jednak dowiedzieć się więcej na temat operatorów najczęściej wykorzystywanych w planach tworzonych przez optymalizator zapytań. Ten temat omówię w następnym rozdziale.

Rozdział 4

Operatory zapytań

W tym rozdziale:

- ▶ Operatory dostępu do danych
- ▶ Agregacje
- ▶ Złączenia
- ▶ Działania równoległe
- ▶ Aktualizacje
- ▶ Podsumowanie



Silnik wykonywania jest w rzeczywistości kolekcją fizycznych operatorów, które są komponentami programowymi wykonującymi funkcje procesora zapytań. Ich celem jest wykonanie zapytania w sposób efektywny. Jeżeli spojrzeć na nie z innej perspektywy, te operacje implementowane przez silnik wykonywania definiują wybory dostępne dla optymalizatora zapytań podczas budowania planu zapytania. Silnik wykonywania i jego operatory omówiłem pokrótce w poprzednich rozdziałach, a teraz omówię niektóre z najczęściej wykorzystywanych operatorów, ich algorytmy i koszt. W tym rozdziale skupię się na operatorach związanych z dostępem do danych, złączeniami, agregacjami, działaniami równoległymi i aktualizacjami, ponieważ są one powszechnie stosowane w zapytaniach, a także są najczęściej wykorzystywane w tej książce. Oczywiście w silniku wykonywania zaimplementowane jest znacznie więcej operatorów, a kompletną ich listę wraz z opisami można znaleźć w dokumentacji SQL Server 2014 Books Online. Ten rozdział pokazuje, w jaki sposób optymalizator podejmuje decyzję w kwestii wyboru spośród różnych operatorów udostępnianych przez silnik wykonywania. Na przykład pokażę, jak procesor zapytań decyduje, czy skorzystać ze złączenia *Nested Loops Join* czy *Hash Join* albo czy skorzystać z agregacji *Stream Aggregate* czy *Hash Aggregate*.

Rozdział rozpoczyna się od przeanalizowania operacji związanych z dostępem do danych, włączając w to operacje wykonujące skanowanie, przeszukiwanie i wyszukiwanie zaznaczeń na strukturach bazy, takich jak sterty, indeksy klastrowe i indeksy nieklastrowe. Wyjaśnię również koncepcje związane z sortowaniem i haszowaniem oraz ich wpływ na niektóre algorytmy fizycznych złączeń i agregacji, które omówię później. Podrozdział o złączeniach prezentuje operatory fizyczne *Nested Loops Join*, *Merge Join* i *Hash Join*. Kolejny podrozdział skupia się na agregacjach i szczegółowo opisuje operatory *Stream Aggregate* i *Hash Aggregate*. Następnie omawiam działania równoległe i wyjaśniam, jak mogą skrócić czas wykonywania zapytania. Rozdział zakończony jest wyjaśnieniem, w jaki sposób procesor zapytań obsługuje operacje aktualizacji.

Operatory dostępu do danych

W tym podrozdziale pokazuję operatory, które uzyskują bezpośredni dostęp do bazy danych, wykorzystując tabele lub indeks — przykładami takiego dostępu są skanowanie lub przeszukiwanie. Skanowanie czyta całą strukturę, która może być stertą, indeksem klastrowym lub indeksem nieklastrowym. Przeszukiwanie natomiast nie wymaga skanowania całej struktury, lecz efektywnie pobiera rekordy, nawigując po indeksie. Dlatego przeszukiwanie może być wykonywane tylko na indeksie klastrowym lub indeksie nieklastrowym. Aby wyjaśnić różnicę między tymi strukturami, dodam, że sterta zawiera wszystkie kolumny tabeli, a dane są przechowywane w postaci nieposortowanej. W przypadku indeksu klastrowego dane przechowywane są logicznie

posortowane na podstawie klucza klastrowego, a oprócz samego klucza indeks zawiera również pozostałe kolumny tabeli. Indeks nieklastrowy natomiast może być zdefiniowany na indeksie klastrowym lub stercie i zazwyczaj zawiera tylko podzbiór kolumn tabeli. Operacje na tych strukturach podsumowuję w tabeli 4.1.

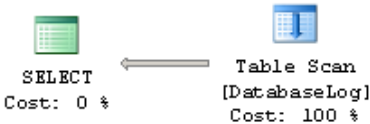
Tabela 4.1. Operatory dostępu do danych

Struktura	Skanowanie	Przeszukiwanie
Sterta	Table Scan (skan tabeli)	
Indeks klastrowy	Clustered Index Scan (skan indeksu klastrowego)	Clustered Index Seek (przeszukiwanie indeksu klastrowego)
Indeks nieklastrowy	Index Scan (skan indeksu)	Index Seek (przeszukiwanie indeksu)

Skanowanie

Zacznijmy od najprostszego przykładu, a mianowicie skanowania sterty, które wykonywane jest, zgodnie z tabelą 4.1, przez operator *Table Scan*. Poniższe zapytanie na bazie AdventureWorks2012 wykorzysta operator *Table Scan* zgodnie z rysunkiem 4.1.

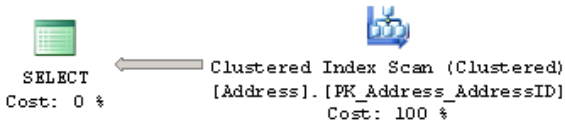
```
SELECT * FROM DatabaseLog
```



Rysunek 4.1. Operator Table Scan

Analogicznie: poniższe zapytanie pokaże operator *Clustered Index Scan*, zgodnie z rysunkiem 4.2. Operacje *Table Scan* i *Clustered Index Scan* są podobne, ponieważ obie skanują całą tabelę bazową, jednak pierwsza działa na stercie, a druga na indeksie klastrowym.

```
SELECT * FROM Person.Address
```



Rysunek 4.2. Operator Clustered Index Scan

Sortowanie to coś, o czym należy pamiętać w kontekście skanowania, ponieważ nawet jeżeli dane przechowywane w indeksie klastrowym są posortowane, wykorzystanie operatora *Clustered Index Scan* nie gwarantuje, że zwrócone dane będą posortowane.

Przez to, że rezultaty nie są automatycznie sortowane, silnik przechowujący może skupić się na znalezieniu najbardziej wydajnego sposobu zwrócenia danych bez konieczności dbania o sortowanie wyników. Przykłady efektywnych metod wykorzystywanych przez silnik przechowujący zawierają zastosowanie skanowania kolejności alokacji opartego na stronach Index Allocation Map (IAM, mapa alokacji indeksu) i skanowania kolejności indeksów na podstawie listy połączonych indeksów. Ponadto zaawansowany mechanizm skanowania o nazwie *merry-go-round scanning* (skanowanie karuzelowe, funkcjonalność dostępna jedynie w wersji Enterprise) pozwala na współdzielenie skanowania tabel przez wiele zapytań w taki sposób, że każde wykonanie może dołączyć do skanowania w dowolnej lokalizacji, a tym samym oszczędzać zasoby, które zostałyby wykorzystane, gdyby każde zapytanie osobno czytało dane.

Jeżeli chcesz wiedzieć, czy Twoje dane zostały posortowane, wartość właściwości *Ordered* (posortowane) pokaże Ci, czy dane z operatora *Clustered Index Scan* zostały zwrócone w formie posortowanej. Na przykład, jeżeli klucz klastrowy tabeli *Person*.
↪ *Address* to *AddressID* i jeżeli uruchomisz poniższe zapytanie i spojrzysz na informacje dotyczące operatora *Clustered Index Scan*, zobaczysz coś podobnego jak na rysunku 4.3.

```
SELECT * FROM Person.Address
ORDER BY AddressID
```

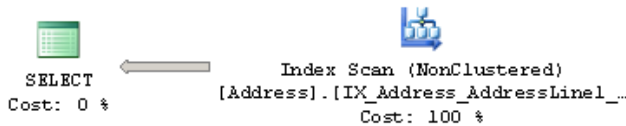
Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	19614
Actual Number of Batches	0
Estimated I/O Cost	0.257199
Estimated Operator Cost	0.278931 (100%)
Estimated Subtree Cost	0.278931
Estimated CPU Cost	0.0217324
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	19614
Estimated Row Size	4241 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
Object	
[AdventureWorks2012].[Person].[Address].	
[PK_Address_AddressID]	
Output List	
[AdventureWorks2012].[Person].[Address].AddressID,	
[AdventureWorks2012].[Person].[Address].AddressLine1,	
[AdventureWorks2012].[Person].[Address].AddressLine2,	
[AdventureWorks2012].[Person].[Address].City,	
[AdventureWorks2012].[Person].[Address].StateProvinceID,	
[AdventureWorks2012].[Person].[Address].PostalCode,	
[AdventureWorks2012].[Person].[Address].SpatialLocation,	
[AdventureWorks2012].[Person].[Address].rowguid,	
[AdventureWorks2012].[Person].[Address].ModifiedDate	

Rysunek 4.3. Właściwości operatora Clustered Index Scan

Zauważ, że właściwość *Ordered* zawiera wartość *true*. Jeżeli uruchomisz to samo zapytanie bez klauzuli *ORDER BY*, właściwość *Ordered* będzie zawierała wartość *false*. W niektórych przypadkach SQL Server może zyskać na odczytywaniu danych w kolejności zgodnej z indeksem klastrowym. Jeden z takich przykładów został pokazany w dalszej części tego rozdziału, na rysunku 4.15, gdzie operator *Stream Aggregate* może skorzystać, jeżeli operator *Clustered Index Scan* pobierze już posortowane dane.

Poniżej pokazuję przykład operatora *Index Scan*. Ten przykład do wykonania zapytania wykorzystuje indeks nieklastrowy, co oznacza, że całe zapytanie może być wykonane bez uzyskiwania dostępu do tabeli bazowej (należy pamiętać, że indeks nieklastrowy zawiera z reguły tylko kilka kolumn tabeli). Uruchomienie poniższego zapytania spowoduje wyświetlenie planu pokazanego na rysunku 4.4.

```
SELECT AddressID, City, StateProvinceID
FROM Person.Address
```



Rysunek 4.4. Operator Index Scan

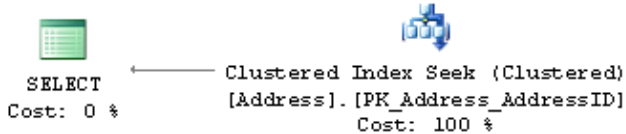
Zauważ, że optymalizator był w stanie rozwiązać zapytanie bez uzyskiwania dostępu do tabeli bazowej *Person.Address* i zdecydował przeskanować indeks *IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode*, który w porównaniu do indeksu klastrowego zawiera mniej stron. Definicja indeksu zawiera kolumny *AddressLine1*, *AddressLine2*, *City*, *StateProvinceID* i *PostalCode*, czyli indeks pokrywa wszystkie kolumny żądane w zapytaniu. Być może zastanawiasz się jednak, skąd indeks bierze kolumnę *AddressID*. Dla tabeli z indeksem klastrowym każdy indeks nieklastrowy będzie zawierał klucz klastrowy. Klucz ten jest wykorzystywany do determinowania, który rekord z indeksu klastrowego jest skojarzony z którym rekordem z indeksu nieklastrowego (podobne podejście dla indeksów nieklastrowych omówię w dalszej części tego podrozdziału). W tym przypadku, jak już wcześniej wspomniałem, *AddressID* to klucz klastrowy dla tabeli i jest przechowywany dla każdego wiersza indeksu nieklastrowego, dlatego właśnie indeks był w stanie pokryć również tę kolumnę.

Przeszukiwanie

Przyjrzyjmy się teraz przeszukiwaniu indeksów. Może ono być wykonywane zarówno przez operator *Clustered Index Seek*, jak i *Index Seek*, które wykorzystywane są, odpowiednio, dla indeksów klastrowych i nieklastrowych. *Index Seek* nie skanuje całego indeksu, lecz zamiast tego nawiguje po strukturze B-drzewa indeksu w celu szybkiego odszukania jednego lub więcej rekordów. Poniższe zapytanie, wraz z planem z rysunku 4.5, przedstawia przykład wykorzystania operatora *Clustered Index Seek*. W porównaniu do

Index Seek zaletą tego operatora jest to, że pokrywa wszystkie kolumny tabeli. Oczywiście, ponieważ rekordy indeksu klastrowego są uporządkowane logicznie na podstawie klucza klastrowego, tabela może mieć tylko jeden indeks klastrowy.

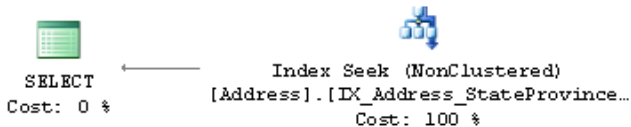
```
SELECT AddressID, City, StateProvinceID FROM Person.Address
WHERE AddressID = 12037
```



Rysunek 4.5. Operator Clustered Index Seek

Następne zapytanie i rysunek 4.6 ilustrują wykorzystanie operatora *Index Seek*. Warto zauważyć, że tabela bazowa nie została wykorzystana, a skanowanie całego indeksu nie było konieczne: na kolumnie *StateProvinceID* założony jest indeks nieklastrowy, który, jak już wcześniej wspomniałem, zawiera również klucz klastrowy *AddressID*.

```
SELECT AddressID, StateProvinceID FROM Person.Address
WHERE StateProvinceID = 32
```



Rysunek 4.6. Operator Index Seek

Chociaż oba przykłady zwracają tylko jeden wiersz, operacja *Index Seek* może także zostać wykorzystana do zwrócenia wielu rekordów dla operatorów równości i nierówności — taka operacja nazywa się częściowym skanowaniem uporządkowanym. Poprzednie zapytanie zwracało tylko jeden wiersz, możesz jednak zmienić parametr, zgodnie z poniższym przykładem:

```
SELECT AddressID, StateProvinceID FROM Person.Address
WHERE StateProvinceID = 9
```

Zapytanie to podlega autoparametryzacji i ten sam plan zostanie wykorzystany dla dowolnego innego parametru. Możesz wymusić zbudowanie nowego planu (np. za pomocą polecenia `DBCC FREEPROCCACHE`), jednak także wtedy zwrócony zostanie taki sam plan. W tym przypadku wykorzystany zostanie plan zgodny z planem z rysunku 4.6 i zwrócone zostaną bez konieczności sięgania do tabeli bazowej 4564 rekordy. Częściowe skanowanie uporządkowane korzysta z indeksu do znalezienia pierwszego rekordu kwalifikującego się do wyniku, a następnie czyta kolejne rekordy,

które są logicznie pogrupowane na stronach tego samego liścia indeksu. Więcej szczegółów na temat struktury indeksu poznasz w rozdziale 5.

Bardziej rozbudowany przykład częściowego skanowania uporządkowanego wiążę się z wykorzystaniem operatora nierówności lub, jak w poniższym zapytaniu, klauzuli BETWEEN:

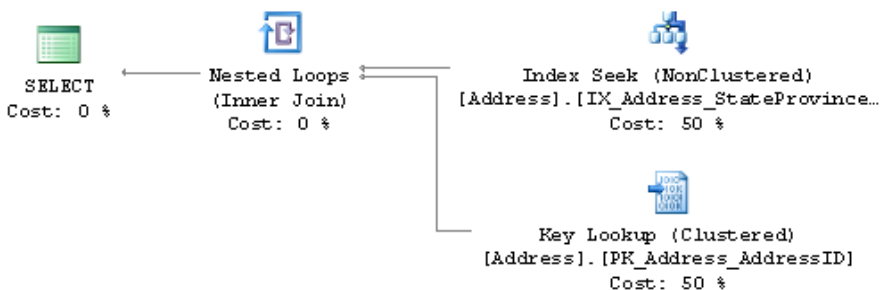
```
SELECT AddressID, City, StateProvinceID FROM Person.Address
WHERE AddressID BETWEEN 10000 and 20000
```

Ponieważ indeks klastrowy jest definiowany za pomocą kolumny *AddressID*, wykorzystany zostanie plan podobny do tego z rysunku 4.5. Operator *Clustered Index Seek* zostanie użyty do odnalezienia pierwszego rekordu pasującego do predykatu filtra, a następnie odczytywane będą kolejne rekordy wiersz po wierszu, dopóki nie zostanie znaleziony ostatni pasujący wiersz. Skanowanie zostanie zatrzymane w momencie natrafienia na pierwszy niekwalifikujący się wiersz.

Wyszukiwanie zaznaczeń

Pojawia się pytanie, co się stanie, jeżeli indeks nieklastrowy jest użyteczny do znalezienia jednego lub więcej rekordów, ale nie pokrywa wszystkich żądanych w zapytaniu kolumn. Innymi słowy: co się stanie, jeżeli indeks nieklastrowy nie zawiera wszystkich kolumn z zapytania? W takim wypadku optymalizator musi zdecydować, czy bardziej wydajne będzie wykorzystanie indeksu nieklastrowego do szybkiego znalezienia tych rekordów, a następnie sięgnięcie do tabeli bazowej w celu pobrania dodatkowych pól, czy bardziej optymalne będzie sięgnięcie do tabeli bazowej od razu i skanowanie jej poprzez czytanie każdego z wierszy i sprawdzanie, czy pasują do predykatów. Na przykład w poprzednim zapytaniu istniejący indeks nieklastrowy pokrywa kolumny *AddressID* i *StateProvinceID*. Co się stanie, jeżeli w tym samym zapytaniu zażądamy również kolumn *City* i *ModifiedDate*? Przykład ten obrazuje następane zapytanie, które zwraca jeden rekord i wykorzystuje plan przedstawiony na rysunku 4.7.

```
SELECT AddressID, City, StateProvinceID, ModifiedDate
FROM Person.Address
WHERE StateProvinceID = 32
```



Rysunek 4.7. Przykład wyszukiwania z zaznaczeniem

Jak w poprzednim przykładzie, optymalizator do szybkiego wyszukiwania rekordów wybiera indeks `IX_Address_StateProvinceID`. Ponieważ jednak indeks nie zawiera wszystkich kolumn, konieczne jest również skorzystanie z tabeli bazowej (w tym przypadku z indeksu klastrowego) do pobrania dodatkowych pól. Ta funkcjonalność nazywa się *wyszukiwaniem zaznaczeń* i jest wykonywana przez operator *Key Lookup* (wyszukanie klucza), który został wprowadzony po to, aby odróżnić wyszukiwanie z zaznaczeniem od zwykłej operacji *Clustered Index Seek*. Tak naprawdę operator *Key Lookup* pojawia się tylko na planie graficznym (i tylko w wersji SQL Server 2005 Service Pack 2 i nowszych). Plany tekstowe i XML pokazują, czy operator *Clustered Index Seek* wykonuje wyszukiwanie z zaznaczeniem za pośrednictwem słowa kluczowego `LOOKUP` i atrybutów `Lookup`, zgodnie z następnym przykładem. Wykonaj następujące zapytanie:

```
SET SHOWPLAN_TEXT ON
GO
SELECT AddressID, City, StateProvinceID, ModifiedDate
FROM Person.Address
WHERE StateProvinceID = 32
GO
SET SHOWPLAN_TEXT OFF
GO
```

Wynik pokaże następujący plan tekstowy zawierający operator *Clustered Index Seek* ze słowem kluczowym `LOOKUP` na końcu:

```
--Nested Loops (Inner Join, OUTER REFERENCES ...)
--Index Seek (OBJECT: ([Address].[IX_Address_StateProvinceID]),
    SEEK: ([Address].[StateProvinceID]=(32)) ORDERED FORWARD)
--Clustered Index Seek (OBJECT: ([Address].[PK_Address_AddressID]),
    SEEK: ([Address].[AddressID]=[Address].[AddressID]) LOOKUP ORDERED FORWARD)
```

Plan XML pokazuje te same informacje w następujący sposób:

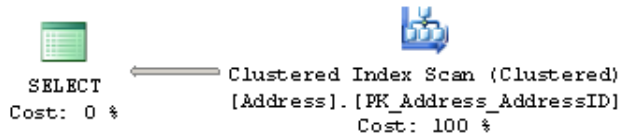
```
<RelOp LogicalOp="Clustered Index Seek" PhysicalOp="Clustered Index Seek" ...>
...
<IndexScan Lookup="true" Ordered="true" ScanDirection="FORWARD" ...>
```

Pamiętaj, że chociaż SQL Server 2000 implementował wyszukiwanie zaznaczeń za pomocą dedykowanego operatora (o nazwie *Bookmark Lookup*), operacja jest w gruncie rzeczy taka sama.

Uruchom teraz to samo zapytanie, ale tym razem zażądaj, aby pole `StateProvinceID` miało wartość 20. Stworzony zostanie plan pokazany na rysunku 4.8.

```
SELECT AddressID, City, StateProvinceID, ModifiedDate
FROM Person.Address
WHERE StateProvinceID = 20
```

Podczas tego wykonania optymalizator wybrał operator *Clustered Index Scan*, a zapytanie zwróciło 308 rekordów, w przeciwieństwie do jednego rekordu zwróconego dla predykatu `StateProvinceID = 32`. Optymalizator tworzy dwa różne plany dla tego samego zapytania, a jedyną różnicą jest wartość parametru `StateProvinceID`. W tym przypadku



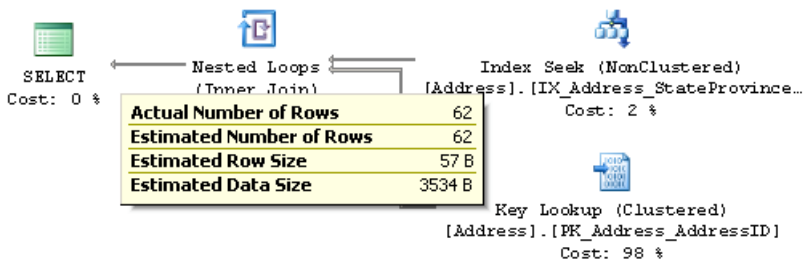
Rysunek 4.8. Zmieniony plan korzystający z Clustered Index Scan

optymalizator wykorzystuje wartość parametru *StateProvinceID* do oszacowania kardynalności predykatu podczas próby wygenerowania wydajnego planu dla zapytania. Dokładniej pokażę ten proces w rozdziale 6.

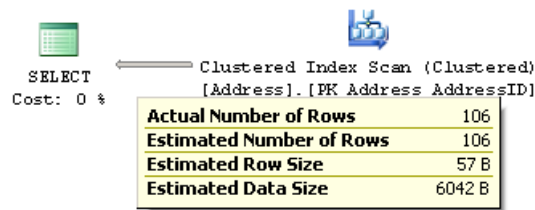
Tym razem optymalizator oszacował, że może zostać zwróconych więcej rekordów, i zdecydował, że taniej będzie wykonać skanowanie tabeli niż wykonywać wiele wyszukiwań zaznaczeń. Być może zastanawiasz się teraz, na jakiej podstawie optymalizator podejmuje decyzję o zmianie jednej metody na drugą. Ponieważ wyszukiwanie zaznaczeń wymaga wykonywania operacji I/O, które są bardzo kosztowne, nie potrzeba wielu rekordów, aby optymalizator wybrał skanowanie indeksu (lub tabeli). Wiemy już, że kiedy zapytanie zwracało jeden rekord, optymalizator wybrał wyszukiwanie zaznaczeń. Wiemy też, że gdy wybraliśmy predykat powodujący zwrócenie 308 rekordów, wybrany został *Clustered Index Scan*. Aby znaleźć punkt, w którym następuje zmiana operatora, musimy więc zażądać liczbę rekordów pomiędzy 1 a 308, prawda?

Być może już podejrzewasz, że jest to decyzja oparta na kosztach operacji, a nie na liczbie rekordów zwracanych przez zapytanie; wpływ na decyzję ma natomiast szacowana liczba rekordów. Możemy sprawdzić, jakie są te szacunki, analizując histogram obiektu statystyk dla indeksu *IX_Address_StateProvinceID* (omówimy to w rozdziale 6.).

Wykonałem to ćwiczenie i okazało się, że najwyższa szacowana liczba rekordów pozwalająca na wykonanie wyszukiwania zaznaczeń dla tego przykładu to 62, a operator *Clustered Index Scan* zaczął być wykorzystywany od 106 rekordów. Przyjrzyjmy się obu przykładom, uruchamiając zapytanie z wartościami *StateProvinceID* równymi 163 i 71. Otrzymamy plany pokazane na rysunkach 4.9 i 4.10.



Rysunek 4.9. Plan dla predykatu *StateProvinceID* = 163



Rysunek 4.10. Plan dla predykatu StateProvinceID = 71

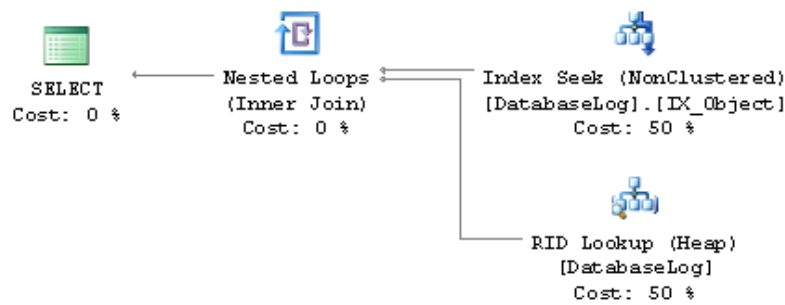
Patrząc na te plany, możemy zauważyć, że dla tego konkretnego przykładu optymalizator wybiera wyszukiwanie zaznaczeń dla 62 szacowanych rekordów i zmienia wybór na *Clustered Index Scan*, kiedy szacowana liczba rekordów wzrasta do 106 (dla tego konkretnego obiektu statystyk histogram nie pokazuje szacowanych wartości pomiędzy 62 a 106). Pamiętaj, że chociaż w tym przypadku szacowana i rzeczywista liczba rekordów są takie same, optymalizator podejmuje decyzję na podstawie liczby szacowanej. Rzeczywista liczba rekordów jest znana dopiero po wygenerowaniu planu, wykonaniu go i zwróceniu rekordów.

Ponieważ indeksy nieklastrowe mogą istnieć zarówno dla stert, jak i dla indeksów klastrowych, możemy również wykonać wyszukiwanie zaznaczeń na stercie. Aby wypróbować kolejny przykład, stwórz indeks na będącej stertą tabeli DatabaseLog, uruchamiając poniższe zapytanie:

```
CREATE INDEX IX_Object ON DatabaseLog(Object)
```

Następnie uruchom poniższe zapytanie, a otrzymasz wynik zgodny z rysunkiem 4.11:

```
SELECT * FROM DatabaseLog
WHERE Object = 'City'
```



Rysunek 4.11. Wyszukiwanie RID

Zauważ, że zamiast poprzednio pokazanego operatora *Key Lookup* ten plan zawiera operator *RID Lookup*. To dlatego, że stopy nie mają klucza klastrowego jak w przypadku indeksów klastrowych, a zamiast niego mają identyfikatory RID. RID to identyfikator wiersza zawierający takie informacje jak plik bazy danych, strona

i numery slotów, które pozwalają łatwo zlokalizować dany wiersz. Każdy wiersz w indeksie nieklastrowym stworzonym na sterce zawiera identyfikator RID wiersza.

Aby posprzątać, usuń po prostu stworzony indeks:

```
DROP INDEX DatabaseLog.IX_Object
```

Agregacje

Agregacje są wykorzystywane w bazach danych do podsumowania informacji o pewnym zbiorze danych. Rezultatem może być pojedyncza wartość, np. średnia płaca w firmie, lub może to być wartość dla każdej grupy, np. średnia płaca per dział. SQL Server ma dwa operatory implementujące agregacje: *Stream Aggregate* (agregacja strumieniowa) i *Hash Aggregate* (agregacja haszowa). Operatory te mogą być stosowane w zapytaniach wykorzystujących funkcje agregujące (takie jak SUM, AVG i MAX), klauzulę GROUP BY lub słowo kluczowe DISTINCT.

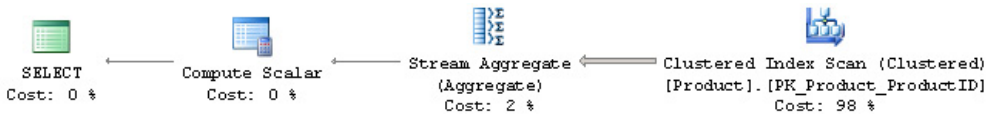
Sortowanie i haszowanie

Zanim przedstawię pozostałe operatory w tym rozdziale, chciałbym dodać kilka słów na temat sortowania i haszowania, ponieważ obie te czynności odgrywają istotną rolę w niektórych operatorach i algorytmach silnika wykonywania. Na przykład dwa z operatorów omawianych w tym rozdziale, *Stream Aggregate* i *Merge Join*, wymagają, aby dane były już posortowane. Aby uzyskać posortowane dane, optymalizator może wykorzystać istniejący indeks lub może niejawnie uruchomić operator *Sort* (sortowanie). Haszowanie jest natomiast wykorzystywane przez operatory *Hash Aggregate* i *Hash Join*, które budują w pamięci tablicę haszy. Operator *Hash Join* wykorzystuje pamięć tylko do przechowywania mniejszego ze swoich dwóch wsadów, który jest określany przez optymalizator.

Sortowanie również wykorzystuje pamięć i, podobnie jak haszowanie, bazę tempdb, jeżeli nie będzie wystarczająco dużo dostępnej pamięci, co może powodować problemy wydajnościowe. Zarówno sortowanie, jak i haszowanie (tylko w trakcie, kiedy wsad jest haszowany; dokładniej wyjaśnię to później) są operacjami blokującymi lub typu *stop-and-go* (zatrzymaj i uruchom); oznacza to, że nie mogą zwrócić żadnych wierszy, dopóki nie przetworzą całego wsadu.

Stream Aggregate

Zacznijmy od operatora *Stream Aggregate* (agregacja strumieniowa) w zapytaniu z funkcją agregującą. Zapytania wykorzystujące funkcję agregującą bez klauzuli GROUP BY nazywane są *agregacjami skalarnymi*, ponieważ zwracają jedną wartość i zawsze implementowane są za pomocą operatora *Stream Aggregate*. Uruchom poniższe zapytanie, które wykorzysta plan pokazany na rysunku 4.12.

**Rysunek 4.12.** Operator Stream Aggregate

```
SELECT AVG(ListPrice) FROM Production.Product
```

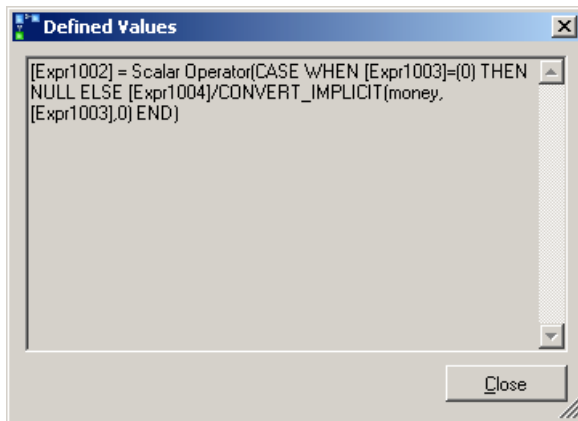
Plan tekstowy może być użyteczny, jeżeli chcemy uzyskać więcej informacji na temat operacji *Stream Aggregate* i *Compute Scalar*. Wykonaj poniższe zapytanie:

```
SET SHOWPLAN_TEXT ON
GO
SELECT AVG(ListPrice) FROM Production.Product
GO
SET SHOWPLAN_TEXT OFF
GO
```

Oto zwrócony plan tekstowy:

```
--Compute Scalar(DEFINE:([Expr1002]=CASE WHEN [Expr1003]=(0) THEN NULL ELSE
[Expr1004]/CONVERT_IMPLICIT(money,[Expr1003],0) END))
--Stream Aggregate(DEFINE:([Expr1003]=Count(*), [Expr1004]=SUM([Product].[ListPrice])))
--Clustered Index Scan(OBJECT:([Product].[PK_Product_ProductID]))
```

Te same informacje można wyciągnąć z planu graficznego, otwierając okno właściwości (lub naciskając *F4*) dla operatorów *Stream Aggregate* i *Compute Scalar*, a następnie wybierając właściwość *Defined Values* (wartości zdefiniowane), zgodnie z rysunkiem 4.13.

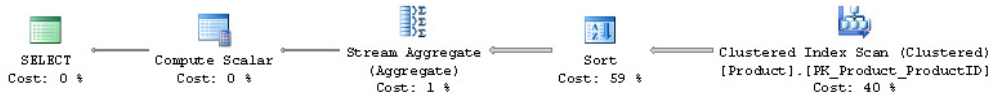
**Rysunek 4.13.** Właściwość Defined Values operatora Compute Scalar

Zauważ, że aby zaimplementować funkcję agregującą AVG, operator *Stream Aggregate* wykonuje zarówno agregację COUNT, jak i SUM, których wyniki będą przechowywane w wyrażeniach Expr1003 i Expr1004. Operator *Compute Scalar* za pośrednictwem wyrażenia CASE sprawdza, czy nie wystąpi błąd dzielenia przez 0. Jak widać w planie tekstowym

wym, jeżeli Expr1003 (wynik agregacji COUNT) wynosi zero, operator *Compute Scalar* zwraca wartość NULL; w przeciwnym wypadku oblicza średnią, dzieląc sumę przez liczbę elementów.

Zobaczmy teraz przykład zapytania z klauzulą GROUP BY. Poniższe zapytanie zwraca plan przedstawiony na rysunku 4.14.

```
SELECT ProductLine, COUNT(*) FROM Production.Product
GROUP BY ProductLine
```

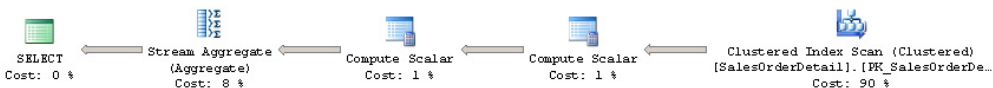


Rysunek 4.14. Stream Aggregate wykorzystujący operator Sort

Operator *Stream Aggregate* zawsze wymaga, aby dane wsadowe były posortowane przez predykat klauzuli GROUP BY, tak więc w tym wypadku operator *Sort* pokazany na planie zwróci dane posortowane po kolumnie *ProductLine*. Kiedy odebrane zostaną posortowane dane wsadowe, rekordy z tej samej grupy będą koło siebie, a operator *Stream Aggregate* będzie mógł z łatwością zliczyć rekordy z każdej grupy. Zauważ, że chociaż pierwszy przykład tego podrozdziału również korzystał z operatora *Stream Aggregate*, nie wymagał posortowanego wsadu: zapytanie bez klauzuli GROUP BY uznaje cały wsad za jedną grupę.

Operator *Stream Aggregate* do posortowania danych wejściowych może także skorzystać z indeksu, zgodnie z poniższym zapytaniem, które wygeneruje plan przedstawiony na rysunku 4.15.

```
SELECT SalesOrderID, SUM(LineTotal) FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
```



Rysunek 4.15. Operator Stream Aggregate wykorzystujący istniejący indeks

W przypadku tego planu nie jest konieczny operator *Sort*, ponieważ operator *Clustered Index Scan* przekazuje dane posortowane po kolumnie *SalesOrderID*, która jest częścią klucza klastrowego tabeli *SalesOrderDetail*. Jak w poprzednim przykładzie operator *Stream Aggregate* przyjmie posortowane dane, ale tym razem obliczy sumę kolumny *LineTotal* dla każdej z grup.

Ponieważ celem operatora *Stream Aggregate* jest agregowanie wartości w grupach, jego algorytm polega na tym, aby dane wsadowe były posortowane na podstawie predykatu z klauzuli GROUP BY, a co za tym idzie, rekordy z tej samej grupy są koło siebie. W algorytmie tym pierwszy odczyt rekordu stworzy pierwszą grupę, a jej wartość zagregowana zostanie zainicjowana. Każdy rekord przeczytany później zostanie sprawdzony pod

kątem tego, czy pasuje do tej samej grupy. Jeżeli pasuje, wartość rekordu zostanie doliczona do wartości zagregowanej tej grupy. Jeżeli natomiast rekord nie pasuje do aktualnej grupy, stworzona zostanie nowa grupa i zainicjowana zostanie jej wartość zagregowana. Proces ten będzie kontynuowany aż do przetworzenia wszystkich rekordów.

Hash Aggregate

Przjrzymy się teraz operatorowi *Hash Aggregate* (agregacja haszowa), na planie pokazywanemu jako *Hash Match* (dopasowanie haszy). W tym rozdziale opisuję dwa algorytmy haszowe: *Hash Aggregate* i *Hash Join*, które działają w podobny sposób i są implementowane przez ten sam fizyczny operator: *Hash Match*. W tym podrozdziale omówię operator *Hash Aggregate*, a w podrozdziale poświęconym złączeniom — operator *Hash Join*.

Optymalizator może wybrać operator *Hash Aggregate* dla dużych tabel, przy czym jeżeli dane nie są posortowane, nie ma potrzeby ich sortowania, a szacunek kardynalności wskazuje tylko na kilka grup. Na przykład tabela `SalesOrderHeader` nie ma indeksu na kolumnie `TerritoryID`, dlatego poniższe zapytanie wykorzysta operator *Hash Aggregate*, zgodnie z rysunkiem 4.16.

```
SELECT TerritoryID, COUNT(*)
FROM Sales.SalesOrderHeader
GROUP BY TerritoryID
```



Rysunek 4.16. Operator Hash Aggregate

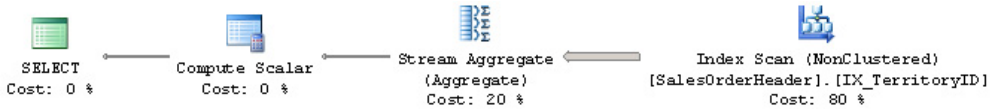
Jak wspomniałem wcześniej w tym rozdziale, operacja haszowa buduje w pamięci tablicę haszy. Klucz hasz wykorzystany dla tej tabeli pokazywany jest w oknie właściwości operatora *Hash Aggregate* jako właściwość *Hash Key Build* (klucz haszowania) — w tym przypadku jest to `TerritoryID`. Ponieważ tabela ta nie jest posortowana po wymaganej kolumnie, każdy skanowany wiersz może należeć do dowolnej grupy.

Algorytm dla operatora *Hash Aggregate* jest podobny do *Stream Aggregate*, z wyjątkiem tego, że w tym przypadku dane wsadowe nie muszą być posortowane. Tabela haszy tworzona jest w pamięci i dla każdego przetwarzanego wiersza obliczana jest wartość hasz. Dla każdej obliczonej wartości hasz algorytm sprawdzi, czy w tabeli haszy istnieje odpowiednia grupa; jeżeli nie istnieje, algorytm stworzy dla niej nowy wpis. W ten sposób wartości dla każdego rekordu zostaną zagregowane w danym wpisie tabeli haszy, a dla każdej grupy w pamięci będzie przechowywany tylko jeden wiersz.

Zauważ, że operator *Hash Aggregate* jest pomocny, jeżeli dane nie są posortowane. Jeśli stworzysz indeks, który będzie w stanie udostępnić posortowane dane, optyma-

lizator może wybrać operator *Stream Aggregate*. Uruchom poniższe polecenie tworzące indeks, a potem jeszcze raz wykonaj poprzednie zapytanie. Operator agregacji zostanie zmieniony na *Stream Aggregate*, zgodnie z rysunkiem 4.17.

```
CREATE INDEX IX_TerritoryID ON Sales.SalesOrderHeader(TerritoryID)
```



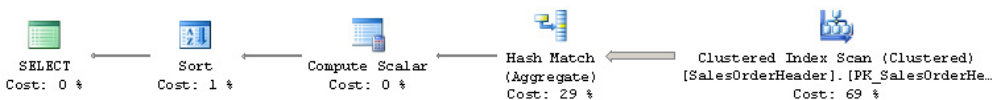
Rysunek 4.17. Operator Stream Aggregate wykorzystujący indeks

Aby posprzątać, usuń indeks za pomocą poniższego polecenia DROP INDEX:

```
DROP INDEX Sales.SalesOrderHeader.IX_TerritoryID
```

Jeżeli wsad nie jest posortowany, a w zapytaniu zostanie jasno określone żądane sortowanie, optymalizator może wprowadzić operator *Sort* wraz z operatorem *Stream Aggregate*, zgodnie z wcześniejszymi wyjaśnieniami, lub może wybrać operator *Hash Aggregate*, a dopiero potem posortować dane, tak jak w poniższym zapytaniu, którego plan został pokazany na rysunku 4.18. Optymalizator oszacuje, która operacja jest mniej kosztowna: czy posortować cały wsad i skorzystać z operatora *Stream Aggregate*, czy też skorzystać z operatora *Hash Aggregate*, a dopiero potem posortować zagregowany wynik.

```
SELECT TerritoryID, COUNT(*)
FROM Sales.SalesOrderHeader
GROUP BY TerritoryID
ORDER BY TerritoryID
```



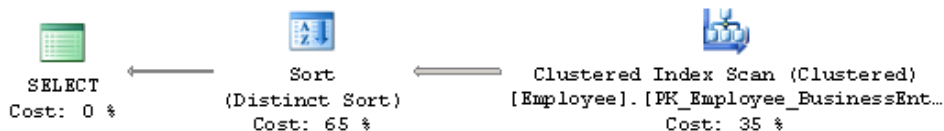
Rysunek 4.18. Operator Hash Aggregate, a po nim operator Sort

Distinct Sort

Zapytanie wykorzystujące słowo kluczowe DISTINCT może być zaimplementowane za pomocą operatorów *Stream Aggregate*, *Hash Aggregate* lub *Distinct Sort* (sortowanie unikalne). Operator *Distinct Sort* jest używany zarówno do usuwania duplikatów, jak i do sortowania danych wsadowych. Tak naprawdę zapytanie ze słowem kluczowym DISTINCT może być przepisane tak, aby wykorzystywało klauzulę GROUP BY, a oba zapytania mogą generować ten sam plan wykonania. Jeżeli dostępny jest indeks zapewniający posortowane dane, optymalizator może wykorzystać operator *Stream Aggregate*. Jeżeli nie ma takiego indeksu, SQL Server może wykorzystać operator *Distinct Sort* lub *Hash Aggregate*.

Przyjrzyjmy się tym przypadkom. Poniższe dwa zapytania zwracają te same dane i generują ten sam plan wykonania, zgodnie z rysunkiem 4.19:

```
SELECT DISTINCT(JobTitle)
FROM HumanResources.Employee
GO
SELECT JobTitle
FROM HumanResources.Employee
GROUP BY JobTitle
```



Rysunek 4.19. Operator Distinct Sort

Zauważ, że plan wykorzystuje operator *Distinct Sort*. Ten operator posortuje wiersze i wyeliminuje duplikaty.

Jeżeli stworzymy indeks, optymalizator może użyć operatora *Stream Aggregate*, ponieważ plan będzie mógł wykorzystać fakt, że dane są już posortowane. Aby to przetestować, uruchom poniższe polecenie:

```
CREATE INDEX IX_JobTitle ON HumanResources.Employee(JobTitle)
```

Teraz uruchom jeszcze raz poprzednie dwa zapytania. Oba zapytania wygenerują teraz plan z operatorem *Stream Aggregate*. Zanim przejdziesz dalej, usuń indeks za pomocą następującego polecenia:

```
DROP INDEX HumanResources.Employee.IX_JobTitle
```

Na koniec, dla dużej tabeli bez indeksu, który mógłby zapewnić posortowane dane, może zostać użyty operator *Hash Aggregate*, tak jak w dwóch poniższych zapytaniach:

```
SELECT DISTINCT(TerritoryID)
FROM Sales.SalesOrderHeader
GO
SELECT TerritoryID
FROM Sales.SalesOrderHeader
GROUP BY TerritoryID
```

Oba zapytania stworzą ten sam plan wykorzystujący operator *Hash Aggregate*, zgodnie z wcześniejszymi przykładami.

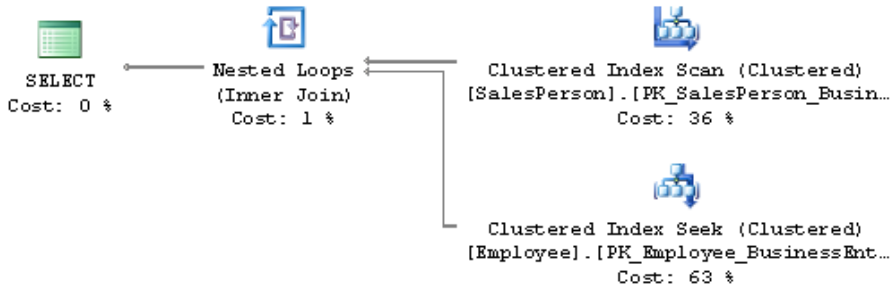
Złączenia

W tym podrozdziale omawiam trzy fizyczne operatory złączeń, których SQL Server używa do implementowania złączeń logicznych: *Nested Loops Join*, *Merge Join* i *Hash Join*. Należy pamiętać, że żaden algorytm nie jest lepszy od innych, a optymalizator wybierze najlepszy w danym przypadku, zgodnie z poniższymi wyjaśnieniami.

Nested Loops Join

Zacznijmy od zapytania wyświetlającego pracowników będących jednocześnie sprzedawcami. Zapytanie stworzy plan z rysunku 4.20, który wykorzystuje operator *Nested Loops Join*.

```
SELECT e.BusinessEntityID, TerritoryID
FROM HumanResources.Employee AS e
JOIN Sales.SalesPerson AS s ON e.BusinessEntityID = s.BusinessEntityID
```



Rysunek 4.20. Operator Nested Loops Join

Dane wsadowe dla operatora *Nested Loops Join* widoczne u góry nazywane są wsadem zewnętrznym, a te u dołu — wsadem wewnętrznym. Algorytm dla operatora *Nested Loops Join* jest bardzo prosty: operator wykorzystywany do uzyskania dostępu do wsadu zewnętrznego jest wykonywany tylko raz, a operator wykorzystywany do uzyskania dostępu do wsadu wewnętrznego jest uruchamiany raz dla każdego kwalifikującego się wiersza z wsadu zewnętrznego. Zauważ, że w tym przykładzie plan jako wsad zewnętrzny skanuje tabelę *SalesPerson* i zwracane jest wszystkie jej 17 rekordów; dlatego, jak wskazuje algorytm, wsad wewnętrzny (operator *Clustered Index Seek*) wykonywany jest 17 razy — raz dla każdego rekordu z tabeli zewnętrznej.

Możesz sprawdzić tę informację, zaglądając do właściwości operatora. Rysunek 4.21 przedstawia właściwości operatora *Clustered Index Scan*, gdzie znajdziesz rzeczywistą liczbę wykonan (która w tym przypadku wynosi 1) i rzeczywistą liczbę rekordów (w tym przypadku 17). Rysunek 4.22 przedstawia właściwości operatora *Clustered Index Seek* i widać na nim, że zarówno liczba wykonan, jak i liczba rekordów wynosiły 17.

Zmieńmy zapytanie, dodając filtr po polu *TerritoryID*:

```
SELECT e.BusinessEntityID, HireDate
FROM HumanResources.Employee AS e
JOIN Sales.SalesPerson AS s ON e.BusinessEntityID = s.BusinessEntityID
WHERE TerritoryID = 1
```

Stworzony plan będzie podobny do poprzednio pokazywanego, wykorzystującego *SalesPerson* jako wsad zewnętrzny i operator *Clustered Index Seek* na tabeli *Employee* jako wsad wewnętrzny. Filtr na tabeli *SalesPerson* wymaga, aby wartość pola *TerritoryID* była równa 1, więc tym razem kwalifikują się tylko trzy rekordy. W rezultacie *Clustered Index*

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	17
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0033007 (36%)
Estimated CPU Cost	0.0001757
Estimated Subtree Cost	0.0033007
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	17
Estimated Row Size	15 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1
Object	
[AdventureWorks2012].[Sales].[SalesPerson].	
[PK_SalesPerson_BusinessEntityID] [s]	
Output List	
[AdventureWorks2012].[Sales].	
[SalesPerson].BusinessEntityID, [AdventureWorks2012].	
[Sales].[SalesPerson].TerritoryID	

Rysunek 4.21. Właściwości operatora Clustered Index Scan

Clustered Index Seek (Clustered)	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	17
Actual Number of Batches	0
Estimated Operator Cost	0.0058127 (63%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Subtree Cost	0.0058127
Number of Executions	17
Estimated Number of Executions	17
Estimated Number of Rows	1
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	2
Object	
[AdventureWorks2012].[HumanResources].[Employee].	
[PK_Employee_BusinessEntityID] [e]	
Output List	
[AdventureWorks2012].[HumanResources].	
[Employee].BusinessEntityID	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2012].	
[HumanResources].[Employee].BusinessEntityID = Scalar	
Operator([AdventureWorks2012].[Sales].[SalesPerson].	
[BusinessEntityID] as [s].[BusinessEntityID])	

Rysunek 4.22. Właściwości operatora Clustered Index Seek

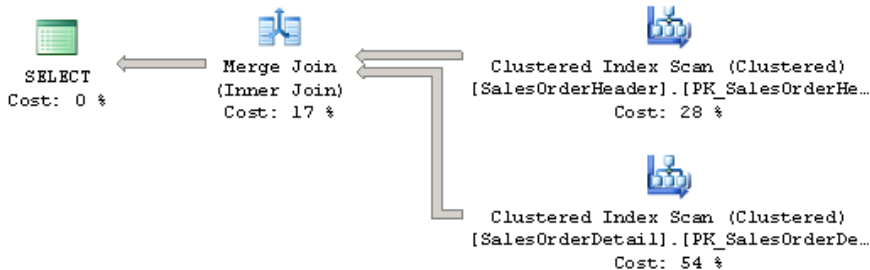
Seek, który jest operatorem dla wsadu wewnętrznego, wykonywany jest tylko trzy razy. Możesz zweryfikować te informacje, przeglądając właściwości każdego z operatorów, jak robiliśmy to w poprzednim przykładzie.

Podsumowując: w algorytmie *Nested Loops Join* operator dla wsadu zewnętrznego będzie wykonywany tylko raz, natomiast operator dla wsadu wewnętrznego będzie wykonywany raz dla każdego kwalifikującego się wiersza z wsadu zewnętrznego. W rezultacie koszt tego algorytmu jest proporcjonalny do rozmiaru wsadu zewnętrznego pomnożonego przez rozmiar wsadu wewnętrznego. W związku z tym optymalizator chętniej wybierze operator *Nested Loops Join*, jeżeli wsad zewnętrzny jest mały, a wsad wewnętrzny ma indeks na kluczu złączenia. Takie złączenie może być szczególnie efektywne, kiedy wsad wewnętrzny jest potencjalnie bardzo duży, ale wyszukiwanych będzie tylko kilka rekordów, wskazanych przez wsad zewnętrzny.

Merge Join

Przyjrzyjmy się teraz przykładowi złączenia *Merge Join*. Uruchom poniższe zapytanie, którego plan widoczny jest na rysunku 4.23.

```
SELECT h.SalesOrderID, s.SalesOrderDetailID, OrderDate
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesOrderDetail s ON h.SalesOrderID = s.SalesOrderID
```



Rysunek 4.23. Przykład złączenia Merge Join

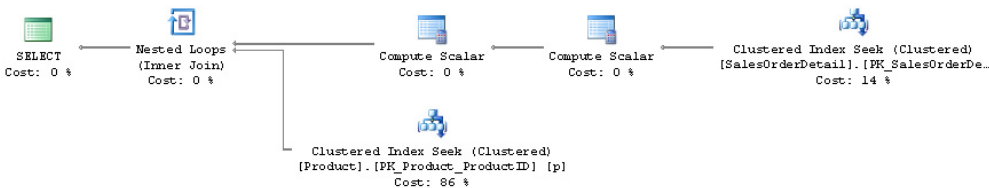
Różnica w stosunku do *Nested Loops Join* polega na tym, że w przypadku *Merge Join* oba operatory wsadowe wykonywane są tylko raz. Możesz to zweryfikować, przeglądając właściwości obu operatorów — przekonasz się wtedy, że liczba wykonań wynosi 1. Kolejną różnicą jest to, że *Merge Join* wymaga operatora równości, a jego wsady muszą być posortowane po predykatcie złączenia. W tym przykładzie predykat złączenia ma znak równości, który wykorzystuje kolumny *SalesOrderID*, a oba indeksy klastrowe są posortowane właśnie po kolumnie *SalesOrderID*.

Operator *Merge Join* korzysta z tego, że oba jego wsady są posortowane po kolumnie z predykatu złączenia, i jednocześnie czyta wiersze z wsadów i porównuje. Jeżeli wiersze pasują, są zwracane. Jeżeli nie pasują, mniejsza wartość może zostać odrzucona — ponieważ oba wsady są posortowane, odrzucony wiersz nie będzie pasował do żadnego

innego wiersza z drugiej tabeli. Proces ten jest kontynuowany, dopóki jedna z tabel nie zostanie zakończona. Nawet jeśli w drugiej tabeli nadal są wiersze, oczywiste jest, że nie będą pasowały do żadnego wiersza z całkowicie przeskanowanej tabeli, nie ma więc sensu kontynuować. Ponieważ obie tabele potencjalnie mogą zostać przeskanowane w całości, maksymalny koszt operatora *Merge Join* równy jest sumie obu wsadów.

Jest mało prawdopodobne, że optymalizator wybierze *Merge Join*, jeżeli oba wsady nie są posortowane. Może jednak zdecydować o posortowaniu wsadów, jeżeli uzna, że koszt będzie niższy niż w przypadku pozostałych możliwości. Zobaczmy przykład, w którym wymusimy użycie *Merge Join* na przykład dla planu, który normalnie wyko-

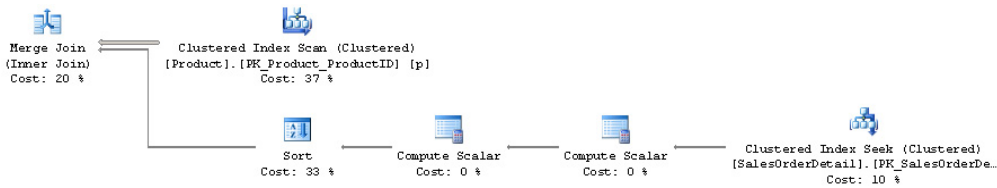
```
SELECT * FROM Sales.SalesOrderDetail s
JOIN Production.Product p ON s.ProductID = p.ProductID
WHERE SalesOrderID = 43659
```



Rysunek 4.24. Plan z podpowiedzią nakazującą wykorzystanie Merge Join

W tym przypadku dobry plan tworzony jest z wykorzystaniem wydajnych operatorów *Clustered Index Seek*. Jeżeli za pomocą podpowiedzi wymusimy zastosowanie operatora *Merge Join*, jak w poniższym zapytaniu, optymalizator będzie musiał wprowadzić posortowane wsady w postaci operatorów *Clustered Index Scan* lub *Sort*. Oba te operatory można znaleźć na planie z rysunku 4.25. Oczywiście te dodatkowe operacje są bardziej kosztowne niż *Clustered Index Seek* i są wprowadzane, ponieważ zmusiliśmy do tego optymalizator.

```
SELECT * FROM Sales.SalesOrderdetail s
JOIN Production.Product p ON s.ProductID = p.ProductID
WHERE SalesOrderID = 43659
OPTION (MERGE JOIN)
```



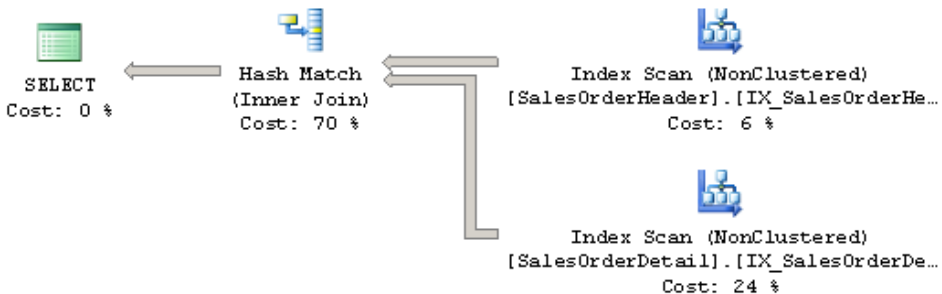
Rysunek 4.25. Plan z podpowiedzią nakazującą wykorzystanie Merge Join

Podsumowując: z uwagi na charakter operatora *Merge Join* optymalizator wybierze ten algorytm, jeżeli wsady będą średnie lub duże, predykat złączenia będzie wykorzystywał operator równości i wsady będą posortowane.

Hash Join

Trzeci algorytm złączeń wykorzystywany przez SQL Server to *Hash Join*. Uruchom poniższe zapytanie, aby uzyskać plan przedstawiony na rysunku 4.26, a potem dokładniej przyjrzymy się, jak działa ten operator.

```
SELECT h.SalesOrderID, s.SalesOrderDetailID FROM Sales.SalesOrderHeader h
JOIN Sales.SalesOrderDetail s ON h.SalesOrderID = s.SalesOrderID
```



Rysunek 4.26. Przykład wykorzystania operatora Hash Join

Podobnie jak *Merge Join*, operator *Hash Join* wymaga wykorzystania operatora równości w predykanie złączenia, w przeciwieństwie jednak do niego nie wymaga, aby wsady były posortowane. Ponadto operacje na obu wsadach wykonywane są tylko raz, co możesz zweryfikować we właściwościach operatorów. *Hash Join* tworzy jednak w pamięci tabelę haszy. Optymalizator wykorzysta szacowanie kardynalności do określenia mniejszego z wsadów, nazywanego *build input* (wsad budowania), który zostanie użyty do zbudowania tabeli haszy w pamięci. Jeżeli nie ma wystarczająco dużo pamięci, aby zmieścić tabelę haszy, SQL Server skorzysta z pliku w bazie *tempdb*, co może niekorzystnie wpłynąć na wydajność. *Hash Join* jest operacją blokującą, ale tylko na czas haszowania wsadu. Kiedy wsad zostanie zhaszowany, druga tabela, nazywana *probe input* (wsad sondowany), będzie odczytywana, a jej rekordy będą porównywane z tabelą haszy za pomocą podobnego mechanizmu jak w przypadku operatora *Hash Aggregate*. Jeżeli rekordy pasują, zostaną zwrócone, a wynik zostanie przesłany dalej. Na planie wykonania tabela znajdująca się u góry zostanie wykorzystana jako wsad budowania, a ta z dołu — jako wsad sondowany.

Zauważ też, że może pojawić się zachowanie nazywane „zmianą ról”. Jeżeli optymalizator nie będzie w stanie poprawnie oszacować, która z tabel jest mniejsza, role wsadu budowania i wsadu sondowanego mogą zostać odwrócone podczas wykonywania zapytania i nie zostanie to pokazane na planie wykonania.

Podsumowując: optymalizator może wybrać *Hash Join* dla dużych tabel, jeżeli w predykcji złączenia wykorzystany zostanie operator równości. Ponieważ obie tabele są skanowane, koszt operacji *Hash Join* jest sumą obu wsadów.

Działania równoległe

SQL Server może wykorzystać działania równoległe, aby wykonując kilka procesorów jednocześnie, pomóc sobie w szybszym wykonywaniu kosztownych zapytań. Jednakże, chociaż zapytanie może uzyskać lepszą wydajność dzięki zastosowaniu planów równoległych, wykorzystane może być więcej zasobów.

Aby SQL Server mógł rozważyć wykorzystanie planów równoległych, instancja musi mieć dostęp do przynajmniej dwóch procesorów lub rdzeni albo do konfiguracji wielowątkowej. Ponadto zarówno maska pokrewieństwa, jak i maksymalny poziom współbieżności muszą być skonfigurowane tak, aby umożliwić wykorzystanie przynajmniej dwóch procesorów. Jak wyjaśnię w dalszej części tego podrozdziału, SQL Server będzie rozważał współbieżność tylko dla zapytań seryjnych, gdzie koszt przewyższa skonfigurowaną wartość graniczną współbieżności (domyślna wartość to 5).

Opcja konfiguracyjna dotycząca maski pokrewieństwa określa, które procesory mogą uruchamiać wątki SQL Servera, a domyślna wartość 0 oznacza, że mogą zostać wykorzystane wszystkie procesory. Opcja dotycząca maksymalnego poziomu współbieżności jest stosowana do ograniczania liczby procesorów, które mogą zostać wykorzystane w planach równoległych, a domyślna wartość 0 również oznacza, że mogą zostać wykorzystane wszystkie dostępne procesory. Jak widzisz, jeżeli masz odpowiedni sprzęt, SQL Server domyślnie zezwala na korzystanie z planów równoległych i nie jest potrzebna żadna dodatkowa konfiguracja.

Plan współbieżny zostanie rozważony, jeżeli szacowany koszt planu seryjnego będzie wyższy niż wartość graniczna zdefiniowana w konfiguracji. Nie gwarantuje to jednak, że współbieżność zostanie zastosowana, ponieważ ostateczna decyzja co do zastosowania równoległego wykonania zapytania zostanie podjęta na podstawie kosztów. Oznacza to, że nie ma gwarancji, iż najlepszy znaleziony plan współbieżny będzie miał niższy koszt niż najlepszy plan seryjny, więc plan seryjny może i tak okazać się lepszy. Współbieżność jest implementowana przez fizyczny operator współbieżny, zwany również operatorem wymiany, który implementuje operatory logiczne *Distribute Streams* (rozproszenie strumieni), *Gather Streams* (zbieranie strumieni) i *Repartition Streams* (repartycjonowanie strumieni).

Aby tworzyć plany współbieżne, musimy w bazie AdventureWorks2012 stworzyć kilka dużych tabel (a jedna z tych tabel będzie również wykorzystywana w następnym podrozdziale). Prosty sposób to kilkukrotne skopiowanie danych z tabeli Sales.SalesOrderDetail za pomocą poniższych zapytań:

```

SELECT *
INTO #temp
FROM Sales.SalesOrderDetail
UNION ALL SELECT * FROM Sales.SalesOrderDetail
UNION ALL SELECT * FROM Sales.SalesOrderDetail
UNION ALL SELECT * FROM Sales.SalesOrderDetail
UNION ALL SELECT * FROM Sales.SalesOrderDetail
UNION ALL SELECT * FROM Sales.SalesOrderDetail

```

```

SELECT IDENTITY(int, 1, 1) AS ID, CarrierTrackingNumber, OrderQty, ProductID, UnitPrice,
↪LineTotal, rowguid, ModifiedDate
INTO dbo.SalesOrderDetail FROM #temp

```

```

SELECT IDENTITY(int, 1, 1) AS ID, CarrierTrackingNumber, OrderQty, ProductID, UnitPrice,
↪LineTotal, rowguid, ModifiedDate
INTO dbo.SalesOrderDetail2 FROM #temp

```

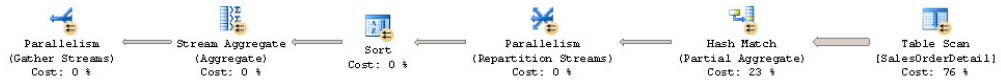
```
DROP TABLE #temp
```

Poniższe zapytanie, wykorzystujące jedną z właśnie stworzonych tabel, zastosuje plan współbieżny. Ponieważ plan ten jest zbyt duży, aby wydrukować go w książce, na rysunku 4.27 przedstawiam tylko część równoległą.

```

SELECT ProductID, COUNT(*)
FROM dbo.SalesOrderDetail
GROUP BY ProductID

```



Rysunek 4.27. Część planu równoległego

Jedną z zalet planów równoległych, w porównaniu do planów XML i tekstowych, jest to, że dzięki symbolowi współbieżności (małe żółte kółko ze strzałkami) umieszczonemu na ikonie operatora można z łatwością odczytać, jakie operatory są wykonywane równolegle.

Aby sprawdzić, dlaczego plan równoległy był rozpatrywany i został wybrany, możesz sprawdzić koszt planu seryjnego. Jednym ze sposobów jest użycie podpowiedzi MAXDOP do wymuszenia wykorzystania planu seryjnego dla tego samego zapytania:

```

SELECT ProductID, COUNT(*)
FROM dbo.SalesOrderDetail
GROUP BY ProductID
OPTION (MAXDOP 1)

```

Koszt planu seryjnego wynosi 9,70746. Ponieważ wartość graniczna kosztu dla współbieżności wynosi 5, wartość ta kwalifikuje się do wyszukania planu współbieżnego. Możesz wykonać interesujące doświadczenie w swoim środowisku testowym, zmieniając wartość graniczną na 10 za pomocą poniższego zapytania:

```
EXEC sp_configure 'cost threshold for parallelism', 10
GO
RECONFIGURE
GO
```

Jeżeli ponownie uruchomisz to samo zapytanie, tym razem bez podpowiedzi MAXDOP, otrzymasz plan seryjny z kosztem 9,70746. Ponieważ wartość graniczna kosztu wynosi teraz 10, optymalizator nawet nie próbował znaleźć planu równoległego. Nie zapomnij przywrócić wartości granicznej z powrotem do domyślnej wartości 5, uruchamiając poniższe zapytanie:

```
EXEC sp_configure 'cost threshold for parallelism', 5
GO
RECONFIGURE
GO
```

Pamiętaj, że zmieniliśmy tę wartość tylko na potrzeby demonstracji; rzadko zachodzi konieczność zmiany tego ustawienia.

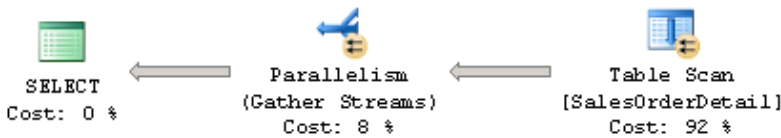
Operator wymiany

Jak wcześniej wspomniałem, współbieżność implementowana jest przez fizyczny operator współbieżny, znany także pod nazwą operatora wymiany. Współbieżność w SQL Serverze polega na podzieleniu zadania na dwie lub więcej kopii tego samego operatora. Na przykład, jeżeli poprosisz, aby SQL Server policzył liczbę rekordów w małej tabeli, wykorzystany zostanie jeden operator *Stream Aggregate*. Jeżeli jednak poprosisz o policzenie liczby rekordów w bardzo dużej tabeli, SQL Server może wykorzystać dwa lub więcej operatorów *Stream Aggregate*, a każdy z nich policzy liczbę rekordów w części tabeli. Te operatory będą działały równolegle, każdy w innym schedulerze, i każdy będzie wykonywał swoją część pracy. Jest to oczywiście uproszczone wyjaśnienie, ale w tym podrozdziale otrzymasz również szczegóły.

Partycjonowanie danych pomiędzy operatorami jest obsługiwane, w większości przypadków, przez operatory współbieżności lub wymiany. Interesujące jest to, że większość operatorów nie musi wiedzieć, iż są wykonywane równolegle. Na przykład operator *Stream Aggregate*, o którym przed chwilą wspomnieliśmy, otrzymuje jedynie rekordy, zlicza je i zwraca rekordy bez wiedzy, że inne operatory *Stream Aggregate* wykonują równolegle tę samą pracę na innych zestawach rekordów. Są oczywiście również operatory świadome tego, że działają współbieżnie, np. *Parallel Scan* (skanowanie współbieżne), który omówimy teraz. Aby zobaczyć, jak działa, uruchom poniższe zapytanie, które zaowocuje stworzeniem planu z rysunku 4.28.

```
SELECT * FROM dbo.SalesOrderDetail
WHERE LineTotal > 3234
```

Jak wspomnieliśmy wcześniej, *Parallel Scan* jest jednym z niewielu operatorów świadomych tego, że działają współbieżnie, i działa on na podstawie procesu współbieżnego dostarczania stron, który przekazuje zestawy stron danych do operatorów



Rysunek 4.28. Zapytanie Parallel Scan

w planie. W pokazanym planie SQL Server wykorzysta dwa lub więcej operatorów *Table Scan*, a współbieżne dostarczanie stron będzie dostarczało do nich strony danych. Ogromną zaletą tej metody jest to, że nie wymaga podzielenia danych dla wątków na równe części, lecz są one dostarczane na żądanie, co może pomóc w przypadkach, kiedy jeden z procesorów jest zajęty przez inne aktywności systemowe i może nie być w stanie przetworzyć wielu rekordów, a tym samym spowalniałby cały proces. Na przykład w moim teście widzę dwa wątki — jeden przetwarza 27 477 rekordów, a drugi 26 535, zgodnie z poniższym fragmentem planu XML (możesz również znaleźć te informacje na planie graficznym poprzez wybranie właściwości operatora *Table Scan* i rozwinięcie sekcji *Actual Number of Rows*).

```
<RunTimeInformation>
  <RunTimeCountersPerThread Thread="1" ActualRows="27477" ... />
  <RunTimeCountersPerThread Thread="2" ActualRows="26535" ... />
  <RunTimeCountersPerThread Thread="0" ActualRows="0" ... />
</RunTimeInformation>
```

Oczywiście zależne jest to od aktywności każdego z procesorów. Aby to zasymulować, uruchomiłem kolejny test, podczas gdy jeden z procesorów był zajęty, a plan pokazał, że jeden z procesorów przetworzył 11 746 rekordów, a drugi 42 266. Zauważ, że plan pokazuje także wątek 0, będący koordynatorem lub głównym wątkiem, który nie przetwarza rekordów.

Chociaż plan na rysunku 4.28 pokazuje tylko operator wymiany *Gather Streams*, operator wymiany może przyjąć trzy różne funkcje lub operacje logiczne, które omówię w dalszej części.

Wymiana *Gather Streams* jest również nazywana rozpoczęciem wymiany współbieżnej i znajduje się zawsze na początku planu lub regionu współbieżnego, biorąc pod uwagę, że wykonywanie planu zaczyna się od lewej. Operator ten przyjmuje kilka strumieni wsadowych, łączy je i tworzy pojedynczy wyjściowy strumień rekordów. Na przykład na planie z rysunku 4.28 operator wymiany *Gather Streams* pobiera dane wyprodukowane przez dwa operatory *Table Scan* działające równolegle i przesyła je do operatora nadrzędnego.

W przeciwieństwie do operatora *Gather Streams*, operator *Distribute Streams* jest nazywany końcem wymiany współbieżnej; pobiera on pojedynczy strumień wsadowy i dzieli go na wiele strumieni wyjściowych.



Trzecim logicznym operatorem wymiany jest operator *Repartition Streams*, który pobiera wiele strumieni i produkuje wiele strumieni rekordów.



W poprzednim przykładzie *Parallel Scan* to operator świadomy działania współbieżnego, w którym współbieżne dostarczanie stron przypisuje zestawy stron do operatorów w planie. Jak już wspomniałem, w większości przypadków operatory nie są świadome tego, że uruchomione są równolegle, dlatego zadaniem operatora wymiany jest przesłanie do nich rekordów za pomocą jednego ze sposobów partycjonowania opisanych w tabeli 4.2. Te typy partycjonowania są pokazywane użytkownikowi we właściwości planu wykonania o nazwie *Partitioning Type* i mają sens tylko dla operatorów *Repartition Streams* i *Distribute Streams*, ponieważ, jak widać na planie z rysunku 4.28, operator *Gather Streams* przekazuje tylko strumień do pojedynczego odbiorcy. Kilka przykładów partycjonowania pokażę w dalszej części tego podrozdziału.

Tabela 4.2. Typy partycjonowania

Typ partycjonowania	Opis
<i>Hash</i>	Wykonuje funkcję haszującą na jednej lub kilku kolumnach wiersza i podejmuje decyzję, gdzie przesłać wiersz.
<i>Round Robin</i>	Każdy pakiet wierszy przekazywany jest do następnego odbiorcy w sekwencji.
<i>Broadcast</i>	Wszystkie wiersze przesyłane są do wszystkich wątków odbierających.
<i>Demand</i>	Każdy nowy wiersz jest przekazywany do odbiorcy, który go zażąda. Jest to jedyny typ wymiany, który wykorzystuje model wciągania zamiast wypychania danych.
<i>Range</i>	Wykonuje funkcję określającą zakres na jednej z kolumn wiersza i podejmuje decyzję, gdzie wysłać wiersz.

Inną właściwością operatorów wymiany jest to, że zachowują kolejność wierszy wsadowych. Kiedy tak jest, nazywane są wymianami *scalającymi* lub *zachowującymi kolejność*. W przeciwnym wypadku nazywają się *niescalającymi* lub *niezachowującymi kolejności*. Wymiany scalające nie wykonują żadnych operacji sortowania; wiersze muszą być już posortowane. Z tego powodu taka operacja zachowująca kolejność ma sens tylko dla wymian *Gather Streams* i *Repartition Streams*. W przypadku *Distribute Streams* jest tylko jeden dostawca danych, nie ma więc czego scalać. Warto również wspomnieć, że wymiany scalające mogą skalować się gorzej niż wymiany niescalające. Na przykład porównaj plany wykonania poniższych dwóch wersji pierwszego przykładu z tego podrozdziału, druga korzysta z klauzuli `ORDER BY`:

```
SELECT ProductID, COUNT(*)
FROM dbo.SalesOrderDetail
GROUP BY ProductID
GO
SELECT ProductID, COUNT(*)
```

```
FROM dbo.SalesOrderDetail
GROUP BY ProductID
ORDER BY ProductID
```

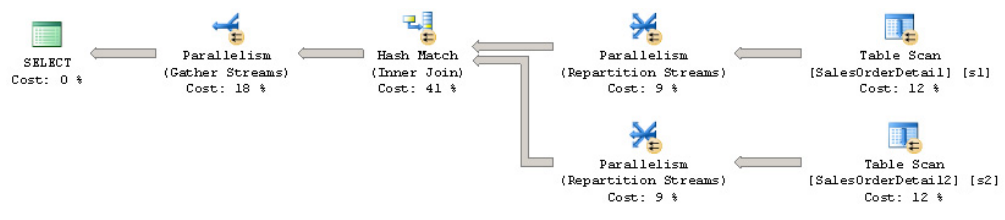
Chociaż oba plany są prawie takie same, ten drugi zwraca wynik posortowany za pośrednictwem operatora wymiany zachowującego kolejność. Możesz to zweryfikować, zaglądając do właściwości operatora wymiany *Gather Streams*, który zawiera sekcję *Order By*, zgodnie z rysunkiem 4.29.

Parallelism	
An operation involving parallelism.	
Physical Operation	Parallelism
Logical Operation	Gather Streams
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	266
Actual Number of Batches	0
Estimated Operator Cost	0.03022 (0%)
Estimated I/O Cost	0
Estimated Subtree Cost	7.71224
Estimated CPU Cost	0.0302168
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	266
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	2
Output List	
[AdventureWorks2012].[dbo].	
[SalesOrderDetail].ProductID, globalagg1005	
Order By	
[AdventureWorks2012].[dbo].	
[SalesOrderDetail].ProductID Ascending	

Rysunek 4.29. Właściwości operatora współbieżności

Partycjonowanie haszowe jest najbardziej powszechnym typem partycjonowania i może być wykorzystywane do równoległego uruchamiania operatorów *Merge Join* lub *Hash Join*, jak w następnym przykładzie. Uruchom poniższe zapytanie, które wygeneruje plan analogiczny do tego z rysunku 4.30, gdzie wykorzystane zostało partycjonowanie haszowe przez operatory wymiany *Repartition Streams*, co zostało również pokazane we właściwości *Partitioning Type* jednego z tych operatorów przedstawionej na rysunku 4.31. W tym przypadku partycjonowanie haszowe rozdziela rekordy budowania i rekordy sondowania pomiędzy poszczególnymi wątkami *Hash Join*.

```
SELECT * FROM dbo.SalesOrderDetail s1 JOIN dbo.SalesOrderDetail s2
ON s1.id = s2.id
```



Rysunek 4.30. Przykład partycjonowania haszowego

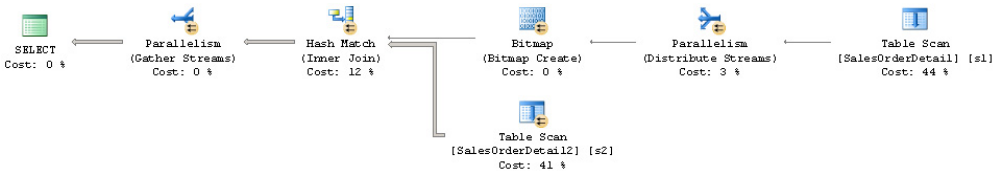
Parallelism	
Repartition streams.	
Physical Operation	Parallelism
Logical Operation	Repartition Streams
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	727902
Actual Number of Batches	0
Estimated I/O Cost	0
Estimated Operator Cost	4.24057 (9%)
Estimated CPU Cost	4.2406
Estimated Subtree Cost	10.1249
Estimated Number of Executions	1
Number of Executions	4
Estimated Number of Rows	727902
Estimated Row Size	95 B
Actual Rebinds	0
Actual Rewinds	0
Partitioning Type	Hash
Node ID	2
Output List	
[AdventureWorks2012].[dbo].[SalesOrderDetail].ID, [AdventureWorks2012].[dbo].[SalesOrderDetail].CarrierTrackingNumber, [AdventureWorks2012].[dbo].[SalesOrderDetail].OrderQty, [AdventureWorks2012].[dbo].[SalesOrderDetail].ProductID, [AdventureWorks2012].[dbo].[SalesOrderDetail].UnitPrice, [AdventureWorks2012].[dbo].[SalesOrderDetail].LineTotal, [AdventureWorks2012].[dbo].[SalesOrderDetail].rowguid, [AdventureWorks2012].[dbo].[SalesOrderDetail].ModifiedDate	
Partition Columns	
[AdventureWorks2012].[dbo].[SalesOrderDetail].ID	

Rysunek 4.31. Właściwości operatora Repartition Streams

Na koniec poniższe zapytanie, które zawiera bardzo selektywny predykat, stworzy plan z rysunku 4.32, wykorzystujący zarówno operator startu współbieżności *Gather Streams*, jak i operator końca współbieżności *Distribute Streams*. Operator *Distribute Streams* wykorzystuje partycjonowanie z rozgłaszaniem, co można zweryfikować w jego właściwościach przedstawionych na rysunku 4.33. Partycjonowanie z rozgłaszaniem przesyła jedyny kwalifikujący się wiersz z tabeli `table1` do wszystkich wątków *Hash Join*.

Operator mapy bitowej jest wykorzystywany do wyeliminowania większości wierszy z tabeli table2, co znacznie poprawia wydajność zapytania.

```
SELECT * FROM dbo.SalesOrderDetail s1
JOIN dbo.SalesOrderDetail s2 ON s1.ProductID = s2.ProductID
WHERE s1.id = 123
```



Rysunek 4.32. Przykład partycjonowania z rozgłaszaniem

Parallelism	
Distribute streams.	
Physical Operation	Parallelism
Logical Operation	Distribute Streams
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	4
Actual Number of Batches	0
Estimated Operator Cost	0.37791 (3%)
Estimated I/O Cost	0
Estimated Subtree Cost	6.66262
Estimated CPU Cost	0.0285128
Number of Executions	4
Estimated Number of Executions	1
Estimated Number of Rows	1.15969
Estimated Row Size	95 B
Actual Rebinds	0
Actual Rewinds	0
Partitioning Type	Broadcast
Node ID	3
Output List	
[AdventureWorks2012].[dbo].[SalesOrderDetail].ID, [AdventureWorks2012].[dbo].[SalesOrderDetail].CarrierTrackingNumber, [AdventureWorks2012].[dbo].[SalesOrderDetail].OrderQty, [AdventureWorks2012].[dbo].[SalesOrderDetail].ProductID, [AdventureWorks2012].[dbo].[SalesOrderDetail].UnitPrice, [AdventureWorks2012].[dbo].[SalesOrderDetail].LineTotal, [AdventureWorks2012].[dbo].[SalesOrderDetail].rowguid, [AdventureWorks2012].[dbo].[SalesOrderDetail].ModifiedDate	

Rysunek 4.33. Właściwości operatora wymiany Distribute Streams

Operatory mapy bitowej omawiam dokładniej w rozdziale 9., gdzie zobaczysz, jak są wykorzystywane do optymalizacji wydajności zapytań wykonywanych na magazynach danych.

Ograniczenia

Możesz spotkać się z przypadkami, kiedy będziesz miał wystarczająco kosztowne zapytanie, które i tak nie będzie wykonywane równolegle. Istnieje kilka funkcjonalności SQL Servera, które mogą powstrzymać wykorzystanie planu równoległego:

- ▶ funkcje skalarne definiowane przez użytkownika;
- ▶ funkcje CLR definiowane przez użytkownika wykorzystujące dostęp do danych;
- ▶ różne funkcje wbudowane, takie jak `OBJECT_ID()`, `ERROR_NUMBER()` i `@@TRANCOUNT`;
- ▶ kursory dynamiczne.

Podobnie istnieje kilka funkcjonalności wymuszających sekcję seryjną w obrębie planu równoległego, których wykorzystanie może prowadzić do problemów wydajnościowych. Funkcjonalności te to:

- ▶ wielopoleceniowe, zwracające tabele funkcje zdefiniowane przez użytkownika;
- ▶ klauzula `TOP`;
- ▶ globalne agregacje skalarne;
- ▶ funkcje sekwencyjne;
- ▶ pula wieloodbiorcowa;
- ▶ skanowanie wsteczne;
- ▶ skanowanie tabel systemowych;
- ▶ zapytanie rekursywne.

Na przykład poniższy kod przedstawia, jak pierwszy przykład z tego podrozdziału zmienia się w plan seryjny za pomocą prostej funkcji zdefiniowanej przez użytkownika:

```
CREATE FUNCTION dbo.ufn_test(@ProductID int)
RETURNS int
AS
BEGIN
RETURN @ProductID
END
GO
SELECT dbo.ufn_test(ProductID), ProductID, COUNT(*)
FROM dbo.SalesOrderDetail
GROUP BY ProductID
```

Jak pokazywałem w rozdziale 1., opcjonalny atrybut `NonParallelPlanReason` elementu `QueryPlan` zawiera opis, z którego wyniku, dlaczego plan równoległy nie został wybrany dla zapytania. W tym przypadku parametr ten ma wartość `CouldNotGenerateValidParallelPlan` (nie można wygenerować poprawnego planu równoległego), zgodnie z poniższym fragmentem planu XML:

```
<QueryPlan ... NonParallelPlanReason="CouldNotGenerateValidParallelPlan" ... >
```

Istnieje również niedokumentowana i tym samym niewspierana flaga, którą możesz wykorzystać do próby wymuszenia planu równoległego. Flagi 8649 możesz użyć do ustawienia granicy kosztu dla planu równoległego na wartość 0, by zachęcić optymalizator do zastosowania planu równoległego, co w niektórych przypadkach może pomóc (głównie ze względów związanych z kosztem). Na potrzeby demonstracji przyjrzyj się poniższemu przykładowi wykorzystującemu małe tabele (zauważ, że jest to tabela SalesOrderDetail ze schematu Sales, a nie większa tabela ze schematu dbo, którą stworzyliśmy wcześniej):

```
SELECT ProductID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
```

Poprzednie zapytanie tworzy plan seryjny z kosztem 0,429621. Korzystając z flagi 8649, jak w poniższym zapytaniu, otrzymamy plan równoległy z nieznacznie niższym kosztem 0,386606 jednostki:

```
SELECT ProductID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
OPTION (QUERYTRACEON 8649)
```

Aktualizacje

Do tej pory skupialiśmy się na tym, jak optymalizator rozwiązuje zapytania SELECT, dlatego w tym podrozdziale będziemy omawiać operacje aktualizacji. Operacje te to część operacji na bazie danych i one również wymagają optymalizacji, dzięki czemu zostaną wykonane najszybciej jak to możliwe. Pamiętaj, że kiedy piszę „aktualizacje”, odnoszę się do dowolnej operacji wykonywanej przez polecenia INSERT, DELETE i UPDATE, a także przez polecenie MERGE, które zostało wprowadzone w SQL Serverze 2008. W tym rozdziale wyjaśnię podstawy operacji aktualizacji i to, w jaki sposób szybko mogą się one skomplikować, jeżeli muszą aktualizować indeksy, uzyskiwać dostęp do wielu tabel i dbać o zachowanie istniejących więzów integralności. Pokażę, jak optymalizator może wybierać plany *per wiersz* lub *per indeks* w celu optymalizacji poleceń UPDATE, a także opiszę problem Halloween dotyczący ochrony i wyjaśnię, jak SQL Server sobie z nim radzi.

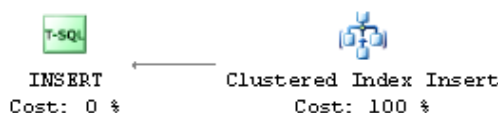
Chociaż podczas wykonywania aktualizacji angażowane są również inne obszary SQL Servera, takie jak transakcje, kontrola współbieżności lub blokowanie, przetwarzanie aktualizacji jest całkowicie zintegrowane wewnątrz frameworka przetwarzania zapytań SQL Servera. Operacje aktualizacji są także optymalizowane, aby mogły być wykonywane jak najszybciej. W tym podrozdziale opowiem więc o aktualizacjach z perspektywy przetwarzania zapytań. Jak wspomniałem wcześniej, na potrzeby tego podrozdziału odnoszę się do operacji wykonywanych przez polecenia INSERT, DELETE, UPDATE i MERGE jako „aktualizacji”.

Plany dla aktualizacji mogą być skomplikowane, ponieważ muszą oprócz danych aktualizować również istniejące indeksy. Także ze względu na obiekty takie jak ograniczenia sprawdzające, więzy integralności referencyjnej i wyzwalacze plany te muszą uzyskiwać dostęp do wielu tabel i dbać o istniejące ograniczenia. Aktualizacje mogą też wymagać modyfikacji wielu tabel, jeżeli konieczna będzie kaskadowa aktualizacja powiązanych tabel lub jeżeli zdefiniowane będą jakieś wyzwalacze. Niektóre z tych operacji, na przykład aktualizacja indeksów, mogą mieć ogromny wpływ na wydajność całej operacji, a omówimy to dokładniej teraz.

Operacje aktualizacji wykonywane są w dwóch krokach, które można podsumować jako etap odczytu, a po nim etap aktualizacji. Pierwszy krok zapewnia szczegóły zmian, które mają zostać wprowadzone, oraz informację o tym, które wiersze mają zostać zaktualizowane. Dla operacji INSERT są to wartości, które mają zostać wstawione, natomiast dla operacji DELETE — klucze rekordów do usunięcia, przy czym mogą to być klucze klastrowe dla indeksów klastrowych lub identyfikatory RID dla stert. Dla operacji UPDATE konieczne są zarówno klucze rekordów do zmodyfikowania, jak i, jeżeli to konieczne, dane do wstawienia. W tym kroku SQL Server może odczytywać tabelę, która ma być aktualizowana, w ten sam sposób jak w przypadku dowolnego zapytania SELECT. W drugim kroku wykonywane są operacje aktualizacji, włączając w to aktualizację indeksów, walidację ograniczeń i wykonywanie wyzwalaczy. Operacja aktualizacji zakończy się niepowodzeniem i zostanie wycofana, jeżeli będzie naruszała więzy integralności.

Zacznę od przykładu bardzo prostej operacji. Wstawienie nowego rekordu do tabeli Person.CountryRegion za pomocą poniższego zapytania stworzy bardzo prosty plan przedstawiony na rysunku 4.34.

```
INSERT INTO Person.CountryRegion (CountryRegionCode, Name)
VALUES ('ZZ', 'Nowy kraj')
```

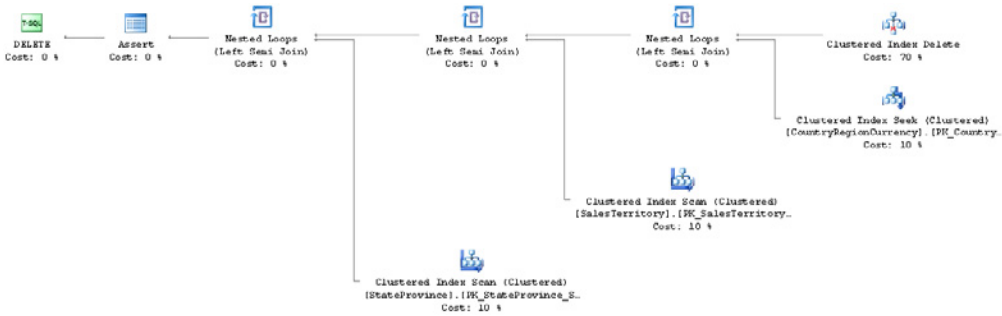


Rysunek 4.34. Przykład operacji INSERT

Operacja może się jednak bardzo szybko skomplikować, jeżeli spróbujesz usunąć ten sam rekord za pomocą poniższego zapytania, zgodnie z rysunkiem 4.35:

```
DELETE FROM Person.CountryRegion
WHERE CountryRegionCode = 'ZZ'
```

Jak widzisz, oprócz CountryRegion wykorzystywane są jeszcze trzy tabele (StateProvince, CountryRegionCurrency i SalesTerritory). Powodem tego jest to, że te trzy tabele mają klucze obce odnoszące się do tabeli CountryRegion, więc SQL Server musi sprawdzić, czy w tych tabelach nie istnieją żadne rekordy odnoszące się do tej konkretnej wartości CountryRegionCode. Dlatego SQL Server uzyskuje dostęp do tych tabel, a operator *Assert*



Rysunek 4.35. Przykład operacji DELETE

jest wykorzystywany na końcu planu do wykonania tej walidacji. Gdyby w którejś z tabel istniała usuwana wartość CountryRegionCode, operator *Assert* zgłosi wyjątek i SQL Server wycofa transakcję i zwróci poniższy błąd:

```
Msg 547, Level 16, State 0, Line 1
The DELETE statement conflicted with the REFERENCE constraint
"FK_CountryRegionCurrency_CountryRegion_CountryRegionCode". The conflict occurred in database
"AdventureWorks2012", table "Sales.CountryRegionCurrency", column 'CountryRegionCode'.
The statement has been terminated.
```

Poprzedni przykład pokazuje, jak operacje aktualizacji mogą uzyskiwać dostęp do innych tabel niezawartych w zapytaniu — w tym przypadku powodem były więzy integralności referencyjnej. Aktualizacja indeksów nieklastrowych zostanie omówiona w następnym podrozdziale.

Plany per wiersz i per indeks

Ważną operacją wykonywaną przez aktualizacje jest modyfikowanie i aktualizowanie istniejących indeksów nieklastrowych, co odbywa się za pośrednictwem planów *per wiersz* i *per indeks* (nazywanych również odpowiednio planami wąskimi i planami szerokimi). W przypadku planu per wiersz aktualizacje na tabeli bazowej i na istniejących indeksach wykonywane są przez ten sam operator, wiersz po wierszu. Natomiast w przypadku planu per indeks tabela bazowa i każdy indeks nieklastrowy są aktualizowane przez osobne operatory.

Z wyjątkiem kilku przypadków, w których plan per indeks jest konieczny, optymalizator, dla każdego z indeksów z osobna, może wybrać pomiędzy planem per wiersz i per indeks na podstawie wydajności. Chociaż czynniki takie jak struktura i rozmiar tabeli oraz inne operacje wykonywane przez polecenie aktualizacji również są brane pod uwagę, wybór między planami per indeks i per wiersz będzie przede wszystkim uzależniony od liczby aktualizowanych rekordów. Jeżeli aktualizowana jest niewielka liczba rekordów, optymalizator wybierze raczej plan per wiersz, natomiast wzrost liczby aktualizowanych rekordów zwiększa prawdopodobieństwo wyboru

planu per indeks, ponieważ jest on lepiej skalowalny. Wadą podejścia per wiersz jest to, że silnik przechowywania aktualizuje indeks nieklastrowy za pomocą klucza klastrowego, co nie jest wydajne, jeżeli trzeba zaktualizować dużą liczbę rekordów.

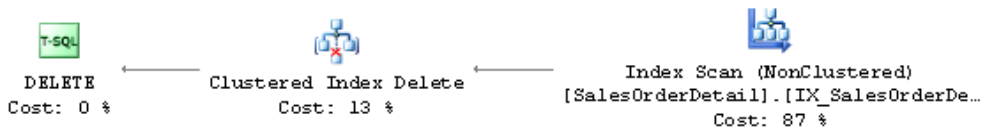


UWAGA

Dwa zapytania z tego podrozdziału aktualizują dane w bazie AdventureWorks2012, więc jeżeli nie chcesz aktualizować danych, być może powinieneś zażądać planu szacowanego. Można również wykorzystać polecenia `BEGIN TRANSACTION` i `ROLLBACK TRANSACTION` do wycofania transakcji. Alternatywnie możesz wykonać kopię zapasową bazy przed uruchomieniem tych zapytań, dzięki czemu po zakończeniu testów będziesz mógł przywrócić oryginalne dane.

Poniższe zapytanie stworzy plan per wiersz, przedstawiony na rysunku 4.36 (z powodu wykonania istniejącego wyzwalacza na planie mogą być widoczne dodatkowe zapytania):

```
DELETE FROM Sales.SalesOrderDetail
WHERE SalesOrderDetailID = 61130
```



Rysunek 4.36. Plan per wiersz

Oprócz aktualizacji indeksu klastrowego operacja usuwania zaktualizuje również dwa istniejące indeksy nieklastrowe, `IX_SalesOrderDetail_ProductID` i `AK_SalesOrderDetail_rowguid`, co można zobaczyć we właściwościach operatora *Clustered Index Delete* (zobacz rysunek 4.37).

Aby zobaczyć plan per indeks, potrzebna jest nam tabela o dużej liczbie wierszy, będziemy więc używać tabeli `dbo.SalesOrderDetail`, którą stworzyliśmy w podrozdziale „Działania równoległe”. Dodajmy do tabeli dwa indeksy nieklastrowe:

```
CREATE NONCLUSTERED INDEX AK_SalesOrderDetail_rowguid
ON dbo.SalesOrderDetail (rowguid)
CREATE NONCLUSTERED INDEX IX_SalesOrderDetail_ProductID
ON dbo.SalesOrderDetail (ProductID)
```

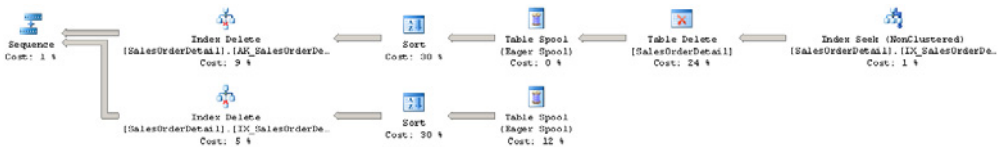
Kiedy aktualizowana jest duża liczba rekordów, optymalizator może wybrać plan per indeks, co zademonstruje poniższe zapytanie, tworząc plan per indeks przedstawiony na rysunku 4.38.

```
DELETE FROM dbo.SalesOrderDetail WHERE ProductID < 953
```

W tym planie tabela bazowa aktualizowana jest za pomocą operatora *Table Delete* (usunięcie z tabeli), natomiast operator *Table Spool* (tabela buforująca) jest wykorzy-

Clustered Index Delete	
Delete rows from a clustered index.	
Physical Operation	Clustered Index Delete
Logical Operation	Delete
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	1
Actual Number of Batches	0
Estimated I/O Cost	0.03
Estimated Operator Cost	0.05184 (13%)
Estimated CPU Cost	0.000003
Estimated Subtree Cost	0.390793
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	1.01239
Estimated Row Size	9 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	0
Object	
[AdventureWorks2012].[Sales].[SalesOrderDetail], [PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID], [AdventureWorks2012].[Sales].[SalesOrderDetail], [AK_SalesOrderDetail_rowguid], [AdventureWorks2012].[Sales]. [SalesOrderDetail].[IX_SalesOrderDetail_ProductID]	

Rysunek 4.37. Właściwości operatora Clustered Index Delete



Rysunek 4.38. Plan per indeks

stwymany do odczytywania danych wartości klucza indeksów do aktualizacji, a operator *Sort* sortuje dane w kolejności zgodnej z indeksem. Ponadto operator *Index Delete* (usuwanie z indeksu) aktualizuje konkretny indeks nieklastrowy w ramach jednej operacji (jego nazwę możesz znaleźć we właściwościach każdego z operatorów). Chociaż operator *Table Spool* widnieje na planie dwukrotnie, jest to tak naprawdę ten sam operator wykorzystany powtórnie. Operator *Sequence* (sekwencja) daje pewność, że operacje *Index Delete* wykonywane są sekwencyjnie, od góry do dołu.

Możesz teraz usunąć tabele stworzone na potrzeby ćwiczeń:

```
DROP TABLE dbo.SalesOrderDetail
DROP TABLE dbo.SalesOrderDetail2
```

Na potrzeby testów mógłbyś użyć nieudokumentowanej i niewspieranej flagi 8790, aby wymusić wykorzystanie planu per indeks dla zapytania, gdzie liczba rekordów nie jest wystarczająco duża, aby plan został wykorzystany bez tej flagi. Na przykład poniższe zapytanie dla tabeli `Sales.SalesOrderDetail` wymaga tej flagi; w przeciwnym wypadku zwrócono zostanie plan per wiersz:

```
DELETE FROM Sales.SalesOrderDetail
WHERE SalesOrderDetailID < 43740
OPTION (QUERYTRACEON 8790)
```

Podsumowując: pamiętaj, że poza kilkoma przypadkami, w których plany per indeks są obowiązkowe, optymalizator może wybierać między planem per wiersz i per indeks dla poszczególnych indeksów, jest więc możliwe, aby oba rodzaje wystąpiły w jednym planie wykonania.

Zabezpieczenie przed problemem Halloween

Zabezpieczenie to odnosi się do problemu Halloween, który pojawia się w przypadku pewnych operacji aktualizacji i został zidentyfikowany ponad 30 lat temu przez badaczy pracujących nad projektem System R w centrum badań Almaden Research Center firmy IBM. Podczas testowania optymalizatora zapytań uruchomili oni zapytanie aktualizujące kolumnę *Salary* tabeli *Employee*. Zapytanie miało dać 10% podwyżki wszystkim pracownikom z pensją niższą niż 25 000 dolarów, jednak po zakończeniu zapytania żaden pracownik nie miał pensji niższej niż 25 000 dolarów. Zauważyli, że optymalizator zapytań wybrał indeks na kolumnie *Salary* i aktualizował niektóre rekordy kilkakrotnie, aż osiągnęły wartość 25 000 dolarów. Ponieważ do skanowania rekordów wykorzystany został indeks na kolumnie *Salary*, kiedy kolumna *Salary* była aktualizowana, niektóre rekordy były przesuwane w indeksie, a potem znów skanowane i w ten sposób były aktualizowane więcej niż raz. Problem został nazwany problemem Halloween, gdyż został odkryty w święto Halloween, prawdopodobnie w roku 1976 lub 1977.

Jak wspomniałem na początku tego podrozdziału, operacje aktualizacji mają sekcję odczytu, a po niej sekcję zapisu i jest to ważne rozróżnienie, o którym powinniśmy pamiętać. Aby uniknąć problemu Halloween, etapy odczytu i zapisu muszą być całkowicie odseparowane; etap odczytu musi być całkowicie zakończony, zanim rozpocznie się etap zapisu. Pokażę, jak SQL Server unika problemu Halloween, w następnym przykładzie. Uruchom poniższe polecenie, aby stworzyć nową tabelę:

```
SELECT * INTO dbo.Product FROM Production.Product
```

Uruchom poniższe polecenie UPDATE, które stworzy plan wykonania przedstawiony na rysunku 4.39.

```
UPDATE dbo.Product SET ListPrice = ListPrice * 1.2
```

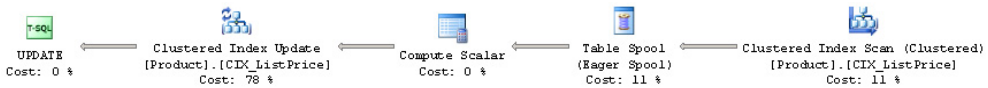


Rysunek 4.39. Aktualizacja bez zabezpieczenia przed problemem Halloween

W tym przypadku nie jest konieczne żadne zabezpieczenie przed problemem Halloween, ponieważ polecenie aktualizuje kolumnę *ListPrice*, która nie jest częścią indeksu, więc aktualizacja danych nie przemieszcza żadnych rekordów. Teraz, aby zademonstrować problem, stwórzmy indeks klastrowy na kolumnie *ListPrice*:

```
CREATE CLUSTERED INDEX CIX_ListPrice ON dbo.Product(ListPrice)
```

Uruchom jeszcze raz poprzednie polecenie UPDATE. Zapytanie pokaże podobny plan, ale tym razem będzie on zawierał również operator *Table Spool*, który jest operatorem blokującym, oddzielającym sekcję odczytu od sekcji zapisu. Operator blokujący musi odczytać wszystkie relewantne wiersze, zanim przekaże do następnego operatora jakiegokolwiek wiersze wynikowe. W tym przykładzie operator *Table Spool* oddziela od siebie operatory *Clustered Index Scan* i *Clustered Index Update* (zobacz rysunek 4.40).



Rysunek 4.40. Aktualizacja z zabezpieczeniem przed problemem Halloween

Operator buforujący skanuje pierwotne dane i przed aktualizacją zapisuje ich kopię w ukrytej tabeli buforującej w bazie tempdb. Operator *Table Spool* jest wykorzystywany do zabezpieczania przed problemem Halloween, ponieważ jest tani. Jeżeli jednak plan zawiera już inny operator, który może zostać użyty, np. *Sort*, wtedy operator *Table Spool* nie będzie potrzebny, a operator *Sort* będzie spełniał funkcję blokującą.

Możesz teraz usunąć tabelę, którą stworzyłeś:

```
DROP TABLE dbo.Product
```

Podsumowanie

W tym rozdziale opisałem system wykonywania jako kolekcję operatorów fizycznych, które definiują również wybory dostępne dla optymalizatora zapytań podczas budowania planów. Przedstawiłem niektóre z najbardziej powszechnych operatorów, opisałem ich algorytmy, relatywne koszty i scenariusze, w których optymalizator może z nich skorzystać. W szczególności przyjrzelśmy się operatorom dostępu do danych, agregacji, złączeń, działań równoległych i aktualizacji.

Omówiłem także koncepcje sortowania i haszowania jako mechanizmy wykorzystywane przez silnik wykonywania do dopasowywania i przetwarzania danych. Omówione operacje dostępu do danych to skanowanie tabel i indeksów, przeszukiwanie indeksów i operacje wyszukiwania zaznaczeń. Opisałem algorytmy agregacji, takie jak *Stream Aggregate* i *Hash Aggregate*, wraz z algorytmami złączeń, takimi jak *Nested Loops Join*, *Merge Join* i *Hash Join*. Zaprezentowałem też wprowadzenie do działań równoległych, a rozdział został zamknięty omówieniem aktualizacji danych, planów per wiersz i per

indeks oraz ochrony przed problemem Halloween. W tym miejscu chciałbym wyrazić słowa uznania dla Craiga Freedmana z zespołu SQL Servera, ponieważ on pierwszy udokumentował większość naszej współczesnej wiedzy o współbieżnym wykonywaniu zapytań.

Zrozumienie, jak działają te operacje oraz jaki jest ich prawdopodobny koszt, zapewni Ci znacznie lepsze rozeznanie na temat tego, co właściwie dzieje się „pod maską”, kiedy będziesz badać, jak wykonywane są Twoje zapytania. To z kolei pomoże Ci znaleźć potencjalne problemy w Twoich planach wykonania i zorientować się, kiedy odwołać się do technik, które opisuję w dalszej części książki.

Rozdział 5

Indeksy

W tym rozdziale:

- ▶ Wprowadzenie
- ▶ Tworzenie indeksów
- ▶ Operacje na indeksach
- ▶ Database Engine Tuning Advisor
- ▶ Brakujące indeksy
- ▶ Fragmentacja indeksów
- ▶ Nieużywane indeksy
- ▶ Podsumowanie



Indeksowanie to jedna z najważniejszych technik wykorzystywanych w dopracowywaniu zapytań i optymalizacji. Korzystając z odpowiednich indeksów, SQL Server może przyspieszyć zapytania i bardzo poprawić wydajność aplikacji. W tym rozdziale opisuję indeksy, pokazuję, w jaki sposób SQL Server z nich korzysta, jak możesz stworzyć lepsze indeksy oraz jak możesz zweryfikować swoje plany wykonania, aby się upewnić, czy te indeksy są poprawnie używane. W SQL Serverze istnieje kilka typów indeksów. W tym rozdziale skupię się na indeksach klastrowych i nieklastrowych i omówię indeksy pokrywające i indeksy filtrowane oraz fragmentację indeksów, a także wyjaśnię, jak wybrać klucz klastrowy indeksu. W rozdziale 9. natomiast szczegółowo omawiam indeksy magazynu kolumn. Inne typy indeksów, takie jak XML, Spatial (przestrzenne) i pełnotekstowe, nie należą do zakresu tematycznego tej książki.

Niniejszy rozdział zawiera również omówienie programu Database Engine Tuning Advisor (doradca optymalizacji bazy danych) i funkcjonalności Missing Indexes (brakujące indeksy). W podrozdziałach na ich temat pokażę, jak korzystać z optymalizatora zapytań w celu uzyskania informacji o tym, jak dostosować indeksy. Należy jednak podkreślić, że niezależnie od tego, jakich rekomendacji udzielią te narzędzia, w ostatecznym rozrachunku to administrator bazy danych lub programista musi wykonać swoją własną analizę indeksów, dokładnie przetestować te rekomendacje, a następnie zdecydować, które z nich zaimplementować.

Omówię także widok DMV `sys.dm_db_index_usage_stats` jako narzędzie pozwalające identyfikować istniejące indeksy, które być może nie są wykorzystywane przez zapytania. Indeksy, które nie są używane, nie są przydatne, będą natomiast zajmować miejsce na dysku i spowalniać operacje aktualizacji, należy więc rozważyć ich usunięcie.

Wprowadzenie

Jak wspomniałem w rozdziale 4., SQL Server może wykorzystywać indeksy w operacjach wyszukiwania i skanowania. Indeksy mogą zostać użyte do przyspieszenia wykonywania zapytania poprzez szybkie wyszukanie rekordów bez konieczności skanowania tabel, poprzez dostarczenie wszystkich żądanych w zapytaniu kolumn bez uzyskiwania dostępu do tabel (czyli pokrycie indeksem, do którego wróć za moment) lub poprzez zapewnienie posortowanych danych, co będzie przydatne w klauzulach `GROUP BY`, `DISTINCT` i `ORDER BY`.

Zadaniem optymalizatora danych jest określenie, czy indeks może być wykorzystany przy danym zapytaniu. Jest to porównanie między kluczem indeksu a stałą lub zmienną. Ponadto optymalizator musi określić, czy indeks pokrywa zapytanie — czyli czy zawiera wszystkie kolumny żądane w zapytaniu (jeżeli tak, nazywany jest indeksem pokrywającym). Musi to potwierdzić, ponieważ zazwyczaj indeks nieklastrowy zawiera tylko podzbiór kolumn tabeli.

SQL Server może również rozważyć wykorzystanie więcej niż jednego indeksu i złączenie ich w celu pokrycia wszystkich kolumn wymaganych w zapytaniu. Operacja ta jest nazywana krzyżowaniem indeksów. Jeżeli niemożliwe jest pokrycie wszystkich niezbędnych kolumn, SQL Server będzie musiał uzyskać dostęp do tabeli bazowej, która może być indeksem klastrowym lub stertą, aby pobrać pozostałe kolumny. Operacja ta nazywa się wyszukiwaniem zaznaczeń (może to być operacja *Key Lookup* lub *RID Lookup*, zgodnie z wyjaśnieniami z rozdziału 4.). Ponieważ jednak wyszukiwanie zaznaczeń wymaga operacji I/O, które są bardzo kosztowne, jego wykorzystanie może być efektywne tylko dla relatywnie małej liczby rekordów.

Pamiętaj także, że chociaż wykorzystany może być jeden lub kilka indeksów, nie znaczy to, iż zostaną one wybrane w planie wykonania — jest to zawsze decyzja podejmowana w oparciu o koszt. Dlatego po stworzeniu indeksu upewnij się, czy indeks będzie używany w planie i, oczywiście, czy Twoje zapytanie działa lepiej, co jest prawdopodobnie głównym powodem, dla którego zdecydowałeś się założyć indeks. Indeks, który nie jest wykorzystywany przez żadne zapytanie, będzie tylko zajmował cenne miejsce i może negatywnie wpłynąć na wydajność operacji aktualizacji. Możliwe jest również, że indeks, który był użyteczny w momencie jego tworzenia, nie jest już wykorzystywany przez żadne zapytanie.

Tworzenie indeksów

Zacznijmy tę sekcję od podsumowania podstawowej terminologii stosowanej podczas pracy z indeksami. Niektórych z tych pojęć użyłem już w poprzednich rozdziałach tej książki.

- ▶ **Sterta** — jest to struktura danych, w której wiersze przechowywane są bez określonego porządku. Innymi słowy: jest to tabela bez indeksu klastrowego.
- ▶ **Indeks klastrowy** — w SQL Serverze cała tabela może być logicznie posortowana na podstawie konkretnego klucza, który na poziomie liścia indeksu zawiera właściwe wiersze z danymi. Dlatego dla tabeli może istnieć tylko jeden indeks klastrowy. Strony danych na poziomie liścia są zorganizowane w podwójnie połączoną listę (każda strona ma wskaźnik do poprzedniej i kolejnej strony). Zarówno indeksy klastrowe, jak i indeksy nieklastrowe są przechowywane w postaci B-drzewa.
- ▶ **Indeks nieklastrowy** — zawiera wartości klucza indeksu i wskaźnik na dane w tabeli bazowej. Indeksy nieklastrowe mogą być tworzone zarówno dla stert, jak i dla indeksów klastrowych. Każda tabela może mieć do 999 indeksów nieklastrowych, zazwyczaj jednak należy się starać, aby liczba indeksów była jak najmniejsza. Indeks nieklastrowy może opcjonalnie zawierać kolumny niezwiązane z kluczem, jeżeli skorzystamy z klauzuli `INCLUDE`, która jest szczególnie użyteczna, kiedy chcemy pokryć zapytanie.

- **Indeks unikalny** — nie pozwala (jak sugeruje jego nazwa), aby dwa wiersze miały taką samą wartość klucza. Tabela może mieć jeden lub więcej unikalnych indeksów, nie jest to jednak powszechne. Domyślnie indeksy unikalne są tworzone jako indeksy nieklastrowe.
- **Klucz główny** — jest to klucz, który unikalnie identyfikuje każdy rekord w tabeli i tworzy indeks unikalny, domyślnie będący również indeksem klastrowym. Oprócz tego, że klucz główny musi być unikalny, kolumny klucza muszą być zdefiniowane jako NOT NULL. Dla tabeli może być zdefiniowany tylko jeden klucz główny.

Chociaż stworzenie klucza głównego jest proste, nie każdy wie, że kiedy klucz główny jest tworzony; domyślnie jest tworzony za pomocą indeksu klastrowego. Może tak być, gdy na przykład tworzymy klucz za pomocą kreatora tabel programu SQL Server Management Studio (dostęp do kreatora tabel można uzyskać, klikając prawym przyciskiem myszy węzeł *Tables*, a następnie klikając pozycję *New Table...*) lub za pomocą poleceń CREATE TABLE i ALTER TABLE, zgodnie z przykładem przedstawionym poniżej. Jeżeli do stworzenia klucza zastosujesz poniższy kod, w którym nie zostały użyte słowa kluczowe CLUSTERED lub NONCLUSTERED, klucz główny zostanie stworzony za pomocą indeksu klastrowego:

```
CREATE TABLE table1 (
    col1 int NOT NULL,
    col2 nchar(10) NULL,
    CONSTRAINT PK_table1 PRIMARY KEY(col1)
)
```

Lub:

```
CREATE TABLE table1
(
    col1 int NOT NULL,
    col2 nchar(10) NULL
)
GO
ALTER TABLE table1 ADD CONSTRAINT
    PK_table1 PRIMARY KEY
(
    col1
)
```

Kod wygenerowany przez kreator tabel jawnie zażąda wykorzystania indeksu klastrowego dla klucza głównego (zazwyczaj jednak nie zobaczysz takiego kodu):

```
ALTER TABLE table1 ADD CONSTRAINT
    PK_table1 PRIMARY KEY CLUSTERED
(
    col1
)
```

Tworzenie indeksu klastrowego wraz z kluczem głównym może mieć konsekwencje dla wydajności, co pokażę w dalszej części tego rozdziału, należy więc zrozumieć, że

jest to zachowanie domyślne. Oczywiście możliwe jest również stworzenie klucza głównego będącego indeksem nieklastrowym, trzeba jednak to jawnie określić. W poprzednim zapytaniu zmienimy słowo kluczowe `CLUSTERED` na `NONCLUSTERED`, aby powstał indeks nieklastrowy:

```
ALTER TABLE table1 ADD CONSTRAINT
    PK_table1 PRIMARY KEY NONCLUSTERED
    (
        col1
    )
```

Po wykonaniu poprzedniego kodu `PK_table1` zostanie stworzony jako unikalny indeks nieklastrowy.

Chociaż poprzedni kod tworzył indeks jako część definicji ograniczenia (w tym przypadku klucza głównego), najprawdopodobniej do tworzenia indeksów będziesz wykorzystywać polecenie `CREATE INDEX`. Poniżej znajduje się uproszczona wersja polecenia `CREATE INDEX`:

```
CREATE [UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX nazwa_indeksu
    ON <obiekt> ( kolumna [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( nazwa_kolumny [ ,...n ] ) ]
    [ WHERE <predykat_filtra> ]
    [ WITH ( <opcja_indeksu_relacyjnego > [ ,...n ] ) ]
```

Klauzula `UNIQUE` tworzy indeks unikalny, w którym dwa wiersze nie mogą mieć tej samej wartości klucza indeksu. Słowo kluczowe `CLUSTERED` i `NONCLUSTERED` definiuje odpowiednio indeks klastrowy lub nieklastrowy. Klauzula `INCLUDE` pozwala wyszczególnić kolumny nienależące do klucza, które zostaną dodane na poziomie liścia do indeksu nieklastrowego. Klauzula `WHERE <predykat_filtra>` pozwala stworzyć indeks filtrowany, który stworzy również statystyki filtrowane. Indeksy filtrowane i klauzulę `INCLUDE` omówię dokładniej w dalszej części tego podrozdziału. Klauzula `WITH <opcja_indeksu_relacyjnego>` pozwala wyspecyfikować opcje, które mają zostać wykorzystane podczas tworzenia indeksu, na przykład `FILLFACTOR`, `SORT_IN_TEMPDB`, `DROP_EXISTING` czy `ONLINE`.

Ponadto polecenie `ALTER INDEX` może zostać użyte do modyfikowania indeksów i wykonywania operacji takich jak wyłączenie, przebudowa i reorganizacja indeksów. Polecenie `DROP INDEX` pozwoli usunąć z bazy dany indeks. Wykorzystanie `DROP INDEX` dla indeksu nieklastrowego spowoduje usunięcie jego stron z bazy danych. Usunięcie indeksu klastrowego nie spowoduje usunięcia danych indeksu, ale dane będą przechowywane na stercie.

Wykonajmy szybkie ćwiczenie, aby przetestować niektóre z koncepcji T-SQL wspomniane w tym podrozdziale. Stwórz nową tabelę, uruchamiając poniższe polecenie:

```
SELECT * INTO dbo.SalesOrderDetail
FROM Sales.SalesOrderDetail
```

Skorzystajmy z katalogu `sys.indexes`, aby przejrzeć właściwości tabeli:

```
SELECT * FROM sys.indexes
WHERE object_id = OBJECT_ID('dbo.SalesOrderDetail')
```

Jak pokazują poniższe wyniki (nie wszystkie kolumny są widoczne), zgodnie z opisem w kolumnach *type* i *type_desc* zostanie stworzona sterta. Sterta zawsze ma indeks *index_id* równy 0.

object_id	name	index_id	type	type_desc	is_unique
1287675635	NULL	0	0	HEAP	0

Stwórzmy indeks nieklastrowy:

```
CREATE INDEX IX_ProductID ON dbo.SalesOrderDetail(ProductID)
```

sys.indexes pokaże poniższe dane, w których możesz zobaczyć, że oprócz sterty mamy teraz indeks nieklastrowy z indeksem *index_id* równym 2. Indeksy nieklastrowe mogą przyjmować indeksy od 2 do 250 oraz od 256 do 1005. Zakres ten pokrywa może 999 indeksów. Wartości od 251 do 255 są zarezerwowane.

object_id	name	index_id	type	type_desc	is_unique
1287675635	NULL	0	0	HEAP	0
1287675635	IX_ProductID	2	2	NONCLUSTERED	0

Teraz stwórz indeks klastrowy:

```
CREATE CLUSTERED INDEX IX_SalesOrderID_SalesOrderDetailID
ON dbo.SalesOrderDetail(SalesOrderID, SalesOrderDetailID)
```

Zauważ, że zamiast sterty mamy teraz indeks klastrowy i wartość *index_id* równą 1. Indeks klastrowy zawsze ma indeks równy 1. Wewnętrznie indeks nieklastrowy został przebudowany, aby jako wskaźnika używać teraz klucza klastrowego zamiast identyfikatora wiersza (RID).

object_id	name	index_id	type	type_desc	is_unique
1287675635	IX_SalesOrderID_SalesOrderDetailID	1	1	CLUSTERED	0
1287675635	IX_ProductID	2	2	NONCLUSTERED	0

Usunięcie indeksu nieklastrowego całkowicie usunie strony indeksu, zostanie tylko indeks klastrowy:

```
DROP INDEX dbo.SalesOrderDetail.IX_ProductID
```

object_id	name	index_id	type	type_desc	is_unique
1287675635	IX_SalesOrderID_SalesOrderDetailID	1	1	CLUSTERED	0

Zauważ jednak, że usunięcie indeksu klastrowego, który zastępuje całą tabelę, nie spowoduje usunięcia samych danych, lecz po prostu zmieni strukturę w stertę:

```
DROP INDEX dbo.SalesOrderDetail.IX_SalesOrderID_SalesOrderDetailID
```

object_id	name	index_id	type	type_desc	is_unique
1287675635	NULL	0	0	HEAP	0

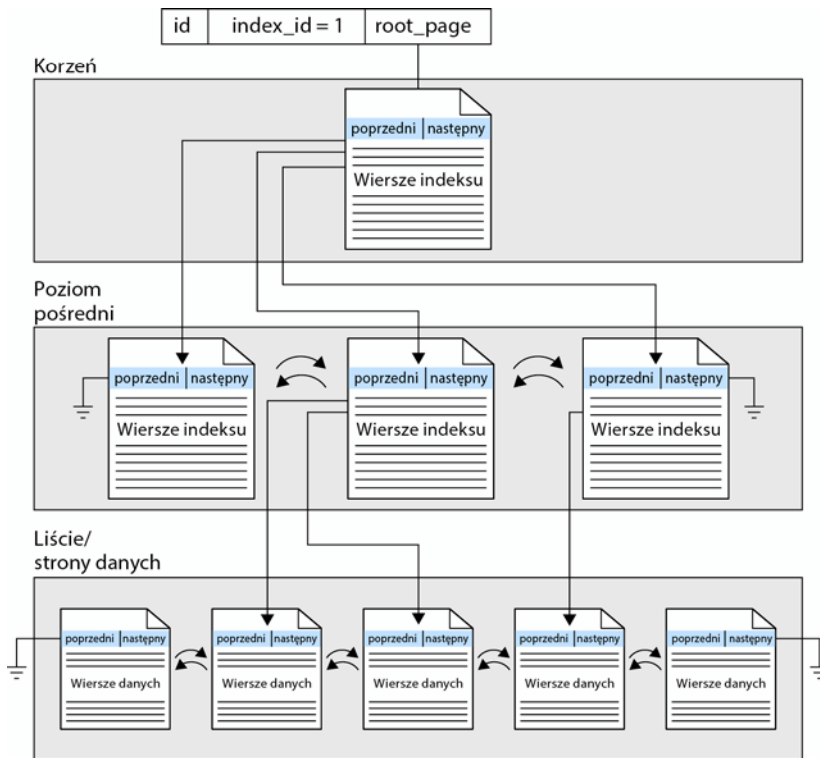
Więcej informacji na temat poleceń `CREATE INDEX`, `ALTER INDEX` i `DROP INDEX` znajdziesz w dokumentacji Books Online.

Jak widać na rysunku 5.1, indeks klastrowy jest zorganizowany w B-drzewo, które składa się z korzenia (najwyższy węzeł B-drzewa), liści (najniższe węzły, które zawierają strony danych) i poziomów pośrednich (węzły pomiędzy korzeniem a liśćmi). Aby znaleźć konkretny wiersz w klastrowym B-drzewie, SQL Server wykorzystuje korzeń i węzły pośrednie jako trasę do liścia, ponieważ korzeń i węzły pośrednie zawierają strony indeksu oraz wskaźnik na stronę poziomu pośredniego lub na stronę liścia. W przykładzie z rysunku 5.1, w którym znajduje się tylko jeden poziom pośredni, aby znaleźć konkretny wiersz, SQL Server musi odczytać trzy strony. Tabela z dużą liczbą rekordów może mieć więcej niż jeden poziom pośredni, wtedy SQL Server będzie musiał odczytać cztery lub więcej stron. Operację tę wykonuje operator *Index Seek*, szczególnie efektywny, gdy wymagany jest tylko jeden wiersz lub gdy do wykonania zapytania może zostać wykorzystane skanowanie częściowe.

Dla wielu rekordów ta operacja może być jednak bardzo kosztowna, ponieważ dla każdego rekordu konieczne będzie uzyskanie dostępu do trzech stron. Problem ten napotykamy zazwyczaj, jeżeli mamy indeks nieklastrowy, który nie pokrywa zapytania i niezbędne jest zwrócenie się do indeksu klastrowego po brakujące kolumny. W takim przypadku SQL Server musi nawigować zarówno po B-drzewie indeksu nieklastrowego, jak i indeksu klastrowego. Optymalizator przypisuje takim operacjom wysoki koszt i to dlatego czasami, kiedy zapytanie wymaga dużej liczby kolumn, zamiast tej operacji SQL Server wykonuje skanowanie indeksu klastrowego. Więcej szczegółów na temat tych operacji znajdziesz w rozdziale 4.

Indeksy klastrowe a sterty

Jedną z decyzji, jakie musisz podjąć, tworząc tabelę, jest to, czy skorzystać z indeksu klastrowego czy ze sterty. Jest to temat częstych debat w społeczności SQL Servera i nie ma jednej właściwej odpowiedzi. Chociaż najlepsze rozwiązanie może być zależne od



Rysunek 5.1. Struktura indeksu klastrowego

Twojej definicji tabeli i obciążenia, zalecane jest tworzenie tabel z indeksem klastrowym, a w tym podrozdziale wyjaśnię dlaczego. Zaczniemy od podsumowania zalet i wad organizacji tabel w indeksach klastrowych i stertach. Oto kilka dobrych powodów na pozostawienie tabeli w postaci sterty:

- *Sterta jest bardzo małą tabelą.* Chociaż indeks klastrowy może dobrze działać również dla małej tabeli.
- *Identyfikator RID jest mniejszy niż kandydat na klucz klastrowy.* Jak pokazałem w rozdziale 4., poszczególne wiersze na stercie identyfikowane są za pomocą identyfikatora wiersza (RID), który pozwala łatwo zlokalizować wiersz i zawiera informacje takie jak plik bazy danych, strona i numery slotów. RID ma wielkość 8 bajtów i w wielu przypadkach może być mniejszy niż klucz indeksu klastrowego. Ponieważ każdy wiersz w każdym indeksie nieklastrowym zawiera RID lub klucz indeksu klastrowego wskazujący na powiązany wiersz, mniejszy rozmiar może bardzo pozytywnie wpłynąć na wykorzystanie zasobów.

Zdecydowanie należy wykorzystać indeksy klastrowe w poniższych przypadkach:

- ▶ *Musisz często zwracać posortowane dane lub odpytować zakresy danych.* W takim przypadku powinieneś stworzyć indeks klastrowy na kolumnie, po której chciałbyś sortować. Możesz potrzebować, aby cała tabela lub tylko zakres danych był w odpowiedniej kolejności. Przykłady tej ostatniej operacji, nazywanej częściowym skanowaniem sortowanym, pokazałem w rozdziale 4.
- ▶ *Musisz często zwracać zgrupowane dane.* W tym przypadku powinieneś stworzyć indeks klastrowy z kluczem na kolumnie wykorzystywanej w klauzuli GROUP BY. Jak wiesz z rozdziału 4., aby wykonać operację agregacji, SQL Server wymaga posortowanych danych, a jeżeli dane nie są posortowane, będzie konieczne dodanie kosztownej operacji sortowania.

W artykule *SQL Server Best Practices* (najlepsze praktyki w SQL Serverze), dostępnym pod adresem <https://technet.microsoft.com/en-us/library/cc917672.aspx>, autorzy wykonali serię testów, aby zobrazować różnice w wydajności przy wykorzystaniu stert i indeksów klastrowych. Chociaż testy te nie muszą przypominać Twojej struktury, obciążenia i zastosowania, możesz wykorzystać je jako wskazówki do oszacowania wpływu zastosowanej w Twoim przypadku metody.

W teście jako klucz klastrowy został użyty indeks klastrowy z trzema kolumnami, który został porównany do sterty z jednym indeksem nieklastrowym zdefiniowanym z wykorzystaniem tych samych trzech kolumn (nie pojawiły się indeksy nieklastrowe). Więcej szczegółów na temat testów, włączając w to scenariusze testowe, znajdziesz w artykule. Wyniki sześciu przeprowadzonych testów są następujące:

- ▶ Wydajność operacji INSERT na tabeli z indeksem klastrowym była o około 3% lepsza niż ta sama operacja dla sterty z założonym indeksem nieklastrowym. Mimo tego, że wstawianie danych na stertę miało o 62,8% mniej podziałów stron na sekundę, zapis w indeksie klastrowym wymagał tylko jednej operacji zapisu, podczas gdy wstawianie na stertę wymagało dwóch — jednej przy zapisie na stertę, drugiej przy aktualizacji indeksu nieklastrowego.
- ▶ Wydajność operacji UPDATE na kolumnie bez indeksu w indeksie klastrowym była o 8,2% lepsza niż ta sama operacja dla sterty z założonym indeksem nieklastrowym. Stało się tak dlatego, że aktualizacja wiersza w indeksie nieklastrowym wymagała tylko wykonania operacji *Index Seek* i następującej po niej aktualizacji, natomiast dla sterty wymagana była operacja *Index Seek* z wykorzystaniem indeksu nieklastrowego, po niej wyszukanie wiersza na sterce za pomocą identyfikatora RID i dopiero po tych operacjach aktualizacja danych.
- ▶ Wydajność operacji DELETE na indeksie klastrowym była o 18,25% lepsza niż w przypadku sterty z założonym indeksem nieklastrowym. Podobnie jak w przypadku operacji UPDATE, usunięcie rekordu z indeksu klastrowego wymagało tylko wykonania operacji *Index Seek* i następującego po niej usunięcia danych, natomiast dla sterty wymagana była operacja *Index Seek* z wykorzystaniem indeksu

nieklastrowego, po niej wyszukanie wiersza na sterce za pomocą identyfikatora RID i dopiero po tych operacjach usunięcie danych. Ponadto wiersz w indeksie nieklastrowym również musiał zostać usunięty.

- ▶ Wydajność operacji SELECT dla jednego wiersza w przypadku indeksu klastrowego była o 13,8% lepsza niż w przypadku sterty z założonym indeksem nieklastrowym. Ten test zakłada, że predykat wyszukiwania bazuje na kluczach indeksu. I znowu, znalezienie wiersza w indeksie klastrowym wymaga tylko operacji przeszukania, a kiedy wiersz zostanie znaleziony, zawiera wszystkie wymagane kolumny. W przypadku sterty wymagana jest operacja *Index Seek* na indeksie nieklastrowym, a po niej wyszukanie identyfikatora RID w celu znalezienia wiersza na sterce.
- ▶ Wydajność operacji SELECT dla zakresu wierszy w przypadku indeksu klastrowego była o 29,41% lepsza niż w przypadku sterty z założonym indeksem nieklastrowym. Zapytanie testowe wybierało 228 rekordów. Po raz kolejny operacja *Clustered Index Seek* pomogła bardzo szybko znaleźć pierwszy rekord, a ponieważ rekordy są posortowane zgodnie z kolejnością kolumn w kluczu, pozostałe rekordy będą na tych samych lub kolejnych stronach. Jak wspomniałem wcześniej, wybranie zakresu wierszy, nazywane częściowym skanowaniem sortowanym, to jeden ze scenariuszy, w którym indeks klastrowy sprawdza się znacznie lepiej niż sterta, ponieważ konieczny jest odczyt mniejszej liczby stron. Pamiętaj również, że w tym przypadku koszt wykonywania wielu wyszukań może być tak wysoki, iż optymalizator może zdecydować się na skanowanie całej sterty. Inaczej jest w przypadku indeksu klastrowego, nawet jeżeli zakres wierszy jest bardzo duży.
- ▶ Test wykorzystania dysku wykonany po operacji INSERT pokazał niewielką różnicę między indeksem klastrowym a stertą z założonym indeksem nieklastrowym. Jednakże w przypadku testu wykorzystania dysku po operacji DELETE różnica między indeksem klastrowym a stertą z założonym indeksem nieklastrowym okazała się znaczna. Indeks klastrowy skurczył się prawie o tyle, ile zajmowały usunięte dane, podczas gdy sterta skurczyła się tylko o ułamek tej wartości. Powodem tej różnicy jest to, że w indeksie klastrowym puste obszary są automatycznie dealokowane, jednak w przypadku sterty obszary te są zachowywane do ponownego wykorzystania w przyszłości. Odzyskanie tej przestrzeni w przypadku sterty wiąże się z reguły z koniecznością wykonania dodatkowych czynności, takich jak przebudowa tabeli (za pomocą polecenia ALTER TABLE REBUILD).
- ▶ W przypadku równoległych operacji INSERT test pokazał, że wraz ze wzrostem liczby procesów, które równocześnie wstawiają rekordy, czas potrzebny na każdą z operacji również wzrasta, a wzrost ten jest większy dla indeksu klastrowego niż dla sterty. Jednym z głównych powodów jest zawartość odnajdywana podczas wstawiania. Testy wykazały, że w przypadku 20 równoległych procesów czas

oczekiwania na wstawienie na stronę był o 12% dłuższy dla indeksu niż dla sterty, a w przypadku 50 równoległych procesów aż o 61%. Jednakże test pokazał także, że w przypadku 50 równoległych procesów narzut ten wynosił tylko 1,2 milisekundy i nie został uznany za znaczący.

- I wreszcie, ponieważ sterta wymagała indeksu nieklastrowego, aby udostępnić te same funkcjonalności wyszukiwania co indeks klastrowy, wykorzystane miejsce na dysku było o 35% mniejsze w przypadku indeksu klastrowego niż w przypadku tabeli zorganizowanej w stertę.

Podsumowaniem badań było stwierdzenie, że zgodnie z przeprowadzonymi testami zazwyczaj korzyści płynące z wykorzystania indeksu klastrowego przewyższają wady. Jednak chociaż indeksy klastrowe są ogólnie polecane, wydajność może być różna w zależności od schematu bazy, obciążenia i konfiguracji, warto więc uważnie przetestować obie opcje.

Klucz indeksu klastrowego

Decyzja co do tego, która kolumna (lub kolumny) będzie częścią klucza indeksu klastrowego, jest również bardzo istotna. W książce *Microsoft SQL Server 2008 Internals* (Microsoft Press, 2009) Kimberly Tripp określiła dobrą praktykę, zgodnie z którą indeksy powinny być unikalne, wąskie, stałe i wciąż rosnące. Pamiętaj jednak, że ta definicja, tak jak wszelkie inne ogólne wskazówki, nie musi się sprawdzać we wszystkich sytuacjach, dlatego powinniśmy wykonywać dokładne testy dla swojej bazy i jej obciążenia.

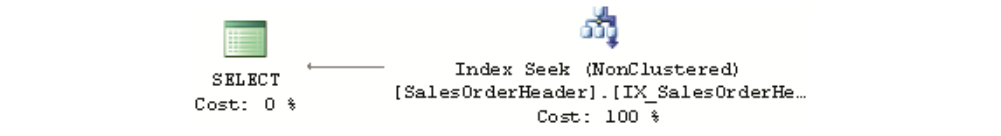
- **Unikalne.** Jeżeli indeks klastrowy nie zostanie zdefiniowany za pomocą klauzuli `UNIQUE`, SQL Server doda 4-bajtowy unikalny identyfikator dla każdego wiersza, co zwiększy rozmiar indeksu klastrowego. Dla porównania: identyfikator RID używany w indeksach nieklastrowych dla stert ma tylko 8 bajtów.
- **Wąskie.** Ponieważ każdy wiersz w każdym indeksie nieklastrowym zawiera, oprócz kolumn definiujących indeks, klucz indeksu klastrowego wskazujący na powiązany z nim wiersz tabeli bazowej, niewielki klucz może pozytywnie wpłynąć na ilość wykorzystywanych zasobów. Mały klucz wymaga mniej powierzchni dyskowej, co również pozytywnie wpłynie na wydajność. Ponownie możemy porównać klucz indeksu klastrowego z identyfikatorem RID, który ma tylko 8 bajtów.
- **Stale (niezmienne).** Aktualizacja indeksu klastrowego może mieć konsekwencje dla wydajności, na przykład mogą powstać podziały stron lub fragmentacja spowodowane relokacją wierszy w obrębie indeksu klastrowego. Dodatkowo, ponieważ każdy indeks nieklastrowy zawiera klucz indeksu klastrowego, wiersze w indeksie nieklastrowym również będą musiały zostać zaktualizowane, aby odzwierciedlić nowy klucz.

- **Wciąż rosnące.** Na indeks klastrowy rosnące wartości klucza są lepsze niż wartości bardziej losowe, jak np. nazwisko. Konieczność wstawiania wierszy w losowych miejscach może powodować podziały stron i fragmentację. Z drugiej strony, musisz wiedzieć, że w niektórych przypadkach wartości rosnące mogą także powodować problemy, ponieważ wiele procesów może pisać na ostatniej stronie tabeli.

Indeksy pokrywające

Indeks pokrywający to bardzo proste, ale też bardzo ważne narzędzie w optymalizacji zapytań — indeks powinien być w stanie rozwiązać lub zwrócić wszystkie kolumny żądane w zapytaniu bez konieczności odwoływania się do tabeli bazowej. Na przykład poniższe zapytanie jest już pokryte przez istniejący indeks, X_SalesOrderHeader_CustomerID, co można odczytać z planu zapytania przedstawionego na rysunku 5.2:

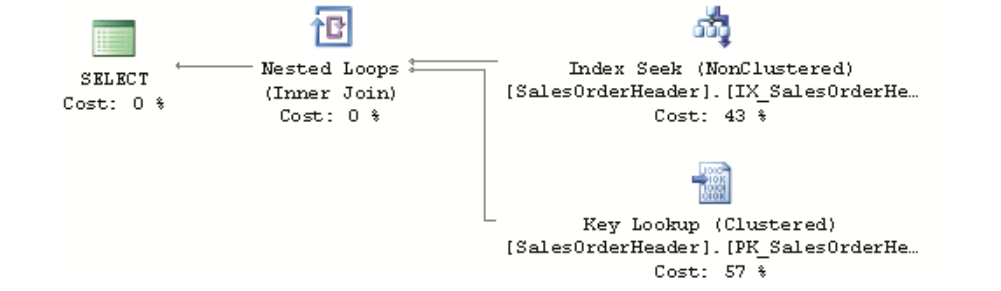
```
SELECT SalesOrderID, CustomerID FROM Sales.SalesOrderHeader
WHERE CustomerID = 16448
```



Rysunek 5.2. Indeks pokrywający

Na planie widać, że nie ma w ogóle potrzeby odwoływania się do tabeli bazowej. Jeżeli delikatnie zmienimy zapytanie, aby żądało również kolumny *SalesPersonID*, tym razem nie będzie indeksu, który pokrywałby wszystkie kolumny, więc stworzony zostanie plan pokazany na rysunku 5.3:

```
SELECT SalesOrderID, CustomerID, SalesPersonID FROM Sales.SalesOrderHeader
WHERE CustomerID = 16448
```



Rysunek 5.3. Plan z operacją Key Lookup

Plan pokazuje, że indeks IX_SalesOrderHeader_CustomerID został użyty do szybkiego odnalezienia żadanego rekordu, ale ponieważ nie zawierał kolumny *SalesPersonID*,

konieczna była również operacja wyszukiująca w indeksie klastrowym. Pamiętaj, że pokrycie zapytania nie oznacza konieczności dodania kolejnej kolumny do klucza indeksu, chyba że przy użyciu tej kolumny chcesz także wykonywać operacje wyszukiwania. Zamiast tego, w celu dodania brakującej kolumny, możesz skorzystać z klauzuli `INCLUDE` polecenia `CREATE` lub `ALTER INDEX`. W tym miejscu możesz po prostu zaktualizować istniejący indeks, aby zawierał brakującą kolumnę, jednak w naszym przykładzie stworzymy nowy indeks:

```
CREATE INDEX IX_SalesOrderHeader_CustomerID_SalesPersonID
ON Sales.SalesOrderHeader(CustomerID)
INCLUDE (SalesPersonID)
```

Jeżeli jeszcze raz uruchomisz zapytanie, optymalizator skorzysta z planu podobnego do użytego poprzednio (zobacz rysunek 5.2), zawierającego tylko operację *Index Seek*, tym razem z nowym indeksem `IX_SalesOrderHeader_CustomerID_SalesPersonID`. Zauważ jednak, że tworzenie wielu indeksów dla tabeli może również powodować problemy z wydajnością, ponieważ wszystkie indeksy muszą być aktualizowane podczas każdej operacji aktualizacji, nie wspominając już o dodatkowym miejscu na dysku potrzebnym do ich przechowywania.

Aby po sobie posprzątać, usuń tymczasowo stworzony indeks:

```
DROP INDEX Sales.SalesOrderHeader.IX_SalesOrderHeader_CustomerID_SalesPersonID
```

Indeksy filtrowane

Możesz wykorzystać indeksy filtrowane dla zapytań, w których kolumna ma tylko niewielką liczbę znaczących wartości. Na przykład, jeżeli chcesz, aby zapytanie skupiło się na pewnych konkretnych wartościach kolumny, w której większość wartości to `NULL`, i musisz odpytać bazę o wartości niebędące `NULL`. Na szczęście indeksy filtrowane mogą również dbać o unikalność filtrowanych danych. Zaletą indeksów filtrowanych jest to, że wymagają mniejszej ilości miejsca na dysku niż zwykłe indeksy, a operacje aktualizacji dla nich są szybsze. Ponadto indeks filtrowany tworzy statystyki filtrowane, które mogą być lepszej jakości niż obiekt statystyk stworzony dla całej tabeli, ponieważ histogram zostanie stworzony tylko dla konkretnego zakresu wartości. Jak pokażę w rozdziale 6., histogram może mieć maksymalnie 200 kroków, co może stanowić ograniczenie przy dużej liczbie unikalnych wartości. Aby stworzyć indeks filtrowany, musisz wyspecyfikować filtr za pomocą klauzuli `WHERE` polecenia `CREATE INDEX`.

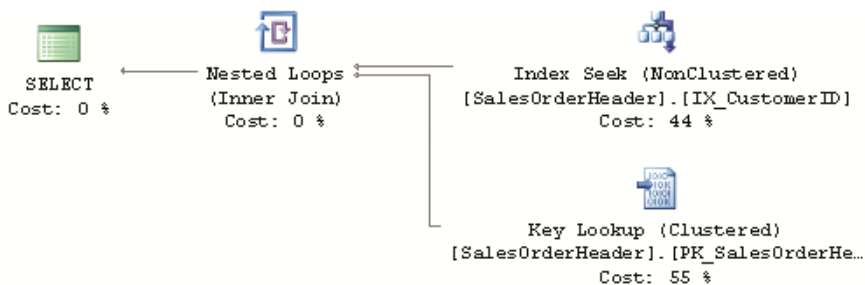
Na przykład, jeżeli spojrzysz na plan dla poniższego zapytania, zobaczysz, że wykorzystuje indeks `IX_SalesOrderHeader_CustomerID` do filtrowania predykatu `CustomerID = 13917` oraz operację *Key Lookup* do znalezienia rekordów w indeksie klastrowym i filtrowania po `TerritoryID = 4` (jak pamiętasz z rozdziału 4., w tym przypadku operacja *Key Lookup* to tak naprawdę *Clustered Index Seek*, co będziesz mógł zobaczyć, korzystając z polecenia `SET SHOWPLAN_TEXT`):

```
SELECT CustomerID, OrderDate, AccountNumber FROM Sales.SalesOrderHeader
WHERE CustomerID = 13917 AND TerritoryID = 4
```

Stwórz następujący indeks filtrowany:

```
CREATE INDEX IX_CustomerID ON Sales.SalesOrderHeader(CustomerID)
WHERE TerritoryID = 4
```

Jeżeli po raz kolejny uruchomisz poprzednie polecenie SELECT, zobaczysz podobny plan jak na rysunku 5.4, ale tym razem z wykorzystaniem przed chwilą stworzonego indeksu. Operator *Index Seek* wykonuje operację wyszukiwania na kolumnie *CustomerID*, jednak *Key Lookup* nie musi już filtrować po kolumnie *TerritoryID*, ponieważ indeks *IX_CustomerID* zawiera już przefiltrowane dane.



Rysunek 5.4. Plan wykorzystujący indeks filtrowany

Pamiętaj, że chociaż może wydawać się, iż indeks filtrowany nie daje żadnych dodatkowych korzyści związanych z wydajnością w porównaniu ze zwykłym indeksem nieklastrowym zdefiniowanym z takimi samymi właściwościami, będzie wykorzystywał mniej przestrzeni na dysku, łatwiej będzie go aktualizować i potencjalnie może udostępniać lepsze statystyki dla optymalizatora, ponieważ statystyki zawierające tylko wiersze z indeksu filtrowanego będą dokładniejsze.

Indeks filtrowany nie może być jednak wykorzystany do rozwiązania zapytania, jeżeli wartość nie jest znana, na przykład kiedy korzystamy ze zmiennych lub parametrów. Ponieważ optymalizator musi stworzyć plan, który będzie działał dla wszystkich możliwych wartości zmiennej lub parametru, indeks filtrowany nie może zostać wybrany. Jak pisałem w rozdziale 1., w takim przypadku na planie możesz zobaczyć ostrzeżenie *UnmatchedIndexes*. Korzystając z naszego aktualnego przykładu, poniższe zapytanie pokaże ostrzeżenie *UnmatchedIndexes*, wskazujące, że plan nie był w stanie skorzystać z indeksu *IX_CustomerID*, nawet jeżeli wartość dla *TerritoryID* była równa 4, czyli taka sama jak wykorzystana do filtrowania indeksu:

```
DECLARE @territory int
SET @territory = 4
SELECT CustomerID, OrderDate, AccountNumber FROM Sales.SalesOrderHeader
WHERE CustomerID = 13917 AND TerritoryID = @territory
```

Zanim przejdziemy dalej, usuń stworzony indeks:

```
DROP INDEX Sales.SalesOrderHeader.IX_CustomerID
```

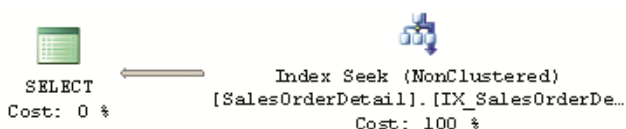
Operacje na indeksach

Aby szybko znaleźć wymagane rekordy bez konieczności skanowania indeksu lub tabeli, podczas operacji wyszukiwania SQL Server nawiguje po indeksie w formie B-drzewa. Przypomina to używanie indeksu zamieszczonego na końcu książki do szybkiego odnalezienia interesującego tematu, by nie czytać całej książki. Kiedy już znaleziony zostanie pierwszy rekord, do odnalezienia pozostałych SQL Server może skanować na poziomie liścia w przód lub w tył. W predykcji mogą zostać wykorzystane zarówno operator równości, jak i operatory nierówności: =, <, >, <=, >=, <>, !=, !<, !>, BETWEEN i IN. Na przykład poniższe predykaty mogą być dopasowane w operacji *Index Seek*, jeżeli na wyspecyfikowanej kolumnie znajduje się indeks lub istnieje indeks wielokolumnowy z tą kolumną jako wiodącą kolumną klucza indeksu:

- ▶ ProductID = 771
- ▶ UnitPrice < 3.975
- ▶ LastName = 'Kowalski'
- ▶ LastName LIKE 'Kowal%'

Przjrzyjmy się poniższemu przykładowemu zapytaniu, które wykorzystuje operator *Index Seek* i tworzy plan pokazany na rysunku 5.5:

```
SELECT ProductID, SalesOrderID, SalesOrderDetailID
FROM Sales.SalesOrderDetail
WHERE ProductID = 771
```



Rysunek 5.5. Plan wykorzystujący operator Index Seek

Tabela SalesOrderDetail ma indeks wielokolumnowy z kolumną *ProductID* jako wiodącą. Właściwości operatora *Index Seek*, które możesz zobaczyć na rysunku 5.6, zawierają poniższe predykaty dla kolumny *ProductID*, które pokazują, że SQL Server mógł efektywnie skorzystać z indeksu do przeszukania tej kolumny:

```
Seek Keys[1]: Prefix: [AdventureWorks].[Sales].[SalesOrderDetail].ProductID =
↳ Scalar Operator (CONVERT_IMPLICIT(int,[@1],0))
```

Indeks nie może być wykorzystany dla pewnych skomplikowanych wyrażeń, wyrażeń korzystających z funkcji lub ciągów znaków ze znakiem maski na początku, na przykład:

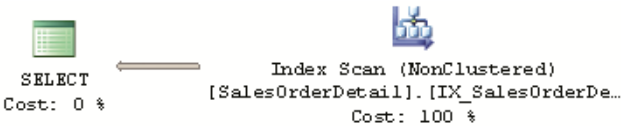
Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	241
Actual Number of Batches	0
Estimated Operator Cost	0.0035471 (100%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0004221
Estimated Subtree Cost	0.0035471
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	241
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
Object [AdventureWorks2012].[Sales].[SalesOrderDetail]. [IX_SalesOrderDetail_ProductID]	
Output List [AdventureWorks2012].[Sales]. [SalesOrderDetail].SalesOrderID, [AdventureWorks2012]. [Sales].[SalesOrderDetail].SalesOrderDetailID, [AdventureWorks2012].[Sales].[SalesOrderDetail].ProductID	
Seek Predicates Seek Keys[1]: Prefix: [AdventureWorks2012].[Sales]. [SalesOrderDetail].ProductID = Scalar Operator (CONVERT_IMPLICIT(int,[@1],0))	

Rysunek 5.6. Właściwości operatora Index Seek

- ▶ ABS(ProductID) = 771
- ▶ UnitPrice + 1 < 3.975
- ▶ LastName LIKE '%Kowalski'
- ▶ UPPER(LastName) = 'Kowalski'

Porównaj poniższe zapytanie z poprzednim przykładem; po dodaniu funkcji ABS do predykatu SQL Server nie może już korzystać z operatora *Index Seek* i zamiast tego wybiera operator *Index Scan*, zgodnie z rysunkiem 5.7:

```
SELECT ProductID, SalesOrderID, SalesOrderDetailID
FROM Sales.SalesOrderDetail
WHERE ABS(ProductID) = 771
```



Rysunek 5.7. Plan wykorzystujący operator Index Seek

Zauważ, że na rysunku 5.8 poniższy predykat jest wciąż rozwiązywany dla operatora *Index Scan*:

```
abs([AdventureWorks].[Sales].[SalesOrderDetail].[ProductID]) = CONVERT_IMPLICIT(int,[@1],0)
```

Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	241
Actual Number of Batches	0
Estimated I/O Cost	0.205347
Estimated Operator Cost	0.338953 (100%)
Estimated Subtree Cost	0.338953
Estimated CPU Cost	0.133606
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	6500.42
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0
Predicate	
abs([AdventureWorks2012].[Sales].[SalesOrderDetail].[ProductID])=CONVERT_IMPLICIT(int,[@1],0)	
Object	
[AdventureWorks2012].[Sales].[SalesOrderDetail].[IX_SalesOrderDetail_ProductID]	
Output List	
[AdventureWorks2012].[Sales].[SalesOrderDetail].[SalesOrderID], [AdventureWorks2012].[Sales].[SalesOrderDetail].[SalesOrderDetailID], [AdventureWorks2012].[Sales].[SalesOrderDetail].[ProductID]	

Rysunek 5.8. Właściwości operatora Index Scan

W przypadku indeksu wielokolumnowego SQL Server może wykorzystać indeks do wyszukiwania drugiej kolumny, tylko jeżeli dla pierwszej został wykorzystany operator równości. Dlatego w poniższych wypadkach SQL Server może skorzystać z indeksu wielokolumnowego do wyszukania obu kolumn (zakładając, że istnieje indeks zawierający obie kolumny w prezentowanej kolejności):

- ▶ ProductID = 771 AND SalesOrderID > 34000
- ▶ LastName = 'Kowalski' AND FirstName = 'Jan'

Jeżeli w pierwszej kolumnie nie został wykorzystany operator równości lub jeżeli predykat dla drugiej kolumny nie może zostać rozwiązany, na przykład w przypadku wyrażenia złożonego, SQL Server i tak może skorzystać z pierwszej kolumny indeksu wielokolumnowego, zgodnie z poniższymi przykładami:

- ▶ ProductID < 771 AND SalesOrderID = 34000
- ▶ LastName > 'Kowalski' AND FirstName = 'Jan'
- ▶ ProductID = 771 AND ABS(SalesOrderID) = 34000

SQL Server nie może jednak wykorzystać indeksu wielokolumnowego dla operatora *Index Seek* dla poniższych przykładów, ponieważ nie może wyszukiwać po pierwszej kolumnie:

- ▶ ABS(ProductID) = 771 AND SalesOrderID = 34000
- ▶ LastName LIKE '%Kowalski' AND FirstName = 'Jan'

Przyjrzyjmy się teraz poniższemu zapytaniu i właściwościom operatora *Index Seek* przedstawionym na rysunku 5.9:

```
SELECT ProductID, SalesOrderID, SalesOrderDetailID
FROM Sales.SalesOrderDetail
WHERE ProductID = 771 AND ABS(SalesOrderID) = 45233
```

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	1
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0035471 (100%)
Estimated CPU Cost	0.0004221
Estimated Subtree Cost	0.0035471
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	12.9133
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
Predicate	
abs([AdventureWorks2012].[Sales].[SalesOrderDetail].[SalesOrderID])=[@2]	
Object	
[AdventureWorks2012].[Sales].[SalesOrderDetail].[IX_SalesOrderDetail_ProductID]	
Output List	
[AdventureWorks2012].[Sales].[SalesOrderDetail].[SalesOrderID], [AdventureWorks2012].[Sales].[SalesOrderDetail].[SalesOrderDetailID], [AdventureWorks2012].[Sales].[SalesOrderDetail].[ProductID]	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2012].[Sales].[SalesOrderDetail].[ProductID] = Scalar Operator (CONVERT_IMPLICIT(int,[@1],0))	

Rysunek 5.9. Operator Index Seek pokazujący właściwości wyszukiwania i predykatów

Predykat wyszukiwania używa tylko kolumny *ProductID*:

```
Seek Keys[1]: Prefix: [AdventureWorks].[Sales].[SalesOrderDetail].ProductID =  
↳ Scalar Operator (CONVERT_IMPLICIT(int,[@1],0))
```

Dodatkowy predykat na kolumnie *SalesOrderID* jest przetwarzany jak każdy inny predykat skanowania:

```
abs([AdventureWorks].[Sales].[SalesOrderDetail].[SalesOrderID])=[@2]
```

Podsumowując: zgodnie z naszymi oczekiwaniami SQL Server był w stanie wykonać operację wyszukiwania dla kolumny *ProductID*, ale ponieważ wykorzystana została funkcja ABS, nie mógł zrobić tego samego dla kolumny *SalesOrderID*. Indeks został wykorzystany do bezpośredniego odnalezienia rekordów odpowiadających pierwszemu predykatowi, ale potem musiało zostać wykonane skanowanie sprawdzające drugi predykat.

Database Engine Tuning Advisor

Aktualnie wszyscy główni producenci baz danych dodają do swoich produktów narzędzia pomagające w tworzeniu indeksów. Kiedy jednak takie narzędzia zaczęły się pojawiać, dostępne były tylko dwa główne podejścia architektoniczne do tego, w jaki sposób powinny rekomendować indeksy. Pierwszym podejściem było zbudowanie osobnego narzędzia mającego własny model kosztów i reguły projektowania. Drugim było zbudowanie narzędzia mogącego korzystać z modelu kosztów optymalizatora zapytań.

Problemem podczas budowania osobnego narzędzia jest konieczność zduplikowania modułu kosztów. Oprócz tego posiadanie narzędzia z własnym modelem kosztów, nawet jeżeli jest lepszy od modelu kosztów optymalizatora, nie jest dobrym pomysłem, ponieważ optymalizator podejmuje decyzje w oparciu o własny model.

Drugie podejście, wykorzystujące optymalizator do pomocy w fizycznym zaprojektowaniu bazy, zostało zaproponowane w społeczności zajmującej się badaniem baz danych już w 1988 roku. Ponieważ to optymalizator wybiera indeksy do planu wykonania, sensowne jest wykorzystanie go do odnajdywania brakujących indeksów, które mogłyby pozytywnie wpłynąć na istniejące zapytania. W tym scenariuszu narzędzie do projektowania wykorzystuje optymalizator do oszacowania kosztu planów z użyciem potencjalnych indeksów. Dodatkową zaletą tego podejścia jest to, że gdy model kosztów optymalizatora ewoluuje, każde stosujące go narzędzie automatycznie korzysta ze zmienionych mechanizmów.

SQL Server był pierwszą komercyjną bazą danych zawierającą takie narzędzie — Index Tuning Wizard został wprowadzony w SQL Serverze 7.0, a potem, w wersji 2005, zastąpiono go narzędziem Database Engine Tuning Advisor (DTA). Oba wykorzystują model szacowania kosztów z optymalizatora zapytań i zostały stworzone w Microsoftzie w ramach projektu AutoAdmin, którego celem było zmniejszenie całkowitego

kosztu posiadania (TCO — *total cost of ownership*) baz danych poprzez wprowadzenia mechanizmów samooptymalizacyjnych i samozarządzających. Oprócz tworzenia indeksów DTA może pomóc w tworzeniu widoków indeksowanych i w partycjonowaniu.

Jednak tworzenie prawdziwych indeksów w DTA nie jest możliwe; narzut mógłby wpłynąć na działające zapytania i obniżyć wydajność bazy danych. Jak więc DTA szacuje koszt korzystania z indeksu, który nie istnieje? Tak naprawdę nawet podczas zwykłej optymalizacji zapytania optymalizator nie wykorzystuje indeksów doszacowania kosztu zapytania. Decyzja o tym, czy użyć indeksu czy nie, oparta jest wyłącznie o metadane i informacje statystyczne dotyczące kolumn indeksu. Same dane indeksu nie są potrzebne podczas optymalizacji, ale będą oczywiście potrzebne podczas wykonania zapytania, jeżeli indeks zostanie wybrany.

Aby więc uniknąć tworzenia prawdziwych indeksów w trakcie sesji DTA, SQL Server korzysta ze specjalnego rodzaju indeksów — indeksów hipotetycznych, które są wykorzystywane przez funkcjonalność Index Tuning Wizard (kreator dostosowywania indeksów). Jak sugeruje nazwa, indeksy hipotetyczne to nie prawdziwe indeksy; zawierają jedynie informacje statystyczne i można je tworzyć za pomocą nieudokumentowanej opcji `WITH STATISTICS_ONLY` polecenia `CREATE INDEX`. Możesz nie być w stanie zobaczyć tych indeksów w trakcie sesji DTA, ponieważ są usuwane automatycznie, kiedy przestają być potrzebne. Jednakże możesz zobaczyć polecenia `CREATE INDEX WITH STATISTICS_ONLY` i `DROP INDEX`, jeżeli uruchomisz sesję SQL Server Profiler, aby podejrzeć, co robi DTA.

Przyjrzyjmy się dokładniej niektórym z tych koncepcji. Aby rozpocząć, stwórz nową tabelę w bazie AdventureWorks:

```
SELECT * INTO dbo.SalesOrderDetail
FROM Sales.SalesOrderDetail
```

Skopiuj poniższe zapytanie i zapisz je w pliku:

```
SELECT * FROM dbo.SalesOrderDetail
WHERE ProductID = 897
```

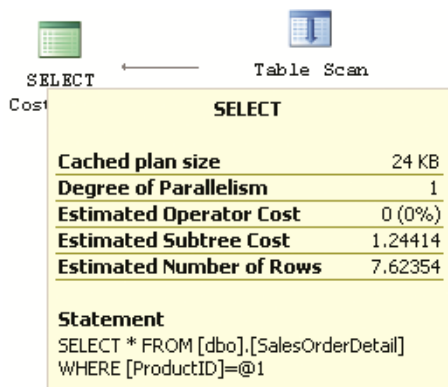
Otwórz nową sesję DTA. DTA znajdziesz w SQL Serverze 2014, w menu *Performance Tools*, pod pełną nazwą *SQL Server 2014 Database Engine Tuning Advisor*. Możesz też uruchomić sesję śledzenia w SQL Server Profiler, jeżeli chcesz obserwować, co robi DTA. W polu *Workload File* wybierz plik zawierający polecenie SQL, który stworzyłeś wcześniej, a potem wybierz bazę AdventureWorks2012 jako bazę do optymalizacji i bazę do analizy zapytań. Kliknij przycisk *Start Analysis*, a kiedy skończy się analiza, uruchom poniższe zapytania, aby sprawdzić zawartość tabeli `msdb..DTA_reports_query`:

```
SELECT * FROM msdb..DTA_reports_query
```

Uruchomienie tego zapytania spowoduje wyświetlenie poniższych wyników (wynik został wyedytowany ze względu na ilość dostępnego miejsca):

StatementString	CurrentCost	RecommendedCost
SELECT * FROM dbo.SalesOrderDetail WHERE ProductID = 897	1.24414	0.00333398

Zauważ, że zapytanie zwraca informacje takie jak analizowane zapytanie oraz aktualny i rekomendowany koszt. Aktualny koszt, wynoszący 1,24414, łatwo sprawdzić, żądając szacowanego planu zapytania (zobacz rysunek 5.10).



Rysunek 5.10. Plan pokazujący całkowity koszt

Ponieważ analiza DTA została zakończona, wymagane indeksy hipotetyczne zostały usunięte. Aby uzyskać indeksy rekomendowane przez DTA, kliknij zakładkę *Recommendations* (rekomendacje) i zajrzyj do sekcji *Index Recommendations* (rekomendowane indeksy), gdzie, klikając kolumnę *Definition* (definicja), znajdziesz kod potrzebny do stworzenia każdego z indeksów. W naszym przykładzie otrzymamy następujący kod:

```
CREATE CLUSTERED INDEX [_dta_index_SalesOrderDetail_c_5_1440724185__K5]
ON [dbo].[SalesOrderDetail]
(
    [ProductID] ASC
)WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]
```

W poniższym zapytaniu, tylko dla celów poglądowych, stworzę indeks rekomendowany przez DTA. Zamiast jednak tworzyć standardowy indeks, stworzę go jako indeks hipotetyczny, dodając klauzulę `WITH STATISTICS_ONLY`. Pamiętaj, że indeksy hipotetyczne nie mogą być wykorzystywane przez zapytania i są użyteczne tylko dla DTA.

```
CREATE CLUSTERED INDEX cix_ProductID ON dbo.SalesOrderDetail(ProductID)
WITH STATISTICS_ONLY
```

Możesz sprawdzić, czy indeks hipotetyczny został stworzony, uruchamiając poniższe zapytanie:

```
SELECT * FROM sys.indexes
WHERE object_id = OBJECT_ID('dbo.SalesOrderDetail')
AND name = 'cix_ProductID'
```

Wynik znajduje się poniżej (zauważ, że pole `is_hypothetical` pokazuje, że jest to tylko indeks hipotetyczny):

object_id	name	index_id	type	type_desc	is_hypothetical
1607676775	cix_ProductID	3	1	CLUSTERED	1

Usuń indeks hipotetyczny, uruchamiając poniższe polecenie:

```
DROP INDEX dbo.SalesOrderDetail.cix_ProductID
```

Możesz teraz zaimplementować rekomendację DTA, tworząc indeks `_dta_index_SalesOrderDetail_c_5_1440724185__K5` za pomocą wcześniej pokazanego kodu. Po zaimplementowaniu rekomendacji i ponownym uruchomieniu zapytania indeks klastrowy zostanie wykorzystany przez optymalizator zapytań. Tym razem plan pokazuje operator *Clustered Index Seek* i szacowany koszt w wysokości 0,0033652, która to wartość jest bardzo zbliżona do wartości przewidzianej przez DTA.

Optymalizacja zapytań i korzystanie z magazynu planów

Oprócz tradycyjnych opcji optymalizacji zapytań za pomocą opcji *File* lub *Table*, które pozwalają wybrać skrypt lub tabelę zawierającą zapytania T-SQL do optymalizacji, od wersji 2012 możesz również wybrać magazyn planów. W tym przypadku DTA wybierze 1000 zdarzeń z magazynu danych na podstawie całkowitego czasu wykonania (czyli kolumny `total_elapsed_time` widoku DMV `sys.dm_exec_query_stats`, zgodnie z informacjami z rozdziału 2.). Wypróbujmy przykład, a aby łatwo było zobaczyć wyniki, wyczyścimy magazyn planów i uruchomimy tylko jedno zapytanie:

```
DBCC FREEPROCCACHE
GO
SELECT SalesOrderID, OrderQty, ProductID
FROM dbo.SalesOrderDetail
WHERE CarrierTrackingNumber = 'D609-4F2A-9B'
```

Po wykonaniu zapytania plan zostanie najprawdopodobniej zachowany w magazynie planów. Otwórz nową sesję DTA. W opcji *Workload* wybierz *Plan Cache* (magazyn planów) i bazę *AdventureWorks2012*, zarówno jako bazę do optymalizacji, jak i do analizy zapytań. Kliknij przycisk *Start Analysis*. Kiedy analiza zostanie zakończona, możesz wybrać zakładkę *Recommendations* i sekcję *Index Recommendations*, która będzie zawierała poniższe rekomendacje (znajdziesz je w kolumnie *Definition*):

```
CREATE NONCLUSTERED INDEX [_dta_index_SalesOrderDetail_5_807673925__K3_1_4_5]
ON [dbo].[SalesOrderDetail]
(
    [CarrierTrackingNumber] ASC
)
INCLUDE ([SalesOrderID],
```

```
[OrderQty],
[ProductID]) WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF)
ON [PRIMARY]
```

Możesz teraz usunąć wcześniej stworzoną tabelę za pomocą poniższego zapytania:

```
DROP TABLE dbo.SalesOrderDetail
```

Rozładowanie narzutu optymalizacji na serwer testowy

Jedną z bardziej interesujących, a jednocześnie mało znanych funkcji DTA jest możliwość wykorzystania go z serwerem testowym do optymalizacji zapytań na serwerze produkcyjnym. Jak już wcześniej wspomniałem, DTA podczas wyszukiwania rekomendacji polega na optymalizatorze zapytań i możesz ustawić go tak, aby wykonywał wywołania optymalizatora na serwerze testowym bez wpływu na wydajność serwera produkcyjnego.

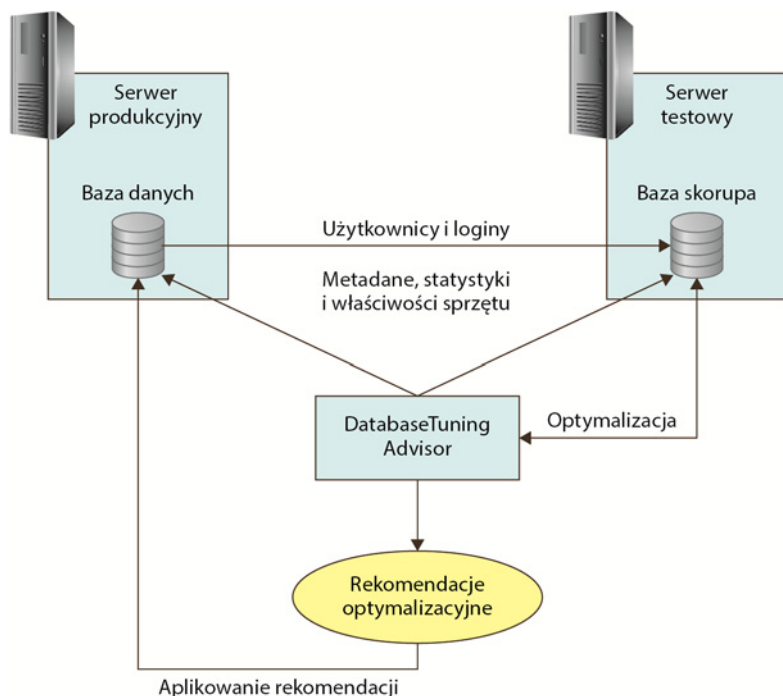
Aby lepiej zrozumieć, jak to działa, zobaczmy najpierw, jakich informacji potrzebuje optymalizator do zoptymalizowania zapytania. Najważniejsze z nich to:

- ▶ metadane bazy danych (definicje tabel i kolumn, indeksy, ograniczenia i tak dalej);
- ▶ statystyki optymalizatora (statystyki indeksów i kolumn);
- ▶ rozmiary tabel (liczba wierszy i stron);
- ▶ dostępna pamięć i liczba procesorów.

DTA może zbierać metadane i statystyki z serwera produkcyjnego i wykorzystać je do stworzenia podobnej bazy, ale bez danych, na innym serwerze. Baza taka nazywana jest skorupą. DTA może również pobrać informacje o dostępnej pamięci i liczbie procesorów na serwerze produkcyjnym za pośrednictwem rozszerzonej procedury przechowywanej `xp_msver` i wykorzystać te informacje w procesie optymalizacji. Należy pamiętać, że w procesie optymalizacji nie są wymagane żadne dane. Proces ten podsumowałem na rysunku 5.11.

Proces ten ma następujące zalety:

- ▶ Nie ma potrzeby wykonywania kosztowych operacji na serwerze produkcyjnym, które mogłyby mieć wpływ na wykorzystanie zasobów. Serwer produkcyjny jest wykorzystywany tylko do początkowych metadanych i wymaganych statystyk.
- ▶ Nie ma potrzeby kopiowania całej bazy danych na serwer testowy (jest to szczególnie ważne w przypadku dużych baz), co pozwala oszczędzić przestrzeń na dysku i czas.
- ▶ Nie jest problemem, jeżeli serwery testowe nie są tak mocne jak serwer produkcyjny, ponieważ sesja DTA weźmie pod uwagę pamięć i liczbę procesorów serwera produkcyjnego.



Rysunek 5.11. Wykorzystanie serwera testowego w pracy z DTA

Pokażę teraz przykład uruchomienia takiej sesji optymalizacyjnej. Przede wszystkim wykorzystanie serwera testowego nie jest wspierane w interfejsie graficznym, a więc konieczne jest użycie programu dta (wersja DTA dla linii komend). Konfiguracja serwera testowego wymaga również pliku XML zawierającego dane wsadowe dla dta. Na potrzeby tego przykładu wykorzystam poniższy plik wsadowy (zapisany jako *input.xml*). Będziesz musiał zmienić nazwy instancji produkcyjnej i testowej w odpowiednich miejscach. Muszą to być różne instancje SQL Servera.

```

<?xml version="1.0" encoding="utf-16" ?>
<DTAXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://schemas.microsoft.com/sqlserver/2004/07/dta">
  <DTAInput>
    <Server>
      <Name>instancja_produkcyjna</Name>
      <Database>
        <Name>AdventureWorks2012</Name>
      </Database>
    </Server>
    <Workload>
      <File>workload.sql</File>
    </Workload>
    <TuningOptions>
      <TestServer>instancja_testowa</TestServer>
      <FeatureSet>IDX</FeatureSet>
    </TuningOptions>
  </DTAInput>
</DTAXML>
  
```

```

    <Partitioning>NONE</Partitioning>
    <KeepExisting>NONE</KeepExisting>
  </TuningOptions>
</DTAInput>
</DTAXML>

```

Elementy Server i Database pliku XML zawierają instancję produkcyjną SQL Servera. Element Workload zawiera definicję skryptu, która zawiera zapytania do optymalizacji. Element TuningOptions zawiera element TestServer, który zawiera instancję testową.

Stwórz plik *workload.sql*, zawierający następujące proste zapytanie:

```

SELECT * FROM AdventureWorks2012.Sales.SalesOrderDetail
WHERE ProductID = 898

```

Wykonaj poniższe zapytanie (zauważ różnicę między -S i -s — wielkość liter ma znaczenie):

```
dta -ix input.xml -S instancja_produkcyjna -s sesja1
```

Pomyślne wykonanie zwróci wynik podobny do poniższego:

```

Microsoft (R) SQL Server Microsoft SQL Server Database Engine Tuning Advisor command line
Utility
Version 12.0.2000.8 ((SQL14_RTM).140220-1832 )
Copyright (c) 2014 Microsoft. All rights reserved.
Tuning session successfully created. Session ID is 3.
Total time used: 00:00:49
Workload consumed: 100%, Estimated improvement: 88%
Tuning process finished.

```

Zostanie stworzona kopia bazy AdventureWorks2012 (bez danych), a na niej wykonana zostanie żądana optymalizacja. Baza skorupa jest automatycznie usuwana po zakończeniu sesji. Opcjonalnie możesz zachować tę bazę (na przykład jeżeli chcesz ją ponownie wykorzystać w innej optymalizacji) za pomocą elementu RetainShellDB wewnątrz elementu TuningOptions, jak w poniższym przykładowym kodzie XML:

```

<TuningOptions>
  <TestServer>instancja_testowa </TestServer>
  <FeatureSet>IDX</FeatureSet>
  <Partitioning>NONE</Partitioning>
  <KeepExisting>NONE</KeepExisting>
  <RetainShellDB>1</RetainShellDB>
</TuningOptions>

```

Jeżeli baza skorupa istnieje już podczas tworzenia sesji optymalizacyjnej, proces tworzenia bazy zostanie pominięty. Będziesz jednak musiał ręcznie usunąć tę bazę, gdy przestanie być potrzebna.

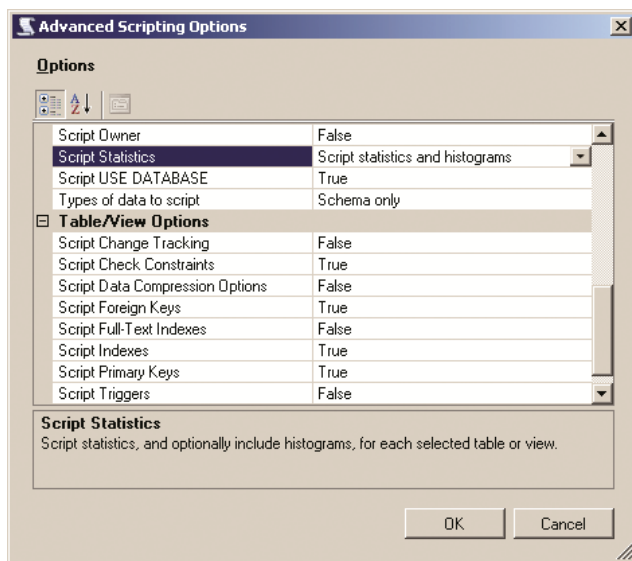
Kiedy sesja zostanie zakończona, będziesz mógł wykorzystać interfejs graficzny do przejrzania rekomendacji. Aby to zrobić, otwórz DTA, otwórz sesję, której używałeś, klikając dwukrotnie jej nazwę (na przykład *sesja1*), i wybierz zakładkę *Recommendations*.

Chociaż DTA automatycznie zbiera metadane i statystyki do budowy bazy skorupy, pokażę, jak stworzyć skrypt zawierający obiekty i statystyki potrzebne do optymalizacji prostego zapytania. Może to być użyteczne, jeżeli nie będziesz chciał kopiować całej bazy danych. Tworzenie skryptów z obiektami bazy danych jest procesem względnie prostym i dobrze znanym osobom pracującym z bazą SQL Server. Dla wielu osób nowe może okazać się tworzenie skryptu statystyk. Stworzone skrypty wykorzystują nieudokumentowane opcje `STATS_STREAM`, `ROWCOUNT` i `PAGECOUNT` oraz polecenie `STATISTICS`.

Aby zoptymalizować poprzednio pokazane zapytanie, w Management Studio wykonaj poniższe czynności:

1. Wybierz węzeł *Databases*.
2. Kliknij prawym przyciskiem myszy bazę *AdventureWorks2012*, wybierz *Tasks/Generate Scripts*, a następnie kliknij *Next*.
3. Wybierz opcję *Select specific database objects* (konkretne obiekty bazy danych).
4. Rozwiń *Tables*.
5. Wybierz *Sales.SalesOrderDetail* i kliknij *Next*.
6. Kliknij *Advanced*.
7. Poszukaj właściwości *Script Statistics* (stwórz skrypt statystyk) i wybierz wartość *Script statistics and histograms* (stwórz skrypt statystyk i histogramów).
8. Dla właściwości *Script Indexes* wybierz wartość *True*.

Okno z zaawansowanymi opcjami powinno wyglądać podobnie do tego z rysunku 5.12.



Rysunek 5.12. Okno z zaawansowanymi opcjami tworzenia skryptu

Kliknij *OK*, aby zakończyć działanie kreatora i wygenerować skrypty. Otrzymasz skrypt z kilkoma poleceniami `UPDATE STATISTICS` podobnymi do poniższego (ze skróconą wartością `STAT_STREAM`, aby zmieściła się na stronie):

```
UPDATE STATISTICS [Sales].[SalesOrderDetail] ([IX_SalesOrderDetail_ProductID])
WITH STATS_STREAM = 0x0100000003000000000000000000000041858B2900000000141A00 ...,
ROWCOUNT = 121317, PAGECOUNT = 274
```

Polecenia `UPDATE STATISTICS` są wykorzystywane do aktualizowania statystyk istniejących indeksów (oczywiście odpowiednie polecenia `CREATE INDEX` również zostały zawarte w skrypcie). Jeżeli tabela ma także statystyki dla kolumn, w skrypcie znajdują się polecenia `CREATE STATISTICS`.

Pokażę teraz, jak skorzystać z zeszyptowanych statystyk do pozyskania planów i szacunków kosztów dla pustej tabeli. Uruchomienie poniższego zapytania dla zwykłej bazy AdventureWorks2012 stworzy plan z szacowaną liczbą wierszy równą 9 i kosztem równym 0,0296836:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 898
```

Stwórzmy nowy plan dla nowej i pustej bazy. Najpierw stwórzmy schemat `Sales`:

```
CREATE SCHEMA Sales
```

Za pomocą poprzednio opisanego procesu stwórz skrypt tabeli `Sales.SalesOrderDetail`. Otrzymasz wiele poleceń, włączając w to poniższe (znów skrócone ze względu na ograniczone miejsce na stronie). Chociaż skrypt stworzył wiele poleceń, te polecenia potrzebne są w tym ćwiczeniu.

```
CREATE TABLE [Sales].[SalesOrderDetail] (
    [SalesOrderID] [int] NOT NULL,
    ...
) ON [PRIMARY]
GO
```

```
CREATE NONCLUSTERED INDEX [IX_SalesOrderDetail_ProductID] ON [Sales].[SalesOrderDetail]
(
    [ProductID] ASC
)
GO
```

```
UPDATE STATISTICS [Sales].[SalesOrderDetail] ([IX_SalesOrderDetail_ProductID])
WITH STATS_STREAM = 0x0100000003000000000000000000000041858B2900000000141A00 ...,
ROWCOUNT = 121317, PAGECOUNT = 274
GO
```

```
UPDATE STATISTICS
[Sales].[SalesOrderDetail] ([PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID])
WITH STATS_STREAM = 0x0100000002000000000000000000000051752A6300000000431500 ...,
ROWCOUNT = 121317, PAGECOUNT = 1237
```

Wykonaj przynajmniej powyższe cztery polecenia, korzystając ze skryptów otrzymanych w poprzednim kroku (pamiętaj, że powyższy listing nie zawiera całych

poleceń — zostały tu zawarte tylko w celach poglądowych). Po uruchomieniu skryptu na pustej bazie i wykonaniu przykładowego zapytania znów otrzymasz plan z kosztem 0,0296836 i szacowaną liczbą rekordów równą 9.

Brakujące indeksy

SQL Server udostępnia drugie podejście, które może pomóc znaleźć użyteczne indeksy dla Twoich istniejących zapytań. Chociaż opcja ta, nazywana funkcjonalnością Missing Indexes (brakujące indeksy), nie jest tak potężna jak DTA, nie wymaga ani tego, aby administrator zdecydował, iż konieczna jest optymalizacja, ani precyzowania, które zapytania wymagają optymalizacji, ani nawet uruchamiania żadnego narzędzia. Jest to niewielka, zawsze włączona funkcjonalność, wprowadzona, podobnie jak DTA, w SQL Serverze 2005. Przyjrzyjmy się, co robi.

Podczas optymalizacji optymalizator definiuje, jakie są najlepsze indeksy dla zapytania, i jeżeli indeksy te nie istnieją, udostępnia odpowiednie informacje w planie XML (informacje te są również dostępne w planie graficznym w SQL Management Studio 2008 lub późniejszych). Alternatywnie: funkcjonalność będzie agregowała te informacje dla zapytań zoptymalizowanych od początku działania instancji i udostępniała je w widoku `sys.dm_db_missing_index`. Zauważ, że wyświetlając te informacje, optymalizator nie tylko informuje, że być może nie korzysta z najbardziej wydajnego planu, lecz także pokazuje, jakie indeksy mogą pomóc w optymalizacji zapytania. Ponadto administratorzy i programiści powinni być świadomi ograniczeń tej funkcjonalności opisanych w dokumentacji Books Online w artykule *Limitations of the Missing Indexes Feature*, dostępnym pod adresem [http://msdn.microsoft.com/en-us/library/ms345485\(v=sql.105\).aspx](http://msdn.microsoft.com/en-us/library/ms345485(v=sql.105).aspx).

Spójrzmy zatem, jak działa ta funkcjonalność. Stwórz tabelę `dbo.SalesOrderDetail` w bazie danych `AdventureWorks2012`, uruchamiając następujące zapytanie:

```
SELECT * INTO dbo.SalesOrderDetail
FROM Sales.SalesOrderDetail
```

Uruchom poniższe zapytanie i zażądaj planu XML lub graficznego:

```
SELECT * FROM dbo.SalesOrderDetail
WHERE SalesOrderID = 43670 AND SalesOrderDetailID > 112
```

To zapytanie mogłoby skorzystać z indeksu na kolumnach `SalesOrderID` i `SalesOrderDetailID`, tym razem jednak nie są pokazywane żadne informacje o brakujących indeksach. Jednym z ograniczeń mechanizmu Missing Indexes, które możemy zaobserwować na podstawie tego przykładu, jest to, że nie działa on z optymalizacjami trywialnymi. Możesz zweryfikować, że jest to plan trywialny, zaglądając do właściwości planu graficznego, gdzie znajdziesz właściwość `Optimization Level` z wartością `TRIVIAL`, lub analizując plan XML, gdzie węzeł `StatementOptmLevel` będzie miał wartość `TRIVIAL`.

Możesz uniknąć optymalizacji trywialnej na kilka sposobów, jak wyjaśniłem w rozdziale 3. W naszym przypadku stworzymy po prostu niezwiązany indeks za pomocą poniższego zapytania:

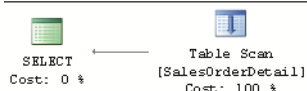
```
CREATE INDEX IX_ProductID ON dbo.SalesOrderDetail(ProductID)
```

Co ważne, chociaż stworzony indeks nie będzie wykorzystywany przez nasze poprzednie zapytanie, zapytanie nie będzie już się kwalifikować do wykorzystania planu trywialnego. Po raz kolejny uruchom zapytanie, a tym razem plan XML będzie zawierał poniższą sekcję:

```
<MissingIndexes>
  <MissingIndexGroup Impact="99.7142">
    <MissingIndex Database="[AdventureWorks2012]" Schema="[dbo]" Table="[SalesOrderDetail]">
      <ColumnGroup Usage="EQUALITY">
        <Column Name="[SalesOrderID]" ColumnId="1" />
      </ColumnGroup>
      <ColumnGroup Usage="INEQUALITY">
        <Column Name="[SalesOrderDetailID]" ColumnId="2" />
      </ColumnGroup>
    </MissingIndex>
  </MissingIndexGroup>
</MissingIndexes>
```

Wpis `MissingIndexes` w planie XML może pokazać się dla trzech grup: `EQUALITY` (równości), `INEQUALITY` (nierówności) i `INCLUDED` (zawierania); w tym przykładzie pokazane są, za pomocą atrybutu `ColumnGroup`, pierwsze dwie. Informacje zawarte w tych grupach mogą być wykorzystane do stworzenia brakującego indeksu; klucz indeksu może być zbudowany z użyciem kolumn równości, po nich kolumn nierówności, a kolumny zawarte można dodać za pomocą klauzuli `INCLUDE` polecenia `CREATE INDEX`. Management Studio (od SQL Servera 2008) może zbudować polecenie `CREATE INDEX`. Tak naprawdę, jeżeli spojrzysz na plan graficzny, zobaczysz ostrzeżenie o brakującym indeksie, zawierające polecenie `CREATE INDEX` (zobacz rysunek 5.13).

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [dbo].[SalesOrderDetail] WHERE [SalesOrderID]=@1 AND [SalesOrderDetailID]>@2
Missing Index (Impact 99.7142): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]...
```



Rysunek 5.13. Plan z ostrzeżeniem o brakującym indeksie

Zwróć uwagę na wartość atrybutu `Impact` (wpływ) wynoszącą 99,7142. Wpływ to liczba z zakresu od 0 do 100, które daje pogląd na średni procentowy zysk, jaki można osiągnąć, zakładając brakujący indeks. Możesz kliknąć prawym przyciskiem myszy na planie graficznym i wybrać opcję *Missing Index Details* (szczegóły brakującego indeksu), aby zobaczyć polecenie `CREATE INDEX`, które stworzy brakujący indeks:

```

/*
Missing Index Details from SQLQuery1.sql - (local).AdventureWorks2012
The Query Processor estimates that implementing the following index could improve the query cost by 99.7142%.
*/

```

```

/*
USE [AdventureWorks2012]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [dbo].[SalesOrderDetail] ([SalesOrderID],[SalesOrderDetailID])

```

```

GO
*/

```

Stwórz rekomendowany indeks, po ustaleniu dla niego nazwy, uruchamiając poniższe polecenie:

```

CREATE NONCLUSTERED INDEX IX_SalesOrderID_SalesOrderDetailID
ON [dbo].[SalesOrderDetail] ([SalesOrderID], [SalesOrderDetailID])

```

Jeżeli ponownie uruchomisz poprzednie polecenie SELECT i spojrzysz na plan wykonania, tym razem zobaczysz, że operator *Index Seek* skorzystał z właśnie stworzonego indeksu, a ostrzeżenie o brakujących indeksach i element *MissingIndex* w planie XML zniknęły.

Usuń teraz stworzoną wcześniej tabelę *dbo.SalesOrderDetail* za pomocą następującego polecenia:

```

DROP TABLE dbo.SalesOrderDetail

```

Fragmentacja indeksów

Chociaż SQL Server automatycznie zajmuje się utrzymaniem indeksów po każdej operacji INSERT, UPDATE, DELETE lub MERGE, konieczne mogą się okazać pewne aktywności konserwacyjne, głównie z powodu fragmentacji indeksów. Fragmentacja ma miejsce, jeżeli logiczna kolejność stron w indeksie nie zgadza się z fizyczną kolejnością danych w pliku. Ponieważ fragmentacja może mieć wpływ na wydajność niektórych zapytań, musisz monitorować poziom fragmentacji swoich indeksów, a jeżeli to konieczne, wykonać dla nich operacje reorganizacji lub przebudowy.

Warto również wyjaśnić, że fragmentacja może mieć wpływ tylko na zapytania wykonujące skanowanie lub skanowanie zakresów, natomiast zapytania wykonujące przeszukania indeksu mogą w ogóle nie odczuć różnicy. Optymalizator również nie bierze pod uwagę fragmentacji, dlatego tworzone przez niego plany będą takie same niezależnie od poziomu fragmentacji indeksów. Oznacza to, że optymalizator nie uwzględnia tego, czy strony w indeksie są fizycznie uporządkowane czy nie. Jednakże jeden z wsadów optymalizatora zapytań to liczba stron wykorzystanych przez tabelę lub indeks, a ta liczba stron może się zwiększać, jeżeli będzie dużo nieużywanej przestrzeni.

Do analizy fragmentacji indeksów możesz wykorzystać widok `sys.dm_db_index_physical_stats`, gdzie będziesz mógł uzyskać informację na temat konkretnej partycji lub indeksu albo przeglądać wszystkie indeksy dla tabeli, bazy, a nawet całej instancji SQL Servera. Poniższy przykład zwróci informacje o fragmentacji tabeli `Sales.SalesOrderDetail` w bazie danych `AdventureWorks2012`:

```
SELECT a.index_id, name, avg
      _fragmentation_in_percent, fragment_count,
      avg_fragment_size_in_pages
FROM sys.dm_db_index_physical_stats (DB_ID('AdventureWorks2012'),
      OBJECT_ID('Sales.SalesOrderDetail'), NULL, NULL, NULL) AS a
JOIN sys.indexes AS b ON a.object_id = b.object_id AND a.index_id = b.index_id
```

W przypadku mojej kopii bazy `AdventureWorks2012` otrzymałem następujące wyniki (nie wszystkie kolumny zostały pokazane):

index_id	name	avg_fragmentation_in_percent
1	PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID	36.13581245
2	AK_SalesOrderDetail_rowguid	2.643171806
3	IX_SalesOrderDetail_ProductID	25.83892617

Chociaż poziom fragmentacji uważany za problematyczny może być różny w zależności od zastosowania i bazy danych, dobrą praktyką jest reorganizowanie indeksów z fragmentacją między 10 a 30%. Operacja przebudowy indeksu może być bardziej stosowna w przypadku indeksów z fragmentacją powyżej 30%. Fragmentacja poniżej 10% nie powinna być uważana za problem.

Operacja reorganizacji indeksu defragmentuje poziom liścia indeksów klastrowych i nieklastrowych i zawsze jest operacją typu online. Podczas przebudowy indeksu możesz opcjonalnie skorzystać z klauzuli `ONLINE = ON` dla większości operacji przebudowy (bardzo krótki okres na początku i na końcu operacji nie pozwoli na równoległą aktywność użytkowników). Przebudowa indeksu usuwa i tworzy go na nowo oraz usuwa fragmentację poprzez kompaktowanie stron indeksu na podstawie wyspecyfikowanego lub istniejącego współczynnika wypełnienia. Współczynnik wypełnienia jest wartością od 1 do 100, która określa wartość procentową tego, na ile pełen ma być poziom liścia każdej strony indeksu podczas tworzenia lub aktualizacji indeksu.

Aby przebudować wszystkie indeksy dla tabeli `SalesOrderDetail`, skorzystaj z następującego zapytania:

```
ALTER INDEX ALL ON Sales.SalesOrderDetail REBUILD
```

Oto fragmentacja w mojej kopii bazy `AdventureWorks2012` po uruchomieniu powyższego zapytania:

index_id	name	avg_fragmentation_in_percent
1	PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID	0.24291498
2	AK_SalesOrderDetail_rowguid	0
3	IX_SalesOrderDetail_ProductID	0

Gdybyś musiał zreorganizować indeks, co w tym przypadku nie ma miejsca, mógłbyś skorzystać z polecenia:

```
ALTER INDEX ALL ON Sales.SalesOrderDetail REORGANIZE
```

Jak wcześniej wspomniałem, fragmentację można również usunąć dla sterty za pomocą polecenia `ALTER TABLE REBUILD`. Może to jednak być kosztowna operacja, ponieważ na skutek zmiany identyfikatorów RID powoduje przebudowanie wszystkich indeksów nieklastrowych. Przebudowanie indeksu ma także wpływ na konserwację statystyk. Więcej informacji na temat konserwacji indeksów i statystyk znajdziesz w rozdziale 6.

Nieużywane indeksy

Rozdział o indeksach zakończę, przedstawiając widok DMV `sys.dm_db_index_usage_stats`, który możesz wykorzystać do pozyskania informacji na temat operacji wykonywanych przez indeksy. Widok ten jest szczególnie użyteczny w odnajdywaniu indeksów, które nie są wykorzystywane przez żadne zapytania lub są wykorzystywane w minimalnym stopniu. Jak już mówiliśmy, nieużywane indeksy nie zapewniają żadnych korzyści, ale zajmują cenne miejsce na dysku i spowalniają aktualizacje, więc powinno się rozważyć ich usunięcie.

Widok `sys.dm_db_index_usage_stats` przechowuje liczbę przeszukań, skanowań, podglądów i aktualizacji dla indeksów wykonywanych zarówno przez zapytania użytkowników, jak i przez zapytania systemowe, jak również czas ostatniego wykonania operacji, a liczniki są resetowane przy starcie usługi SQL Server. Pamiętaj, że ten widok DMV, oprócz indeksów nieklastrowych, zawiera także sterty, oznaczone wartością 0 w kolumnie `index_id`, oraz indeksy klastrowe z wartością 1. Na potrzeby tego podrozdziału być może zechcesz skupić się tylko na indeksach nieklastrowych, które mają wartość 2 i wyższe w kolumnie `index_id`. Ponieważ sterty i indeksy klastrowe zawierają dane z tabeli, mogą w ogóle nie być kandydatami do usunięcia.

Sprawdzając wartości `user_seeks`, `user_scans` i `user_lookup` indeksów nieklastrowych, możesz zobaczyć, w jaki sposób wykorzystywane są Twoje indeksy. Możesz również spojrzeć na wartości `user_updates`, aby zobaczyć, ile aktualizacji zostało wykonanych na indeksie. Wszystkie te informacje dadzą Ci pogląd na to, jak użyteczny jest indeks.

Pamiętaj, że pokażę jedynie, jak pobrać informacje z tego widoku i jakie sytuacje wywołają aktualizacje zawartych w nim danych.

Jako przykład uruchom poniższy kod tworzący nową tabelę z indeksem nieklastrowym:

```
SELECT * INTO dbo.SalesOrderDetail
FROM Sales.SalesOrderDetail
CREATE NONCLUSTERED INDEX IX_ProductID ON dbo.SalesOrderDetail(ProductID)
```

Jeżeli chcesz śledzić wartości z tego przykładu, uważnie wykonaj poniższe kroki, ponieważ każde wykonanie zapytania może zmienić statystyki wykorzystania indeksów. Kiedy uruchomisz poniższe zapytanie, początkowo będzie zawierało tylko jeden rekord, który został stworzony po uzyskaniu dostępu do tabeli w trakcie tworzenia indeksu IX_ProductID:

```
SELECT DB_NAME(database_id) AS database_name,
       OBJECT_NAME(s.object_id) AS object_name, i.name, s.*
FROM sys.dm_db_index_usage_stats s JOIN sys.indexes i
     ON s.object_id = i.object_id AND s.index_id = i.index_id
AND OBJECT_ID('dbo.SalesOrderDetail') = s.object_id
```

Jednak wartości, które będą nas interesowały w tym przypadku — user_seeks, user_scans, user_lookups i user_updates — są ustawione na 0. Teraz uruchom poniższe zapytanie, powiedzmy, trzy razy:

```
SELECT * FROM dbo.SalesOrderDetail
```

To zapytanie wykorzystuje operator *Table Scan*, więc jeżeli ponownie uruchomisz poprzednie zapytanie korzystające z widoku sys.dm_db_index_usage_stats, zobaczysz wartość 3 w kolumnie user_scans. Zauważ, że kolumna index_id ma wartość 0, co oznacza stertę; podana jest również nazwa tabeli (ponieważ sterta to tabela bez indeksu klastrowego). Dwukrotnie uruchom następne zapytanie, które korzysta z operatora *Index Seek*. Po wykonaniu zapytania dodany zostanie nowy rekord dla indeksu klastrowego, a wartość licznika user_seeks będzie równa 2.

```
SELECT ProductID FROM dbo.SalesOrderDetail
WHERE ProductID = 773
```

Teraz uruchom poniższe zapytanie cztery razy — wykorzystuje ono zarówno operator *Index Seek*, jak i *RID Lookup*. Ponieważ licznik user_seeks dla indeksu nieklastrowego miał wartość 2, teraz będzie równy 6, a wartość user_lookups dla sterty zostanie zmieniona na 4.

```
SELECT * FROM dbo.SalesOrderDetail
WHERE ProductID = 773
```

Na koniec raz uruchom poniższe zapytanie:

```
UPDATE dbo.SalesOrderDetail
SET ProductID = 666
WHERE ProductID = 927
```

Zauważ, że polecenie UPDATE wykorzystuje operatory *Index Seek* i *Table Update*, więc wartość `user_seek` zostanie zaktualizowana dla indeksu, a wartość `user_updates` zostanie zaktualizowane zarówno dla indeksu nieklastrowego, jak i dla sterty. Oto ostateczny wynik naszego zapytania wykorzystującego widok `sys.dm_db_index_usage_stats` (wyedytowany ze względu na ograniczoną ilość dostępnego miejsca):

name	index_id	user_seeks	user_scans	user_lookups	user_updates
NULL	0	0	3	4	1
IX_ProductID	2	7	0	0	1

Możesz teraz usunąć tabelę, którą stworzyliśmy:

```
DROP TABLE dbo.SalesOrderDetail
```

Podsumowanie

W tym rozdziale opisałem indeksowanie jako jedną z najważniejszych technik używanych w optymalizacji zapytań oraz omówiłem indeksy klastrowe i indeksy nieklastrowe wraz z tematami z nimi związanymi, takimi jak sposób wykorzystania indeksów przez SQL Server, wybór klucza indeksu klastrowego i naprawa fragmentacji indeksu.

Wyjaśniłem również, jak zdefiniować klucz indeksów, tak aby był wykorzystywany w operacjach wyszukiwania, co może poprawić wydajność zapytań i pomóc w szybkim wyszukiwaniu rekordów. Przeanalizowałem predykaty w kontekście indeksów jedno- i wielokolumnowych, a także kwestię sprawdzenia w planie wykonania, czy indeksy zostały wybrane i poprawnie wykorzystane przez SQL Server.

Przedstawiłem funkcjonalności Database Engine Tuning Advisor i Missing Indexes, wprowadzone w SQL Serverze 2008. Omówiłem też wykorzystanie przez nie optymalizatora zapytań do budowania rekomendacji dotyczących optymalizacji indeksów.

Na koniec przedstawiłem widok `sys.dm_db_index_usage_stats` oraz ważne informacje dotyczące wykorzystania indeksów nieklastrowych, które udostępnia. Chociaż nie mieliśmy czasu zająć się wszystkimi aspektami wykorzystania tego widoku, omówiliśmy ich wystarczająco dużo, abyś mógł odszukać indeksy nieklastrowe, które nie są stosowane przez Twoją instancję SQL Servera.

Rozdział 6

Statystyki

W tym rozdziale:

- ▶ Statystyki
- ▶ Histogram
- ▶ Nowy estymator kardynalności
- ▶ Błędy estymacji kardynalności
- ▶ Statystyki inkrementacyjne
- ▶ Statystyki dla kolumn wyliczeniowych
- ▶ Statystyki filtrowane
- ▶ Statystyki dla kluczy rosnących
- ▶ UPDATE STATISTICS z ROWCOUNT i PAGECOUNT
- ▶ Statystyki dla serwerów połączonych
- ▶ Konserwacja statystyk
- ▶ Szacunek kosztów
- ▶ Podsumowanie



Optymalizator w SQL Serverze działa w oparciu o koszty, a więc jakość planów, które generuje, jest bezpośrednio związana z trafnością szacunków kosztów. Analogicznie: szacowany koszt planu oparty jest na wykorzystanych algorytmach i operatorach oraz ich estymacji kardynalności. Z tego powodu, aby poprawnie oszacować koszt planu wykonania, optymalizator musi oszacować, najdokładniej jak to możliwe, liczbę rekordów zwracaną przez dane zapytanie.

Podczas optymalizacji zapytania SQL Server tworzy wiele potencjalnych planów, szacuje ich relatywne koszty i wybiera najbardziej wydajny z nich. Dlatego błędne oszacowanie kardynalności i kosztu może sprawić, że optymalizator wybierze nieefektywny plan, co może mieć negatywny wpływ na wydajność bazy danych.

W tym rozdziale omówię statystyki wykorzystywane przez optymalizator zapytań, opiszę, jak upewnić się, czy zapewniasz najlepsze możliwe statystyki, oraz zasugeruję, co zrobić w sytuacji, gdy błędny szacunek kardynalności jest nieunikniony. Statystyki optymalizatora zawierają trzy główne elementy informacji: histogram, informację o gęstości i statystyki ciągów znaków, a każdy z nich pomaga w innej części procesu szacowania kardynalności. Pokażę Ci, jak statystyki są tworzone i utrzymywane oraz jak są wykorzystywane przez optymalizator zapytań. Podam Ci również informacje dotyczące wykrywania błędów szacowania kardynalności, które mogą negatywnie wpłynąć na jakość planów wykonania, oraz sposoby naprawiania tych błędów.

Omówię także nowy mechanizm szacowania kardynalności wprowadzony w SQL Serverze 2014 oraz różnice w stosunku do poprzedniej jego wersji. Rozdział zakończę ogólnym omówieniem modelu szacowania kosztów, który szacuje koszt I/O oraz CPU dla każdego operatora, aby po ich zsumowaniu otrzymać całkowity koszt planu.

Statystyki

SQL Server tworzy i utrzymuje statystyki, aby pomóc optymalizatorowi w szacowaniu kardynalności. **Szacunek kardynalności** jest szacowaną liczbą wierszy, która zostanie zwrócona przez zapytanie lub konkretny operator, taki jak złączenie lub filtr. **Selektywność** to koncepcja podobna do szacunku kardynalności i można ją opisać jako część wierszy w zestawie, która satysfakcjonuje predykat i zawsze jest wartością od 0 do 1 włącznie. Wysoko selektywny predykat zwraca małą liczbę wierszy. Więcej szczegółów dotyczących tych koncepcji poznasz w dalszej części tego rozdziału.

Tworzenie i aktualizacja statystyk

Na początek przyjrzyjmy się różnym sposobom tworzenia i aktualizacji statystyk. Statystyki są tworzone na kilka sposobów: automatycznie przez optymalizator zapytań (jeżeli ustawiona jest opcja automatycznego tworzenia statystyk — `AUTO_CREATE_`

➤`STATISTICS`), kiedy tworzony jest indeks oraz bezpośrednio (na przykład za pomocą polecenia `CREATE STATISTICS`). Statystyki mogą być tworzone dla jednej lub wielu kolumn, a zarówno tworzenie indeksu, jak i metoda bezpośrednia wspierają jedno- i wielokolumnowe statystyki. Jak już wstępnie wspomniałem, statystyki składają się z histogramu, informacji o gęstości oraz statystyk ciągów znaków. Zarówno histogram, jak i statystyki ciągów znaków tworzone są tylko dla pierwszej kolumny obiektu statystyk, a statystyka ciągów znaków tylko wówczas, jeżeli kolumna jest typu `string`.

Informacja o gęstości, którą omawiam bardzo dokładnie w dalszej części tego rozdziału, jest obliczana dokładnie dla każdego zestawu kolumn, tworząc prefiks obiektu statystyk. Natomiast statystyki filtrowane nie są tworzone automatycznie przez optymalizator, lecz tylko wtedy, kiedy tworzony jest indeks filtrowany lub kiedy wywołane zostanie polecenie `CREATE STATISTICS` z klauzulą `WHERE`. Zarówno indeksy filtrowane, jak i statystyki są funkcjonalnościami wprowadzonymi w SQL Serverze 2008. Indeksy filtrowane omówiłem w rozdziale 5., a ze statystykami filtrowanymi zetknijemy się w dalszej części tego rozdziału.

W domyślnej konfiguracji (jeżeli `AUTO_UPDATE_STATISTICS` jest włączone) optymalizator zapytań automatycznie aktualizuje statystyki, jeżeli nie są aktualne. Jak wcześniej zaznaczyłem, optymalizator nie tworzy automatycznie statystyk wielokolumnowych ani filtrowanych, lecz jeżeli zostaną stworzone za pomocą którejś z wcześniej opisanych metod, mogą zostać automatycznie zaktualizowane. Alternatywnie: operacje przebudowy indeksów i polecenia takie jak `UPDATE STATISTICS` mogą również zostać wykorzystane do aktualizacji statystyk. Ponieważ obie opcje, autotworzenia i autoaktualizacji, zazwyczaj dają statystyki o dobrej jakości, zaleca się, aby pozostawić te opcje włączone. Normalnie masz także możliwość wykorzystania kilku innych poleceń, które dadzą Ci większą kontrolę nad jakością statystyk.

A więc domyślnie statystyki mogą być automatycznie tworzone (jeżeli nie istnieją) i automatycznie aktualizowane (jeżeli są nieaktualne) podczas optymalizacji zapytań. Jako „nieaktualne” rozumiemy dane, które zostały zmienione, przez co statystyki przestają być dla tych danych reprezentatywne (więcej na temat tego mechanizmu w dalszej części). Jeżeli plan wykonania dla danego zapytania istnieje w magazynie planów, a statystyki wykorzystane do zbudowania planu nie są aktualne, plan jest usuwany, statystyki aktualizowane i tworzony jest nowy plan. W podobny sposób aktualizacja statystyk, manualna lub automatyczna, sprawia, że wszystkie plany, które korzystały z tych statystyk, stają się nieaktualne i przy następnym wywołaniu zapytania wywołana zostanie nowa optymalizacja.

Jeżeli chodzi o określanie jakości statystyk, warto rozważyć rozmiar próbki tabeli będącej celem obliczania statystyk. Optymalizator określa statystycznie znaczącą próbkę, kiedy tworzy lub aktualizuje statystykę, a minimalny rozmiar to 8 MB (1024 stron) lub rozmiar tabeli, jeżeli jest ona mniejsza niż 8 MB. Rozmiar próbki będzie rósł dla większych tabel, ale wciąż może to być tylko mały procent tabeli.

Jeżeli zajdzie taka konieczność, możesz wykorzystać polecenia `CREATE STATISTICS` i `UPDATE STATISTICS` do bezpośredniego zażądania większej próbki lub skanowania całej tabeli w celu budowy lepszych statystyk. Aby to zrobić, musisz wyspecyfikować rozmiar próbki lub użyć opcji `WITH FULLSCAN` (z pełnym skanowaniem), aby przeskanować całą tabelę. Rozmiar próbki można wyspecyfikować jako liczbę rekordów lub wartość procentową, a ponieważ optymalizator musi przeskanować wszystkie dane na stronie danych, wartości te są przybliżone. Użycie `WITH FULLSCAN` lub większej próbki może być korzystne, szczególnie w przypadku danych, które nie są losowo rozłożone w całej tabeli. Skanowanie całej tabeli da Ci oczywiście najdokładniejsze statystyki. Zauważ, że jeżeli statystyki są budowane po przeskanowaniu 50% danych, SQL Server założy, iż pozostałe 50% danych jest dokładnie takie samo jak dane, które przeskanował. Biorąc pod uwagę fakt, że statystyki są tworzone wraz z indeksami i że ta operacja i tak skanuje całą tabelę, tworzenie statystyki zawsze przypomina wykorzystanie opcji `WITH FULLSCAN`. Jednakże jeżeli optymalizator musi automatycznie zaktualizować te statystyki, wówczas musi wrócić do próbki domyślnej, ponieważ ponowne skanowanie całej tabeli mogłoby trwać zbyt długo.

Domyślnie SQL Server musi poczekać na zakończenie operacji aktualizacji statystyk przed optymalizacją i wykonaniem zapytania; oznacza to, że statystyki aktualizowane są synchronicznie. Wprowadzona w SQL Serverze 2005 opcja konfiguracji bazy danych, `AUTO_UPDATE_STATISTICS_ASYNC`, może zostać wykorzystana do zmiany tego domyślnego zachowania i sprawienia, aby statystyki były aktualizowane asynchronicznie. Jak zapewne zgadłeś, w przypadku aktualizacji asynchronicznej optymalizator nie czeka na zakończenie operacji aktualizacji statystyk i w procesie optymalizacji wykorzystuje aktualne statystyki. To może pomóc w sytuacjach, w których w aplikacjach występuje problem z przekroczeniem czasu z powodu dodatkowego czasu potrzebnego na aktualizację statystyk. Chociaż aktualna optymalizacja wykorzysta nieaktualne statystyki, zostaną one zaktualizowane w tle i użyte w późniejszych optymalizacjach. Jednakże asynchroniczne aktualizacje są zazwyczaj korzystne jedynie dla zapytań OLTP i mogą nie być dobrym rozwiązaniem w przypadku bardziej kosztownych zapytań, gdzie uzyskanie lepszego planu zapytania jest ważniejsze niż rzadko występujące opóźnienie związane z aktualizacją statystyk.

SQL Server określa, kiedy statystyki nie są aktualne, na podstawie liczników modyfikacji kolumn, zwanych *colmodctr*, które zliczają całkowitą liczbę modyfikacji dla głównych kolumn statystyk od ostatniej aktualizacji statystyk. Dla tabel większych niż 500 rekordów obiekt statystyk jest uważany za nieaktualny, jeżeli wartość *colmodctr* głównej kolumny zmieniła się o więcej niż 500 plus 20% liczby rekordów w tabeli. Ta sama formuła jest wykorzystywana dla statystyk filtrowanych, ale ponieważ są budowane tylko z podzbioru rekordów tabeli, *colmodctr* jest najpierw mnożona przez selektywność filtra. Wartość *colmodctr* można znaleźć w kolumnie *modification_counter* funkcji DMF `sys.dm_db_stats_properties`, która dostępna jest od wersji SQL Server 2008 R2 Service Pack 2 i SQL Server 2012 Service Pack 1 (wcześniej wartość ta była dostępna

tylko poprzez specjalnie dedykowane połączenie administracyjne w kolumnie *rcmodified* tabeli systemowej *sys.sysrscols* w SQL Serverze 2008 i w kolumnach *sysrowset* w SQL Serverze 2005).



UWAGA

SQL Server 2000 wykorzystywał *rowmodctr*, lub inaczej liczniki modyfikacji, śledzące liczbę zmian w tabeli lub indeksie. Najistotniejsza różnica polega na tym, że *rowmodctr* śledzą dowolne zmiany wiersza, natomiast *colmodctr* śledzą zmiany tylko dla głównych kolumn obiektów statystyk. Aktualnie polecenie *sp_updatestats*, będące kolejnym sposobem aktualizacji, wciąż bazuje na licznikach *rowmodctr*, które dostępne są w kolumnie *rowmodctr* widoku *sys.sysindexes*.

W SQL Server 2008 R2 Service Pack 1, jako sposób automatycznej aktualizacji statystyk w niższych i dynamicznych zakresach procentowych, zamiast wspomnianych 20%, została wprowadzona flaga 2371. Z dynamicznym zakresem procentowym im wyższa jest liczba rekordów w tabeli, tym niższa liczba aktualizacji będzie potrzebna do wyzwolenia automatycznej aktualizacji statystyk. Tabele z liczbą rekordów mniejszą niż 25000 będą wykorzystywały wartość 20%, ale wraz ze wzrostem liczby rekordów w tabeli ta liczba będzie coraz niższa. Więcej szczegółów na temat tej flagi znajdziesz w artykule *Changes to Automatic Update Statistics in SQL Server — Traceflag 2371*.

Informacja o gęstości dla statystyk wielokolumnowych może poprawić jakość planów wykonania w przypadku kolumn skorelowanych lub statystycznych korelacji pomiędzy kolumnami. Jak wcześniej wspomniałem, informacje o gęstości są przechowywane dla wszystkich kolumn w obiekcie statystyk, w takiej kolejności, w jakiej pojawiają się w definicji statystyki. Domyślnie SQL Server zakłada, że kolumny są niezależne, a zatem jeżeli pomiędzy kolumnami istnieje relacja lub zależność, statystyki wielokolumnowe mogą pomóc w rozwiązaniu problemów z szacowaniem kardynalności w zapytaniach korzystających z tych kolumn. Informacje o gęstości pomogą również w przypadku filtrów i klauzul *GROUP BY*, jak zobaczymy w dalszej części, w podrozdziale „Gęstość”. Statystyki filtrowane, które także opiszę w dalszej części tego rozdziału, również mogą być wykorzystywane podczas rozwiązywania problemów przy szacowaniu kardynalności dla kolumn skorelowanych. Więcej szczegółów na temat założenia niezależności przedstawię w podrozdziale „Nowy mechanizm szacowania kardynalności”.

Sprawdzanie obiektów statystyk

Przyjrzyjmy się przykładowemu obiektowi statystyk i sprawdźmy, jakie dane przechowuje. Najpierw jednak upewnijmy się, czy korzystasz z nowego mechanizmu szacowania kardynalności, poprzez uruchomienie poniższego zapytania:

```
ALTER DATABASE AdventureWorks2012
SET COMPATIBILITY_LEVEL = 120
```

Istniejące statystyki dla konkretnego obiektu można wyświetlić za pomocą widoku katalogu systemowego sys.stats:

```
SELECT * FROM sys.stats
WHERE object_id = OBJECT_ID('Sales.SalesOrderDetail')
```

Wynik będzie podobny do poniższego (wyedytowany ze względu na rozmiar):

object_id	name	stats_id
1154103152	PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID	1
1154103152	AK_SalesOrderDetail_rowguid	2
1154103152	IX_SalesOrderDetail_ProductID	3

Pokazany zostanie jeden wiersz dla każdego obiektu statystyk. Aby wyświetlić szczegóły na temat obiektu statystyk, możesz użyć polecenia DBCC SHOW_STATISTICS, podając nazwę kolumny lub nazwę obiektu statystyk.

Uruchom na przykład poniższe polecenie, aby potwierdzić, że dla kolumny *UnitPrice* z tabeli Sales.SalesOrderDetail nie ma statystyk:

```
DBCC SHOW_STATISTICS ('Sales.SalesOrderDetail', UnitPrice)
```

Jeżeli obiekt statystyk nie istnieje, jak w tym przypadku dla świeżej instalacji bazy AdventureWorks2012, otrzymasz następujący komunikat o błędzie:

```
Msg 2767, Level 16, State 1, Line 1
Could not locate statistics 'UnitPrice' in the system catalogs.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

Ale przy uruchomieniu poniższego zapytania SQL Server automatycznie stworzy statystyki dla kolumny *UnitPrice*, która wykorzystywana jest w predykcji zapytania:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE UnitPrice = 35
```

Powtórne uruchomienie poprzedniego polecenia DBCC SHOW_STATISTICS pokaże teraz obiekt statystyk podobny do poniższego (pokazany jako tekst i wyedytowany ze względu na ograniczoną ilość miejsca na stronie):

Name		Updated	Rows	Rows Sampled	Steps
_WA_Sys_00000007_44CA3770		Jan 6 2014 11:55PM	121317	110388	200
All density	Average Length Columns				
0.003205128	8	UnitPrice			
RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS	

1.3282	0	1	0	1
1.374	35.19722	142.3062	0	370.2226
2.29	35.19722	2747.751	0	370.2226
2.994	417.5509	341.9168	3	123.5255
3.975	35.19722	1	0	370.2226
3.99	35.19722	2061.052	0	370.2226

Wynik podzielony jest na trzy zestawy rezultatów, nazywane nagłówkiem, wektorem gęstości i histogramem, chociaż nagłówek został przycięty, aby dane zmieściły się na stronie, i pokazane zostało tylko kilka wierszy histogramu. Przyjrzyjmy się kolumnom nagłówka w poprzednim przykładzie obiektu statystyk, pamiętając, że niektóre kolumny, które będę opisywał, nie są widoczne na powyższym listingu z wynikiem:

- ▶ **Name: `_WA_Sys_00000007_44CA3770`** — jest to nazwa obiektu statystyk i prawdopodobnie będzie inna w Twojej instancji SQL Servera. Wszystkie automatycznie generowane statystyki mają nazwę zaczynającą się od `_WA_Sys`. Wartość `00000007` to wartość `column_id` kolumny, na której bazują te statystyki, i można znaleźć ją w katalogu `sys.columns`, a `44CA3770` to heksadecymalny ekwiwalent wartości `object_id` (co można łatwo potwierdzić za pomocą kalkulatora dostępnego w Windowsie). `WA` pochodzi od *Washington*, czyli Waszyngtonu — stanu w USA, w którym pracuje zespół SQL Servera.
- ▶ **Updated: Jan 6 2014 11:55PM** — jest to data i godzina stworzenia lub ostatniej aktualizacji obiektu.
- ▶ **Rows: 121317** — jest to liczba rekordów, jakie istniały w tabeli w momencie stworzenia lub ostatniej aktualizacji obiektu.
- ▶ **Rows Sampled: 110388** — jest to liczba próbkowanych wierszy w momencie stworzenia lub ostatniej aktualizacji obiektu.
- ▶ **Steps: 200** — jest to liczba kroków histogramu. Wartość ta zostanie omówiona w dalszej części.
- ▶ **Density: 0.06236244** — gęstość wszystkich próbkowanych wartości oprócz `RANGE_HI_KEY` (wartość `RANGE_HI_KEY` zostanie omówiona w podrozdziale „Histogram”). Wartość ta jest już nieużywana przez optymalizator zapytań i pozostała tylko ze względu na wsteczną kompatybilność.
- ▶ **Average key length: 8** — jest to średnia liczba bajtów dla kolumn w obiekcie statystyk.
- ▶ **String Index: NO** — ta wartość wskazuje na to, czy obiekt statystyk zawiera statystyki ciągów znaków, a jedyne opcje to YES i NO. Statystyki dla ciągów znaków zawierają rozkład danych w podciągach znaków w kolumnie i mogą być wykorzystywane do określania kardynalności zapytań z warunkiem LIKE. Jak już wcześniej wspomniałem, statystyki ciągów znaków tworzone są tylko dla pierwszej kolumny i tylko, jeżeli kolumna jest typu związanego z ciągami znaków.

- **Filter Expression i Unfiltered Rows** — te kolumny omówię w podrozdziale „Statystyki filtrowane”.

Pod nagłówkiem znajdziesz wektor gęstości, który zawiera wiele potencjalnie użytecznych informacji o gęstości i zostanie wyjaśniony w następnym podrozdziale.

Gęstość

Aby lepiej wyjaśnić wektor gęstości, uruchom poniższe polecenie, które pozwala sprawdzić statystyki indeksu `IX_SalesOrderDetail_ProductID`:

```
DBCC SHOW_STATISTICS ('Sales.SalesOrderDetail', IX_SalesOrderDetail_ProductID)
```

Wyświetlony zostanie poniższy wektor gęstości, który pokaże gęstości kolumny *ProductID* oraz kombinacji kolumn *ProductID* i *SalesOrderID*, a następnie *ProductID*, *SalesOrderID* i *SalesOrderDetailID*:

All density	Average Length	Columns
0.003759399	4	ProductID
8.242868E-06	8	ProductID, SalesOrderID
8.242868E-06	12	ProductID, SalesOrderID, SalesOrderDetailID

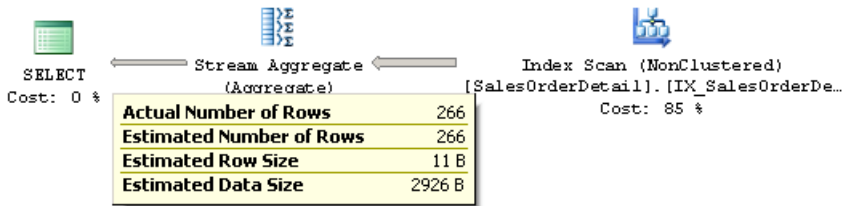
Gęstość zdefiniowana jako *1/liczba unikalnych wartości* jest wyświetlona w kolumnie *All density* i obliczana dla każdego zestawu kolumn, tworząc prefiks dla kolumn w obiekcie statystyk. Na przykład pokazany obiekt statystyk został stworzony dla kolumn *ProductID*, *SalesOrderID* i *SalesOrderDetailID*, dlatego wektor gęstości pokaże trzy różne wartości gęstości: jedną dla *ProductID*, drugą dla połączonych *ProductID* i *SalesOrderID*, a trzecią dla kombinacji kolumn *ProductID*, *SalesOrderID* i *SalesOrderDetailID*. Nazwy analizowanych kolumn będą wyświetlone w kolumnie *Columns*, a kolumna *Average Length* będzie zawierała średnią liczbę bajtów dla każdej wartości gęstości. W poprzednim przykładzie wszystkie kolumny zostały zdefiniowane jako typu `int`, dlatego średnia długość dla każdej wartości gęstości będzie wynosiła odpowiednio 4, 8 i 12 bajtów. Teraz, kiedy wiemy już, jaką strukturę mają informacje o gęstości, zobaczmy, jak są wykorzystywane.

Informacje o gęstości mogą być stosowane do poprawienia szacunków optymalizatora zapytań dla operacji `GROUP BY` i dla predykatów z operatorem równości, jeżeli wartość jest nieznana, jak w przypadku zapytania korzystającego ze zmiennych lokalnych. Aby zobaczyć, jak to się odbywa, rozważmy na przykład liczbę unikalnych wartości w kolumnie *ProductID* w tabeli `Sales.SalesOrderDetail` — jest ich 266. Gęstość może zostać obliczona, jak już wspomniałem, jako *1/liczba unikalnych wartości*, co w tym przypadku oznacza *1/266*, czyli 0,003759399, zgodnie z pierwszą wartością gęstości w poprzednim przykładzie.

Tak więc optymalizator zapytań może wykorzystać informacje o gęstości do szacowania kardynalności w zapytaniach `GROUP BY`. Zapytania `GROUP BY` mogą skorzystać z informacji o szacowanej liczbie unikalnych wartości, a ta wartość jest dostępna

w informacji o gęstości. Na przykład, aby oszacować kardynalność dla poniższego zapytania wykorzystującego klauzulę `GROUP BY ProductID`, możemy obliczyć odwrotność gęstości. W tym przypadku to $1/0,003759399$, co daje 266 — szacowaną liczbę rekordów pokazaną w planie na rysunku 6.1.

```
SELECT ProductID FROM Sales.SalesOrderDetail
GROUP BY ProductID
```



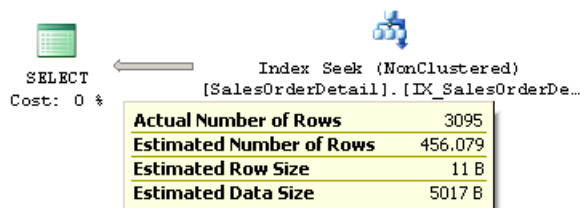
Rysunek 6.1. Przykład szacunku kardynalności dla klauzuli `GROUP BY`

Podobnie, aby przetestować `GROUP BY ProductID, SalesOrderID`, potrzebujemy $1/8,242868E-06$, czyli 121317 unikalnych kombinacji w kolumnach *ProductID* i *SalesOrderID*. Możesz to również potwierdzić z planem wykonania tego zapytania.

Poniżej znajduje się przykład tego, jak gęstość może zostać wykorzystana do szacowania kardynalności zapytania stosującego zmienne lokalne:

```
DECLARE @ProductID int
SET @ProductID = 921
SELECT ProductID FROM Sales.SalesOrderDetail
WHERE ProductID = @ProductID
```

W tym przypadku optymalizator w czasie optymalizacji nie zna wartości zmiennej lokalnej `@ProductID`, nie może więc skorzystać z histogramu (który omówimy niedługo) i korzysta z informacji o gęstości. Szacowana liczba rekordów jest uzyskiwana z mnożenia gęstości i liczby rekordów w tabeli, co w naszym przypadku wynosi $0,003759399 \cdot 121317$, czyli 456079 (zobacz rysunek 6.2).



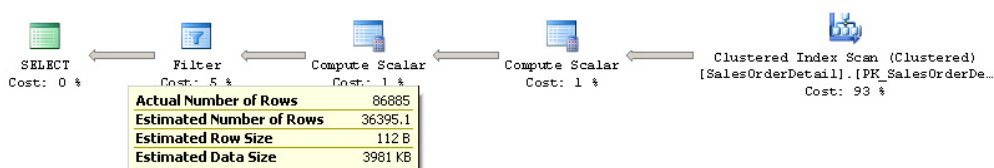
Rysunek 6.2. Przykład szacunku kardynalności dla zmiennej lokalnej

Właściwie, ponieważ w momencie optymalizacji optymalizator zapytań nie zna wartości `@ProductID`, wartość 921 z poprzedniego listingu nie ma znaczenia, bo każda inna wartość da dokładnie tę samą szacowaną liczbę wierszy i taki sam plan wykonania,

ponieważ to jest średnia liczba rekordów dla wartości. Uruchom teraz to samo zapytanie z operatorem nierówności:

```
DECLARE @pid int = 897
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID < @pid
```

Tak jak poprzednio, wartość 897 nie ma znaczenia, bo każda inna wartość da tę samą szacowaną liczbę rekordów i ten sam plan. Jednakże tym razem optymalizator nie może wykorzystać informacji o gęstości i wykorzystuje standardową wartość 30% selektywności dla operatora nierówności. Oznacza to, że dla dowolnego operatora nierówności szacowana liczba rekordów wynosi zawsze 30% całkowitej liczby rekordów — w tym przypadku 30% z 121317 to 36395,1 (zobacz rysunek 6.3).



Rysunek 6.3. Szacowanie kardynalności z użyciem wartości zgadywanej 30%

Jednakże wykorzystanie w zapytaniu zmiennych lokalnych ogranicza jakość szacunków kardynalności podczas używania informacji o gęstości z operatorem równości. Co gorsze, zmienne lokalne powodują brak szacunku podczas korzystania z informacji o gęstości i operatora nierówności, co owocuje wartością zgadywaną. Z tego powodu powinno się unikać stosowania w zapytaniach zmiennych lokalnych, a zamiast nich powinny być stosowane parametry lub literały. Kiedy używane są parametry lub literały, optymalizator może skorzystać z histogramu, co przełoży się na lepsze szacunki niż w przypadku wykorzystania samej informacji o gęstości.

Tak się składa, że ostatnia sekcja wyniku DBCC SHOW_STATISTICS to histogram, który objaśnię w następnym podrozdziale.

Histogram

W SQL Serverze histogramy są tworzone tylko dla pierwszej kolumny obiektu statystyk i zawierają informacje na temat rozkładu wartości w tej kolumnie, rozdzielane na podzbiory nazywane wiadrami lub krokami. Maksymalna liczba kroków w histogramie to 200, ale nawet jeżeli dane wsadowe mają 200 lub więcej unikalnych wartości, histogram może i tak mieć mniej niż 200 kroków. Aby zbudować histogram, SQL Server odszukuje unikalne wartości w kolumnie i próbuje odnaleźć najczęściej pojawiające się wartości, korzystając z wariacji algorytmu *maxdiff*, aby zaprezentowane były informacje najbardziej statystycznie znaczące. *Maxdiff* to jeden z dostępnych histogramów, których rolą jest dokładne przedstawienie rozkładu wartości w bazach relacyjnych.



UWAGA

Uproszczoną wersję algorytmu wykorzystywanego do tworzenia histogramów możesz znaleźć w artykule *Statistics Used by the Query Optimizer in Microsoft SQL Server 2008* autorstwa Erica Hansona i Yavora Angelova.

Aby zobaczyć, jak wykorzystywany jest histogram, uruchom poniższe polecenie wyświetlające aktualne statystyki dla indeksu `IX_SalesOrderDetail_ProductID` tabeli `Sales.SalesOrderDetail`:

```
DBCC SHOW_STATISTICS ('Sales.SalesOrderDetail', IX_SalesOrderDetail_ProductID)
```

Zarówno indeks wielokolumnowy, jak i obiekt statystyk zawierają kolumny *ProductID*, *SalesOrderID* i *SalesOrderDetailID*, ale ponieważ histogram tworzony jest tylko dla pierwszej kolumny, te dane dostępne są tylko dla niej.

Pokażę teraz kilka przykładów tego, jak histogram może być wykorzystany do szacowania kardynalności prostych predykatów. Przyjrzyjmy się części histogramu przedstawionej na poniższym listingu:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
826	0	305	0	1
831	110	198	3	36.66667
832	0	256	0	1

`RANGE_HI_KEY` to górna granica kroku histogramu, wartość 826 to górna granica dla pierwszego wyświetlonego kroku, a 831 to górna granica dla drugiego wyświetlonego kroku. Oznacza to, że drugi krok może zawierać tylko wartości z zakresu 827 – 831. Wartości `RANGE_HI_KEY` to z reguły wartości częściej pojawiające się w rozkładzie.

Aby lepiej zrozumieć resztę struktury histogramu i to, jak jest on agregowany, uruchom poniższe zapytanie, które zwraca prawdziwą liczbę rekordów dla rekordów z wartościami *ProductID* z zakresu 827 – 831; porównamy je z histogramem:

```
SELECT ProductID, COUNT(*) AS Total
FROM Sales.SalesOrderDetail
WHERE ProductID BETWEEN 827 AND 831
GROUP BY ProductID
```

Otrzymasz następujący wynik:

ProductID	Total
827	31
828	46
830	33
831	198

Wracając do histogramu: EQ_ROWS to szacowana liczba rekordów, dla których wartość kolumny jest równa wartości RANGE_HI_KEY. Tak więc w naszym przykładzie dla wartości RANGE_HI_KEY równej 831 kolumna EQ_ROWS pokazuje 198, a wiemy, że jest to dokładna liczba rekordów z *ProductID* równym 831.

RANGE_ROWS to szacowana liczba rekordów, dla których wartości kolumny mieszczą się w zakresie wyznaczanym przez krok z wyłączeniem górnej granicy. W naszym przykładzie jest to liczba rekordów z wartościami od 827 do 830 (górna granica RANGE_HI_KEY, 831, nie jest brana pod uwagę). Histogram pokazuje 110 rekordów — możemy uzyskać tę samą wartość, sumując 31 rekordów z *ProductID* równym 827, 46 rekordów z *ProductID* równym 828, 0 rekordów z *ProductID* równym 829 i 33 rekordy z *ProductID* równym 830.

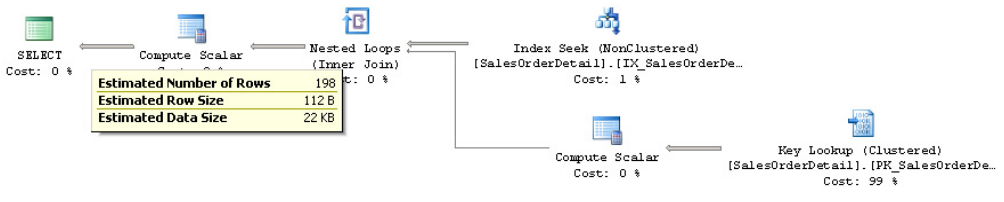
DISTINCT_RANGE_ROWS jest szacowaną liczbą rekordów z unikalną wartością kolumny w ramach zakresu, tu także z wyłączeniem górnej granicy. W naszym przykładzie mamy rekordy dla trzech unikalnych wartości(827, 828, i 830), więc wartość DISTINCT_RANGE_ROWS jest równa 3. Nie ma rekordów dla *ProductID* równego 829 i 831, co jako górna granica jest wyłączone.

Wreszcie AVG_RANGE_ROWS to średnia liczba rekordów dla unikalnej wartości z wyłączeniem górnej granicy — jest ona obliczana jako RANGE_ROWS/DISTINCT_RANGE_ROWS. W naszym przykładzie mamy 110 rekordów dla trzech DISTINCT_RANGE_ROWS, czyli $110/3 = 36,6667$ — wartość ta została przedstawiona w drugim kroku poprzednio pokazanego histogramu. Histogram zakłada, że wartość 110 będzie równo podzielona pomiędzy wszystkie trzy *ProductID*.

Zobaczmy teraz, jak histogram jest wykorzystywany do szacowania selektywności niektórych zapytań. Najpierw spójrzmy na zapytanie:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 831
```

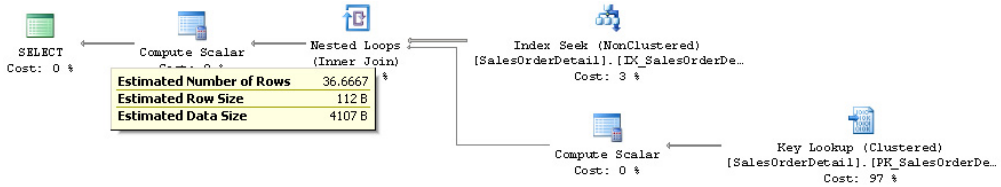
Ponieważ 831 to RANGE_HI_KEY, dla drugiego kroku poprzednio pokazanego histogramu optymalizator zapytań wykorzysta wartość EQ_ROWS (szacowana liczba rekordów, dla których wartość kolumny jest równa RANGE_HI_KEY), a szacowana liczba rekordów będzie równa 198 (zobacz rysunek 6.4).



Rysunek 6.4. Przykład szacowania kardynalności z użyciem wartości RANGE_HI_KEY

Teraz uruchom to samo zapytanie z wartością 828. Tym razem wartość należy do zakresu drugiego kroku, ale nie jest równa RANGE_HI_KEY wewnątrz histogramu, więc optymalizator wykorzystuje wartość obliczoną dla AVG_RANGE_ROWS (średnia liczba rekor-

dów dla unikalnych wartości), która wynosi 36,6667, zgodnie z histogramem. Plan dla tego zapytania został pokazany na rysunku 6.5 i, co nie dziwne, otrzymamy tę samą wartość dla każdej innej wartości wewnątrz zakresu (oczywiście poza RANGE_HI_KEY). To dotyczy także wartości 829, dla której w tabeli nie ma żadnych rekordów.



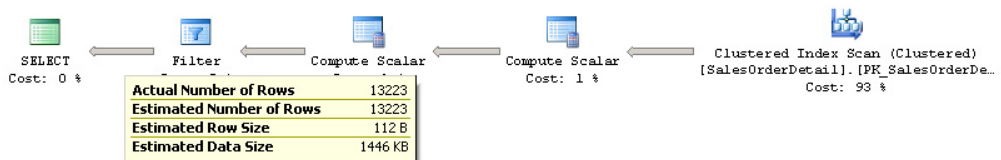
Rysunek 6.5. Przykład szacowania kardynalności z użyciem wartości AVG_RANGE_ROWS

Skorzystajmy teraz z operatora nierówności i spróbujmy znaleźć rekordy z *ProductID* mniejszym niż 714. Ponieważ to wymaga wszystkich rekordów, zarówno wewnątrz, jak i w górnej granicy kroku, musimy obliczyć sumę wartości kolumn RANGE_ROWS i EQ_ROWS dla kroków od 1 do 7, zgodnie z poniższym histogramem, co da nam 13223 rekordy:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
707	0	3083	0	1
708	0	3007	0	1
709	0	188	0	1
710	0	44	0	1
711	0	3090	0	1
712	0	3382	0	1
713	0	429	0	1
714	0	1218	0	1
715	0	1635	0	1

Poniżej znajduje się rozpatrywane zapytanie, a szacowana liczba rekordów została pokazana na rysunku 6.6:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID < 714
```



Rysunek 6.6. Przykład szacowania kardynalności z użyciem operatora nierówności

Histogramy są również wykorzystywane do szacowania kardynalności dla wielu predykatów, jednakże od SQL Servera 2014 szacunki te zależą od wersji zastosowanego szacowania kardynalności — tę kwestię omówimy w następnym podrozdziale.

Nowy mechanizm szacowania kardynalności

Jak wspomniałem we wprowadzeniu, SQL Server 2014 zawiera nowy mechanizm szacowania kardynalności, a jednocześnie jest w tej wersji dostępny także stary mechanizm. W tym podrozdziale wyjaśnię, czym jest mechanizm szacowania kardynalności, dlaczego zbudowany został nowy oraz jak włączać nowy i stary mechanizm.

Mechanizm szacowania kardynalności to komponent procesora zapytań, którego zadaniem jest szacowanie liczby rekordów zwracanych przez operacje relacyjne w zapytaniu. Te informacje, wraz z innymi danymi, są wykorzystywane przez optymalizator zapytań podczas wybierania wydajnego planu. Szacowanie kardynalności jest z natury niedokładne, ponieważ jest modelem matematycznym polegającym na informacjach statystycznych. Opiera się również na kilku założeniach, które chociaż nieudokumentowane, są znane od lat — niektóre z nich to jednolitość, niezależność, zawieranie i inkluzja. Poniżej znajduje się krótki opis tych założeń:

- ▶ **Jednolitość** — kiedy rozkład dla atrybutu nie jest znany, na przykład wewnątrz zakresu rekordów w kroku histogramu, lub kiedy histogram nie jest dostępny.
- ▶ **Niezależność** — kiedy atrybuty w relacji są niezależne, chyba że znana jest korelacja między nimi.
- ▶ **Zawieranie** — kiedy dwa atrybuty mogą być takie same; w tym przypadku zakłada się, że są takie same.
- ▶ **Inkluzja** — podczas porównywania atrybutu ze stałą; zakłada się, że zawsze są zgodne.

Aktualny mechanizm szacowania kardynalności został napisany wraz z całym procesorem zapytań w SQL Serverze 7.0, udostępnionym w 1998 roku. Oczywiście komponent ten uległ przez lata i na przestrzeni kilku wersji bazy SQL Servera wielu zmianom, które obejmowały poprawianie błędów, modyfikacje i rozszerzenia, mające dostosować szacowanie kardynalności do nowych funkcjonalności T-SQL. Możesz więc się zastanawiać: po co zmieniać komponent, który był z powodzeniem wykorzystywany przez 15 lat?

W artykule *Testing Cardinality Estimation Models in SQL Server* Campbell Fraser i inni autorzy tłumaczą niektóre powody tej zmiany, w tym następujące:

- ▶ Potrzeba dostosowania mechanizmu szacowania kardynalności do nowych wzorców wykonywanych zadań.
- ▶ Zmiany wprowadzane w mechanizmie szacowania kardynalności w ciągu tych lat sprawiły, że komponent był trudny do zrozumienia i debugowania oraz nieprzewidywalny.
- ▶ Próba ulepszenia aktualnego modelu była trudna przy użyciu aktualnej architektury, dlatego stworzona została nowa architektura, skupiająca się na separacji zadań: (a) decydowania, jak przetworzyć dany szacunek, oraz (b) wykonywania obliczeń.

Zdziwiłem się również, czytając, że autorzy przyznają, iż zgodnie z ich doświadczeniem poprzednio przyjęte założenia są „często niepoprawne”.

Pierwszą kwestią, jaka przychodzi do głowy przy tak dużej zmianie wewnątrz optymalizatora zapytań, jest regresja planów. Strach przed regresją planów jest uznawany za największą przeszkodę w ulepszeniach optymalizatora zapytań. Regresje to problemy pojawiające się po wprowadzeniu poprawki do optymalizatora zapytań. Może się tak stać, kiedy dwa błędne szacunki — na przykład jeden przeszacowujący wartość, a drugi niedoszacowujący — wykluczają się wzajemnie, zbiegiem okoliczności dając dobrą wartość. Poprawienie tylko jednej z tych wartości może prowadzić do błędnego szacunku, co może negatywnie wpłynąć na wybór planu, a tym samym spowodować regresję.

Aby uniknąć regresji związanych z nowym mechanizmem szacowania kardynalności, SQL Server udostępnia sposób na włączanie i wyłączanie go w zależności od poziomu kompatybilności bazy danych. Mechanizm można zmieniać, korzystając z polecenia `ALTER DATABASE`, jak pokazałem wcześniej. Ustawienie bazy na poziom kompatybilności równy 120 spowoduje wykorzystanie starego mechanizmu szacowania kardynalności. Ponadto kiedy już korzystasz z konkretnego mechanizmu, do jego zmiany możesz wykorzystać dwie flagi śledzenia. Flagi 2312 można użyć do włączenia nowego mechanizmu szacowania kardynalności, natomiast flagi 9481 — do jego wyłączenia. Możesz nawet użyć flag dla pojedynczego zapytania, korzystając z podpowiedzi `QUERYTRACEON`. Zarówno flagi śledzenia, jak i ich wykorzystanie za pomocą podpowiedzi `QUERYTRACEON` są udokumentowane i wspierane.



UWAGA

W tym rozdziale poziom kompatybilności bazy danych będzie zmieniany kilkakrotnie na potrzeby demonstracji zachowania starego i nowego mechanizmu. W środowisku produkcyjnym poziom kompatybilności powinien być stały i niezmienny lub bardzo rzadko zmieniany.

SQL Server zawiera też kilka nowych zdarzeń rozszerzonych, które możemy zastosować do diagnozowania problemów z szacunkiem kardynalności lub po prostu do sprawdzania jego działania. Te zdarzenia to: `query_optimizer_estimate_cardinality`, `inaccurate_cardinality_estimate`, `query_optimizer_force_both_cardinality_estimation_behaviors` i `query_rpc_set_cardinality`.

Przykłady

W tym rozdziale pokazuję różnice w szacunkach między nowym i starym mechanizmem szacowania kardynalności związane z predykatami zawierającymi słowa kluczowe `AND` i `OR`, zwanymi odpowiednio koniunkcjami i dysjunkcjami. Inne podrozdziały tego rozdziału wyjaśniają różnice w innych kwestiach (na przykład w wykorzystaniu statystyk lub kluczy rosnących).

Najpierw spójrzmy na zachowanie tradycyjne. Aby to zrobić, upewnij się, że korzystasz ze starego mechanizmu szacowania kardynalności, uruchamiając następujące polecenie na bazie AdventureWorks2012:

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 110
```

A potem uruchom poniższe zapytanie:

```
SELECT * FROM Person.Address WHERE City = 'Burbank'
```

Patrząc na plan zapytania, pokazany poprzednio, możemy zobaczyć szacunek 196 rekordów. Analogicznie poniższe polecenie spowoduje oszacowanie 194 rekordów:

```
SELECT * FROM Person.Address WHERE PostalCode = '91502'
```

Oba szacunki dla pojedynczych predykatów używają histogramu, zgodnie z wcześniejszymi wyjaśnieniami. Jeżeli użyjemy obu predykatów, otrzymamy poniższe zapytanie, które będzie miało liczbę rekordów oszacowaną na 1,93862:

```
SELECT * FROM Person.Address  
WHERE City = 'Burbank' AND PostalCode = '91502'
```

Ponieważ SQL Server nie wie nic o korelacjach pomiędzy dwoma predykatami, zakłada, że są niezależne, i wykorzystuje histogramy, tak jak pokazałem to poprzednio, do znalezienia części wspólnej obu zbiorów rekordów, mnożąc selektywność obu klauzul. Selektywność predykatu `City = 'Burbank'` jest obliczana jako $196/19614$ (gdzie 19 614 to całkowita liczba rekordów w tabeli), czyli 0,009992862. Selektywność predykatu `PostalCode = '91502'` jest obliczana jako $194/19614$, czyli 0,009890894. Aby uzyskać część wspólną tych zestawów, musimy pomnożyć selektywności obu klauzul, 0,009992862 i 0,009890894, w wyniku czego otrzymamy wartość 9,88383E-05. Na koniec obliczona selektywność jest mnożona przez całkowitą liczbę rekordów — $9,88383E-05 * 19614$ — w wyniku czego otrzymujemy szacunek: 1,93862. Bardziej bezpośredni wzór pozwalający otrzymać ten sam wynik to: $(196 * 194) / 19614$.

Zobaczmy, jak wygląda ten szacunek w przypadku użycia nowego mechanizmu szacowania kardynalności:

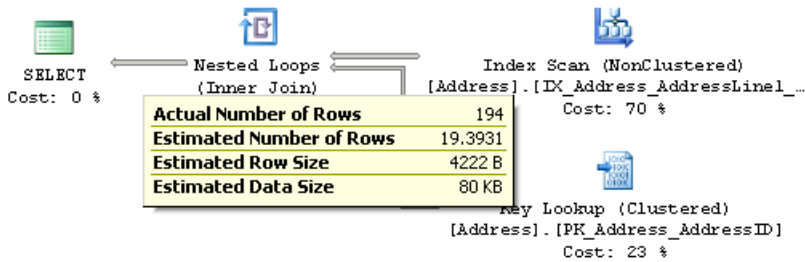
```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 120  
GO  
SELECT * FROM Person.Address WHERE City = 'Burbank' AND PostalCode = '91502'
```

Powtórne uruchomienie tego samego zapytania da szacunek równy 19,3931 rekordu, jak na rysunku 6.7.

Nowy wzór wykorzystany w tym przypadku to:

*selektywność najbardziej selektywnego filtra * SQRT(selektywność drugiego najbardziej selektywnego filtra)*

lub $(194/19614) * \text{SQRT}(196/19614) * 19614$, co daje wynik 19,393. Gdyby zapytanie miało trzy (lub więcej) predykaty, wzór zostałby rozszerzony poprzez dodanie SQRT dla każdego predykatu:



Rysunek 6.7. Predykaty z AND oszacowane za pomocą nowego mechanizmu

*selektywność najbardziej selektywnego filtra * SQRT(selektywność następnego najbardziej selektywnego filtra) * SQRT(SQRT(selektywność następnego najbardziej selektywnego filtra))*

i tak dalej.

Pamiętaj, że stary mechanizm szacowania kardynalności korzysta z założenia niezależności. Nowy mechanizm wykorzystuje nową formułę, rozluźnia założenie, które teraz nazywa się wycofaniem wykładniczym. Ta nowa formuła także nie zakłada całkowitej korelacji, ale przynajmniej nowy szacunek jest lepszy od poprzedniego. Zauważ też, że optymalizator nie może wiedzieć, czy dane są skorelowane, dlatego stosuje te formuły niezależnie od tego, czy dane są skorelowane. W naszym przykładzie dane są skorelowane — kod pocztowy 91502 związany jest z miastem Burbank w stanie Kalifornia.

Sprawdźmy teraz ten sam przykład wykorzystujący predykaty z OR, najpierw za pomocą starego mechanizmu:

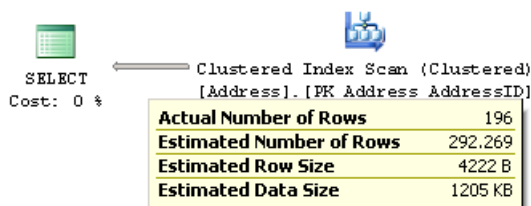
```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 110
GO
SELECT * FROM Person.Address WHERE City = 'Burbank' OR PostalCode = '91502'
```

Z definicji predykaty z OR to unia bez duplikatów zestawów rekordów obu klauzul. Oznacza to, że powinny to być rekordy dla predykatu `City = 'Burbank'` plus rekordy szacowane dla `PostalCode = '91502'`, ale jeżeli istnieją jakieś rekordy należące do obu zbiorów, powinny zostać zaliczone tylko raz. Jak wskazałem w poprzednim przykładzie, szacowana liczba rekordów dla predykatu `City = 'Burbank'` to 196, a dla predykatu `PostalCode = '91502'` to 194. Szacowana liczba rekordów należących do obu zbiorów to szacowana liczba rekordów z predykatem AND: 1,93862. Zatem szacowana liczba rekordów z predykatem OR to $196 + 194 - 1,93862$, czyli 388,061.

Sprawdzenie tego samego przykładu dla nowego mechanizmu szacowania kardynalności zwróci szacunek 292,269 rekordu (zobacz rysunek 6.8).

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 120
GO
SELECT * FROM Person.Address WHERE City = 'Burbank' OR PostalCode = '91502'
```

Wzór na obliczenie tego predykatu został zasugerowany w artykule *Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator*, w którym wskazuje się, że SQL



Clustered Index Scan (Clustered) [Address]. [PK Address AddressID]	
Actual Number of Rows	196
Estimated Number of Rows	292,269
Estimated Row Size	4222 B
Estimated Data Size	1205 KB

Rysunek 6.8. Predykaty z OR w nowym mechanizmie szacowania kardynalności

Server „oblicza tę wartość najpierw transformując dysjunkcję do negacji koniunkcji”. W tym przypadku możemy zmienić poprzednią formułę na następującą: $(1 - (1 - (196 / 19614)) * \text{SQRT}(1 - (194 / 19614))) * 19614$, co daje wynik 292,269.

Jeżeli włączyłeś nowy mechanizm szacowania kardynalności na poziomie bazy danych, ale chcesz wyłączyć go dla pojedynczego zapytania, aby uniknąć regresji planu, możesz skorzystać z flagi 9481, zgodnie z wcześniejszymi wyjaśnieniami:

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 120
GO
SELECT * FROM Person.Address WHERE City = 'Burbank' AND PostalCode = '91502'
OPTION (QUERYTRACEON 9481)
```



UWAGA

Jak pokazałem w rozdziale 1., odpowiedź QUERYTRACEON jest wykorzystywana do wywołania odpowiedzi na poziomie zapytania, ale na razie wspierana jest tylko przez ograniczoną liczbę scenariuszy, włączając w to wspomniane w tym podrozdziale flagi 2312 i 9481.

Od wersji 2014 dostępne są dwa mechanizmy szacowania kardynalności, dlatego potrzebny będzie sposób na określenie, który z nich został wykorzystany przy optymalizacji zapytania, szczególnie podczas diagnozowania problemów. Możesz znaleźć taką informację na planach wykonania, spoglądając na właściwość `CardinalityEstimationModelVersion` (wersja modelu szacowania kardynalności) planu graficznego lub na atrybut `CardinalityEstimationModelVersion` elementu `StmtSimple` w planie XML, zgodnie z poniższym fragmentem:

```
<StmtSimple ... CardinalityEstimationModelVersion="120" ...>
```

Wartość 120 oznacza, że wykorzystany został nowy mechanizm szacowania kardynalności. Wszystkie inne poziomy kompatybilności bazy danych dostępne w SQL Serverze 2014, takie jak 90, 100 i 110, spowodują zwrócenie wartości 70 dla `CardinalityEstimationModelVersion`, co oznacza stary mechanizm szacowania kardynalności.

Flaga 4137

Flaga 4137 może być pomocna w przypadku skorelowanych predykatów ze słowem kluczowym AND. Flaga ta (która pierwotnie została wprowadzona jako poprawka do SQL Server 2008 Service Pack 2, SQL Server 2008 R2 Service Pack 1 i SQL Server 2012 RTM) spowoduje wybranie najniższego szacunku kardynalności listy predykatów połączonych AND. Na przykład poniższy kod pokaże szacowaną liczbę 194 rekordów, ponieważ 194 to najmniejszy wynik obu szacunków kardynalności dla predykatów City = 'Burbank' i PostalCode = '91502', zgodnie z wcześniejszymi wyjaśnieniami:

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 110
GO
SELECT * FROM Person.Address
WHERE City = 'Burbank' AND PostalCode = '91502'
OPTION (QUERYTRACEON 4137)
```



UWAGA

Korzystanie z flagi 4137 wraz z podpowiedzią QUERYTRACEON jest udokumentowane i wspierane.

Aby jeszcze trochę skomplikować sprawę, flaga 4137 nie działa dla nowego mechanizmu szacowania kardynalności i konieczne jest skorzystanie z flagi 9471. Kolejną różnicą w przypadku flagi 9471 jest to, że wpływa zarówno na predykaty koniunkcyjne, jak i dysjunkcyjne, podczas gdy flaga 4137 działa tylko dla predykatów koniunkcyjnych.

Statystyki filtrowane, omawiane w dalszej części tego rozdziału, mogą być pomocne w przypadkach, gdy dane są mocno skorelowane.

Błędy szacunku kardynalności

Błędy w szacunku kardynalności mogą prowadzić do błędnych decyzji podejmowanych przez optymalizator zapytań, a co za tym idzie, do mało wydajnych planów wykonania. Na szczęście można łatwo sprawdzić, czy występują błędy w szacowaniu kardynalności, porównując wartość szacowaną z rzeczywistą liczbą rekordów pokazaną w planach graficznych lub XML lub korzystając z polecenia SET STATISTICS PROFILE. W następnym zapytaniu pokażę, jak korzystać z polecenia SET STATISTICS PROFILE z jednym z poprzednich przykładów, w którym SQL Server zgaduje selektywność pewnych kolumn:

```
SET STATISTICS PROFILE ON
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE OrderQty * UnitPrice > 10000
```

```
GO
SET STATISTICS PROFILE OFF
GO
```

Oto wynik — kolumna *EstimateRows* została ręcznie przeniesiona za kolumnę *Rows*, a tekst wyedytowany tak, aby zmieścił się na stronie:

Rows	EstimateRows	StmtText
772	36395.1	SELECT * FROM [Sales].[SalesOrderDetail] WHERE [OrderQty] * [U
772	36395.1	--Filter(WHERE:([Expr1003]>(\$10000.0000)))
0	121317	--Compute Scalar(DEFINE:([AdventureWorks2012].[Sales]
0	121317	--Compute Scalar(DEFINE:([AdventureWorks2012].[S
121317	121317	--Clustered Index Scan(OBJECT:([AdventureWo

Korzystając z tych danych, można dla każdego operatora w planie łatwo porównać właściwą liczbę rekordów, pokazaną w kolumnie *Rows*, z szacowaną liczbą rekordów, pokazaną w kolumnie *EstimateRows*. Zdarzenie rozszerzone *inaccurate_cardinality_estimate*, wprowadzone w wersji 2012, również może zostać wykonane do wykrywania błędów w szacowaniu kardynalności poprzez wyszukanie zapytań, które zwracają znacząco więcej wierszy, niż oszacował to optymalizator zapytań.

Ponieważ każdy operator jako dane wsadowe przyjmuje dane wyjściowe poprzedniego operatora, błędy w szacowaniu kardynalności mogą wzrastać wykładniczo w całym planie. Na przykład błąd w szacunku kardynalności dla operatora *Filter* może mieć wpływ na szacunek kardynalności wszystkich operatorów przyjmujących dane tego operatora. Jeżeli Twoje zapytanie nie jest wydajne i znajdziesz błędy w szacowaniu kardynalności, poszukaj problemów takich jak brakujące lub nieaktualne statystyki, wykorzystanie bardzo małych próbek, korelacja pomiędzy kolumnami, wykorzystanie wyrażeń skalarnych, zgadywanie selektywności i tak dalej.

Zalecenia, jak radzić sobie z tymi problemami, zostały przedstawione w tym rozdziale i dotyczą między innymi takich tematów jak domyślna konfiguracja automatycznie tworzonych i aktualizowanych statystyk, aktualizacje statystyk za pomocą polecenia *WITH FULLSCAN*, unikanie zapytań ze zmiennymi lokalnymi, unikanie zmiennych lub skomplikowanych wyrażeń w predykatkach, wykorzystanie kolumn wyliczeniowych oraz rozważenie zastosowania statystyk wielokolumnowych lub filtrowanych. Ponadto zapytania wyczuwające parametry i czułe na parametry zostały dokładnie omówione w rozdziale 8. To dość długa lista, ale powinna pomóc przekonać Cię, że zostałeś uzbrojony w użyteczne informacje.

Niektóre funkcjonalności SQL Servera, takie jak zmienne tablicowe, nie mają statystyk, więc jeżeli napotkasz związane z nimi problemy wydajnościowe lub błędy w szacowaniu kardynalności, możesz rozważyć zastąpienie ich tabelą tymczasową lub standardową tabelą. Wielopoleceniowe funkcje, definiowane przez użytkownika i zwracające zmienne tablicowe, również nie mają statystyk. W takim przypadku możesz rozważyć wykorzystanie tabeli tymczasowej lub standardowej tabeli jako miejsca tymczasowo przechowywanego dane wynikowe. W obu przypadkach (zmiennych tablicowych i funkcji definiowanych przez użytkownika zwracających zmienne tabli-

cowe) optymalizator zgadnie jeden wiersz (co zostało zmienione na 100 wierszy dla funkcji w SQL Serverze 2014). Ponadto dla skomplikowanych zapytań, które nie są wydajne przez błędy w szacowaniu kardynalności, możesz rozważyć rozbić zapytania na dwa lub więcej kroków, przechowując wyniki pośrednie w tabeli tymczasowej. Pozwoli to, aby SQL Server tworzył statystyki dla wyników pośrednich, co pomoże optymalizatorowi stworzyć lepszy plan wykonania. Więcej szczegółów na temat rozbijania skomplikowanych zapytań przedstawię w rozdziale 10.



UWAGA

W SQL Server 2012 Service Pack 2 została wprowadzona nowa flaga śledzenia, która pozwala uzyskać lepsze szacunki kardynalności podczas korzystania ze zmiennych tablicowych. Flaga ta prawdopodobnie zostanie również zaimplementowana w SQL Serverze 2014. Więcej szczegółów znajdziesz pod adresem <http://support.microsoft.com/kb/2952444>.

Statystyki inkrementacyjne

Głównym problemem przy aktualizacji statystyk dla dużych tabel w SQL Serverze było to, że nawet jeżeli zmienione zostały tylko najświeższe dane, próbkowana musiała być cała tabela. Dotyczyło to również sytuacji, gdy stosowane było partycjonowanie: nawet jeżeli od ostatniej aktualizacji statystyk zmiany zostały dokonane tylko dla najnowszej partycji, aktualizacja statystyk wymagała próbkowania całej tabeli, włącznie z partycjami, które nie były zmieniane. Pomóc mogą statystyki inkrementacyjne, nowa funkcjonalność SQL Servera 2014.

Używając statystyk inkrementacyjnych, możesz aktualizować tylko statystyki dla partycji, które zostały zmienione, a informacje na nich zawarte będą scalone z istniejącymi informacjami w celu utworzenia ostatecznego obiektu statystyk. Kolejną zaletą statystyk inkrementacyjnych jest to, że wartość procentowa danych, które muszą ulec zmianie, aby wyzwolona została aktualizacja statystyk, obowiązuje teraz na poziomie partycji, co oznacza, że wymagana jest zmiana tylko 20% danych (dla głównej kolumny statystyki) dla partycji. Niestety w tej wersji SQL Servera histogram nadal ograniczony jest do 200 kroków dla całego obiektu statystyki.

Spójrzmy na przykład pokazujący, jak zaktualizować statystyki na poziomie partycji. Najpierw musimy stworzyć tabelę partycjonowaną, korzystając z bazy AdventureWorks2012:

```
CREATE PARTITION FUNCTION TransactionRangePF1 (datetime)
AS RANGE RIGHT FOR VALUES
(
    '20071001', '20071101', '20071201', '20080101',
    '20080201', '20080301', '20080401', '20080501',
    '20080601', '20080701', '20080801'
)
GO
```

```

CREATE PARTITION SCHEME TransactionsPS1 AS PARTITION TransactionRangePF1 TO
(
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY]
)
GO
CREATE TABLE dbo.TransactionHistory
(
    TransactionID int NOT NULL,
    ProductID int NOT NULL,
    ReferenceOrderID int NOT NULL,
    ReferenceOrderLineID int NOT NULL DEFAULT (0),
    TransactionDate datetime NOT NULL DEFAULT (GETDATE()),
    TransactionType nchar(1) NOT NULL,
    Quantity int NOT NULL,
    ActualCost money NOT NULL,
    ModifiedDate datetime NOT NULL DEFAULT (GETDATE()),
    CONSTRAINT CK_TransactionType
    CHECK (UPPER(TransactionType) IN (N'W', N'S', N'P'))
)
ON TransactionsPS1 (TransactionDate)
GO

```



UWAGA

Więcej szczegółów na temat partycjonowania i poleceń CREATE PARTITION FUNCTION/SCHEME znajdziesz w rozdziale „Partitioned Tables and Indexes” dokumentacji Books Online.

Mamy aktualnie dane do zasilenia 12 partycji. Na razie zacznijmy od zapełnienia 11:

```

INSERT INTO dbo.TransactionHistory
SELECT * FROM Production.TransactionHistory
WHERE TransactionDate < '2008-08-01'

```

Jeżeli chcesz, za pomocą poniższego zapytania możesz sprawdzić zawartość partycji:

```

SELECT * FROM sys.partitions
WHERE object_id = OBJECT_ID('dbo.TransactionHistory')

```

Stwórzmy inkrementacyjny obiekt statystyk, korzystając z polecenia CREATE STATISTICS z nową klauzulą INCREMENTAL ustawioną na ON (OFF to wartość domyślna):

```

CREATE STATISTICS incrstats ON dbo.TransactionHistory(TransactionDate)
WITH FULLSCAN, INCREMENTAL = ON

```

Możesz również stworzyć statystyki inkrementacyjne podczas tworzenia indeksu, korzystając z nowej klauzuli STATISTICS_INCREMENTAL polecenia CREATE INDEX. Możesz sprawdzić stworzony obiekt statystyk za pomocą poniższego zapytania:

```

DBCC SHOW_STATISTICS('dbo.TransactionHistory', incrstats)

```

Zauważ, że stworzony histogram ma 200 kroków (tutaj pokazane zostały tylko ostatnie 3):

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS
198	2008-07-25 00:00:00.000	187	100	2
199	2008-07-27 00:00:00.000	103	101	1
200	2008-07-31 00:00:00.000	281	131	3

Mamy więc już maksymalną liczbę kroków w obiekcie statystyk. Co się stanie, jeżeli dodamy dane do nowej partycji? Dodajmy dane do partycji 12.:

```
INSERT INTO dbo.TransactionHistory
SELECT * FROM Production.TransactionHistory
WHERE TransactionDate >= '2008-08-01'
```

Teraz zaktualizujemy obiekt statystyk za pomocą następującego polecenia:

```
UPDATE STATISTICS dbo.TransactionHistory(incrstats)
WITH RESAMPLE ON PARTITIONS(12)
```

Zwróć uwagę na nową składnię specyfikującą partycję, gdzie możesz podać wiele partycji, oddzielonych przecinkami. Polecenie `UPDATE STATISTICS` odczytuje wyszczególnione partycje i scala ich rezultaty z istniejącym obiektem statystyk w celu zbudowania statystyki globalnej. Zwróć uwagę na klauzulę `RESAMPLE` — jest wymagana, ponieważ obiekty statystyk partycji muszą mieć taki sam współczynnik próbkowania, aby mogły zostać scalone. Chociaż skanowana była tylko wyszczególniona partycja, możesz zobaczyć, że SQL Server przebudował histogram. Ostatnie trzy kroki pokazują teraz dane dla dodanej partycji. Możesz również porównać oryginalny histogram z tym zmienionym, a zauważysz kilka innych drobnych różnic:

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS
197	2008-07-31 00:00:00.000	150	131	2
198	2008-08-12 00:00:00.000	300	36	9
199	2008-08-22 00:00:00.000	229	43	7
200	2008-09-03 00:00:00.000	363	37	11

Jeżeli z jakiegoś powodu chcesz wyłączyć inkrementacyjny obiekt statystyk, możesz wykorzystać poniższe polecenie, aby powrócić do pierwotnego zachowania (lub po prostu usunąć obiekt i stworzyć nowy):

```
UPDATE STATISTICS dbo.TransactionHistory(incrstats)
WITH FULLSCAN, INCREMENTAL = OFF
```

Po wyłączeniu statystyk inkrementacyjnych poprzednio pokazana próba zaktualizowania partycji zwróci następujący błąd:

```
Msg 9111, Level 16, State 1, Line 1
UPDATE STATISTICS ON PARTITIONS syntax is not supported for non-incremental statistics
```

Możesz również włączyć statystyki inkrementacyjne dla statystyk automatycznych na poziomie bazy danych — za pomocą klauzuli `INCREMENTAL = ON` polecenia `ALTER DATABASE`, co oczywiście wymaga również włączonej opcji `AUTO_CREATE_STATISTICS`.

Aby posprzątać obiekty stworzone na potrzeby tego ćwiczenia, uruchom poniższe polecenia:

```
DROP TABLE dbo.TransactionHistory
DROP PARTITION SCHEME TransactionsPS1
DROP PARTITION FUNCTION TransactionRangePF1
```

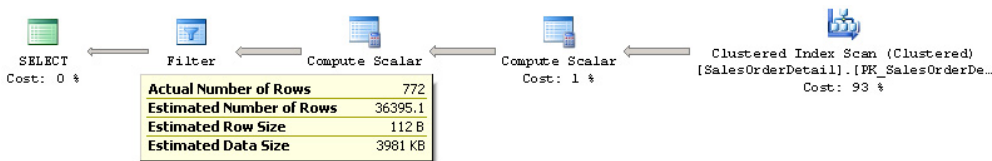
Statystyki dla kolumn wyliczeniowych

Kolejnym interesującym krokiem podczas optymalizacji zapytań jest automatyczne dopasowywanie kolumn wyliczeniowych. Chociaż kolumny wyliczeniowe były dostępne w poprzednich wersjach SQL Servera, funkcjonalność automatycznego dopasowywania została wprowadzona dopiero w wersji 2005. W tym podrozdziale pokażę, jak działa ta funkcjonalność, i wyjaśnię, w jaki sposób kolumny wyliczeniowe mogą pomóc w poprawie wydajności zapytań.

Problem, z jakim boryka się część zapytań wykorzystujących wyrażenia skalarne, polega na tym, że z reguły nie mogą one korzystać ze statystyk, a bez statystyk optymalizator dla wyrażeń nierówności uzna stałą selektywności na poziomie 30%, przez co wynikowe plany zapytania mogą być nieefektywne. Rozwiązaniem tego problemu jest wykorzystanie kolumn wyliczeniowych, ponieważ SQL Server może automatycznie tworzyć i aktualizować statystyki dla tych kolumn. Wielką zaletą tego rozwiązania jest to, że nie musisz podawać nazwy kolumny wyliczeniowej w zapytaniach, aby SQL Server korzystał ze statystyk. Optymalizator zapytań automatycznie dopasowuje definicję kolumny wyliczeniowej do wyrażenia skalarne w zapytaniu, więc Twoje aplikacje nie muszą być zmieniane.

Aby zobaczyć przykład, uruchom to zapytanie, które stworzy plan przedstawiony na rysunku 6.9:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE OrderQty * UnitPrice > 10000
```



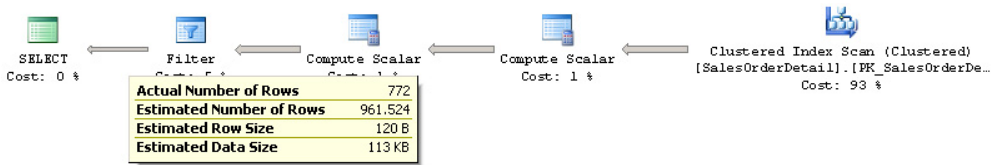
Rysunek 6.9. Przykład szacowania kardynalności z użyciem stałej wartości 30%

Szacowana liczba rekordów to 36395,1, co stanowi 30% całkowitej liczby rekordów (121317), chociaż zapytanie zwraca tylko 772 rekordy. SQL Server oczywiście wykorzystuje stałą wartość selektywności, ponieważ nie jest w stanie oszacować selektywności wyrażenia $\text{OrderQty} * \text{UnitPrice} > 10000$.

Stwórz teraz kolumnę wyliczeniową:

```
ALTER TABLE Sales.SalesOrderDetail
ADD cc AS OrderQty * UnitPrice
```

Uruchom poprzednie zapytanie i zauważ, że tym razem szacowana liczba rekordów uległa zmianie i jest bliska właściwej liczbie zwracanych przez zapytanie rekordów (zobacz rysunek 6.10). Możesz przetestować zmianę wartości 10000 na jakąś inną, na przykład 10, 100, 1000, 5000, i porównać właściwą i szacowaną liczbę rekordów.



Rysunek 6.10. Przykład szacowania kardynalności z użyciem kolumn wyliczeniowych

Zwróć uwagę, że tworzenie kolumny wyliczeniowej nie powoduje stworzenia statystyki, ponieważ statystyki są tworzone przy pierwszej optymalizacji zapytania i możesz uruchomić poniższe zapytanie, aby wyświetlić informacje dotyczące obiektów statystyk dla tabeli Sales.SalesOrderDetail:

```
SELECT * FROM sys.stats
WHERE object_id = OBJECT_ID('Sales.SalesOrderDetail')
```

Nowo stworzony obiekt statystyk znajdzie się na końcu listy. Skopiuj nazwę obiektu i użyj poniższego polecenia, aby wyświetlić szczegóły na temat obiektu (wykorzystałem nazwę mojego lokalnego obiektu, ale powinieneś zmienić ją na swój obiekt):

```
DBCC SHOW_STATISTICS ('Sales.SalesOrderDetail', _WA_Sys_0000000E_44CA3770)
```

Jako nazwę obiektu możesz również wykorzystać cc i otrzymasz te same efekty. Kolumna cc powinna być pokazana w polu Columns w sekcji dotyczącej gęstości. Tak czy inaczej, liczba rekordów jest szacowana z użyciem histogramu stworzonego obiektu statystyk, zgodnie z wyjaśnieniami dotyczącymi porównywania wyrażeń nierówności.

```
DBCC SHOW_STATISTICS ('Sales.SalesOrderDetail', cc)
```

Niestety, aby zadziałało automatyczne dopasowanie, wyrażenie musi być dokładnie takie samo jak definicja kolumny wyliczeniowej. Dlatego jeżeli zmienię wyrażenie na $\text{UnitPrice} * \text{OrderQty}$ zamiast $\text{OrderQty} * \text{UnitPrice}$, plan wykonania znów pokaże szacunek w wysokości 30%, jak dla poniższego zapytania:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE UnitPrice * OrderQty > 10000
```

Usuń teraz kolumnę wyliczeniową:

```
ALTER TABLE Sales.SalesOrderDetail
DROP COLUMN cc
```

Statystyki filtrowane

Statystyki filtrowane to statystyki tworzone dla podzbioru rekordów w tabeli. Są automatycznie tworzone, kiedy tworzony jest indeks filtrowany, ale można je również stworzyć ręcznie za pomocą klauzuli `WHERE` polecenia `CREATE STATISTICS` i w takim przypadku indeks filtrowany nie jest wymagany. Jak możesz sobie wyobrazić, statystyki filtrowane mogą pomóc w zapytaniach uzyskujących dostęp do konkretnego podzbioru danych. Mogą również być przydatne na przykład w przypadku kolumn skorelowanych, w szczególności jeżeli jedna z tych kolumn ma niewielką liczbę wartości unikalnych i możesz stworzyć kilka statystyk filtrowanych dla każdej z tych wartości. Jak pokazałem w podrozdziale „Histogram”, przy korzystaniu z wielu predykatów SQL Server zakłada, że każda klauzula w zapytaniu jest niezależna. Gdyby kolumny wykorzystane w tym zapytaniu były skorelowane, szacunek kardynalności byłby niepoprawny. Nawet w przypadku wycofania wykładniczego stosowanego przez nowy mechanizm szacowania kardynalności możesz wciąż uzyskać lepszy szacunek za pomocą statystyk filtrowanych, ponieważ każdy obiekt statystyk ma swój własny histogram. Statystyki filtrowane mogą również pomóc w przypadku ogromnych tabel, dla których duża liczba unikalnych wartości nie jest odpowiednio reprezentowana przez ograniczenie histogramu do 200 kroków.

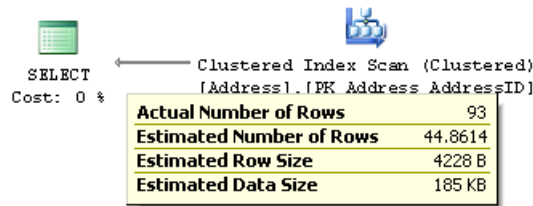
Pokażę teraz, jak skorzystać ze statystyk filtrowanych w przypadku problemu z kolumnami skorelowanymi. Uruchomienie poniższego zapytania poprawnie oszacuje liczbę rekordów na 93:

```
SELECT * FROM Person.Address
WHERE City = 'Los Angeles'
```

Podobnie dla poniższego zapytania liczba rekordów zostanie poprawnie oszacowana na 4564:

```
SELECT * FROM Person.Address
WHERE StateProvinceID = 9
```

Ponieważ jednak `StateProvinceID = 9` odnosi się do stanu Kalifornia (co możesz potwierdzić w tabeli `Person.StateProvince`), możliwe jest, że ktoś uruchomi poniższe zapytanie, które zwróci znacznie mniej dokładny szacunek 44,8614 wiersza (wynik ten uzyskiwany jest za pomocą nowego mechanizmu szacowania kardynalności, a z użyciem starego wynik to 21,6403), zgodnie z planem z rysunku 6.11:



Rysunek 6.11. Szacowanie kardynalności z wycofaniem wykładniczym

```
SELECT * FROM Person.Address
WHERE City = 'Los Angeles' AND StateProvinceID = 9
```

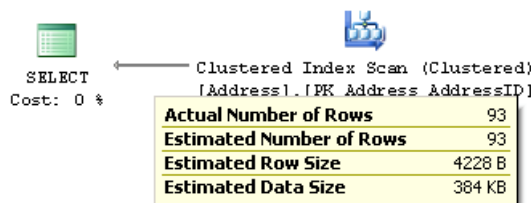
Przez założenie niezależności SQL Server pomnoży kardynalność obu predykatów, zgodnie z wcześniejszymi wyjaśnieniami. Obliczenie w postaci $(93 \cdot 4564) / 19614$ da wynik 21,6403 dla starego mechanizmu szacowania kardynalności (19614 to całkowita liczba rekordów w tabeli). W przypadku wycofania wykładniczego wykorzystywanego przez nowy mechanizm wyliczenie to $(93/19614) \cdot \text{SQRT}(4564/19614) \cdot 19614$, co w przybliżeniu daje 44,861 rekordu.

Zarówno jednak założenie niezależności, jak i wycofanie wykładnicze są w tym przykładzie niepoprawne, ponieważ kolumny są statystycznie skorelowane. Aby poradzić sobie z tym problemem, możesz stworzyć statystyki filtrowane dla stanu Kalifornia, zgodnie z poniższym poleceniem:

```
CREATE STATISTICS california
ON Person.Address(City)
WHERE StateProvinceID = 9
```

Wyczyszczenie magazynu planów i powtórne uruchomienie poprzedniego zapytania da znacznie lepszy szacunek (zobacz rysunek 6.12).

```
DBCC FREEPROCCACHE
GO
SELECT * FROM Person.Address
WHERE City = 'Los Angeles' AND StateProvinceID = 9
```



Rysunek 6.12. Szacunek kardynalności ze statystykami filtrowanymi

Przjrzyjmy się teraz obiektowi statystyk filtrowanych. Uruchom następujące polecenie:

```
DBCC SHOW_STATISTICS('Person.Address', california)
```

Uzyskasz poniższy wynik (wyedytowany ze względu na ograniczoną ilość miejsca), pokazujący w kolumnie EQ_ROWS szacowane 93 rekordy dla Los Angeles:

Name	Rows	Rows Sampled	Filter Expression	Unfiltered Rows
california	4564	4564	([StateProvinceID]=(9))	19614

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
Alhambra	0	1	0	1
Alpine	0	1	0	1
Altadena	0	2	0	1
Auburn	0	1	0	1
Baldwin Park	0	1	0	1
Barstow	0	2	0	1
...				
Los Angeles	0	93	0	1

Zauważ, że definicja filtra pokazana jest w polu Filter Expression i że pole Unfiltered Rows pokazuje całkowitą liczbę rekordów w tabeli w momencie utworzenia statystyki filtrowanej. Zauważ też, że tym razem wartość kolumny Rows jest niższa niż całkowita liczba rekordów w tabeli i nawiązuje do liczby rekordów odpowiadających predykatowi filtra w momencie tworzenia obiektu statystyki. Definicja filtra znajduje się również w kolumnie filter_definition katalogu sys.stats. Zauważ także, że histogram pokazuje tylko miasta w Kalifornii, zgodnie z definicją filtra (pokazane zostało tylko kilka kroków).

Usuń teraz obiekt statystyk poprzez uruchomienie poniższego polecenia:

```
DROP STATISTICS Person.Address.california
```

Statystyki dla kluczy rosnących

Statystyki dla kluczy rosnących reprezentują problem szacowania kardynalności, który istnieje we wszystkich wersjach SQL Servera od wersji 7. Najlepszym rozwiązaniem tego problemu było użycie flag 2389 i 2390. Od wersji 2014 możesz również wykorzystać nowy mechanizm szacowania kardynalności, pozwalający uzyskać dokładnie te same informacje bez konieczności stosowania flag.

Najpierw jednak pozwól mi wyjaśnić, na czym polega problem. Jak już wcześniej widzieliśmy, SQL Server buduje histogram na pierwszej kolumnie każdego obiektu statystyk. W przypadku statystyk dla rosnących lub malejących kluczy, takich jak IDENTITY, lub kolumn ze stemplem czasowym, nowo wstawione wartości zazwyczaj wpadają poza zakres wartości przechowywanych w histogramie. Ponadto liczba dodanych wartości może być zbyt mała, aby wyzwolić automatyczną aktualizację statystyk. Ponieważ ostatnio dodane rekordy nie są uwzględnione w histogramie, gdy znacząca ich liczba powinna być w nim uwzględniona, uruchomienie zapytania wykorzystującego te wartości może zaowocować niepoprawnymi szacunkami kardynalności, co z kolei może spowodować zwracanie nieoptymalnych planów.

Tradycyjna rekomendacja Microsoftu polegała na ręcznym aktualizowaniu statystyk po wstawieniu danych, niestety jednak może to również wiązać się z częstszymi aktualizacjami, co nie zawsze może być pożądane. Aby pomóc w rozwiązaniu tego problemu, w SQL Server 2005 Service Pack 1 zostały wprowadzone flagi 2389 i 2390, przedstawione przez Iana Jose z Microsoftu w artykule *Ascending Keys and Auto Quick Corrected Statistic*.

Aby pokazać, na czym polega problem i jak działają flagi, zacznijmy od stworzenia tabeli w bazie AdventureWorks2012. Najpierw jednak upewnijmy się, że korzystasz ze starego mechanizmu szacowania kardynalności:

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 110
GO
CREATE TABLE dbo.SalesOrderHeader (
    SalesOrderID int NOT NULL,
    RevisionNumber tinyint NOT NULL,
    OrderDate datetime NOT NULL,
    DueDate datetime NOT NULL,
    ShipDate datetime NULL,
    Status tinyint NOT NULL,
    OnlineOrderFlag dbo.Flag NOT NULL,
    SalesOrderNumber nvarchar(25) NOT NULL,
    PurchaseOrderNumber dbo.OrderNumber NULL,
    AccountNumber dbo.AccountNumber NULL,
    CustomerID int NOT NULL,
    SalesPersonID int NULL,
    TerritoryID int NULL,
    BillToAddressID int NOT NULL,
    ShipToAddressID int NOT NULL,
    ShipMethodID int NOT NULL,
    CreditCardID int NULL,
    CreditCardApprovalCode varchar(15) NULL,
    CurrencyRateID int NULL,
    SubTotal money NOT NULL,
    TaxAmt money NOT NULL,
    Freight money NOT NULL,
    TotalDue money NOT NULL,
    Comment nvarchar(128) NULL,
    rowguid uniqueidentifier NOT NULL,
    ModifiedDate datetime NOT NULL
)
```

Zasil tabelę danymi i stwórz dla niej indeks (zauważ, że obie tabeli mają tę samą nazwę, ale są w różnych schematach: dbo i Sales):

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate < '2008-07-20 00:00:00.000'
CREATE INDEX IX_OrderDate ON SalesOrderHeader(OrderDate)
```

Po stworzeniu indeksu SQL Server utworzy również obiekt statystyk, dlatego poniższe zapytanie będzie miało poprawny szacunek kardynalności równy 35 rekordów (ponieważ istnieją dane dla 19 czerwca i dane te są uwzględnione w ostatnim korku histogramu, co możesz zweryfikować za pomocą polecenia DBCC SHOW_STATISTICS):

```
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-19 00:00:00.000'
```

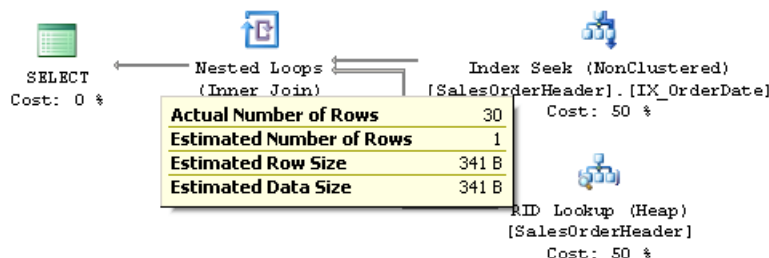
Załóżmy teraz, że dodamy nowe dane dla 20 czerwca:

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate = '2008-07-20 00:00:00.000'
```

Zmieńmy też zapytanie, aby szukało danych z tej daty:

```
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-20 00:00:00.000'
```

Ponieważ liczba dodanych rekordów nie jest wystarczająca, aby wyzwolić automatyczną aktualizację statystyk, wartość z 20 czerwca nie jest reprezentowana w istniejącym histogramie. Oznacza to, że SQL Server oszacuje liczbę rekordów na 1 (zobacz rysunek 6.13).



Rysunek 6.13. Zły szacunek kardynalności po dodaniu nowych rekordów

Chociaż plany dla obu zapytań są bardzo podobne, błędny szacunek kardynalności może w przypadku bardziej realistycznych scenariuszy spowodować powstanie złego planu.

Flaga 2389

Zobaczmy teraz, w jaki sposób flaga 2389 może pomóc w rozwiązaniu tego problemu. Uruchom poniższe polecenia (zauważ, że nie wspominałem wcześniej o flagze 2388 — objaśnię ją w dalszej części):

```
DBCC TRACEON (2388)
DBCC TRACEON (2389)
```

Flaga 2389, wprowadzona w SQL Server 2005 Service Pack 1, zaczyna śledzić charakterystykę kolumn na podstawie kolejnych operacji aktualizacji statystyk. Jeżeli mechanizm zaobserwuje, że statystyki wzrastają trzy razy z rzędu, kolumna zostanie oznakowana jako *Ascending* („rosnąca”). Flaga 2388 nie jest wymagana, aby włączyć opisane w tym podrozdziale zachowanie, ale umożliwia wyświetlenie poprzednio ukrytych metadanych, które będą przydatne, aby pokazać, jak działają flagi 2389 i 2390, i pozwolą sprawdzić, czy kolumna została oznakowana jako *rosnąca*. Flaga zmienia

wynik działania polecenia `DBCC SHOW_STATISTICS`, tak aby dać Ci wgląd w historię ostatnich operacji aktualizacji statystyk.

Flaga 2390 udostępnia podobne zachowanie jak flaga 2389, nawet jeżeli rosnący charakter kolumny nie jest znany, ale nie będę jej omawiał. Uruchom polecenie `DBCC SHOW_STATISTICS`:

```
DBCC SHOW_STATISTICS ('dbo.SalesOrderHeader', 'IX_OrderDate')
```

Wynik będzie następujący (wyedytowany ze względu na ograniczoną ilość miejsca):

Updated	Rows Above	Rows Below	Inserts Since Last Update	Deletes Since Last Update	Leading column Type
Jan 11 2014 2:12PM	NULL	NULL	NULL	NULL	Unknown

Na razie jest to niewiele danych. Jednakże pokażę Ci te dane po trzech kolejnych wstawieniach danych i aktualizacjach obiektu statystyk. Uruchom poniższe polecenie, aby zaktualizować statystyki danymi z 20 czerwca, które wstawiłeś ostatnio:

```
UPDATE STATISTICS dbo.SalesOrderHeader WITH FULLSCAN
```

`DBCC SHOW_STATISTICS` pokaże teraz:

Updated	Rows Above	Rows Below	Inserts Since Last Update	Deletes Since Last Update	Leading column Type
Jan 11 2014 2:13PM	30	0	30	0	Unknown
Jan 11 2014 2:12PM	NULL	NULL	NULL	NULL	NULL

Pola *Rows Above* i *Inserts Since Last Update* zawierają 30 poprzednio dodanych wierszy (być może będziesz musiał przewinąć wynik w prawo). Teraz uruchom drugie wstawienie:

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate = '2008-07-21 00:00:00.000'
```

Uruchomienie tego zapytania potwierdzi, że SQL Server nadal szacuje kardynalność na 1 wiersz:

```
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-21 00:00:00.000'
```

Teraz znów zaktualizuj statystyki:

```
UPDATE STATISTICS dbo.SalesOrderHeader WITH FULLSCAN
```

`DBCC SHOW_STATISTICS` pokaże teraz poniższe informacje. Zwróć uwagę na nowy wiersz z polami *Rows Above* i *Inserts Since Last Update* zawierającymi wartość 27. Typ kolumny głównej to nadal *Unknown* (nieznany).

Updated	Rows Above	Rows Below	Inserts Since Last Update	Deletes Since Last Update	Leading column Type
Jan 11 2014 2:15PM	27	0	27	0	Unknown
Jan 11 2014 2:13PM	30	0	30	0	NULL
Jan 11 2014 2:12PM	NULL	NULL	NULL	NULL	NULL

Czas na trzecią porcję danych:

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate = '2008-07-22 00:00:00.000'
```

I ostatnią aktualizację statystyk:

```
UPDATE STATISTICS dbo.SalesOrderHeader WITH FULLSCAN
```

DBCC SHOW_STATISTICS pokaże teraz poniższe dane:

Updated	Rows Above	Rows Below	Inserts Since Last Update	Deletes Since Last Update	Leading column Type
Jan 11 2014 2:16PM	32	0	32	0	Ascending
Jan 11 2014 2:15PM	27	0	27	0	NULL
Jan 11 2014 2:13PM	30	0	30	0	NULL
Jan 11 2014 2:12PM	NULL	NULL	NULL	NULL	NULL

Oprócz nowego rekordu informującego o dodaniu 32 wierszy możesz zauważyć również, że typ kolumny został zmieniony na *Ascending* (rosnąca). Po zmienieniu typu kolumny SQL Server będzie mógł lepiej szacować kardynalność, bez konieczności ręcznej aktualizacji statystyk.

Aby to sprawdzić, uruchom poniższe wstawienie:

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate = '2008-07-23 00:00:00.000'
```

A następnie zapytanie:

```
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-23 00:00:00.000'
```

Tym razem otrzymamy lepszy szacunek kardynalności. Zauważ, że teraz polecenie `UPDATE STATISTICS` nie było konieczne.

Zamiast szacunku 1 wiersza otrzymamy teraz 27,9677. Ale skąd pochodzi ta wartość? Optymalizator używa teraz informacji dotyczących gęstości obiektu statystyk. Jak już tłumaczyłem, definicja gęstości to *1/liczba unikalnych wartości*, a szacowana liczba rekordów jest ustalana za pomocą gęstości pomnożonej przez liczbę rekordów w tabeli, co w tym przypadku wynosi $0,000896861 \cdot 31184$, czyli zgodnie z informacją w planie 27,967713424. Zauważ również, że informacje dotyczące gęstości są używane tylko dla wartości niezawartych w histogramie (informację na temat gęstości możesz uzyskać, korzystając z tego samego polecenia DBCC SHOW_STATISTICS, ale w sesji, w której flaga 2388 jest wyłączona).

Ponadto jeżeli będziemy szukać danych, które nie istnieją, nadal otrzymamy szacunek 1 wiersza, który jest zawsze adekwatny, ponieważ zapytanie zwróci 0 wierszy.

```
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-24 00:00:00.000'
```

Zauważ, że oznaczenie kolumny jako *rosnącej* wymaga, aby statystyki wzrosły trzy razy z rzędu. Jeżeli następnie wstawimy starsze dane, przerywając sekwencję wzrastających danych, kolumna Leading column Type pokaże wartość *Stationary* (stacjonarne), a procesor zapytań wróci do oryginalnego zachowania w procesie szacowania kardynalności. Trzy kolejne aktualizacje z danymi rosnącymi sprawią, że kolumna znowu zostanie oznaczona jako rosnąca.

Wreszcie możesz skorzystać z flag dla zapytania, bez definiowania ich na poziomie sesji lub globalnym, korzystając z odpowiedzi QUERYTRACEON:

```
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-23 00:00:00.000'
OPTION (QUERYTRACEON 2389, QUERYTRACEON 2390)
```

Flagi 2389 i 2390 nie są już potrzebne, jeżeli korzystasz z nowego mechanizmu szacowania kardynalności, bo bez nich otrzymasz dokładnie takie samo zachowanie i szacunki. Aby zobaczyć, jak to działa, usuń tabelę dbo.SalesOrderHeader:

```
DROP TABLE dbo.SalesOrderHeader
```

Wyłącz flagi 2388 i 2389 tak jak poniżej lub otwórz nową sesję:

```
DBCC TRACEOFF (2388)
DBCC TRACEOFF (2389)
```



UWAGA

Korzystając z polecenia DBCC TRACESTATUS, możesz upewnić się, czy flagi nie są włączone.

Stwórz tabelę dbo.SalesOrderHeader zgodnie z instrukcją z początku tego podrozdziału. Wstaw dane i stwórz indeks:

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate < '2008-07-20 00:00:00.000'
CREATE INDEX IX_OrderDate ON SalesOrderHeader(OrderDate)
```

Teraz dodaj dane dla 20 czerwca:

```
INSERT INTO dbo.SalesOrderHeader SELECT * FROM Sales.SalesOrderHeader
WHERE OrderDate = '2008-07-20 00:00:00.000'
```

Tak jak poprzednio, ponieważ liczba dodanych rekordów jest niewielka, nie będzie wystarczająca, aby wyzwolić automatyczną aktualizację statystyk. Jak widzieliśmy wcześniej, uruchomienie poniższego zapytania w przypadku starego mechanizmu szacowania kardynalności zwróci 1 wiersz:

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 110
GO
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-20 00:00:00.000'
```

Uruchomienie tego samego zapytania z nowym mechanizmem szacowania kardynalności da lepszy szacunek, wynoszący 27,9631, bez konieczności korzystania z flag 2389 i 2390:

```
ALTER DATABASE AdventureWorks2012 SET COMPATIBILITY_LEVEL = 120
GO
SELECT * FROM dbo.SalesOrderHeader WHERE OrderDate = '2008-07-20 00:00:00.000'
```

Wartość 27,9631 jest szacowana w dokładnie ten sam sposób, jaki wcześniej opisałem — za pomocą gęstości pomnożonej przez liczbę rekordów w tabeli, co w tym przypadku wynosi $0,0008992806 \cdot 31095$, czyli 27,9631302570. Pamiętaj jednak o skorzystaniu z polecenia DBCC SHOW_STATISTICS dla nowej wersji tej tabeli, aby pozyskać nową gęstość.

UPDATE STATISTICS z ROWCOUNT i PAGECOUNT

Nieudokumentowane opcje ROWCOUNT i PAGECOUNT polecenia UPDATE STATISTICS są wykorzystywane przez mechanizm Data Engine Tuning Advisor (DTA) do tworzenia skryptów i kopiowania statystyk, jeżeli chcesz skonfigurować serwer testowy do optymalizowania zapytań serwera produkcyjnego. Możesz również zobaczyć te polecenia w akcji, tworząc skrypt obiektu statystyk. Jako przykład spróbuj następujących czynności w Management Studio: wybierz *Databases*, kliknij prawym przyciskiem myszy bazę AdventureWorks2012, wybierz *Tasks, Generate Scripts...*, kliknij *Next*, wybierz *Select specific database objects*, rozwiń *Tables*, wybierz Sales.SalesOrderDetail, kliknij *Next*, kliknij *Advanced*, poszukaj *Script Statistics* i wybierz *Script statistics and histograms*. W polu *Script Indexes* wybierz *True*. Kliknij *OK* i zakończ działanie kreatora, aby wygenerować skrypty. Otrzymasz skrypt z poleceniami UPDATE STATISTICS podobnymi do poniższego (wartość STATS_STREAM została skrócona ze względu na ograniczoną ilość miejsca):

```
UPDATE STATISTICS [Sales].[SalesOrderDetail] ([IX_SalesOrderDetail_ProductID])
WITH STATS_STREAM = 0x010000000300000000000000 ..., ROWCOUNT = 121317,
PAGECOUNT = 274
```

W tym podrozdziale pokażę, jak możesz użyć opcji ROWCOUNT i PAGECOUNT polecenia UPDATE STATISTICS w przypadku, gdy chcesz zobaczyć, które plany wykonania zostałyby wykorzystane dla ogromnych tabel (zawierających miliony rekordów), ale potem przetestować te plany dla mniejszych lub nawet pustych tabel. Jak zapewne się domyślasz, opcje te mogą być użyteczne przy testach, jeżeli nie chcesz poświęcać czasu i przestrzeni dyskowej na kopiowanie dużych tabel.

Korzystając z tej metody, prosisz optymalizator, aby wygenerował plany wykonania, korzystając z szacunków kardynalności, jak gdyby tabela rzeczywiście miała miliony rekordów, nawet jeżeli w rzeczywistości tabela ta jest bardzo mała. Zauważ, że ta opcja, dostępna od wersji 2005, pomocna jest jedynie w tworzeniu planów zapytania dla Twoich zapytań. Właściwe uruchomienie zapytania wykorzysta prawdziwe dane z Twojej tabeli testowej, co oczywiście spowoduje, że zapytanie wykona się szybciej niż dla tabeli z milionami rekordów.

Korzystanie z opcji polecenia UPDATE STATISTICS nie zmienia statystyk, a jedynie liczniki liczby rekordów i stron tabeli. Jak niedługo pokażę, optymalizator korzysta z tych informacji do szacowania kardynalności zapytań. Zanim jednak spojrzymy na przykłady, pamiętaj, że są to opcje nieudokumentowane i niewspierane, i nie powinny być wykorzystywane w środowisku produkcyjnym.

Spojrzymy na przykład. Uruchom poniższe zapytanie, aby stworzyć nową tabelę w bazie AdventureWorks2012:

```
SELECT * INTO dbo.Address
FROM Person.Address
```

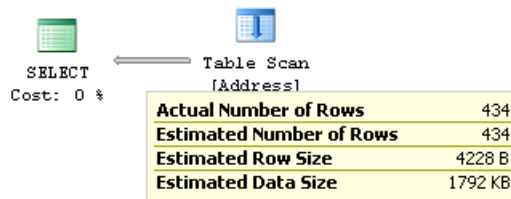
Sprawdź liczbę rekordów za pomocą poniższego zapytania; kolumna row_count powinna pokazywać 19614 rekordów:

```
SELECT * FROM sys.dm_db_partition_stats
WHERE object_id = OBJECT_ID('dbo.Address')
```

Teraz uruchom poniższe zapytanie i sprawdź plan graficzny:

```
SELECT * FROM dbo.Address
WHERE City = 'London'
```

Uruchomienie tego zapytania spowoduje stworzenie nowego obiektu statystyk dla kolumny City i wyświetli plan przedstawiony na rysunku 6.14. Zauważ, że szacowana liczba rekordów wynosi 434 i że plan wykorzystuje prosty operator *Table Scan*.



Rysunek 6.14. Przykład szacowania kardynalności dla małej tabeli

Możemy sprawdzić, skąd optymalizator bierze szacowaną liczbę rekordów, przeglądając obiekt statystyk. Korzystając z metodologii pokazanej wcześniej w podrozdziale „Histogram”, możesz znaleźć nazwę obiektu statystyk, a następnie skorzystać z polecenia DBCC SHOW_STATISTICS, aby wyświetlić histogram. W histogramie możesz znaleźć wartość 434 w kolumnie EQ_ROWS w wierszu mającym wartość RANGE_HI_KEY równą London.

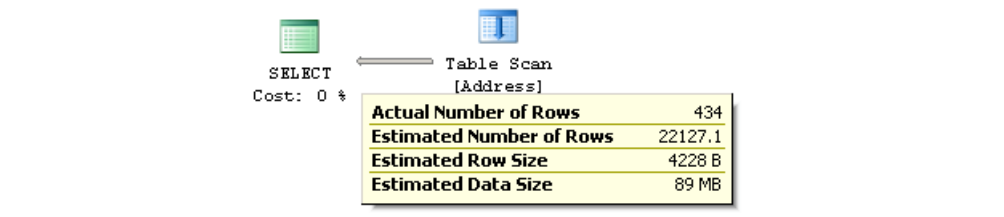
Teraz uruchom poniższe polecenie UPDATE STATISTICS WITH ROWCOUNT, PAGECOUNT (dla ROWCOUNT i PAGECOUNT możesz podać dowolne inne wartości):

```
UPDATE STATISTICS dbo.Address WITH ROWCOUNT = 1000000, PAGECOUNT = 100000
```

Jeżeli znów sprawdzisz liczbę rekordów za pomocą sys.dm_db_partition_stats, otrzymasz wartość 1000000 wierszy (nowa liczba stron pokazywana jest w kolumnie in_row_data_page_count). Wyczyść magazyn planów i ponownie uruchom zapytanie.

```
DBCC FREEPROCCACHE
GO
SELECT * FROM dbo.Address WHERE City = 'London'
```

Zauważ, że szacowana liczba rekordów zmieniła się z 434 na 22127,1 (zobacz rysunek 6.15).



Rysunek 6.15. Szacunek kardynalności z użyciem ROWCOUNT i PAGECOUNT

Jeżeli jednak, korzystając z DBCC SHOW_STATISTICS, jeszcze raz spojrzysz na obiekt statystyk, zobaczysz, że histogram nie uległ zmianie. Jednym ze sposobów na uzyskanie szacowanej liczby rekordów pokazanej w nowym planie wykonania jest obliczenie wartości procentowej (lub ułamka) rekordów dla wartości London dla próbki statystyk, która w tym przypadku wynosi 19614. Ułamek to 434/19614, czyli 0,022127052. Następnie stosujemy ten sam ułamek dla nowej liczby rekordów, co daje 1000000 0,022127052; w ten sposób otrzymujemy 22127,1, co jest szacowaną liczbą rekordów wyświetloną w planie z rysunku 6.15.

Jeżeli chcesz przywrócić prawdziwe wartości dla rekordów i stron, być może aby wykonać dodatkowe testy, możesz wykorzystać polecenie DBCC UPDATEUSAGE. Polecenie to można użyć do poprawienia liczników stron i rekordów w widokach katalogu. Uruchom poniższe polecenie:

```
DBCC UPDATEUSAGE(AdventureWorks2012, 'dbo.Address') WITH COUNT_ROWS
```

Jednak po skończonych testach zalecam usunąć tę tabelę, aby uniknąć pozostawionych niechcący nieścisłości w liczbie rekordów i stron:

```
DROP TABLE dbo.Address
```

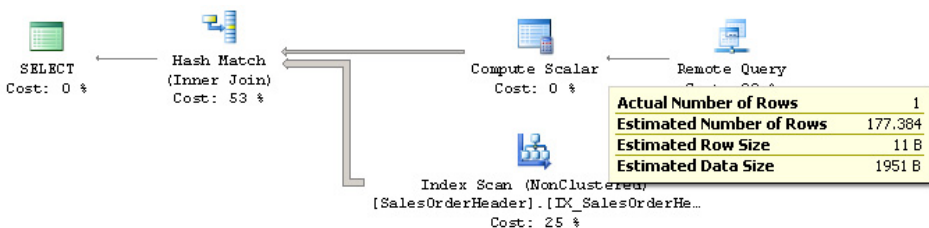
Statystyki na serwerach połączonych

Znanym problemem podczas łączenia się z serwerami połączonymi na instancjach działających na wersji SQL Server 2012 RTM i starszych jest to, że procesor zapytań nie jest w stanie pozyskać wymaganych statystyk ze zdalnej instancji z powodu uprawnień użytkownika ustawionych na serwerze połączonym. W tych wersjach, aby pozyskać wszystkie dostępne statystyki, użytkownik musi być właścicielem tabeli albo być członkiem roli serwerowej *sysadmin*, roli bazodanowej *db_owner* lub roli *db_ddladmin* na serwerze połączonym. W SQL Server 2012 Service Pack 1 i nowszych wersjach możliwy jest dostęp do statystyk poprzez uruchomienie DBCC SHOW_STATISTICS z uprawnieniami SELECT. Chociaż jest to również domyślne zachowanie w SQL Serverze 2014, i tak musisz zdawać sobie sprawę z istnienia tego problemu, jeżeli łączysz się ze starszymi wersjami SQL Servera lub jeżeli napotkasz regresję i zechcesz wrócić do starego zachowania.

Zobaczmy więc, jak to działa z wykorzystaniem serwera połączonego (tutaj oznaczonego jako remote) poprzez połączenie się z instancją SQL Server 2012 RTM lub starszą.

```
SELECT l.SalesOrderID, l.CustomerID
FROM AdventureWorks2012.Sales.SalesOrderHeader l
JOIN [remote].AdventureWorks2012.Sales.SalesOrderHeader r
ON l.SalesOrderID = r.SalesOrderID
WHERE r.CustomerID = 11000
```

Moja konfiguracja korzysta z serwera połączonego wymagającego logowania z uprawnieniami do danych tylko do odczytu (na przykład *db_datareader*). Uruchomienie powyższego zapytania dla zdalnej instancji SQL Servera zwraca zły szacunek kardynalności i plan pokazany na rysunku 6.16, wykorzystujący operator *Hash Join*. Ponieważ lokalny procesor zapytań nie ma dostępu do statystyk na serwerze zdalnym, musi polegać na przypuszczeniu (w tym przypadku 177,384 rekordu). Możesz zauważyć dużą różnicę między właściwą i szacowaną liczbą rekordów.



Rysunek 6.16. Zły szacunek kardynalności dla serwera połączonego

Jeżeli to samo zapytanie uruchomimy na serwerze w wersji SQL Server 2012 Service Pack 1 lub nowszej, otrzymamy lepszy szacunek kardynalności, a optymalizator będzie w stanie podjąć lepszą decyzję. Otrzymamy nowy plan wykorzystujący operator *Nested Loops Join*, który jest bardziej odpowiedni w przypadku niewielkiej liczby rekordów. Chociaż nowe zachowanie jest domyślnie włączone dla serwerów połączonych działających na SQL Server 2012 Service Pack 1 i nowszych, możesz je również wyłączyć, korzystając z flagi 9485, której możesz użyć w przypadku regresji i pogorszenia się wydajności jakichś zapytań. Włączenie flagi 9485 wyłącza nowy mechanizm sprawdzania uprawnień i przywraca poprzednie zachowanie.

Uruchom poniższe zapytanie, aby wyłączyć flagę 9485:

```
DBCC TRACEON (9485)
```

Powtórne uruchomienie zapytania spowoduje ponowne stworzenie pierwszego planu wykorzystującego *Hash Join* i oszacowanie kardynalności na 177,384 rekordu. Możesz również potrzebować polecenia do usunięcia aktualnego planu lub wymuszenia nowej optymalizacji podczas testów (na przykład `DBCC FREEPROCCACHE`).

Konserwacja statystyk

Jak już wspominałem, optymalizator zapytań domyślnie automatycznie zaktualizuje statystyki, jeżeli będą nieaktualne. Statystyki można również zaktualizować za pomocą polecenia `UPDATE STATISTICS`, które możesz ustawić jako regularnie uruchamiane zadanie. Kolejnym często stosowanym poleceniem jest `sp_updatestats`, które w tle także uruchamia polecenie `UPDATE STATISTICS`.

Są dwie ważne zalety regularnego aktualizowania statystyk. Pierwsza z nich jest taka, że zapytania będą korzystały z aktualnych statystyk bez konieczności oczekiwania na zakończenie aktualizacji automatycznej, a tym samym unikając opóźnień w optymalizacji zapytań (choć z tym problemem można również częściowo poradzić sobie za pomocą aktualizacji asynchronicznych). Drugą zaletą jest to, że możesz wykorzystać większą próbkę, niż zrobiłby to optymalizator, lub możesz nawet przeskanować całą tabelę. Dzięki temu możesz uzyskać lepszej jakości statystyki dla dużych tabel, w szczególności dla tych, w których dane nie są losowo rozmieszczone na stronach danych. Ręczna aktualizacja statystyk może również być korzystna po operacjach takich jak ładowanie danych w partiach lub aktualizacja dużych partii danych.

Z drugiej strony, zauważ również, że aktualizacja statystyk w zadaniach spowoduje rekompilację planów znajdujących się już w magazynie planów i korzystających z tych statystyk, więc być może nie powinieneś aktualizować ich zbyt często.

Dodatkowym elementem, jaki należy brać pod uwagę w przypadku ręcznej aktualizacji statystyk, jest ich powiązanie z zadaniami przebudowywania indeksów, co powoduje również aktualizację ich statystyk. Miej na uwadze poniższe aspekty, jeżeli

łączysz zadania konserwacyjne dla indeksów i statystyk, pamiętaj, że istnieją statystyki dla kolumn związane i niezwiązane z indeksami i że oczywiście operacje na indeksach wpływają tylko na te pierwsze:

- ▶ Przebudowa indeksu (na przykład za pomocą polecenia `ALTER INDEX ... REBUILD`) spowoduje przebudowanie statystyk poprzez skanowanie rekordów w tabeli, co jest równoznaczne z wykorzystaniem `UPDATE STATISTICS WITH FULLSCAN`. Przebudowanie indeksów nie ma wpływu na statystyki kolumn.
- ▶ Reorganizacja indeksu (na przykład za pomocą polecenia `ALTER INDEX ... REORGANIZE`) nie aktualizuje statystyk, nawet tych związanych z indeksem.
- ▶ Domyślnie polecenie `UPDATE STATISTICS` aktualizuje statystyki indeksów i kolumn. Wykorzystanie opcji `INDEX` spowoduje aktualizację tylko statystyk dla indeksów, natomiast użycie opcji `COLUMNS` zaktualizuje tylko statystyki dla kolumn niezwiązane z indeksami.

Dlatego w zależności od Twoich zadań konserwacyjnych i skryptów może istnieć kilka scenariuszy. Najprostszy plan to przebudowa wszystkich indeksów i wszystkich statystyk. Jak już wspomniałem, jeżeli przebudujesz wszystkie indeksy, to również statystyki z nimi związane zostaną automatycznie zaktualizowane poprzez skanowanie rekordów w tabeli. Wtedy pozostanie tylko zaktualizowanie statystyk niezwiązanych z indeksami za pomocą polecenia `UPDATE STATISTICS WITH FULLSCAN, COLUMNS`. Ponieważ zadanie przebudowania indeksów aktualizuje tylko statystyki indeksów, a to drugie aktualizuje tylko statystyki kolumn, nie ma znaczenia, które z nich zostanie uruchomione jako pierwsze.

Oczywiście mogą istnieć bardziej skomplikowane plany — na przykład kiedy indeksy są przebudowywane lub reorganizowane w zależności od poziomu ich fragmentacji (temat ten omówiłem w rozdziale 5.). Powinieneś pamiętać o aspektach wspomnianych wcześniej, aby uniknąć podwójnej aktualizacji statystyk, co mogłoby mieć miejsce, gdybyś wykonał zarówno operację przebudowy indeksu, jak i aktualizacji statystyki. Musisz także unikać usuwania efektów wcześniej wykonanych zadań — na przykład w przypadku, gdy przebudujesz indeksy tabeli, co również spowoduje aktualizację statystyk poprzez skanowanie całej tabeli, a następnie uruchomisz zadanie aktualizujące statystyki z domyślną lub mniejszą próbką. W tym przypadku poprzednio zaktualizowane statystyki są zamieniane przez statystyki, które potencjalnie mają niższą jakość.

Pozwól, że za pomocą kilku przykładów pokażę Ci, jak działają te polecenia. Stwórz nową tabelę o nazwie `dbo.SalesOrderDetail`:

```
SELECT * INTO dbo.SalesOrderDetail FROM Sales.SalesOrderDetail
```

Następne zapytanie korzysta z widoku katalogowego `sys.stats`, aby pokazać, że dla nowej tabeli nie ma żadnych obiektów statystyk:

```
SELECT name, auto_created, STATS_DATE(object_id, stats_id) AS update_date
FROM sys.stats
WHERE object_id = OBJECT_ID('dbo.SalesOrderDetail')
```

Teraz uruchom poniższe zapytanie:

```
SELECT * FROM dbo.SalesOrderDetail
WHERE SalesOrderID = 43670 AND OrderQty = 1
```

Użyj poprzedniego zapytania wykorzystującego `sys.stats`, aby potwierdzić, że obiekty statystyk zostały stworzone: jeden dla kolumny *SalesOrderID* i drugi dla kolumny *OrderQty*. Teraz stwórz poniższy indeks i po raz kolejny uruchom zapytanie z `sys.stats`, aby sprawdzić, czy nowy obiekt statystyk dla kolumny *ProductID* został stworzony:

```
CREATE INDEX IX_ProductID ON dbo.SalesOrderDetail(ProductID)
```

Wynik zapytania `sys.stats` na tym etapie powinien być następujący:

name	auto_created	update_date
_WA_Sys_00000004_4DD47EBD	1	1/11/2014 8:11:34 PM
_WA_Sys_00000001_4DD47EBD	1	1/11/2014 8:11:34 PM
IX_ProductID	0	1/11/2014 8:14:31 PM

Zwróć uwagę, że wartość kolumny `auto_created`, która wskazuje na to, czy statystyki zostały stworzone przez optymalizator, dla obiektu statystyk `IX_ProductID` ma wartość 0. Uruchom poniższe polecenie, aby zaktualizować tylko statystyki dla kolumn:

```
UPDATE STATISTICS dbo.SalesOrderDetail WITH FULLSCAN, COLUMNS
```

Możesz sprawdzić, że zaktualizowane zostały tylko statystyki kolumn, porównując wartość w kolumnie *update_date* z poprzednim wynikiem. Kolumna *update_date*, do wyświetlenia czasu ostatniej aktualizacji, wykorzystuje funkcję `STATS_DATE`, zgodnie z poniższym wynikiem:

name	auto_created	update_date
_WA_Sys_00000004_4DD47EBD	1	1/11/2014 8:16:54 PM
_WA_Sys_00000001_4DD47EBD	1	1/11/2014 8:16:55 PM
IX_ProductID	0	1/11/2014 8:14:31 PM

Poniższe polecenie zrobi to samo tylko dla statystyki indeksu:

```
UPDATE STATISTICS dbo.SalesOrderDetail WITH FULLSCAN, INDEX
```

A te polecenia zaktualizują zarówno statystyki indeksów, jak i kolumn:

```
UPDATE STATISTICS dbo.SalesOrderDetail WITH FULLSCAN
UPDATE STATISTICS dbo.SalesOrderDetail WITH FULLSCAN, ALL
```


Jak wcześniej wspomniałem, jeżeli po każdym z poniższych dwóch zapytań uruchomisz zapytanie z `sys.stats`, zobaczysz, że polecenie `ALTER INDEX REBUILD` aktualizuje tylko statystyki indeksu:

```
ALTER INDEX ix_ProductID ON dbo.SalesOrderDetail REBUILD
```

A reorganizacja indeksu nie aktualizuje żadnych statystyk:

```
ALTER INDEX ix_ProductID ON dbo.SalesOrderDetail REORGANIZE
```

Aby utrzymać bazę w dobrej kondycji, usuń stworzoną tabelę:

```
DROP TABLE dbo.SalesOrderDetail
```

Szacowanie kosztów

Jak pokazałem, jakość planów wykonania generowanych przez optymalizator zapytań jest bezpośrednio związana z trafnością szacowania kosztów. Nawet jeżeli optymalizator jest w stanie wygenerować plany o niskim koszcie, nieprawidłowy szacunek kosztów może prowadzić do wyboru nieefektywnego planu, co może negatywnie odbić się na wydajności Twojej bazy danych. Podczas optymalizacji zapytań optymalizator bada wiele potencjalnych planów, szacuje ich koszty, a następnie wybiera najbardziej wydajny z nich.

Koszty szacowane są dla każdego częściowego lub kompletnego planu, zgodnie z wyjaśnieniami z rozdziału 3., w którym zbadaliśmy zawartość struktury Memo. Obliczenie kosztów wykonywane jest per operator, a całkowity koszt planu jest sumą kosztów poszczególnych operatorów w planie. Koszt każdego z operatorów zależy od jego algorytmu i szacowanej liczby zwracanych rekordów. Niektóre operatory, takie jak *Sort* lub *Hash Join*, uwzględniają również ilość dostępnej w systemie pamięci. Wysokopoziomowy przegląd kosztów algorytmów dla niektórych z najczęściej stosowanych operatorów przedstawiłem w rozdziale 4.

Tak więc każdy operator ma przypisany koszt CPU, niektóre z nich będą również miały koszt I/O, a całościowy koszt operatora to suma tych kosztów. Operatory takie jak *Clustered Index Scan* ma zarówno koszt CPU, jak i I/O, podczas gdy inne operatory, takie jak *Stream Aggregate*, będą miały przypisany tylko koszt CPU. Ponieważ sposób obliczania tych kosztów nie jest udokumentowany, pokażę Ci prosty przykład takich obliczeń.

Aby pokazać to na przykładzie, spójrzmy na największą tabelę w bazie AdventureWorks. Uruchom poniższe zapytanie i spójrz na pokazany na rysunku 6.17 szacowany koszt CPU i I/O dla operatora *Clustered Index Scan*:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE LineTotal = 35
```

Zauważ, że w starszych wersjach SQL Servera koszt oznaczał szacowany czas na wykonanie zapytania na dostępnym sprzęcie w sekundach, ale aktualnie wartość ta

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	121317
Actual Number of Batches	0
Estimated I/O Cost	0.918681
Estimated Operator Cost	1.05229 (93%)
Estimated CPU Cost	0.133606
Estimated Subtree Cost	1.05229
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	121317
Estimated Row Size	95 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	3
Object	
[AdventureWorks2012].[Sales].[SalesOrderDetail].	
[PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID]	

Rysunek 6.17. Właściwości operatora Clustered Index Scan

jest bezużyteczna jako obiektywna jednostka miary i nie powinna być tak interpretowana. Jej celem jest wyłącznie wewnętrzny użytek przy wyborze najlepszego spośród potencjalnych planów.

Dla operatora *Clustered Index Scan* koszt CPU to 0,0001581 dla pierwszego rekordu, plus 0,0000011 dla każdego dodatkowego rekordu. Ponieważ w tym konkretnym przypadku mamy szacowane 121317 rekordów, możemy obliczyć, że koszt wyniesie $0,0001581 + 0,0000011 \cdot (121317 - 1)$, czyli 0,133606, i taka wartość znajduje się na rysunku 6.17 w pozycji *Estimated CPU Cost*. Podobnie minimalny koszt I/O to 0,003125 dla pierwszej strony, a następnie rośnie o 0,00074074 dla każdej następnej strony. Ponieważ operator ten skanuje całą tabelę, mogę wykorzystać poniższe zapytanie do sprawdzenia liczby stron (liczba ta to 1237):

```
SELECT in_row_data_page_count, row_count
FROM sys.dm_db_partition_stats
WHERE object_id = OBJECT_ID('Sales.SalesOrderDetail')
AND index_id = 1
```

W tym przypadku daje to $0,003125 + 0,00074074 \cdot (1234 - 1)$, czyli około 0,918681, co jest wartością pokazaną w pozycji *Estimated I/O Cost*.

Musimy teraz dodać oba koszty, $0,133606 + 0,918681$, i uzyskamy 1,05229, co stanowi całkowity szacowany koszt operatora. W ten sam sposób dodanie kosztu wszystkich operatorów da nam całkowity koszt planu. W tym przypadku koszt operatora *Clustered Index Scan* (1,05229), koszt operatora *Compute Scalar* (0,01213), drugiego operatora *Compute Scalar* (0,01213) i operatora *Filter* (0,05823) składają się na całkowity koszt planu: 1,13478.

Podsumowanie

W tym rozdziale zobaczyłeś, jak wykorzystywane są statystyki do szacowania kardynalności oraz kosztu operatorów i planów zapytania. Pokazałem i objaśniłem najważniejsze elementy obiektów statystyk: histogram, informację o gęstości i statystyki ciągów znaków. Zamieściłem przykłady wykorzystania histogramów, włącznie z zapytaniami z operatorami równości i nierówności połączonymi koniunkcjami i dysjunkcjami. Przedstawiłem sposób wykorzystania informacji o gęstości w zapytaniach z klauzulą `GROUP BY`, a także w sytuacjach, w których optymalizator nie może skorzystać z histogramu, na przykład w wypadku zmiennych lokalnych.

Omówiłem również konserwację statystyk, z zaznaczeniem, jak aktywnie aktualizować statystyki, aby uniknąć opóźnień podczas optymalizacji zapytań, i jak poprawić jakość statystyk przez skanowanie całej tabeli zamiast domyślnej próbki. Pokazałem też, jak wykrywać błędy w szacowaniu kardynalności, które mogą negatywnie wpłynąć na jakość planów wykonania, a także omówiłem sposoby naprawiania tych błędów.

Rozdział 7

OLTP w pamięci — Hekaton

W tym rozdziale:

- ▶ Architektura
- ▶ Tabele i indeksy
- ▶ Natywnie kompilowane procedury przechowywane
- ▶ Ograniczenia
- ▶ Narzędzie AMR
- ▶ Podsumowanie



Społeczność badaczy baz danych zasugerowała niedawno, że ponieważ systemy zarządzania bazami danych zostały zaprojektowane w późnych latach siedemdziesiątych XX wieku, kiedy sprzęt zdecydowanie różnił się od obecnego, potrzebne jest nowe podejście projektowe i architektoniczne, uwzględniające aktualne możliwości sprzętowe. Systemy zarządzania bazami relacyjnymi zostały zaprojektowane z założeniem, że pamięć jest ograniczona i kosztowna, a bazy są wielokrotnie większe niż pamięć główna, i dlatego dane powinny być przechowywane na dysku. Ponieważ aktualny sprzęt ma pamięć i dyski tysiące razy większe i procesory tysiące razy szybsze, założenia te przestały być prawdziwe. Co więcej, czas dostępu do dysku podlegający fizycznym ograniczeniom nie uległ usprawnieniu w podobnym tempie i nadal jest najwolniejszym ogniwem systemu. Chociaż pojemność pamięci znacznie wzrosła — a rozmiar baz OLTP nie — jedną z nowych koncepcji projektowych dotyczących baz OLTP jest stwierdzenie, że silnik OLTP powinien działać w pamięci, a nie na dysku, gdyż większość baz OLTP jest już w stanie w całości zmieścić się w pamięci głównej.

Ponadto analizy wykonywane w Microsoftzie dla projektu Hekaton pokazywały, że nawet w przypadku sprzętu dostępnego dzisiaj, korzystając z mechanizmów Microsoftu, możemy uzyskać od 10 do 100 razy wyższą wydajność, wymagałoby to jednak radykalnej zmiany sposobu projektowania systemów zarządzania danymi. SQL Server jest już wysoko zoptymalizowany, więc wykorzystanie aktualnych technik nie przyniesie drastycznych wzrostów wydajności ani wielokrotnego przyspieszenia. Nawet po zbudowaniu głównego silnika działającego w pamięci wciąż dużo czasu zajmowało przetwarzanie zapytań, a więc badacze zdali sobie sprawę, że muszą znacznie zredukować liczbę wykonywanych instrukcji. Wykorzystanie dostępnej pamięci nie polega tylko na wczytaniu do pamięci większej liczby stron, ale raczej konieczne jest powtórne zaprojektowanie systemów zarządzania danymi z innym podejściem, aby maksymalnie wykorzystać nowy sprzęt. Ponadto standardowe mechanizmy kontroli współbieżności dostępne obecnie nie skalują się do wysokich szybkości transakcji osiąganych przez bazy zoptymalizowane do działania w pamięci, więc blokowanie staje się kolejnym wąskim gardłem.

Na rynku pojawiły się specjalistyczne silniki baz działające w pamięci, takie jak Oracle TimesTen czy IBM SolidDB. Microsoft również rozpoczął sprzedaż technologii działających w pamięci w postaci systemu analitycznego xVelocity, będącego częścią SQL Server Analysis Services i indeksów magazynu kolumn zoptymalizowanych do działania w pamięci xVelocity zintegrowanych z silnikiem SQL Servera. Obecnie, w SQL Serverze 2014, Microsoft wprowadził działające w pamięci bazy OLTP, nazwane Hekaton, jako nowy silnik OLTP. Wzrost wydajności Hekatona oparty jest na trzech głównych obszarach architektonicznych: optymalizacji dostępu do pamięci głównej, kompilacji procedur do kodu natywnego i szacowaniu blokad. Poniżej przyjrzymy się tym trzem obszarom.

W tym rozdziale omawiam Hekaton, funkcjonalność dostępną tylko w wersji Enterprise Edition. Indeksy xVelocity omówię w rozdziale 9. Ponieważ Hekaton jest nową funkcjonalnością SQL Servera 2014, postaram się wyjaśnić tę technologię najdokładniej jak to możliwe. Jednakże tematy takie jak transakcje i kontrola współbieżności mogą nie zostać omówione wystarczająco szczegółowo, gdyż niniejsza książka dotyczy optymalizacji zapytań. Aby rozróżnić terminologię silnika Hekaton od standardowego silnika SQL Servera omawianego w tej książce, procedury i tabele Hekatona będą nazywane tabelami zoptymalizowanymi do działania w pamięci i procedurami kompilowanymi natywnie. Standardowe tabele i procedury będą nazywane tabelami dyskowymi i procedurami standardowymi lub interpretowanymi.

Architektura

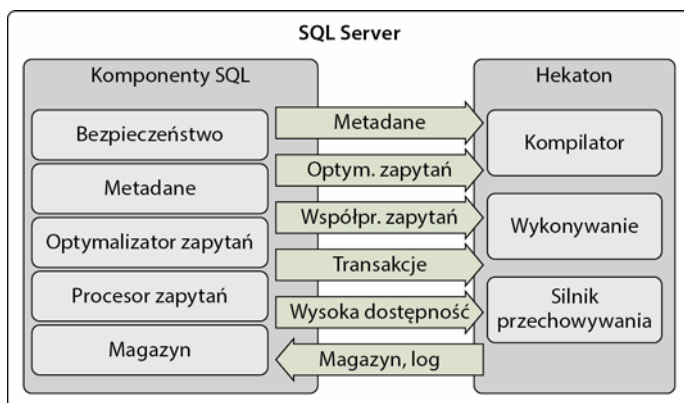
Jedną z głównych strategicznych decyzji podjętych podczas projektowania Hekatona polegała na zbudowaniu nowego silnika baz danych w pełni zintegrowanego z bazą SQL Servera zamiast tworzenia nowego produktu, jak zrobili to inni producenci. Mogłoby to dać użytkownikom kilka korzyści, takich jak działanie istniejących aplikacji bez konieczności wprowadzania zmian i brak konieczności zakupu i wdrażania się w nowy produkt. Jak już wspomniałem, Hekaton może wielokrotnie zwiększyć wydajność baz dzięki poniższym mechanizmom:

- ▶ **Tabele i indeksy zoptymalizowane do działania w pamięci głównej.** Tabele i indeksy Hekatona są zaprojektowane i zoptymalizowane do działania w pamięci. Nie są przechowywane jako strony bazodanowe i nie wykorzystują puli bufora pamięci.
- ▶ **Procedury kompilowane do kodu natywnego.** Procedury przechowywane mogą być najpierw zoptymalizowane przez optymalizator zapytań SQL Servera, jak standardowa procedura przechowywana, a następnie skompilowane do wysoko wydajnego kodu maszynowego.
- ▶ **Wylimitowanie blokad.** Hekaton implementuje nowy, optymistyczny wielowersyjowy mechanizm kontroli współbieżności (MVCC), który wykorzystuje nową strukturę danych, aby wyeliminować tradycyjne blokady, dzięki czemu nie ma konieczności oczekiwania na odblokowanie tabeli.

Brak mechanizmu blokującego nie oznacza, że podczas robienia migawki, wykonywania powtarzalnego odczytu i izolowanych transakcji zapanuje chaos, co omówię w dalszej części. Silnik Hekaton składa się z trzech głównych komponentów (zobacz rysunek 7.1):

- ▶ **Silnik przechowywania Hekaton** — zarządza tabelami i indeksami, zapewniając wsparcie dla przechowywania, transakcji, odzyskiwania, wysokiej dostępności itd.

- **Kompilator Hekaton** — kompiluje procedury i tabele do kodu natywnego, który jest ładowany do plików DLL i procesu SQL Servera.
- **System wykonywania Hekaton** — zapewnia integrację z innymi zasobami SQL Servera.



Rysunek 7.1. Silnik bazodanowy Hekaton

Tabele Hekatona zoptymalizowane do działania w pamięci są przechowywane w pamięci przez cały czas, z wykorzystaniem nowych struktur całkowicie różnych od tabel przechowywanych na dysku. Oczywiście są również przechowywane na dysku, ale jedynie po to, aby zapewnić trwałość danych, są więc odczytywane z dysku tylko podczas odtwarzania bazy danych, na przykład przy uruchomieniu instancji. Tabele Hekatona spełniają warunki ACID — czyli gwarantują atomowość (*Atomicity*), spójność (*Consistency*), izolację (*Isolation*) i trwałość (*Durability*) — chociaż dostępna jest także nietrwała wersja tabel i w tym przypadku dane nie są zapisywane na dysku. Tabele Hekatona muszą mieć przynajmniej jeden indeks i nie mają struktury danych będącej odpowiednikiem sterty. SQL Server wykorzystuje ten sam log transakcji do logowania operacji na tabelach dyskowych i na tabelach Hekatona. Indeksy Hekatona są utrzymywane wyłącznie w pamięci — nie są zapisywane na dysk i w konsekwencji operacje na nich nie są logowane. Indeksy są przebudowywane, kiedy dane są ładowane do pamięci przy starcie instancji SQL Servera.

Chociaż natywnie kompilowane procedury przechowywane mogą uzyskiwać dostęp jedynie do tabel zoptymalizowanych do działania w pamięci, SQL Server udostępnia operatory pozwalające odczytywać i aktualizować tabele Hekatona w procedurach interpretowanych. Funkcjonalność ta jest udostępniana przez nowy komponent odpowiedzialny za współpracę między silnikami (*query interop*), który pozwala umożliwić aktualizację obu typów tabel w jednej transakcji.

Hekaton może pomóc w przypadku aplikacji mających problemy z operacjami I/O, szybkością CPU i blokowaniem tabel. Poprzez wykorzystanie natywnie kompilowanych

procedur przechowywanych jest wykonywanych mniej instrukcji niż w przypadku procedur interpretowanych, co pomaga w przypadkach, kiedy czas wykonywania jest zdominowany przez kod procedur. Poza logowaniem podczas normalnego działania bazy nie są wymagane żadne operacje I/O, a Hekaton wymaga też mniej logowania niż operacje na standardowych tabelach. Aplikacje mające problem ze współbieżnością, na przykład przez blokady fizyczne, blokady logiczne lub semaforey, również znacznie skorzystają na zastosowaniu silnika Hekaton, ponieważ nie wykorzystuje on blokad podczas dostępu do danych.



UWAGA

Szczególnym przypadkiem, w którym Hekaton może okazać się bardzo dobrym rozwiązaniem, jest problem wstawiania na ostatniej stronie, gdzie w wyniku ciągłych prób aktualizowania ostatniej strony przez wszystkie wątki z powodu zastosowania klucza inkrementacyjnego następuje fizyczna blokada. Eliminując blokady fizyczne i logiczne, Hekaton powoduje, że operacje te są bardzo szybkie.

Hekaton nie może jednak być wykorzystany do poprawy wydajności, jeżeli Twoja aplikacja ma problemy z pamięcią lub siecią. Hekaton wymaga, aby wszystkie tabele zdefiniowane jako tabele zoptymalizowane do działania w pamięci były w stanie zmieścić się w pamięci, tak więc Twój serwer musi mieć dość pamięci RAM, aby pomieścić je wszystkie. I wreszcie: Hekaton dostępny jest tylko w 64-bitowych systemach SQL Servera i tylko w wersji Enterprise Edition.

Tabele i indeksy

Jak już wyjaśniałem, do tabel Hekatona można uzyskać dostęp za pomocą natywnie skompilowanych procedur lub poprzez standardowy T-SQL, czyli na przykład zapytania *ad hoc* lub standardowe procedury przechowywane. Tabele przechowywane są w pamięci i każdy wiersz może potencjalnie mieć wiele wersji. Wersje przechowywane są w pamięci zamiast bazy tempdb, jak to mam miejsce w przypadku mechanizmu wersjonowania standardowej bazy danych. Wersje, które nie są już potrzebne — czyli takie, które nie są już widoczne dla żadnej transakcji — są usuwane w celu uniknięcia zapelniania dostępnej pamięci. Ten proces nazywa się *usuwaniami odpadów* (*garbage collection*).

W rozdziale 5. omówiłem indeksy dla tabel standardowych. Tabele przechowywane w pamięci również korzystają z indeksów i w tym podrozdziale omawiam te indeksy oraz różnice między nimi i ich odpowiednikami dla tabel dyskowych. Jak już tłumaczyłem, indeksy Hekatona nie są nigdy zapisywane na dysku — istnieją tylko w pamięci i z tego powodu ich operacje nie są logowane w logu transakcji. Zapisywane są tylko metadane indeksów, a same indeksy są odbudowywane podczas ładowania danych do pamięci przy starcie instancji SQL Servera.

W silniku Hekaton funkcjonują dwa różne rodzaje indeksów: indeksy haszowe i indeksy zakresowe, a oba są implementacjami pozbawionymi blokad. Indeksy haszowe wspierają przeszukiwanie indeksów dla predykatów równości, nie mogą jednak być wykorzystywane dla predykatów nierówności lub do zwracania posortowanych danych. Indeksy zakresowe mogą być wykorzystywane do przeszukiwania indeksów dla predykatów równości, indeksy haszowe oferują jednak znacznie lepszą wydajność i są zalecane dla tego typu operacji.



UWAGA

W najnowszej wersji dokumentacji SQL Servera indeksy zakresowe są po prostu nazywane *indeksami nieklastrowymi* lub *indeksami nieklastrowymi zoptymalizowanymi do działania w pamięci*. Pamiętaj o tym, jeżeli w przyszłości nie znajdziesz w dokumentacji wyrażenia *indeksy zakresowe*.

Indeksy haszowe nie mają kolejności i ich skanowanie spowoduje zwrócenie rekordów w losowej kolejności. Oba rodzaje to indeksy pokrywające; indeks zawiera wskaźniki pamięci do rekordów tabeli, gdzie mogą zostać pobrane wszystkie kolumny. Nie ma bezpośredniego dostępu do rekordu bez wykorzystania indeksu, dlatego wymagany jest przynajmniej jeden indeks, aby móc zlokalizować dane w pamięci.

Chociaż koncepcje fragmentacji i współczynnika wypełnienia przedstawione w rozdziale 5. nie mają odniesienia do indeksów Hekatona, możemy napotkać podobne zachowanie w przypadku konfiguracji wypełnienia wiadra: możliwe jest, że indeks haszowy będzie miał puste wiadra, co zaowocuje zmarnowanym miejscem i wpłynie na wydajność skanowania indeksu, lub będzie przechowywał dużą liczbę rekordów w jednym wiadrze, co może wpłynąć na wydajność przeszukiwania. Ponadto aktualizowanie i usuwanie rekordów może stworzyć nowy rodzaj fragmentacji na powiązanym magazynie dyskowym.

Tworzenie tabel Hekatona

Tworzenie tabeli zoptymalizowanej do działania w pamięci wymaga grupy plików zoptymalizowanej do działania w pamięci. Jeżeli spróbujesz stworzyć tabelę, nie mając takiej grupy, otrzymasz następujący błąd:

```
Msg 41337, Level 16, State 0, Line 22
The MEMORY_OPTIMIZED_DATA filegroup does not exist or is empty. Memory optimized tables
↳ cannot be created for database until it has one MEMORY_OPTIMIZED_DATA filegroup that is not
↳ empty. // Grupa plików MEMORY_OPTIMIZED_DATA nie istnieje lub jest pusta. Tabele zoptymalizowane
↳ do działania w pamięci nie zostaną stworzone, dopóki w bazie nie będzie grupy plików
↳ MEMORY_OPTIMIZED_DATA, która nie jest pusta.
```

Możesz albo stworzyć nową bazę danych z grupą plików zoptymalizowaną do działania w pamięci, albo dodać taki plik do istniejącej bazy. Poniższe polecenie pokazuje pierwszy scenariusz:

```
CREATE DATABASE Test
ON PRIMARY (NAME = Test_data,
FILENAME = 'C:\DATA\Test_data.mdf', SIZE=500MB),
FILEGROUP Test_fg CONTAINS MEMORY_OPTIMIZED_DATA
(NAME = Test_fg, FILENAME = 'C:\DATA\Test_fg')
LOG ON (NAME = Test_log, Filename='C:\DATA\Test_log.ldf', SIZE=500MB)
COLLATE Latin1_General_100_BIN2
```

Poniższy kod pokazuje, jak dodać grupę plików zoptymalizowaną do działania w pamięci do istniejącej bazy danych:

```
CREATE DATABASE Test
ON PRIMARY (NAME = Test_data,
FILENAME = 'C:\DATA\Test_data.mdf', SIZE=500MB)
LOG ON (NAME = Test_log, Filename='C:\DATA\Test_log.ldf', SIZE=500MB)
COLLATE Latin1_General_100_BIN2
GO
ALTER DATABASE Test ADD FILEGROUP Test_fg CONTAINS MEMORY_OPTIMIZED_DATA
GO
ALTER DATABASE Test ADD FILE (NAME = Test_fg, FILENAME = N'C:\DATA\Test_fg')
TO FILEGROUP Test_fg
GO
```



UWAGA

Być może zauważyłeś klauzulę COLLATE dla obu poleceń CREATE DATABASE. Podczas pracy z Hekatonem należy brać pod uwagę kilka kwestii związanych z kodowaniem. Więcej szczegółów w dalszej części tego podręcznika.

Kiedy już masz bazę z grupą plików zoptymalizowaną do działania w pamięci, jesteś gotowy na stworzenie swojej pierwszej tabeli Hekatona. W tym ćwiczeniu skopiujesz dane z bazy AdventureWorks2012 do nowo stworzonej bazy *Test*. Możesz zauważyć, że próba stworzenia skryptów tabel z bazy AdventureWorks2012 i wykorzystania ich do stworzenia tabel zoptymalizowanych do działania w pamięci od razu ujawni pierwsze ograniczenie Hekatona: nie wszystkie właściwości tabel są wspierane, co dokładnie omówię w dalszej części. Możesz również skorzystać z mechanizmu Memory Optimization Advisor, aby pomógł Ci w migracji tabel dyskowych do tabel zoptymalizowanych do działania w pamięci. Mechanizm Memory Optimization Advisor omawiam w dalszej części tego rozdziału.

Stworzenie tabeli Hekatona wymaga wykorzystania klauzuli MEMORY_OPTIMIZED, która musi być ustawiona na ON. Zalecane jest również jawne zadeklarowanie DURABILITY i SCHEMA_AND_DATA, SCHEMA_AND_DATA jest jednak wartością domyślną, więc nie musi być podawane.

Możesz spróbować zdefiniować tabelę, korzystając tylko z klauzuli `MEMORY_OPTIMIZED`:

```
CREATE TABLE TransactionHistoryArchive (
    TransactionID int NOT NULL,
    ProductID int NOT NULL,
    ReferenceOrderID int NOT NULL,
    ReferenceOrderLineID int NOT NULL,
    TransactionDate datetime NOT NULL,
    TransactionType nchar(1) NOT NULL,
    Quantity int NOT NULL,
    ActualCost money NOT NULL,
    ModifiedDate datetime NOT NULL
) WITH (MEMORY_OPTIMIZED = ON)
```

Jednak otrzymasz poniższy błąd:

```
Msg 41321, Level 16, State 7, Line 17
The memory optimized table 'TransactionHistoryArchive' with DURABILITY=SCHEMA_AND_DATA must
have a primary key.
Msg 1750, Level 16, State 0, Line 17
Could not create constraint or index. See previous errors.
```

Ponieważ błąd wskazuje, że tabela zoptymalizowana do działania w pamięci musi mieć klucz główny, możesz go zdefiniować, zmieniając poniższą linię:

```
TransactionID int NOT NULL PRIMARY KEY,
```

Jednak tym razem otrzymasz poniższy komunikat:

```
Msg 12317, Level 16, State 76, Line 17
Clustered indexes, which are the default for primary keys, are not supported with memory
optimized tables. // Indeksy klastrowe, będące domyślnymi dla kluczy głównych, nie są wspierane dla tabel
zoptymalizowanych do działania w pamięci.
```

Zmień poprzednią linię, aby specyfikowała indeks nieklastrowy:

```
TransactionID int NOT NULL PRIMARY KEY NONCLUSTERED,
```

Teraz polecenie zakończy się powodzeniem i stworzona zostanie tabela z indeksem zakresowym.

Dla klucza głównego można również stworzyć indeks haszowy poprzez wykorzystanie klauzuli `HASH`, zgodnie z poniższym przykładem, co wymaga także podania opcji `BUCKET_COUNT`. Najpierw usuńmy nową tabelę za pomocą polecenia `DROP TABLE`, w dokładnie ten sam sposób jak w przypadku tabel dyskowych:

```
DROP TABLE TransactionHistoryArchive
```

A następnie stwórzmy nową:

```
CREATE TABLE TransactionHistoryArchive (
    TransactionID int NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH
        (BUCKET_COUNT = 100000),
    ProductID int NOT NULL,
    ReferenceOrderID int NOT NULL,
    ReferenceOrderLineID int NOT NULL,
    TransactionDate datetime NOT NULL,
    TransactionType nchar(1) NOT NULL,
```

```

Quantity int NOT NULL,
ActualCost money NOT NULL,
ModifiedDate datetime NOT NULL
) WITH (MEMORY_OPTIMIZED = ON)

```

Tak więc tabela zoptymalizowana do działania w pamięci musi również mieć klucz główny, który może być indeksem haszowym lub zakresowym. Aktualnie maksymalna liczba indeksów to 8 i oczywiście na tej samej tabeli możemy mieć indeksy haszowe i zakresowe, zgodnie z poniższym przykładem (znowu musisz usunąć poprzednio stworzoną tabelę):

```

CREATE TABLE TransactionHistoryArchive (
    TransactionID int NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH
        (BUCKET_COUNT = 100000),
    ProductID int NOT NULL,
    ReferenceOrderID int NOT NULL,
    ReferenceOrderLineID int NOT NULL,
    TransactionDate datetime NOT NULL,
    TransactionType nchar(1) NOT NULL,
    Quantity int NOT NULL,
    ActualCost money NOT NULL,
    ModifiedDate datetime NOT NULL,
    INDEX IX_ProductID NONCLUSTERED (ProductID)
) WITH (MEMORY_OPTIMIZED = ON)

```

Powyższy kod tworzy indeks haszowy na kolumnie *TransactionID*, która jest również kluczem głównym tabeli, oraz indeks zakresowy na kolumnie *ProductID*. Słowo kluczowe *NONCLUSTERED* jest opcjonalne dla indeksów zakresowych z tego przykładu. Kiedy tabela zostanie stworzona, jesteśmy gotowi do zasilenia jej danymi z bazy *AdventureWorks2012*. Jednak poniższe zapytanie nie zadziała:

```

INSERT INTO TransactionHistoryArchive
SELECT * FROM AdventureWorks2012.Production.TransactionHistoryArchive

```

Otrzymamy poniższy błąd:

```

Msg 41317, Level 16, State 3, Line 28
A user transaction that accesses memory optimized tables or natively compiled procedures
↳ cannot access more than one user database or databases model and msdb, and it cannot write
↳ to master. // Transakcja użytkownika korzystająca z tabel zoptymalizowanych do działania w pamięci nie może
↳ korzystać z więcej niż jednej tabeli użytkownika lub modelu baz danych i msdb, nie może też zapisywać w bazie master.

```

Jak informuje komunikat, transakcja wykonująca operacje na tabeli zoptymalizowanej do działania w pamięci nie może jednocześnie uzyskiwać dostępu do więcej niż jednej bazy danych. Z tego powodu poniższy kod, łączący dwie tabele z dwóch różnych baz, również nie zadziała:

```

SELECT * FROM TransactionHistoryArchive tha
JOIN AdventureWorks2012.Production.TransactionHistory ta
ON tha.TransactionID = ta.TransactionID

```

Możesz jednak skopiować te dane w inny sposób, na przykład korzystając z kreatora *Import and Export Wizard* lub z tabel tymczasowych:

```
SELECT * INTO #temp
FROM AdventureWorks2012.Production.TransactionHistoryArchive
GO
INSERT INTO TransactionHistoryArchive
SELECT * FROM #temp
```

Jak jednak wyjaśniałem wcześniej, transakcje wykonujące operacje na tabeli zoptymalizowanej do działania w pamięci i tabeli dyskowej w ramach tej samej bazy są dozwolone (poza natywnie kompilowanymi procedurami przechowywanymi). W poniższym przykładzie zostanie stworzona tabela dyskowa, a następnie zostanie złączona z tabelą zoptymalizowaną do działania w pamięci:

```
CREATE TABLE TransactionHistory (
    TransactionID int,
    ProductID int)
GO
SELECT * FROM TransactionHistoryArchive tha
JOIN TransactionHistory ta ON tha.TransactionID = ta.TransactionID
```

Minimalne wymagania do stworzenia tabeli zoptymalizowanej do działania w pamięci to wykorzystanie klauzuli `MEMORY_OPTIMIZED` i zdefiniowanie klucza głównego, który z kolei potrzebuje indeksu. Druga i opcjonalna klauzula to `DURABILITY`, która jest powszechnie wykorzystywana i wspiera opcje `SCHEMA_AND_DATA` i `SCHEMA_ONLY`. `SCHEMA_AND_DATA` jest wartością domyślną i pozwala, aby zarówno schemat, jak i dane były zapisywane na dysku. `SCHEMA_ONLY` oznacza, że dane tabeli nie będą zapisywane na dysku przy restarcie instancji; zapisywany jest tylko schemat tabeli. `SCHEMA_ONLY` tworzy nietrwałe tabele, które mogą poprawić wydajność transakcji poprzez znaczne zmniejszenie operacji I/O dla zapytań i mogą być wykorzystywane w scenariuszach takich jak przechowywanie sesji i tabele pośrednie w przetwarzaniu ETL.

Jak wcześniej wspomniałem, podczas pracy z tabelami Hekatona jest kilka aspektów, które należy brać pod uwagę, a dotyczą one zestawów znaków. Tabele zoptymalizowane do działania w pamięci i natywnie skompilowane procedury przechowywane podlegają następującym ograniczeniom:

- Kolumny typu `char` i `varchar` w tabelach zoptymalizowanych do działania w pamięci muszą wykorzystywać stronę kodową 1252. Zauważ, że to ograniczenie nie dotyczy typów `nchar` i `nvarchar`. Możesz uzyskać listę wspieranych zestawów znaków korzystających ze strony kodowej 1252, uruchamiając poniższe zapytanie:

```
SELECT * FROM sys.fn_helpcollations()
WHERE COLLATIONPROPERTY(name, 'codepage') = 1252
```

Próba wykorzystania innej strony kodowej w kolumnie `char` lub `varchar` zwróci poniższy komunikat:

```
Msg 12329, Level 16, State 107, Line 10
The data types char(n) and varchar(n) using a collation that has a code page other
than 1252 are not supported with memory optimized tables. // Typy danych char(n)
```

→ i `varchar(n)` używające zestawu znaków mającego inną stronę kodową niż 1252 nie są wspierane dla tabel zoptymalizowanych do działania w pamięci.

- Indeksy dla kolumn z ciągami znaków (`char`, `nchar`, `varchar` lub `nvarchar`) mogą być specyfikowane tylko z zestawami znaków BIN2. Listę takich zestawów możesz uzyskać za pomocą następującego zapytania:

```
SELECT * FROM sys.fn_helpcollations() WHERE name like '%BIN2'
```

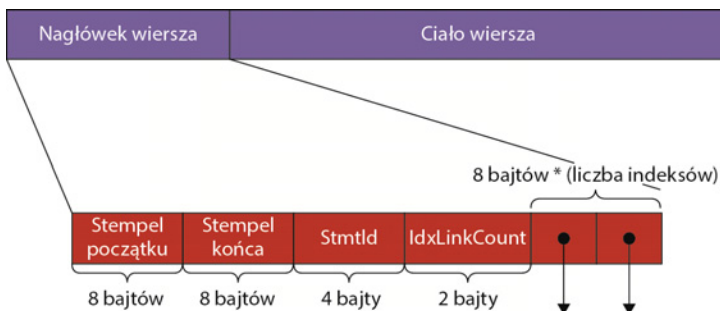
Podobnie próba stworzenia indeksu dla kolumny z ciągiem znaków z zestawem znaków innym niż BIN2 zwróci następujący komunikat:

Msg 12328, Level 16, State 106, Line 10
Indexes on character columns that do not use a *_BIN2 collation are not supported with
→ indexes on memory optimized tables. // Indeksy dla kolumn z ciągami znaków z zestawem znaków
→ innym niż *_BIN2 nie są wspierane dla indeksów tabel zoptymalizowanych do działania w pamięci.

- Porównania, sortowanie i modyfikacje ciągów znaków wewnątrz natywnie kompilowanych procedur przechowywanych również wymagają, aby zestaw znaków korzystał z BIN2. Nieprzestrzeganie tego ograniczenia skutkuje poniższym komunikatem:

Msg 12327, Level 16, State 105, Procedure test, Line 44
Comparison, sorting, and manipulation of character strings that do not use a *_BIN2
→ collation is not supported with natively compiled stored procedures. // Porównania,
→ sortowanie i manipulacja ciągami znaków z zestawem znaków innym niż *_BIN2 nie są wspierane
→ w procedurach przechowywanych kompilowanych natywnie.

Rysunek 7.2 przedstawia strukturę rekordu, gdzie możemy zidentyfikować dwie główne części: nagłówek wiersza i ciało wiersza. Nagłówek wiersza zaczyna się od stempla czasowego początku i końca, który determinuje, dla jakich transakcji będzie kwalifikowany wiersz (szczegóły tego procesu przedstawię w dalszej części).



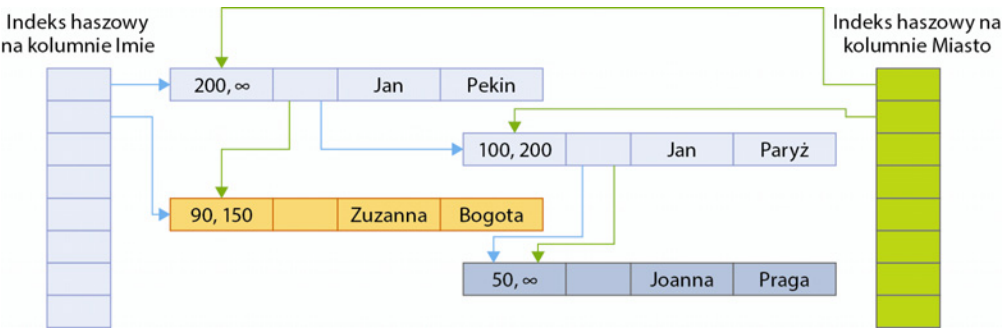
Rysunek 7.2. Struktura wiersza Hekatona

Następne jest pole `StmtId`, zawierające identyfikator instrukcji, która stworzyła wiersz. Ostatnia sekcja nagłówka składa się ze wskaźników indeksu, jeden dla każdego indeksu tabeli. Ciało wiersza zawiera właściwy rekord, czyli kolumny kluczy indeksów oraz pozostałe kolumny wiersza. Ponieważ ciało wiersza zawiera wszystkie kolumny rekordów, możemy powiedzieć, że w Hekatonie nie ma potrzeby definiowania indeksów

pokrywających jak w przypadku tradycyjnych indeksów: indeks zoptymalizowany do działania w pamięci jest indeksem pokrywającym, czyli wszystkie kolumny są zawarte w indeksie. Indeks zawiera wskaźnik na właściwy wiersz tabeli.

Indeksy haszowe

Poprzednio pokrótce przedstawiłem pojęcie indeksów haszowych. Przyjrzyjmy się im teraz dokładniej. Rysunek 7.3 przedstawia przykład indeksów haszowych zawierających trzy rekordy: Zuzanna, Joanna i Jan. Zauważ, że wiersz Jan ma dwie wersje, a ich relacje zostaną wytłumaczone w dalszej części. Zdefiniowane są dwa indeksy: pierwszy na kolumnie Imię, a drugi na kolumnie Miasto, więc zaalokowana została tabela haszy dla każdego z nich, a każde pole reprezentuje inne wiadro haszy. Pierwsza część rekordu (na przykład 90, 150) to stempel czasowy początku i końca pokazany na rysunku 7.2. Symbol nieskończoności (∞) na końcu stempla czasowego oznacza, że jest to aktualna wersja wiersza. Dla uproszczenia przykład zakłada, że każde wiadro zawiera rekordy w zależności od pierwszej litery klucza, czyli wartości w kolumnie Imię lub Miasto, chociaż tak naprawdę nie tak działa funkcja haszująca, ale to wyjaśnię w dalszej części.



Rysunek 7.3. Przykład rekordów i indeksów Hekatona

Ponieważ tabela ma dwa indeksy, każdy rekord ma dwa wskaźniki, po jednym dla każdego indeksu, zgodnie z rysunkiem 7.2. Pierwsze wiadro dla indeksu Imię ma łańcuch trzech rekordów: dwie wersje rekordu Jan i jedną wersję rekordu Joanna. Połączenia pomiędzy tymi rekordami zostały przedstawione za pomocą czarnych strzałek. Drugie wiadro dla indeksu Imię ma tylko jeden rekord — Zuzanna — dlatego nie ma żadnych połączeń. Podobnie rysunek 7.3 przedstawia dwa wiadra dla indeksu Miasto — pierwsze z dwoma rekordami dla miast Pekin i Bogota, a drugie dla miast Paryż i Praga. Połączenia zostały przedstawione za pomocą szarych strzałek. Rekord Jan, Pekin ma czas aktualności od 200 do nieskończoności, co oznacza, że został stworzony przez transakcję w chwili czasu 200 i nadal jest aktualny. Druga wersja, Jan, Paryż, była aktualna w czasie od 100 do 200, kiedy to została zaktualizowana na Jan, Pekin. W systemie MVCC operacje UPDATE są traktowane jako kombinacja operacji INSERT i DELETE. Rekordy

są widoczne tylko od początkowego stempla czasowego do końcowego (nie włącznie), więc w tym przypadku istniejąca wersja straciła ważność poprzez ustawienia końcowego stempla czasowego i stworzona została nowa wersja, ze stemplem czasowym końca ustawionym na nieskończoność.

Transakcja rozpoczęta w czasie 150 widziałaby rekord Jan, Paryż, ale gdyby rozpoczęła się w czasie 220, znalazłaby najnowszą wersję, czyli Jan, Pekin. Jak jednak SQL Server znajduje rekordy z wykorzystaniem indeksu? Tutaj do akcji wkracza funkcja haszująca.

Dla zapytania:

```
SELECT * FROM Tabela WHERE Miasto = 'Pekin'
```

SQL Server wykona funkcję haszującą dla predykatu. Pamiętaj, że dla uproszczenia nasza przykładowa funkcja haszująca działa na podstawie pierwszego znaku ciągu, tak więc w tym przypadku rezultat to B. SQL Server zajrzy następnie do wiadra i wyciągnie wskaźnik na pierwszy rekord. Mając pierwszy rekord, porówna ciągi znaków; jeżeli są takie same, zwróci żądane szczegóły wiersza. Następnie będzie przemieszczał się po wskaźnikach indeksu, porównując wartości i w odpowiednich miejscach zwracając rekordy.

Zauważ, że w przykładzie wykorzystałem dwa indeksy haszowe, ale ten sam mechanizm zadziała również, gdybyśmy wykorzystali, na przykład, indeks haszowy na kolumnie Imie i indeks zakresowy na kolumnie Miasto lub dwa indeksy zakresowe.

BUCKET_COUNT kontroluje rozmiar tabeli haszowej, dlatego wartość tę należy wybierać uważnie. Zaleca się, aby była dwukrotnie większa od maksymalnej oczekiwanej liczby unikalnych wartości klucza indeksu i zaokrąglona do najbliższej potęgi liczby 2, chociaż jeżeli zajdzie taka potrzeba, Hekaton sam wykona zaokrąglenie. Nieadekwatna liczba BUCKET_COUNT może powodować problemy z wydajnością: zbyt duża wartość może prowadzić do dużej liczby pustych wiader w tabeli haszowej. Może to skutkować większym wykorzystaniem pamięci, ponieważ każde wiadro wykorzystuje 8 bajtów. Ponadto skanowanie indeksu będzie bardziej kosztowne, gdyż konieczne jest przeskanowanie tych pustych wiader. Jednakże duża wartość BUCKET_COUNT nie ma wpływu na wydajność przeszukiwania indeksu. Natomiast zbyt mała wartość BUCKET_COUNT może prowadzić do długich łańcuchów rekordów, co spowoduje, że wyszukiwanie konkretnego rekordu będzie bardziej kosztowne, ponieważ silnik będzie musiał przejść wiele wartości, zanim znajdzie konkretny rekord.

Zobaczmy kilka przykładów działania BUCKET_COUNT, zaczynając od pustej tabeli. Usuń istniejącą tabelę TransactionHistoryArchive:

```
DROP TABLE TransactionHistoryArchive
```

A następnie stwórz ją ponownie z wartością BUCKET_COUNT ustawioną na 100000:

```
CREATE TABLE TransactionHistoryArchive (
    TransactionID int NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH
        (BUCKET_COUNT = 100000),
    ProductID int NOT NULL,
```

```
ReferenceOrderID int NOT NULL,
ReferenceOrderLineID int NOT NULL,
TransactionDate datetime NOT NULL,
TransactionType nchar(1) NOT NULL,
Quantity int NOT NULL,
ActualCost money NOT NULL,
ModifiedDate datetime NOT NULL
) WITH (MEMORY_OPTIMIZED = ON)
```

Aby zobaczyć statystyki dotyczące indeksów haszowych, możesz wykorzystać widok `sys.dm_db_xtp_hash_index_stats`. Uruchom poniższe zapytanie:

```
SELECT * FROM sys.dm_db_xtp_hash_index_stats
```

Otrzymasz następujący wynik:

object_id	table_name	index_id	index_name	total_bucket_count	empty_bucket_count	avg_chain_len	max_chain_len
309576141	TransactionHistoryArchive	2	PK_Transact__55433A4A5BA80FBC	131072	131072	0	0

Jak widzisz w kolumnie *total_bucket_count*, zamiast 100000 wiader jest 131072, ponieważ SQL Server zaokrąglił do najbliższej potęgi liczby 2 (w tym przypadku 2¹⁷, czyli 131072).

Jeszcze raz wstaw te same dane za pomocą poniższych poleceń:

```
DROP TABLE #temp
GO
SELECT * INTO #temp
FROM AdventureWorks2012.Production.TransactionHistoryArchive
GO
INSERT INTO TransactionHistoryArchive
SELECT * FROM #temp
```

Tym razem wstawione zostaną 89253 rekordy. Jeszcze raz zajrzyj do widoku `sys.dm_db_xtp_hash_index_stats`. Tym razem otrzymasz dane podobne do poniższych:

object_id	table_name	index_id	index_name	total_bucket_count	empty_bucket_count	avg_hain_len	max_chain_len
309576141	TransactionHistoryArchive	2	PK_Transact__55433A4A5BA80FBC	131072	91881	2	4

Możemy zaobserwować kilka rzeczy. Całkowita liczba wiader (131072) pomniejszona o liczbę pustych wiader (91881), pokazaną w kolumnie *empty_bucket_count*, daje liczbę wykorzystanych wiader (39191). Ponieważ wstawiliśmy 89253 rekordy do 39191 wiader, otrzymujemy średnią liczbę rekordów w wiadrze na poziomie 2,28, co widać

w kolumnie *avg_chain_len*, i jest to średnia długość łańcucha rekordów we wszystkich wiadrach indeksu. W kolumnie *max_chain_len* zawarta jest maksymalna długość łańcucha wierszy w wiadrach.

Aby pokazać skrajny przypadek, który może mieć wpływ na wydajność, uruchom to samo ćwiczenie, ale w klauzuli `BUCKET_COUNT` zażądaj stworzenia tylko 1024 wiader. Po wstawieniu 89253 rekordów dostalibyśmy następujący wynik:

object_id	table_name	index_id	index_name	total_bucket_count	empty_bucket_count	avg_hain_len	max_chain_len
373576369	Transaction HistoryArchive	2	PK_Transact_55433A4A89A2F661	1024	0	87	89

W tym przypadku mamy 89253 rekordy podzielone na 1024 (czyli 87,16 na każde wiadro). Kolizje haszy zdarzają się, jeżeli dwa lub więcej kluczy indeksu jest zmapowanych na to samo wiadro, jak w tym przykładzie, a duża liczba kolizji haszy może negatywnie wpływać na wydajność operacji wyszukiwania.

Drugim ekstremalnym przypadkiem jest zbyt wiele wiader w odniesieniu do liczby rekordów. Uruchomienie tego samego przykładu dla 1000000 wiader i wstawienie tych samych danych da nam poniższy wynik:

object_id	table_name	index_id	index_name	total_bucket_count	empty_bucket_count	avg_hain_len	max_chain_len
405576483	Transaction HistoryArchive	2	PK_Transact_55433A4A0A843683	1048576	959323	1	1

W tym przykładzie mamy 91,49% nieużywanych wiader, które będą wykorzystywać miejsce i negatywnie wpływać na wydajność operacji skanowania, ponieważ konieczne będzie czytanie wielu nieużywanych wiader.

Zastanawiasz się, co się stanie, jeżeli będziemy mieć taką samą liczbę wiader jak rekordów?

Zmień poprzedni kod, aby stworzył 65 536 wiader, a następnie uruchom to:

```
DROP TABLE #temp
GO
SELECT TOP 65536 * INTO #temp
FROM AdventureWorks2012.Production.TransactionHistoryArchive
GO
INSERT INTO TransactionHistoryArchive
SELECT * FROM #temp
```

Otrzymasz poniższy wynik:

object_id	table_name	index_id	index_name	total_bucket_count	empty_bucket_count	avg_hain_len	max_chain_len
437576597	Transaction HistoryArchive	2	PK_Transact_ 55433A4AFB5FAACA	65536	47828	3	7

Po przeanalizowaniu tych ekstremalnych przypadków warto przypomnieć, że zaleca się skonfigurowanie `BUCKET_COUNT` na wartość dwukrotnie wyższą od maksymalnej oczekiwanej liczby unikalnej wartości w kluczu indeksu, pamiętając, że zmiana tej wartości nie jest prosta i wymaga usunięcia i powtórnego stworzenia tabeli.

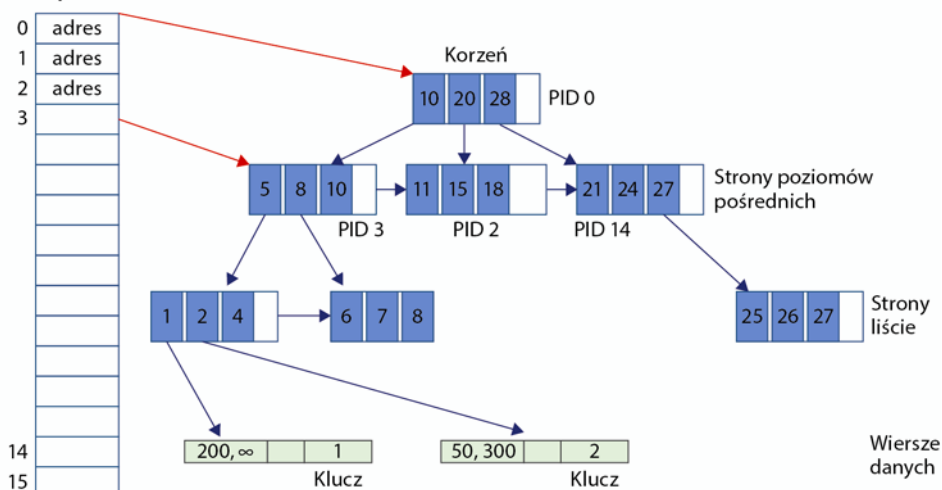
SQL Server wykorzystuje tę samą funkcję haszującą dla wszystkich indeksów haszowych i funkcja ta jest deterministyczna, co oznacza, że ten sam klucz indeksu zawsze zostanie zmapowany na to samo wiadro. Jak widzieliśmy w przykładzie, również wiele kluczy może być zmapowanych na to samo wiadro, a ponieważ wartości nie są równomiernie rozmieszczone w wiadrach, może istnieć wiele pustych wiader, a wiadra wykorzystane mogą zawierać po jednym lub więcej rekordów.

Indeksy zakresowe

Chociaż indeksy haszowe wspierają przeszukiwanie indeksów dla predykatów równości i są najlepszym wyborem dla wyszukiwań punktowych, przeszukania dla predykatów równości, czyli wykorzystujące operatory `>`, `<`, `<=` i `>=`, nie są wspierane w takich indeksach. Dodatkowo, ponieważ wszystkie kolumny klucza indeksu są wykorzystywane do obliczenia wartości hasza, wyszukiwania na indeksie haszowym nie mogą być wykonywane, jeżeli w predykatcie znajduje się tylko podzbiór tych kolumn. Na przykład, jeżeli indeks zdefiniowany jest jako nazwisko i imię, nie może zostać użyty w predykatcie równości wykorzystującym tylko nazwisko lub tylko imię. Indeksy haszowe nie mogą być wykorzystywane do zwracania rekordów posortowanych według definicji indeksu. Indeksy zakresowe mogą być stosowane we wszystkich tych scenariuszach, a ich dodatkową zaletą jest brak konieczności definiowania liczby wiader. Chociaż indeksy zakresowe mają kilka zalet w porównaniu z indeksami haszowymi, pamiętaj, że mogą prowadzić do nieoptymalnego wykonywania wyszukiwań w sytuacjach, gdzie zalecane są indeksy haszowe.

Indeksy zakresowe to nowy rodzaj B-drzew, nazywany Bw-drzewami, który został zaprojektowany z myślą o nowym sprzęcie wykorzystującym pamięć podręczną nowoczesnych procesorów wielordzeniowych. Są to struktury przechowywane w pamięci, które osiągają ogromną wydajność dzięki technikom wolnym od blokad i zostały pierwotnie opisane przez Justina Levandoskiego i innych członków Microsoft Research w artykule *The Bw-Tree: A B-tree for New Hardware Platforms*. Rysunek 7.4 przedstawia ogólną strukturę Bw-drzewa.

Tabela mapowania stron

**Rysunek 7.4.** Struktura Bw-drzewa

Podobnie jak w B-drzewie, opisanym w rozdziale 5., w Bw-drzewie korzeń i strony poziomów pośrednich wskazują na strony indeksu na kolejnym poziomie drzewa, a strony liście zawierają zestaw posortowanych wartości klucza ze wskaźnikami na wiersz. Każda strona niebędąca liściem ma logiczny identyfikator strony (PID), a tabela mapująca strony zawiera mapowanie tych identyfikatorów na ich fizyczne adresy w pamięci. Na przykład na rysunku 7.4 najwyższy wpis tabeli mapującej strony ma numer 0 i wskazuje na PID 0. Klucze na stronach poziomów pośrednich zawierają najwyższą możliwą wartość, do której odnosi się strona. Strony liście nie mają identyfikatora PID, lecz jedynie adres pamięci wskazujący na rekordy danych. Aby więc znaleźć klucz równy 2, SQL Server zacząłby od korzenia (PID 0), podążyłby połączeniem do klucza równego 10 (ponieważ 10 jest większe niż 2), które wskazuje na PID 3, a następnie do klucza równego 5 (który także jest większy niż 2). Wskazywana strona liść zawiera klucz 2, który przechowuje adres pamięci wskazujący na wymagany rekord. Jeżeli interesują Cię dalsze szczegóły tych nowych struktur, możesz zajrzeć do wspomnianego wyżej artykułu.

Przykłady

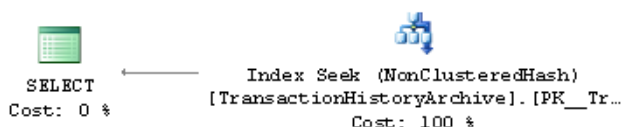
Uruchommy teraz kilka przykładowych zapytań, aby zrozumieć, jak działają te indeksy i jakie plany wykonania tworzą. W tym podrozdziale używam zapytań T-SQL *ad hoc*, które w Hekatonie korzystają z funkcjonalności współpracy. Przykłady korzystają z poniższej tabeli, mającej zarówno indeks haszowy, jak i zakresowy oraz dane załadowane zgodnie z poprzednim opisem:

```
CREATE TABLE TransactionHistoryArchive (
    TransactionID int NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH
        (BUCKET_COUNT = 100000),
    ProductID int NOT NULL,
    ReferenceOrderID int NOT NULL,
    ReferenceOrderLineID int NOT NULL,
    TransactionDate datetime NOT NULL,
    TransactionType nchar(1) NOT NULL,
    Quantity int NOT NULL,
    ActualCost money NOT NULL,
    ModifiedDate datetime NOT NULL,
    INDEX IX_ProductID NONCLUSTERED (ProductID)
) WITH (MEMORY_OPTIMIZED = ON)
```

Najpierw uruchom poniższe zapytanie:

```
SELECT * FROM TransactionHistoryArchive
WHERE TransactionID = 8209
```

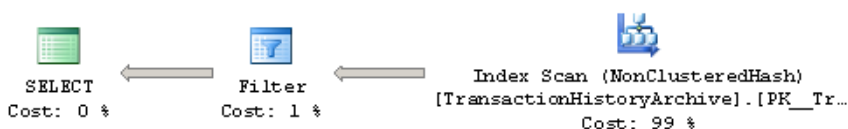
Ponieważ dla kolumny TransactionID zdefiniowaliśmy indeks haszowy, otrzymamy plan pokazany na rysunku 7.5, który wykorzystuje operator *Index Seek*.



Rysunek 7.5. Operacja Index Seek dla indeksu haszowego

Wykorzystany indeks ma nazwę PK__Transact__55433A4A7EB94404. Nazwa została nadana automatycznie, ale możliwe jest też samodzielne nadanie nazwy indeksowi, zgodnie z następnym przykładem. Jak już wspominałem, indeksy haszowe są wydajne w przypadku wyszukikań punktowych, nie wspierają jednak przeszukiwania indeksu dla predykatów z nierównościami. Jeżeli zmienisz poprzednie zapytanie, aby korzystało z operatora nierówności, jak poniżej, stworzony zostanie plan z operatorem *Index Scan* (zobacz rysunek 7.6).

```
SELECT * FROM TransactionHistoryArchive
WHERE TransactionID > 8209
```



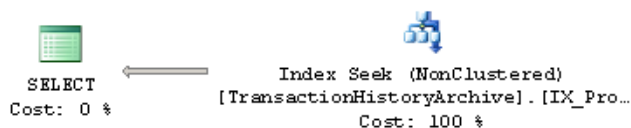
Rysunek 7.6. Operacja Index Scan na indeksie haszowym

Wypróbujmy teraz inne zapytanie:

```
SELECT * FROM TransactionHistoryArchive
WHERE ProductID = 780
```

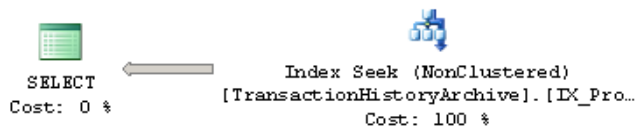
Ponieważ dla kolumny *ProductID* mamy zdefiniowany indeks zakresowy, wykorzystana zostanie operacja *Index Seek* na tym indeksie, *IX_ProductID* (zobacz rysunek 7.7). Wyprobujmy teraz operator nierówności dla tej samej kolumny:

```
SELECT * FROM TransactionHistoryArchive
WHERE ProductID < 780
```



Rysunek 7.7. Operacja Index Seek dla indeksu zakresowego

Tym razem do pobrania danych może być wykorzystany indeks zakresowy, a operacja *Indeks Scan* nie jest konieczna. Wynikowy plan został pokazany na rysunku 7.8.



Rysunek 7.8. Operacja Index Seek dla indeksu haszowego z predykatem nierówności

Indeksy haszowe nie mogą być stosowane do zwracania posortowanych danych, ponieważ wiersze przechowywane są w losowej kolejności. Poniższe zapytanie wykorzystuje operator *Sort* do posortowania żądanych danych (zobacz rysunek 7.9).

```
SELECT * FROM TransactionHistoryArchive
ORDER BY TransactionID
```

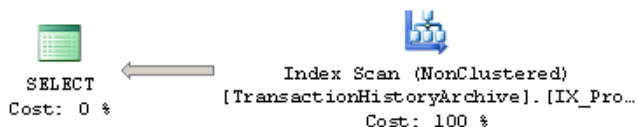


Rysunek 7.9. Operacja Sort zastosowana do posortowania danych z indeksu haszowego

Indeks zakresowy może być wykorzystany do zwrócenia posortowanych danych. Uruchomienie poniższego zapytania przeskanuje indeks zakresowy bez konieczności dodatkowego sortowania danych (zobacz rysunek 7.10).

```
SELECT * FROM TransactionHistoryArchive
ORDER BY ProductID
```

Jednak, w przeciwieństwie do indeksów dyskowych, indeksy zakresowe są jednokierunkowe, co może stanowić zaskoczenie. Dlatego wykonanie tego samego zapytania



Rysunek 7.10. Wykorzystanie indeksu zakresowego do pobrania posortowanych danych

z klauzulą `ORDER BY ProductID DESC` nie spowoduje wykorzystania danych posortowanych przez indeks, a zamiast tego zastosowany zostanie plan z operatorami *Index Scan* i *Sort*, podobny do planu z rysunku 7.9.

Spójrzmy teraz na przykład indeksu haszowego z dwiema kolumnami. Załóżmy, że masz poniższą wersję tabeli `TransactionHistoryArchive`, która używa indeksu haszowego założonego na kolumnach `TransactionID` i `ProductID`:

```

CREATE TABLE TransactionHistoryArchive (
    TransactionID int NOT NULL,
    ProductID int NOT NULL,
    ReferenceOrderID int NOT NULL,
    ReferenceOrderLineID int NOT NULL,
    TransactionDate datetime NOT NULL,
    TransactionType nchar(1) NOT NULL,
    Quantity int NOT NULL,
    ActualCost money NOT NULL,
    ModifiedDate datetime NOT NULL,
    CONSTRAINT PK_TransactionID_ProductID PRIMARY KEY NONCLUSTERED
        HASH (TransactionID, ProductID) WITH (BUCKET_COUNT = 100000)
) WITH (MEMORY_OPTIMIZED = ON)
  
```

Ponieważ, zgodnie z wcześniejszymi wyjaśnieniami, aby obliczyć wartość hasza, konieczne są obie kolumny, dla poniższego zapytania SQL Server będzie w stanie wykorzystać indeks `PK_TransactionID_ProductID`:

```

SELECT * FROM TransactionHistoryArchive
WHERE TransactionID = 7173 AND ProductID = 398
  
```

Ale nie dla poniższego — tutaj konieczny będzie operator *Index Scan*:

```

SELECT * FROM TransactionHistoryArchive
WHERE TransactionID = 7173
  
```

Chociaż plany wykonania pokazane w tym podrozdziale korzystają ze znajomo wyglądających operatorów *Index Scan* i *Index Seek*, pamiętaj, że te indeksy i dane znajdują się w pamięci, a dysk nie jest wykorzystywany. W tym przypadku zarówno tabela, jak i indeksy znajdują się w pamięci i korzystają z innych struktur. Jak więc je zidentyfikować w planach wykonania, szczególnie jeżeli odpytujesz zarówno tabele przechowywane w pamięci, jak i na dysku? Możesz spojrzeć na właściwość `Storage` (magazyn) we właściwościach operatora — ma ona wartość `MemoryOptimized` (zobacz rysunek 7.11). Chociaż dla indeksów haszowych po nazwie operatora możesz zobaczyć `NonClustered Hash`, indeksy zakresowe pokazują tylko `NonClustered` i jest to wartość identyczna jak dla indeksów przechowywanych na dysku.

Index Scan (NonClusteredHash)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	MemoryOptimized
Actual Number of Rows	89253
Actual Number of Batches	0
Estimated I/O Cost	0
Estimated Operator Cost	2.16269 (99%)
Estimated Subtree Cost	2.16269
Estimated CPU Cost	2.16269
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	131072
Estimated Row Size	54 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1
Object	
[Test],[dbo].[TransactionHistoryArchive].	
[PK_TransactionID_ProductID]	

Rysunek 7.11. Właściwości operatora Index Scan dla indeksu haszowego

Natywnie kompilowane procedury przechowywane

Jak już wspomniałem, aby stworzyć procedury kompilowane natywnie, Hekaton korzysta z optymalizatora SQL Servera do stworzenia efektywnego planu wykonania, który jest potem kompilowany do kodu natywnego i ładowany jako biblioteki DLL do procesu SQL Servera. Procedury kompilowane natywnie przydają się głównie w sytuacjach, w których wydajność jest kluczowa, lub w przypadku często uruchamianego procesu. Musisz jednak mieć świadomość ograniczeń funkcjonalności T-SQL wspieranych dla natywnie kompilowanych procedur w SQL Serverze 2014, które również omówię w dalszej części tego podrozdziału.

Tworzenie natywnie kompilowanych procedur przechowywanych

Stwórzmy teraz natywnie kompilowaną procedurę przechowywaną, która, jak pokażę w tym przykładzie, wymaga klauzuli `NATIVE_COMPILATION`:

```
CREATE PROCEDURE test
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
```

```

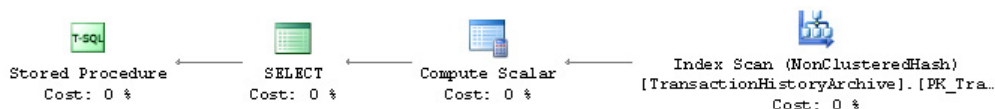
LANGUAGE = 'us_english')
SELECT TransactionID, ProductID, ReferenceOrderID
FROM dbo.TransactionHistoryArchive
WHERE ProductID = 780
END

```

Po stworzeniu procedury możesz ją uruchomić za pomocą poniższego polecenia:

```
EXEC test
```

Możesz również wyświetlić wybrany plan wykonania za pomocą opcji *Display Estimated Execution Plan* (wyświetl szacowany plan wykonania) lub polecenia SET SHOWPLAN_XML ON. W obu przypadkach otrzymamy plan pokazany na rysunku 7.12, gdzie korzeniem planu nie jest już operator SELECT, lecz operator *Stored Procedure* (procedura przechowywana).



Rysunek 7.12. Plan wykonania procedury kompilowanej natywnie

Oprócz NATIVE_COMPILATION kod pokazuje inne wymagane klauzule: SCHEMABINDING, EXECUTE AS i BEGIN ATOMIC. Pominięcie którejkolwiek z nich spowoduje błąd. Opcje te muszą być ustawione w czasie kompilacji i mają na celu zminimalizowanie sprawdzeń i operacji w czasie wykonywania, a tym samym podniesienie wydajności.

SCHEMABINDING odnosi się do faktu, że natywnie kompilowane procedury przechowywane muszą być powiązane ze schematem, co oznacza, że tabele, do których odnosi się procedura, nie mogą zostać usunięte. Pozwala to uniknąć kosztownych blokad stabilności schematu przed wykonaniem. Próba usunięcia tabeli dbo.TransactionHistoryArchive spowoduje wystąpienie poniższego błędu:

```

Msg 3729, Level 16, State 1, Line 65
Cannot DROP TABLE 'dbo.TransactionHistoryArchive' because it is being referenced by object
'<u>'test'</u>'. // Nie można usunąć tabeli 'dbo.TransactionHistoryArchive', ponieważ jest wykorzystywana przez obiekt
'<u>'test'</u>'.

```

EXECUTE AS skupia się na uniknięciu sprawdzania uprawnień w czasie wykonywania. W procedurach kompilowanych natywnie domyślna opcja EXECUTE AS CALLER (wykonaj jako uruchamiający) nie jest dostępna, więc musisz wykorzystać jedną z trzech dostępnych opcji: EXECUTE AS SELF (wykonaj jako ty sam), EXECUTE AS OWNER (wykonaj jako właściciel) lub EXECUTE AS nazwa_uzytkownika.

BEGIN ATOMIC jest częścią standardu ANSI SQL i w SQL Serverze wykorzystywana jest do definiowania atomowego bloku, gdzie cały blok kończy się powodzeniem lub wszystkie polecenia w bloku są wycofywane. Jeżeli procedura zostanie wywołana poza kontekstem aktywnej transakcji, BEGIN ATOMIC rozpocznie nową transakcję, a blok będzie definiował początek i koniec transakcji. Jeżeli jednak transakcja jest już rozpoczęta, granice transakcji będą definiowane przez polecenia BEGIN TRANSACTION,

COMMIT TRANSACTION i ROLLBACK TRANSACTION. BEGIN ATOMIC wspiera pięć opcji: TRANSACTION ISOLATION LEVEL (poziom izolacji transakcji) i LANGUAGE (język), które są wymagane, oraz opcjonalne DELAYED_DURABILITY (opóźniona trwałość), DATEFORMAT (format daty) i DATEFIRST (najpierw data).

TRANSACTION ISOLATION LEVEL definiuje poziom izolacji transakcji, który ma być użyty w natywnie kompilowanej procedurze, a wspierane wartości to: SNAPSHOT (migawka), REPEATABLE READ (powtarzalny odczyt) i SERIALIZABLE (serializowalny). LANGUAGE definiuje język wykorzystywany przez procedurę przechowywaną i determinuje formaty daty i godziny oraz komunikaty systemowe. Języki zdefiniowane są w sys.syslanguages. DELAYED_DURABILITY jest wykorzystywana do specyfikowania trwałości transakcji i domyślnie ma wartość OFF, co oznacza, że transakcje są w pełni trwałe. Kiedy opcja DELAYED_DURABILITY jest włączona, zapisy danych transakcji są asynchroniczne i mogą podnieść wydajność transakcji poprzez zapis rekordów logów transakcji w partiach, ale może to również prowadzić do utraty danych w przypadku awarii systemu.



UWAGA

Opóźniona trwałość transakcji to nowa funkcjonalność SQL Servera, która może być również wykorzystywana poza mechanizmami Hekatona i jest przydatna, kiedy występują problemy z wydajnością przez opóźnienie w zapisach logów transakcji, a ewentualna utrata danych jest tolerowalna. Więcej informacji na temat opóźnionej trwałości znajdziesz pod adresem [http://msdn.microsoft.com/en-us/library/dn449490\(v=sql.120\).aspx](http://msdn.microsoft.com/en-us/library/dn449490(v=sql.120).aspx).

Jak już wspomniałem, tabele Hekatona wspierają poziomy izolacji SNAPSHOT, REPEATABLE READ oraz SERIALIZABLE i korzystają z mechanizmu MVCC. Poziom izolacji można wyspecyfikować za pośrednictwem klauzuli ATOMIC, jak zrobiliśmy to wcześniej, lub bezpośrednio w interpretowanym kodzie T-SQL za pomocą polecenia SET TRANSACTION ISOLATION LEVEL. Warto zauważyć, że chociaż mechanizmy wersjonowania rekordów są dostępne również dla tabel dyskowych, wykorzystują tylko poziomy SNAPSHOT i READ_COMMITTED_SNAPSHOT. Ponadto wersje w Hekatonie nie są przechowywane w tempdb, co ma miejsce w przypadku tabel dyskowych, lecz w pamięci jako część struktur zoptymalizowanych do działania w pamięci. W przypadku tabel zoptymalizowanych do działania w pamięci nie ma blokad. Brak konfliktów jest optymistycznym założeniem, więc jeżeli dwie transakcje spróbują zaktualizować ten sam rekord, wystąpi konflikt zapis – zapis. Ponieważ Hekaton nie wykorzystuje blokad, także podpowiedzi blokad nie są dostępne.

Zalecane jest ręczne aktualizowanie statystyk przed stworzeniem natywnie kompilowanych procedur przechowywanych. Chociaż statystyki są automatycznie tworzone dla tabel zoptymalizowanych do działania w pamięci, nie są automatycznie aktualizowane wraz ze zmianami danych, co ma miejsce dla domyślnej konfiguracji tabel dyskowych. Oznacza to, że kiedy tworzysz tabelę, dane statystyk nie istnieją, ponieważ tabela jest pusta, i tak samo jest, kiedy dane są ładowane lub zapisywane w tabeli.

Na przykład uruchomienie poniższego zapytania po stworzeniu tabeli Transaction ↪HistoryArchive da wynik pokazany na rysunku 7.13:

```
DBCC SHOW STATISTICS(TransactionHistoryArchive, PK_TransactionID_ProductID)
```

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	PK_TransactionID_ProductID	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
<div> All density Average Length Columns </div>										
<div> RANGE_HI_KEY RANGE_ROWS EQ_ROWS DISTINCT_RANGE_ROWS AVG_RANGE_ROWS </div>										

Rysunek 7.13. Wynik polecenia DBCC SHOW_STATISTICS dla tabeli TransactionHistoryArchive

Ponieważ statystyki nie są aktualizowane automatycznie, otrzymasz ten sam wynik niezależnie od tego, ile danych zostanie załadowanych do tabeli. Aby zaktualizować statystyki, uruchom poniższe polecenie:

```
UPDATE STATISTICS TransactionHistoryArchive WITH FULLSCAN, NORECOMPUTE
```

DBCC SHOW_STATISTICS zwróci teraz lepsze informacje statystyk, zgodnie z rysunkiem 7.14.

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	PK_TransactionID_ProductID	Apr 9 2014 11:22PM	89253	89253	3	1	8	NO	NULL	89253

	All density	Average Length	Columns
1	1.120411E-05	4	TransactionID
2	1.120411E-05	8	TransactionID, ProductID

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1	0	1
2	89252	89250	1	89250	1
3	89253	0	1	0	1

Rysunek 7.14. Wynik polecenia DBCC SHOW_STATISTICS po aktualizacji statystyk

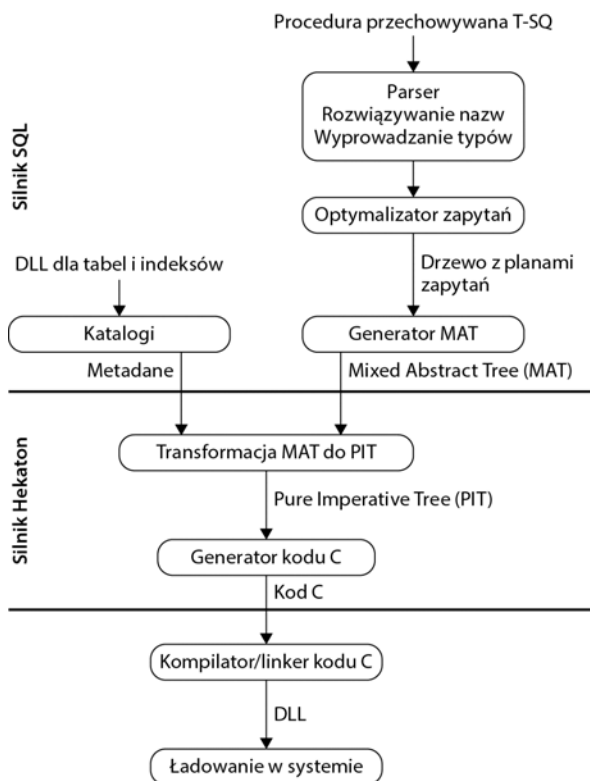
Zauważ, że klauzule FULLSCAN i NORECOMPUTE dla tabel zoptymalizowanych do pracy w pamięci są obowiązkowe.

DLL

Silnik Hekaton wykorzystuje pewne komponenty silnika SQL Servera służące do kompilacji procedur natywnych, a dokładniej stos przetwarzania zapytań, który omówiliśmy w rozdziałach 1. i 3., do parsowania, łączenia i optymalizacji zapytań. Ponieważ jednak ostatecznym celem jest wyprodukowanie kodu natywnego, dodane są również inne operacje budujące wynikową bibliotekę DLL.

Przełożenie planu stworzonego przez optymalizator na kod C nie jest prostym zadaniem i do wykonania tej operacji konieczne są dodatkowe kroki. Pamiętaj, że kod natywny jest również tworzony, kiedy zakładana jest tabela. Operacje na tabelach,

Stworzony plan jest najpierw wykorzystywany do stworzenia struktury danych o nazwie Mixed Abstract Tree (MAT), która jest później transformowana do struktury, którą łatwiej przełożyć na kod C — struktura ta nazywa się Pure Imperative Tree (PIT). W następnym etapie kod C generowany jest za pomocą PIT, a końcowym krokiem jest skompilowanie kodu C/C++ do stworzenia biblioteki DLL. W procesie tym wykorzystywane są programy Microsoft C/C++ Optimizing Compiler i Microsoft Incremental Linker w formie plików *cl.exe* i *link.exe* dostępnych w ramach instalacji SQL Servera 2014.



Rysunek 7.15. Architektura kompilera Hekatona

Biblioteki DLL dla tabel i procedur kompilowanych są rekompilowane podczas odtwarzania bazy przy każdym starcie instancji SQL Servera. SQL Server przechowuje informacje konieczne do odtworzenia bibliotek w postaci metadanych.

Warto zauważyć, że wygenerowane pliki DLL nie są przechowywane w bazie danych, lecz w systemie plików, i możesz je znaleźć, wraz z kilkoma innymi pośrednimi plikami, przeglądając lokalację zwróconą przez poniższe zapytanie:

```
SELECT name, description FROM sys.dm_os_loaded_modules
where description = 'XTP Native DLL'
```

Ja otrzymałem poniższy wynik:

name	description
C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\xtp\↪8\xtp_t_8_885578193.dll	XTP Native DLL
C:\Program Files\Microsoft SQL Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\xtp\↪8\xtp_p_8_917578307.dll	XTP Native DLL

W katalogu *xtp* tworzony jest katalog tożsamy z identyfikatorem bazy danych, w tym przypadku 8. Nazwy plików DLL zaczynają się od *xtp*, potem *t* dla tabel i *p* dla procedur. Znow w nazwie wykorzystywany jest identyfikator bazy danych (w tym przypadku 8), a po nim identyfikator obiektu, czyli 885578193 dla tabeli Transaction ↪HistoryArchive i 917578307 dla naszej natywnie skompilowanej procedury testowej — test.

Jeżeli zajrzysz do tych katalogów, znajdziesz tam pliki DLL, kod źródłowy w języku C i kilka innych plików, zgodnie z rysunkiem 7.16.

Nazwa	Data modyfikacji	Typ	Rozmiar
xtp_p_8_565577053.c	2015-05-30 12:40	Plik C	12 KB
xtp_p_8_565577053.dll	2015-05-30 12:40	Rozszerzenie aplik...	76 KB
xtp_p_8_565577053.obj	2015-05-30 12:40	Plik OBJ	97 KB
xtp_p_8_565577053.out	2015-05-30 12:40	Plik OUT	1 KB
xtp_p_8_565577053.pdb	2015-05-30 12:40	Plik PDB	587 KB
xtp_p_8_565577053.xml	2015-05-30 12:40	Plik XML	9 KB
xtp_t_8_277576027.c	2015-05-30 07:51	Plik C	9 KB
xtp_t_8_277576027.dll	2015-05-30 07:52	Rozszerzenie aplik...	74 KB
xtp_t_8_277576027.obj	2015-05-30 07:51	Plik OBJ	85 KB
xtp_t_8_277576027.out	2015-05-30 07:52	Plik OUT	1 KB
xtp_t_8_277576027.pdb	2015-05-30 07:52	Plik PDB	595 KB
xtp_t_8_277576027.xml	2015-05-30 07:51	Plik XML	3 KB

Rysunek 7.16. Pliki stworzone podczas procesu kompilacji

Nie ma potrzeby dbania o te pliki w żaden sposób, ponieważ SQL Server automatycznie usuwa je, kiedy przestaną być potrzebne. Jeżeli usuniesz tabelę i procedurę, które stworzyliśmy wcześniej, wszystkie te pliki zostaną automatycznie usunięte przez mechanizm *garbage collector*, chociaż może się to stać dopiero po jakimś czasie, szczególnie w przypadku tabel. Aby to sprawdzić, wykonaj poniższe polecenie:

```
DROP PROCEDURE test
DROP TABLE TransactionHistoryArchive
```

Zauważ również, że najpierw musisz usunąć procedurę, aby uniknąć błędu 3729, zgodnie z wcześniejszymi wyjaśnieniami.

Przechowywanie tych plików w systemie plików nie przedstawia zagrożenia bezpieczeństwa (na przykład w przypadku, gdyby zostały ręcznie zmienione). Za każdym razem, kiedy Hekaton musi załadować pliki DLL, na przykład wtedy, kiedy instancja jest restartowana lub baza jest wyłączana i znowu włączana, SQL Server skompiluje pliki od nowa, a istniejące pliki nie są nigdy wykorzystywane.

Ograniczenia

Bez wątplenia największym ograniczeniem Hekatona, przynajmniej w SQL Serverze 2014, jest brak możliwości modyfikowania tabel: konieczne jest stworzenie nowej tabeli z wymaganymi zmianami. Tak jest w przypadku każdej zmiany, jaką chciałbyś wprowadzić dla tabeli, na przykład dodania nowej kolumny, dodania indeksu lub zmiany liczby wiader w indeksie haszowym. Oczywiście stworzenie nowej tabeli wymaga kilku innych operacji, takich jak skopiowanie danych z tabeli do innego miejsca, usunięcie tabeli, stworzenie nowej tabeli ze zmianami i powtórne wstawienie danych, co wiąże się z tym, że przez pewien okres aplikacja nie będzie działać. To ograniczenie będzie prawdopodobnie największym problemem dla działających aplikacji i będzie wymagało przemyślenia i zaprojektowania odpowiedniej architektury pozwalającej uniknąć lub zminimalizować zmiany po wdrożeniu tabel zoptymalizowanych do działania w pamięci w środowisku produkcyjnym.

Ponieważ pamięć nie jest zwalniana natychmiast po usunięciu tabeli, potrzebna jest dodatkowa pamięć, nawet jeżeli do tymczasowego przechowywania danych wykorzystana zostanie tabela dyskowa. Jako tabelę tymczasową można również wykorzystać tabelę zoptymalizowaną do działania w pamięci, ale to jeszcze bardziej powiększy ilość wymaganej pamięci.

Ponadto usunięcie i stworzenie tabeli zazwyczaj implikuje konieczność wykonania innych operacji, na przykład stworzenia skryptu ze wszystkimi uprawnieniami. A ponieważ procedury kompilowane natywnie są powiązane ze schematem, one także będą musiały zostać usunięte, zanim będzie można usunąć tabelę. Musisz więc stworzyć skrypt tych procedur, usunąć je i stworzyć na nowo po powtórnym założeniu

tabeli. Aktualizacja statystyk z opcją `FULLSCAN` również jest bardzo zalecana po stworzeniu tabeli i załadowaniu do niej wszystkich danych, aby optymalizator mógł stworzyć najlepszy możliwy plan.

Natycznie kompilowanych procedur przechowywanych również nie można zmieniać — ani nawet ich zrekompilować (poza kilkoma wyjątkami, na przykład kiedy restartowana jest instancja SQL Servera lub kiedy baza danych jest wyłączana i włączana powtórnie). Jak już wspomniałem, aby zmodyfikować procedurę, będziesz musiał stworzyć skrypt uprawnień, usunąć procedurę, stworzyć nową wersję procedury i na nowo ustawić uprawnienia. To oznacza, że podczas tego procesu procedura będzie niedostępna.

Jest też kilka różnic dotyczących statystyk i rekompilacji pomiędzy Hekatonem oraz standardowymi tabelami dyskowymi i tradycyjnymi procedurami przechowywanymi. Podczas gdy zmianie ulegają dane w Hekatonie, statystyki nigdy nie są aktualizowane automatycznie; będziesz więc musiał aktualizować je ręcznie za pomocą polecenia `UPDATE STATISTICS` z opcją `FULLSCAN` lub `NORECOMPUTE`. Nawet po zaktualizowaniu statystyk istniejące natycznie skompilowane procedury nie będą mogły automatycznie z nich skorzystać i, jak już wspomniałem, nie możesz ich zrekompilować. Będziesz musiał ręcznie usunąć i powtórnie stworzyć procedury.

Tabele Hekatona i procedury przechowywane nie wspierają wszystkich poleceń T-SQL dostępnych w standardowych procedurach przechowywanych. Na przykład poniższe funkcjonalności nie są dostępne dla tabel zoptymalizowanych do działania w pamięci:

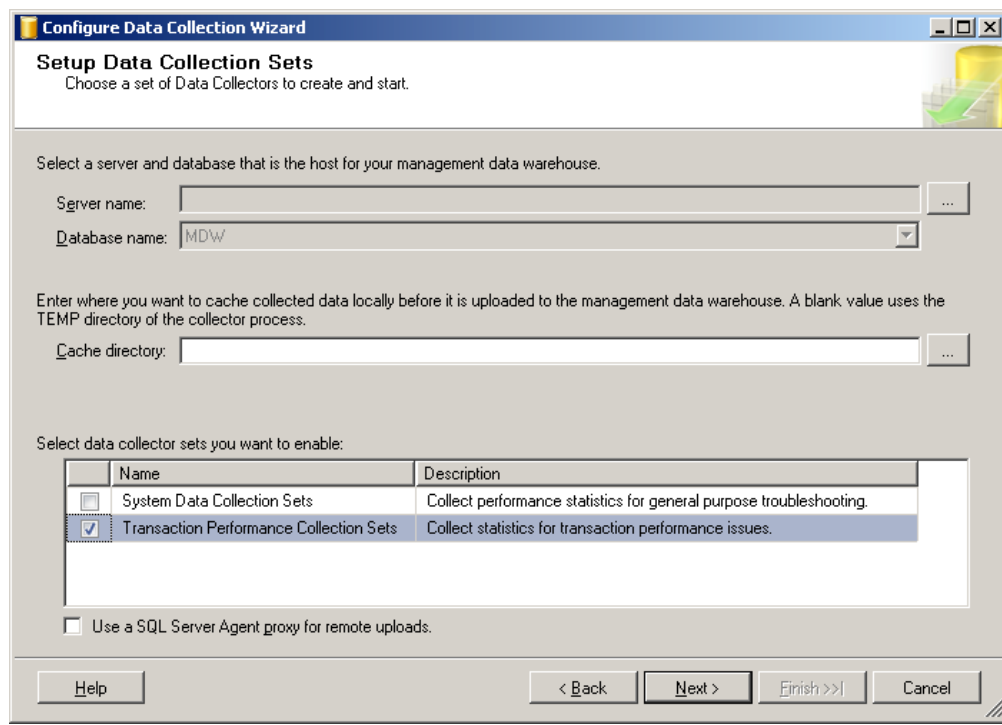
- ▶ kolumny `IDENTITY` — tylko częściowo wspierane (więcej szczegółów znajdziesz pod adresem [http://msdn.microsoft.com/en-us/library/dn247640\(v=sql.120\).aspx](http://msdn.microsoft.com/en-us/library/dn247640(v=sql.120).aspx));
- ▶ ograniczenia `FOREIGN KEY`;
- ▶ ograniczenia `CHECK`;
- ▶ ograniczenia `DEFAULT`;
- ▶ kolumny wyliczeniowe;
- ▶ wyzwalacze `DML`;
- ▶ niektóre typy danych.

Pełną listę funkcjonalności niewspieranych przez tabele i procedury Hekatona znajdziesz pod adresem [http://msdn.microsoft.com/en-us/library/dn246937\(v=sql.120\).aspx](http://msdn.microsoft.com/en-us/library/dn246937(v=sql.120).aspx).

Narzędzie AMR

SQL Server 2014 zawiera narzędzie pomagające zdecydować, które tabele i procedury przechowywane warto przenieść do mechanizmów Hekatona lub OLTP w pamięci. Jest to AMR (*Analysis, Migration, Reporting* — analiza, migracja, raportowanie), który pokrótce opiszę w tym podrozdziale. Narzędzie to jest zintegrowane z mechanizmem

SQL Server Data Collector i aby je włączyć, musisz włączyć nowe zestawy Transaction Performance Collection Sets (zbieranie informacji o wydajności transakcji) w kreatorze Configure Data Collection Wizard (zobacz rysunek 7.17). Data Collector został przedstawiony w rozdziale 2.



Rysunek 7.17. Konfiguracja narzędzia AMR w kreatorze Data Collection Wizard

Oprócz trzech poprzednio dostępnych zestawów stworzone zostaną dwa nowe zestawy zbierania danych — Stored Procedure Usage Analysis (analiza wykorzystania procedur przechowywanych) i Table Usage Analysis (analiza wykorzystania tabel).

Kiedy skonfigurujesz nowe zestawy w mechanizmie Data Collector, będziesz gotowy do przetestowania narzędzia AMR. Najpierw musisz wygenerować aktywność w bazie danych. W tym przypadku przetestujemy poniższe procedury przechowywane na bazie AdventureWorks2012:

```
CREATE PROCEDURE test1
AS
SELECT * FROM Sales.SalesOrderHeader soh
      JOIN Sales.SalesOrderDetail sod ON soh.SalesOrderID = sod.SalesOrderID
WHERE ProductID = 870

CREATE PROCEDURE test2
AS
SELECT ProductID, SalesOrderID, COUNT(*)
```

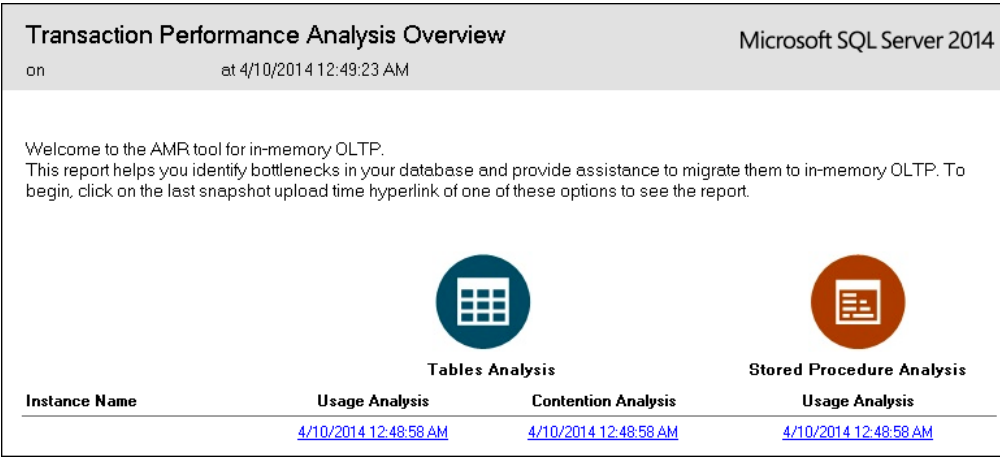
```
FROM Sales.SalesOrderDetail
GROUP BY ProductID, SalesOrderID
```

Kilkakrotnie wykonaj procedury:

```
EXEC test1
GO
EXEC test2
GO
```

Po wygenerowaniu aktywności będziesz prawdopodobnie musiał poczekać na następne wykonanie zadania aktualizacji mechanizmu Data Collector. Na przykład zadanie Stored Procedure Usage Analysis uruchamiane jest co 30 minut, a Table Usage Analysis co 15 minut. Możesz również wywołać te zadania ręcznie (odpowiednio `collection_set_5_upload` i `collection_set_6_upload`).

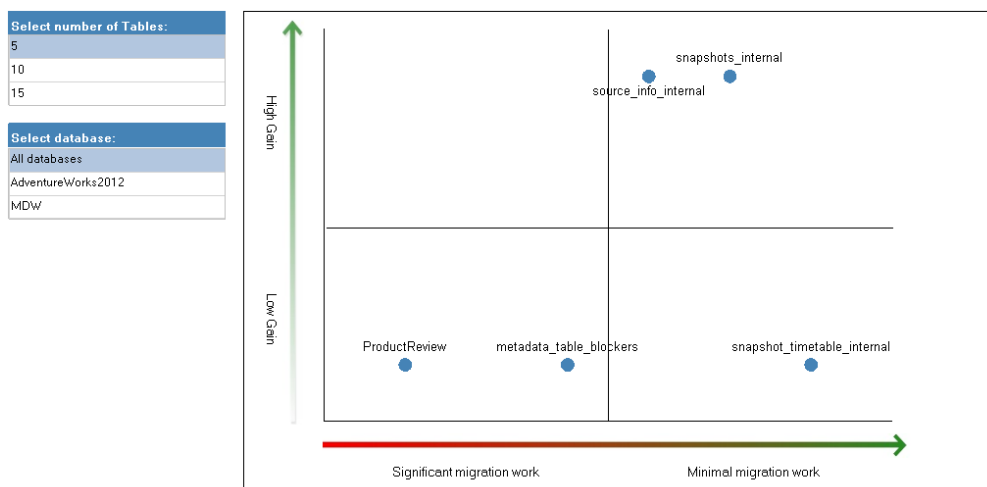
Aby uzyskać dostęp do raportów AMR, kliknij prawym przyciskiem myszy bazę *MDW*, wybierz *Reports, Management Data Warehouse* i *Transaction Performance Analysis Overview*. Raport narzędzia AMR został pokazany na rysunku 7.18.



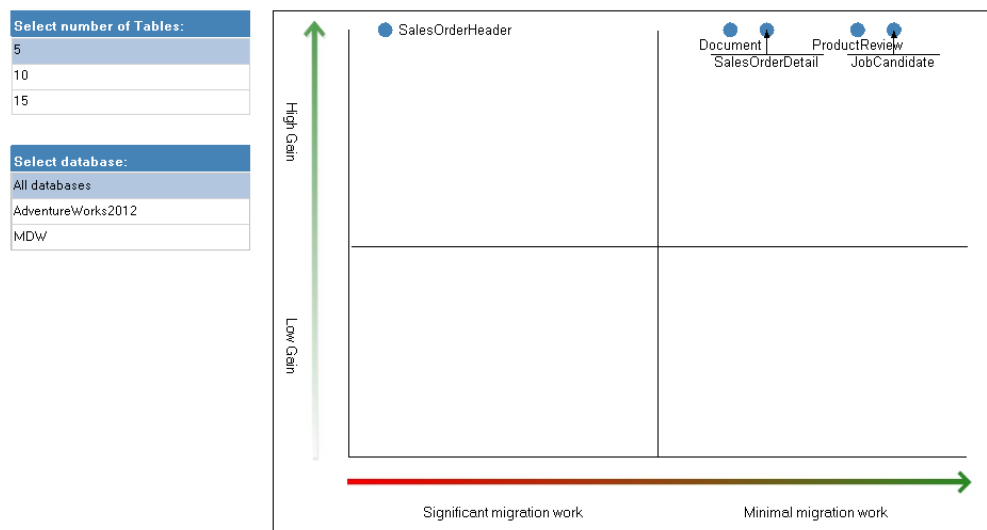
Rysunek 7.18. Raport Transaction Performance Analysis Overview

Kliknięcie *Usage Analysis* w sekcji *Tables Analysis* spowoduje wyświetlenie raportu *Recommended Tables Based on Usage* (rekomendowane tabele na podstawie wykorzystania), który na podstawie wzorców dostępu i wykonywanych zapytań przedstawi najlepszych kandydatów do przeniesienia do pamięci (zobacz rysunek 7.19). Graf w raporcie pokazuje zysk wydajności z takiej optymalizacji, a także wysiłek niezbędny do przeniesienia tabel do mechanizmów OLTP w pamięci uzależniony od tego, ile niewspieranych funkcjonalności występuje w tabeli.

Wybranie na głównym raporcie *Tables Analysis*, *Contention Analysis* spowoduje wyświetlenie raportu *Recommended Tables Based on Contention* (rekomendowane tabele na podstawie połączeń), pokazanego na rysunku 7.20.



Rysunek 7.19. Raport Recommended Tables Based on Usage



Rysunek 7.20. Raport Recommended Tables Based on Contention

W obu przypadkach raporty zalecają rozpoczęcie od tabel znajdujących się najbliżej prawego górnego rogu raportu. Kliknięcie tabeli na którymś z poprzednich raportów pokaże jej szczegóły statystyk wydajności, włączając w to statystyki wykorzystania tabeli i połączeń. Przykład został pokazany na rysunku 7.21.

		Lookup Statistics		Range Scan Statistics		Interop Gain		Native Gain	
Table Name	% of total accesses	Count	Average per Transaction	Count	Average per Transaction	Lookup	Range	Lookup	Range
SalesOrderDetail	0.8	0	0	736	27.26	1.5X-2.5X	1X-4X	2.5X	1X-4X

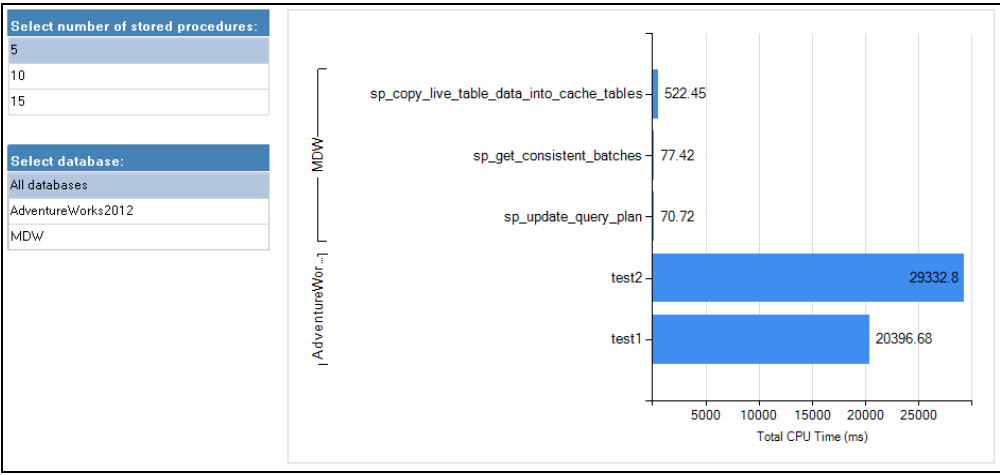
		Latch Statistics		Lock Statistics		
Table Name	% of total waits	Page latch wait count	Average wait time per latch wait (ms)	Page lock count	Page lock wait count	Average wait time per lock wait (ms)
SalesOrderDetail	0	0	0	556048	0	0

Table Name	Number of Migration Blockers
SalesOrderDetail	11

[See information for all user tables in database: AdventureWorks2012](#)

Rysunek 7.21. Raport statystyk wydajności tabeli

Natomiast wybranie na głównym raporcie *Stored Procedure Analysis, Usage Analysis* spowoduje wyświetlenie raportu *Recommended Stored Procedures Based on Usage* (rekomendowane procedury przechowywane na podstawie wykorzystania), przedstawionego na rysunku 7.22. Ten raport pokazuje procedury, które mają największy sumaryczny czas wykonywania w milisekundach.



Rysunek 7.22. Raport Recommended Stored Procedures Based on Usage

Podobnie kliknięcie konkretnej procedury przechowywanej spowoduje pokazanie statystyk dotyczących wykonywania procedury i tabel, na których ona operuje. Przykład dla procedury test2 został pokazany na rysunku 7.23.

Możesz również kliknąć wymienione tabele, aby otrzymać omawiane poprzednio informacje dotyczące ich wydajności.

Podsumowując: narzędzie AMR, które można wykorzystać z wersjami od 2008, będzie bardzo pomocne w określaniu, które tabele i procedury przechowywane mogą

Stored Procedure's Execution Statistics				
Cached Time	Total CPU Time (ms)	Total Execution Time (ms)	Total Cache Missed	Execution Count
4/10/2014 12:17:44 AM	29332.8	319281.76	289	422
Stored Procedure's Table References:				
Referenced Server	Referenced Database	Referenced Schema	Referenced Table	
	AdventureWorks2012			
		Sales	SalesOrderDetail	

Rysunek 7.23. Raport statystyk wykonywania procedury przechowywanej

zostać przeniesione do mechanizmów Hekatona. Możesz również zaktualizować bazę do SQL Servera 2014 i uruchomić to narzędzie do przeanalizowania prawdziwych danych wydajnościowych i iteracyjnie przenosić tabele i procedury przechowywane do struktury OLTP w pamięci.

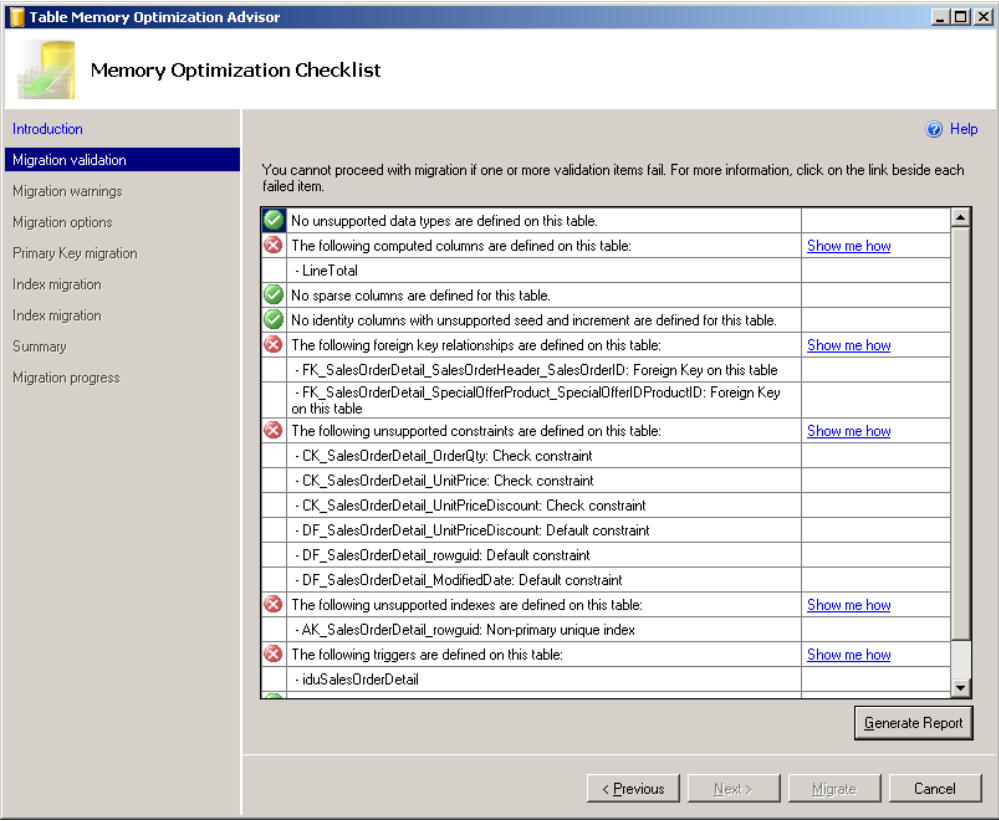
Kiedy już zidentyfikujesz tabele, które chcesz przenieść do mechanizmów Hekatona, możesz wykorzystać funkcjonalność Memory Optimization Advisor (doradca optymalizacji pamięci) w procesie migracji. Aby uzyskać dostęp do tego narzędzia, w oknie *Object Explorer* kliknij prawym przyciskiem myszy tabelę, którą chcesz zmigrować, i wybierz *Memory Optimization Advisor*. Memory Optimization Advisor może pomóc zidentyfikować niekompatybilne funkcjonalności z tabelą Hekatona, dając jednocześnie informacje o tym, jakie zmiany mogą być konieczne. Na przykład rysunek 7.24 przedstawia takie informacje dla tabeli AdventureWorks2012.Sales.SalesOrderDetail.

Memory Optimization Advisor pozwoli Ci również między innymi skonfigurować Twoją nową tabelę Hekatona przez zdefiniowanie grupy plików zoptymalizowanej do działania w pamięci, zmienić oryginalną tabelę, skopiować jej zawartość do nowej tabeli Hekatona, zdefiniować klucz główny. Na końcu kreatora będziesz mógł wybrać, czy wygenerować skrypt zmian, czy wykonać je natychmiast.

Mechanizm Native Compilation Advisor możesz też wykorzystać do zidentyfikowania niekompatybilnych elementów T-SQL w procedurach przechowywanych. Aby uzyskać do niego dostęp, kliknij prawym przyciskiem myszy procedurę, którą chcesz przeanalizować, a następnie wybierz *Native Compilation Advisor*. Program wygeneruje listę niewspieranych funkcjonalności T-SQL, które znajdują się w procedurze.

Podsumowanie

W tym rozdziale omówiłem bazy OLTP przechowywane w pamięci, znane również pod nazwą Hekaton, które bez wątpienia są najważniejszą nową funkcjonalnością w SQL Serverze 2014. Silnik Hekaton jest odpowiedzią na nowe podejście projektowe i architektoniczne mające na celu wyciągnąć jak najwięcej korzyści z aktualnie dostępnego



Rysunek 7.24. Lista sprawdzeń w programie Memory Optimization Advisor

sprzętu. Chociaż optymalizacja dla dostępu do głównej pamięci jest najważniejszym elementem, usprawnienia wydajnościowe w Hekatonie dotyczą kilku innych znaczących obszarów architektury, na przykład kompilacji procedur przechowywanych do kodu natywnego i eliminowania blokad.

W tym rozdziale omówiłem architekturę silnika Hekaton i jego główne komponenty: tabele zoptymalizowane do działania w pamięci, indeksy haszowe i indeksy zakresowe, a także bardzo dokładnie opisałem procedury przechowywane kompilowane do kodu natywnego. Objąsniałem również działanie narzędzia AMR, które może pomóc w podjęciu decyzji, które tabele i procedury przechowywane warto przenieść do OLTP w pamięci. Chociaż w porównaniu ze standardowymi bazami Hekaton ma wiele ograniczeń, Microsoft zapowiada, że ograniczenia te zostaną w przyszłości zlikwidowane.

Rozdział 8

Magazynowanie planów

W tym rozdziale:

- ▶ Kompilacja zestawów zapytań i replikacja
- ▶ Przeglądanie magazynu planów
- ▶ Parametryzacja
- ▶ Podsluchiwanie parametrów
- ▶ Podsumowanie



O mówiliśmy, w jaki sposób proces optymalizacji wytwarza plan. W tym rozdziale opisuję, co się dzieje z tymi planami. Zrozumienie tego, jak działa magazyn planów, jest bardzo ważne z punktu widzenia Twoich zapytań i wydajności samego SQL Servera. Optymalizacja zapytań to relatywnie kosztowna operacja, więc jeżeli plany mogą być przechowywane do ponownego wykorzystania, kosztu optymalizacji można uniknąć. Próba zminimalizowania tego kosztu oszczędza czas optymalizacji i zasoby serwera, na przykład CPU. Magazynowanie planów musi też uwzględniać potrzebę utrzymania jak najmniejszego magazynu, aby zasoby pamięci mogły być wykorzystywane przez zapytanie.

Są jednak przypadki, w których powtórne wykorzystanie tego samego planu może nie być pożądane i mogłoby powodować problemy wydajnościowe. W tym rozdziale pokażę, jak zidentyfikować te problemy i jakie są możliwe rozwiązania. Chociaż podsłuchiwanie parametrów jest czasami postrzegane jako coś złego, tak naprawdę jest optymalizacją wydajnościową, która pozwala, aby SQL Server zoptymalizował zapytanie dla konkretnych wartości parametrów przekazanych za pierwszym razem. Podsłuchiwanie parametrów zyskało złą sławę przez fakt, że mechanizm ten nie we wszystkich przypadkach działa dobrze — zjawisko to nazywane jest problemem podsłuchiwania parametrów.

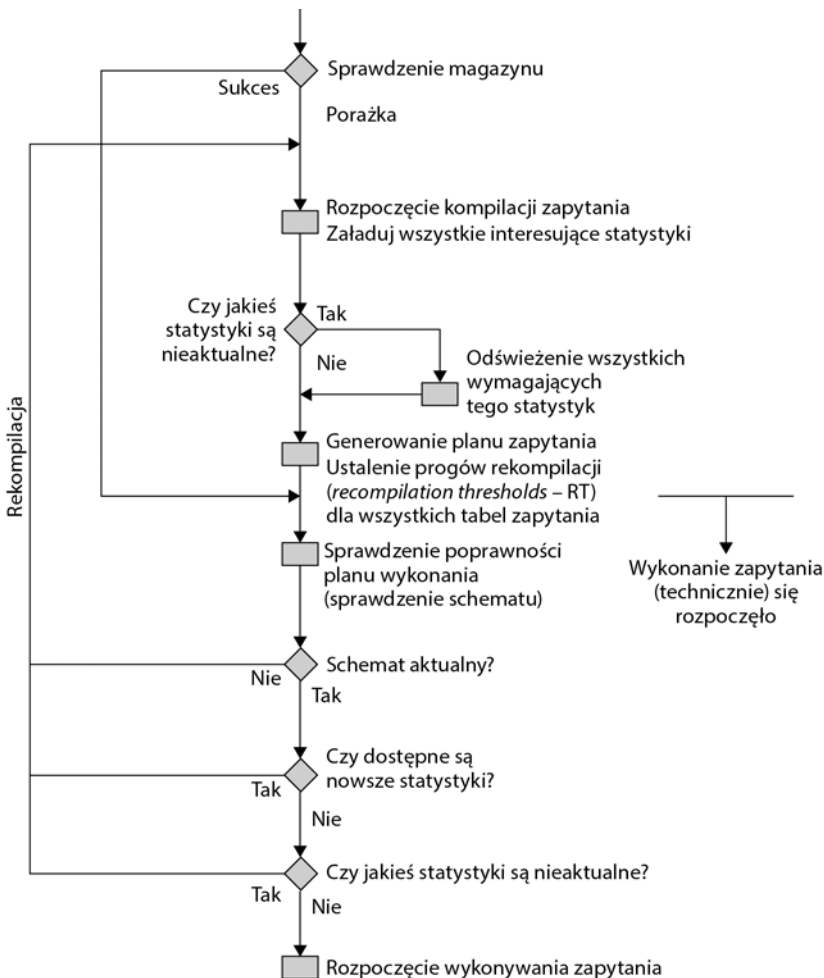
Kompilacja i rekompilacja zestawów zapytań

Jak wspominałem w rozdziale 1., za każdym razem, kiedy zestaw zapytań jest przesyłany do serwera, SQL Server najpierw sprawdza magazyn planów, aby zobaczyć, czy dla tego zestawu nie istnieje już plan wykonania. Optymalizacja zapytań to relatywnie kosztowna operacja, dlatego jeżeli w magazynie istnieje plan, który może zostać wykorzystany, optymalizację można pominąć i uniknąć związanych z nią kosztów w czasie procesora, czasie optymalizacji itd. Jeżeli plan nie zostanie znaleziony, zestaw jest kompilowany, aby wygenerować plany dla wszystkich zapytań w procedurze, wyzwalaczu lub zestawie dynamicznych zapytań SQL.

Optymalizator zaczyna od załadowania wszystkich interesujących statystyk i sprawdzenia, czy któreś z nich nie są przestarzałe. Następnie aktualizuje ewentualne przestarzałe statystyki — poza przypadkami, kiedy użyta zostanie opcja konfiguracji `AUTO_UPDATE_STATISTICS_ASYNC`, bo wówczas optymalizator skorzysta z istniejących statystyk, nawet jeżeli są nieaktualne. Statystyki zostaną zaktualizowane asynchronicznie i będą gotowe dla następnego procesu optymalizacji. Optymalizator rozpoczyna potem proces optymalizacji, który dokładnie opisuję w rozdziale 3.

Jeżeli plan zostanie znaleziony w magazynie planów lub zostanie stworzony nowy plan, zapytanie może zostać wykonane. W tym miejscu rozpoczyna się, zgodnie z rysunkiem 8.1, wykonanie zapytania, ale sprawdzana jest jeszcze poprawność planu, między innymi pod kątem ewentualnych zmian schematu. Jeżeli schemat nie jest

aktualny, plan jest odrzucany i zestaw lub pojedyncze zapytanie jest kompilowane powtórnie. Jeżeli schemat jest aktualny, optymalizator sprawdza zmiany w danych statystycznych, szukając nowych lub nieaktualnych statystyk. Jeżeli dostępne są nowsze statystyki, plan jest odrzucany i zestaw lub pojedyncze zapytanie jest kompilowane powtórnie. Takie kompilacje są znane jako *rekompilacje*. Jak być może zauważyłeś, rekompilacje są wykonywane z ważnych powodów — aby zapewnić poprawność planu i jego optymalność (czyli aby uzyskać potencjalnie szybsze plany). Rekompilacje również mogą być monitorowane, aby upewnić się, że nie występują zbyt często i nie powodują problemów wydajnościowych. Cały proces kompilacji i rekompilacji podsumowują na rysunku 8.1.



Rysunek 8.1. Proces kompilacji i rekompilacji

Nadmiarowych kompilacji i rekompilacji możesz szukać za pośrednictwem liczników *SQL Compilations/sec* (kompilacje SQL/sek.) i *SQL Re-Compilations/sec* (rekompilacje SQL/sek.) obiektu SQLServer:SQL Statistics mechanizmu Windows System Monitor. Licznik *SQL Compilations/sec* pozwala sprawdzić liczbę kompilacji na sekundę. Ponieważ plany są przechowywane i powtórnie wykorzystywane, jeżeli aktywność użytkowników SQL Servera jest stabilna, wartość ta również powinna być stabilna. *SQL Re-Compilations/sec* pozwala sprawdzić liczbę rekompilacji na sekundę. Jak pokazałem wcześniej, rekompilacje wykonywane są z ważnych powodów, ale generalnie liczba ta powinna być jak najniższa.

Kiedy już będziesz wiedzieć, jak wysoka jest liczba rekompilacji, możesz skorzystać ze zdarzeń śledzenia *SP:Recompile* i *SQL:StmtRecompile* lub zdarzenia rozszerzonego *sql_statement_recompile* do poszukiwania problemów i w celu uzyskania dodatkowych informacji. Spójrzmy na poniższe przykładowe ćwiczenie. Uruchom sesję SQL Profiler na instancji testowej i wybierz poniższe zdarzenia (niektóre z nich omówiłem w rozdziale 2). Są zlokalizowane w klasach zdarzeń Stored Procedures i TSQL.

- ▶ *SP:Recompile*
- ▶ *SQL:StmtRecompile*
- ▶ *SP:Starting*
- ▶ *SP:StmtStarting*
- ▶ *SP:Completed*
- ▶ *SP:StmtCompleted*

Uruchom poniższy kod:

```
DBCC FREEPROCCACHE
GO
CREATE PROCEDURE test
AS
CREATE TABLE #table1 (name varchar(40))
SELECT * FROM #table1
GO
EXEC test
```

Powinieneś zobaczyć poniższą sekwencję zdarzeń, która zawiera 3 – *Deferred compile* (kompilacja odroczone) w kolumnie EventSubClass dla zdarzeń *SP:Recompile* i *SQL:StmtRecompile*.

EventClass	TextData	EventSubClass
SP:Starting	EXEC test	
SP:StmtStarting	CREATE TABLE #table1 (name varchar(40))	
SP:StmtCompleted	CREATE TABLE #table1 (name varchar(40))	
SP:StmtStarting	SELECT * FROM #table1	

EventClass	TextData	EventSubClass
SP:Recompile	SELECT * FROM #table1	3 - Deferred compile
SQL:StmtRecompile	SELECT * FROM #table1	3 - Deferred compile
SP:StmtStarting	SELECT * FROM #table1	
SP:StmtCompleted	SELECT * FROM #table1	
SP:Completed	EXEC test	

Przechwycone zdarzenia pokazują odroczoną kompilację spowodowaną przez polecenie SELECT jako przyczynę rekompilacji. Pamiętaj, że kiedy procedura przechowywana jest wykonywana po raz pierwszy, jest również optymalizowana, a w wyniku optymalizacji tworzony jest plan wykonania. Plan dla polecenia CREATE TABLE znajdującego się wewnątrz procedury może zostać stworzony. Polecenie SELECT nie może zostać zoptymalizowane, ponieważ odnosi się do tabeli #table1, która w tym momencie nie istnieje. Pamiętaj, że jest to wciąż proces optymalizacji i aby stworzyć tabelę, plan wynikowy musi być najpierw wykonany. Dopiero po stworzeniu tabeli w trakcie wykonywania procedury SQL Server będzie w stanie zoptymalizować polecenie SELECT, tym razem jednak będzie to rekompilacja.

Kompilacja odroczonej to jedna z możliwych wartości pola EventSubClass. Pozostałe udokumentowane wartości możesz sprawdzić, uruchamiając następujące zapytanie:

```
SELECT map_key, map_value FROM sys.dm_xe_map_values
WHERE name = 'statement_recompile_cause'
```

Uruchomienie tego zapytania pokazuje poniższy wynik. Opisy zostały zaczerpnięte z artykułu *Plan Caching and Recompile in SQL Server 2012*.

SubclassName	SubclassValue	Szczegółowy powód rekompilacji
Schema changed	1	Schemat, łączenia lub uprawnienia uległy zmianie między kompilacją a wykonaniem.
Statistics changed	2	Statystyki zostały zmienione.
Deferred compile	3	Rekompilacja z powodu odroczonego rozwiązywania nazw (DNR — <i>Deferred Name Resolution</i>).
Set option change	4	W zestawie zmieniona została opcja SET OPTION.
Temp table changed	5	Schemat tabeli tymczasowej, łączenia lub uprawnienia uległy zmianie.
Remote rowset changed	6	Schemat zdalnego zestawu wierszy, łączenia lub uprawnienia uległy zmianie.
For browse permissions changed	7	Zmianie uległy uprawnienia dla FOR BROWSE (zdeprecjonowana opcja DBLIB).

SubclassName	SubclassValue	Szczegółowy powód rekompilacji
Query notification environment changed	8	Zmianie uległo środowisko informowania o zapytaniach.
Partition view changed	9	Dla zapytań na niektórych widokach indeksowych SQL Server czasami do klauzuli WHERE dodaje predykaty zależne od danych. Jeżeli dane ulegną zmianie, takie predykaty przestaną być aktualne, a powiązany plan wykonywania będzie wymagał rekompilacji.
Cursor options changed	10	Zmiana w opcjach kursora.
Option (Recompile) requested	11	Zażądano rekompilacji.
Parameterized plan flushed	12	Plan parametryzowany został usunięty z magazynu (SQL Server 2008 i wersje późniejsze).
Test plan linearization	13	Tylko na potrzeby testów wewnętrznych (SQL Server 2008 i wersje późniejsze).
Plan affecting database version changed	14	Tylko na potrzeby testów wewnętrznych (SQL Server 2008 i wersje późniejsze).
QDS plan forcing policy changed	15	Tylko na potrzeby testów wewnętrznych (SQL Server 2014).
QDS plan forcing failed	16	Tylko na potrzeby testów wewnętrznych (SQL Server 2014).

Przeglądanie magazynu planów

Jak widziałeś w rozdziale 2., do zwrócenia zagregowanych statystyk wydajnościowych dotyczących zmagazynowanych planów wykonywania możesz wykorzystać widok `sys.dm_exec_query_stats`, w którym każdy wpis reprezentuje polecenie w ramach zmagazynowanego planu. Widziałeś przykłady tego, jak znaleźć najbardziej kosztowne zapytania, korzystając z różnych kryteriów, takich jak czas CPU lub czas wykonywania. Wskazaliśmy również, że aby uzyskać te same informacje z przeszłości, musiałbyś uruchomić potencjalnie kosztowny proces śledzenia i analizować zebrane dane z użyciem zewnętrznych aplikacji lub stworzonych przez siebie metod, co jest bardzo czasochłonnym procesem. Chociaż informacje z widoku `sys.dm_exec_query_stats` dostępne są automatycznie bez żadnej dodatkowej konfiguracji, ma on również kilka ograniczeń — przede wszystkim nie każde zapytanie trafi do magazynu, a plan może zostać usunięty z magazynu w dowolnej chwili. Mimo tego wykorzystanie DMV jest i tak wielkim usprawnieniem w porównaniu z koniecznością uruchamiania procesów śledzenia.

Dodatkowo, jak pokazałem w rozdziale 5., mechanizm Database Engine Tuning Advisor (DTA), jeżeli wyspecyfikujesz magazyn jako zestaw zapytań do optymalizacji,

może wykorzystać informacje z widoku i wybrać najdłużej trwające zapytania. Oznacza to, że nie musisz nawet szukać najbardziej kosztownych zapytań i przekazywać ich do DTA — wszystko dostępne jest bezpośrednio po zaledwie kilku kliknięciach myszką.

Niezależnie od metody, jaką zastosujesz do przechwycenia najbardziej kosztownych zapytań, powinieneś zawsze brać pod uwagę sytuację, w której samo zapytanie nie wykorzystuje wiele zasobów (na przykład cykli CPU), ale koszt sumaryczny może być bardzo wysoki ze względu na częste wykorzystanie zapytania.

Kolejnym widokiem DMV użytecznym podczas analizowania magazynu planów jest `sys.dm_exec_cached_plans`, który zwraca wiersz dla każdego zmagazynowanego planu. Użyjemy tego widoku do eksplorowania magazynu planów w pozostałych podrozdziałach tego rozdziału, gdzie skupimy się głównie na poniższych trzech kolumnach:

- ▶ ***usecounts*** — ile razy obiekt w magazynie był pobierany.
- ▶ ***cacheobjtype*** — typ obiektu w magazynie danych, który może przyjmować jedną z poniższych wartości:
 - ▶ Compiled Plan (skompilowany plan).
 - ▶ Compiled Plan Stub (zaślepka skompilowanego planu).
 - ▶ ParseTree (drzewo parsowania) — jak wspomniałem w rozdziale 3., komponent procesora zapytań o nazwie *algebrizer* tworzy drzewo reprezentujące logiczną strukturę zapytania. Struktura nazywana jest drzewem *algebrizera*, czasami jednak może być nazywana również drzewem parsowania lub drzewem znormalizowanym i jest następnie przekazywana do optymalizatora, który buduje na jej podstawie plan wykonania. Ponieważ stworzony plan jest magazynowany, nie ma potrzeby przechowywania tych drzew (wyjątkiem są drzewa dla widoków, wartości domyślne i ograniczeń), ponieważ może się do nich odnosić wiele różnych zapytań.
 - ▶ Extended Proc — zmagazynowane obiekty, które śledzą metadane dla rozszerzonej procedury przechowywanej.
 - ▶ CLR Compiled Func (funkcja kompilowana CLR).
 - ▶ CLR Compiled Proc (procedura kompilowana CLR).
- ▶ ***objtype*** — typ obiektu, który może przyjmować jedną z poniższych wartości:
 - ▶ Proc (procedura przechowywana).
 - ▶ Prepared (polecenie przygotowane).
 - ▶ Adhoc (zapytanie *ad hoc*).
 - ▶ Rep1Proc (procedura filtra replikacji).
 - ▶ Trigger (wyzwalacz).
 - ▶ View (widok).
 - ▶ Default (wartość domyślna).

- ▶ `usrTab` (tabela użytkownika).
- ▶ `sysTab` (tabela systemowa).
- ▶ `Check` (sprawdzenie ograniczenia).
- ▶ `Rule` (reguła).

Informacje o liczbie wpisów w magazynie danych oraz ilości wykorzystanej i zaalokowanej pamięci możesz uzyskać za pomocą widoku `sys.dm_os_memory_cache_counters`. Poniższe zapytanie przedstawia szybkie podsumowanie tego, co dokładnie możesz zobaczyć w widoku `sys.dm_exec_cached_plans`:

```
SELECT * FROM sys.dm_os_memory_cache_counters
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP', 'CACHESTORE_PHDR', 'CACHESTORE_XPROC')
```

Zauważ, że zawężamy wybór do czterech poniższych magazynów:

- ▶ **CACHESTORE_OBJCP** — dla procedur, funkcji i wyzwalaczy.
- ▶ **CACHESTORE_SQLCP** — dla zapytań *ad hoc* i poleceń przygotowanych.
- ▶ **CACHESTORE_PHDR** — dla drzew algebry dla widoków, wartości domyślnych i ograniczeń.
- ▶ **CACHESTORE_XPROC** — dla procedur rozszerzonych.

Chociaż nie jest to bezpośrednio związane z magazynem planów, SQL Server 2014 (i tylko w wersji Enterprise Edition) pozwala na wykorzystanie magazynu stałego, zazwyczaj dysków typu SSD (*solid-state drives*), jako rozszerzenia dla podsystemu pamięci zamiast podsystemu dyskowego. Funkcjonalność ta jest nazywana *rozszerzeniem puli bufora* i można ją skonfigurować za pomocą klauzuli `BUFFER POOL EXTENSION` polecenia `ALTER SERVER CONFIGURATION`. Dodatkowo możesz użyć kolumny `is_in_bpool_extension` widoku `sys.dm_os_buffer_descriptors` do sprawdzenia wszystkich stron danych znajdujących się aktualnie w puli bufora SQL Servera, które są również używane przez funkcjonalność rozszerzenia puli bufora. Więcej informacji na temat tej funkcjonalności znajdziesz w dokumentacji Books Online.

Jak usuwać plany?

Dotychczas w tej książce intensywnie korzystaliśmy z polecenia `DBCC FREEPROCCACHE`, za którego pomocą możemy bardzo łatwo wyczyścić cały magazyn planów na potrzeby testowe — powinieneś już zdawać sobie sprawę, że musisz być bardzo ostrożny podczas wykonywania tego typu czynności na środowisku produkcyjnym. Istnieją jeszcze inne polecenia, które pozwalają na bardziej selektywne czyszczenie magazynu planów, który jest współdzielony przez wszystkie bazy instancji. Możesz usunąć plany dla konkretnej bazy danych, puli zarządcy zasobów lub nawet pojedynczego planu. Oto podsumowanie tych poleceń:

- ▶ `DBCC FREEPROCCACHE [({ uchwyt_planu | uchwyt_zapytania | nazwa_puli })]`

Tego polecenia można użyć do usunięcia wszystkich wpisów w magazynie, konkretnego planu poprzez wyspecyfikowanie uchwytu do planu bądź uchwytu do zapytania albo wszystkich planów powiązanych z pulą zasobów.

- ▶ `DBCC FREESYSTEMCACHE ('ALL' [,nazwa_puli])`

To polecenie usuwa wszystkie nieużywane wpisy we wszystkich magazynach, nie tylko w magazynie planów. `ALL` można użyć do wyspecyfikowania wszystkich magazynów wspierających, a *nazwa_puli* — do wyspecyfikowania magazynu zarządcy zasobów.

- ▶ `DBCC FLUSHPROCINDB(db_id)`

To polecenie można zastosować do usunięcia wszystkich wpisów w magazynie dla konkretnej bazy danych.

Ponadto musisz mieć świadomość, że wiele innych poleceń uruchamianych na instancji SQL Servera może usunąć plany dla całej instancji lub konkretnej bazy danych. Na przykład odłączenie lub przywrócenie bazy danych albo zmiana pewnych opcji konfiguracyjnych może usunąć wszystkie plany z całego magazynu. Niektóre polecenia `ALTER DATABASE` mogą usunąć wszystkie plany dla konkretnej bazy. Pełną listę znajdziesz w dokumentacji Books Online.

Parametryzacja

Pokrótce wspomniałem o autoparametryzacji w rozdziale 2., podczas omawiania wartości `query_hash` i `plan_hash`. Aby zrozumieć, w jaki sposób SQL Server przechowuje plany, wraz z różnymi mechanizmami pozwalającymi na powtórne wykorzystanie planu, musisz dokładniej zrozumieć parametryzację. Parametryzacja umożliwia powtórne wykorzystanie planu poprzez automatyczną zamianę literałów w poleceniach na parametry. Przyjrzyjmy się tym zapytaniom raz jeszcze, tym razem korzystając z widoku `sys.dm_exec_cached_plans`, który pozwoli zwrócić plany aktualnie przechowywane w magazynie SQL Servera. Jedną z kolumn, *usecounts*, będzie użyteczna, ponieważ zwraca liczbę wywołań konkretnego obiektu w magazynie planów, wskazując tym samym, ile razy wykorzystany został dany plan. Kolumny *cacheobjtype* i *objtype*, które zostały przedstawione w poprzednim podrozdziale, również będą użyteczne.

Spójrzmy na poniższe zapytanie:

```
DBCC FREEPROCCACHE
GO
SELECT * FROM Person.Address
WHERE StateProvinceID = 79
GO
SELECT * FROM Person.Address
WHERE StateProvinceID = 59
GO
```

```
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

Otrzymamy poniższy wynik (dostosowany do rozmiaru strony). Możesz zignorować pierwsze dwa rezultaty pokazujące dane z tabeli `Person.Address` i skupić się na zapytaniu korzystającym z `sys.dm_exec_cached_plans`, którego wynik pokaże się w trzecim rezultacie.

usecounts	cacheobjtype	objtype	text
1	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 59
1	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 79

W tym przypadku możemy zobaczyć, że każde zdanie lub zestaw zostało skompilowane do własnego planu wykonania, nawet jeżeli różnią się tylko wartością *StateProvinceID*. SQL Server jest domyślnie bardzo konserwatywny, jeżeli chodzi o autoparametryzowanie zapytań, dlatego w tym wypadku żaden plan nie został wykorzystany ponownie, ponieważ nie jest to bezpieczne („nie jest bezpieczne” oznacza, że taka operacja grozi potencjalną degradacją wydajności). Jeżeli przyjrzymy się planowi (na przykład za pomocą funkcji `sys.dm_exec_query_plan` omówionej w rozdziale 1.) oraz kolumnie *plan_handle* widoku `sys.dm_exec_cached_plans`, pokazanej w poniższym zapytaniu, zobaczysz, że są to różne plany zapytania — jeden wykorzystuje kombinację *Index Seek/Key Lookup*, a drugi *Clustered Index Scan*. Zapytanie zwróci tekst zapytania wraz z odnośnikiem, po którego kliknięciu zostaniesz przeniesiony do graficznego planu zapytania.

```
SELECT text, query_plan FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
WHERE text like '%Person%'
```

Jak wspomniałem w rozdziale 2., ponieważ filtr na porównaniu równości dla *StateProvinceID* może zwrócić zero, jeden lub więcej rekordów, SQL Server uznaje, że autoparametryzacja nie jest bezpieczna. Jeżeli optymalizator uzna, że dla różnych parametrów mogą być wykorzystane różne plany, parametryzacja nie jest bezpieczna.

Autoparametryzacja

Użyjmy teraz drugiej wersji zapytań:

```
DBCC FREEPROCCACHE
GO
SELECT * FROM Person.Address
WHERE AddressID = 12
GO
SELECT * FROM Person.Address
```



```
WHERE AddressID = 37
GO
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

Teraz otrzymamy poniższy wynik (także dostosowany do rozmiaru strony):

usecounts	cacheobjtype	objtype	text
1	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 59
1	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 79
2	Compiled Plan	Adhoc	(@1 tinyint)SELECT * FROM [Person].[Address] WHERE [AddressID]=@1

Ponieważ w tym przypadku *AddressID* jest częścią indeksu unikalnego, predykat równości dla tej kolumny zawsze zwróci maksymalnie jeden rekord, zatem autoparametryzacja i powtórne wykorzystanie tego samego planu jest bezpieczne. Jak widać w ostatnim rekordzie, wartość *usecounts* wynosi 2, a *objtype* ma wartość Prepared. Autoparametryzacja jest nazywana również *prostą parametryzacją* i jest zazwyczaj stosowana dla zapytań, dla których forma sparametryzowana zaowocuje planem trywialnym. Pierwsze dwa rekordy z tego przykładu są rozważanymi zapytaniami skorupowymi i nie zawierają pełnego planu zapytania, co można zweryfikować za pomocą funkcji `sys.dm_exec_query_plan` zgodnie z wcześniejszymi wyjaśnieniami.

Opcja optymalizacji dla zapytań ad hoc

Opcja *Optimize for Ad Hoc Workloads* (optymalizacja dla zapytań *ad hoc*) jest opcją konfiguracyjną wprowadzoną w SQL Serverze 2008 i może być bardzo pomocna w przypadkach, w których masz dużą liczbę zapytań *ad hoc* z niską lub żadną możliwością powtórnego wykorzystywania planów. Jeżeli zostanie użyta ta opcja, SQL Server w momencie pierwszej optymalizacji przechowa w magazynie planów małą skompilowaną zaślepkę planu zamiast pełnego planu. Dopiero po drugiej optymalizacji zaślepka zostanie zastąpiona pełnym planem. Unikanie planów, które nigdy nie są wykorzystywane ponownie, może pomóc w minimalizacji rozmiaru magazynu planów, a tym samym zwolnić pamięć systemową. Właściwie nie ma żadnych wad użycia tej opcji, możesz więc rozważyć jej włączenie dla wszystkich instalacji SQL Servera.

Spójrzmy na przykład użycia `sp_configure` do włączenia tej opcji. Wykonaj następujące polecenia:

```
EXEC sp_configure 'optimize for ad hoc workloads', 1
RECONFIGURE
DBCC FREEPROCCACHE
GO
SELECT * FROM Person.Address
WHERE StateProvinceID = 79
```

```
GO
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

W tym momencie włączyliśmy opcję *Optimize for Ad Hoc Workloads* na poziomie instancji, a po wykonaniu pierwszego polecenia SELECT otrzymamy poniższy wynik:

usecounts	size_in_bytes	cacheobjtype	objtype	text
1	352	Compiled Plan Stub	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 79

Jak widzisz, zaślepka to mały obiekt wykorzystujący niewielką liczbę bajtów (w tym przypadku 352). Kolumna *usecounts* dla skompilowanej zaślepki planu zawsze wynosi 1, ponieważ nigdy nie zostanie ona wykorzystana powtórnie. Warto również wyjaśnić, że zaślepka nie jest tym samym co zapytanie skorupowe wspomniane wcześniej w tym podrozdziale.

Uruchom teraz poniższe polecenia:

```
SELECT * FROM Person.Address
WHERE StateProvinceID = 79
GO
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

Otrzymamy poniższy wynik:

usecounts	size_in_bytes	cacheobjtype	objtype	text
1	16384	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 79

Po drugiej optymalizacji zapytania zaślepka jest zamieniana na pełen plan wykonywania, zgodnie z kolumną *cacheobjtype*. Zauważ również, że rozmiar planu jest zdecydowanie większy niż w przypadku zaślepki (tutaj 16384 bajty).

Pamiętaj jednak, że chociaż opcja ta może być użyteczna w scenariuszach, w których możesz nie mieć wpływu na zapytania wykonywane na serwerze, nie oznacza to, iż pisanie dużej liczby zapytań *ad hoc* jest zalecane. Zalecane jest korzystanie z bezpośredniej parametryzacji (na przykład poprzez stosowanie procedur przechowywanych). Chociaż warto mieć tę opcję włączoną, nie zapomnij jej wyłączyć, aby kontynuować testowanie kodu z książki z domyślną konfiguracją:

```
EXEC sp_configure 'optimize for ad hoc workloads', 0
RECONFIGURE
```

Wymuszona parametryzacja

Pamiętasz pierwszy przykład tego podrozdziału, korzystający z predykatu `StateProvinceID = 79`, który nie był odpowiedni do parametryzacji? Mogą występować przypadki specjalne, w których uznasz, że warto sparametryzować podobne zapytania, ponieważ użycie tego samego planu pozwoli uzyskać lepszą wydajność. Chociaż mógłbyś stworzyć procedury przechowywane, które umożliwią uzyskanie tego samego efektu, jeżeli Twoja aplikacja generuje zapytania *ad hoc*, w SQL Serverze jest opcja, która pozwoli Ci to osiągnąć bez konieczności zmieniania choćby jednej linii kodu aplikacji. Ta opcja to wymuszona parametryzacja i może być ustawiona na poziomie bazy danych lub dla konkretnego zapytania. Wymuszona parametryzacja odnosi się do poleceń `SELECT`, `INSERT`, `UPDATE` i `DELETE` i podlega pewnym ograniczeniom udokumentowanym w Books Online.

Aby włączyć tę funkcjonalność i przetestować jej działanie, włącz wymuszoną parametryzację na poziomie bazy danych za pomocą poniższego zapytania:

```
ALTER DATABASE AdventureWorks2012 SET PARAMETERIZATION FORCED
```

A następnie po raz kolejny uruchom poniższe zapytania:

```
DBCC FREEPROCCACHE
GO
SELECT * FROM Person.Address
WHERE StateProvinceID = 79
GO
SELECT * FROM Person.Address
WHERE StateProvinceID = 59
GO
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

W przeciwieństwie do pierwszego przykładu, kiedy to otrzymaliśmy dwa odrębne plany dedykowane dla poszczególnych zapytań, tym razem otrzymamy tylko jeden, pokazany w poniższym wyniku:

usecounts	cacheobjtype	objtype	text
1	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 59
1	Compiled Plan	Adhoc	SELECT * FROM Person.Address WHERE StateProvinceID = 79
2	Compiled Plan	Adhoc	(@0 int)select * from Person . Address where StateProvinceID = @0

W tym przypadku mamy tylko jeden plan, pokazany w trzecim wierszu. Pierwsze dwa wiersze to nie plany wykonania, lecz zapytania skorupowe.

Pewnie pamiętasz pierwotne plany stworzone w pierwszym przykładzie tego podrozdziału, gdzie jeden korzystał z operatora *Clustered Index Scan*, a drugi z kombinacji

operatorów *Index Seek/Key Lookup*. Być może zastanawiasz się, który plan został wybrany jako plan współdzielony. Skoro dotarłeś aż do tego miejsca w książce, na pewno zgadłeś, że plan został zdefiniowany podczas pierwszej optymalizacji. Jeżeli najpierw wykorzystałeś zapytanie z predykatem `StateProvinceID = 79`, otrzymasz plan wykorzystujący *Clustered Index Scan* dla obu zapytań, jeżeli jednak najpierw wykonałeś zapytanie z predykatem `StateProvinceID = 59`, dla obu zapytań będzie wykorzystywany plan z operatorami *Index Seek/Key Lookup*. Użycie innych wartości dla *StateProvinceID* może spowodować stworzenie innych planów.

Ponieważ jednak wszystkie podobne zapytania będą korzystały z tego samego planu, może to nie być odpowiednie we wszystkich przypadkach i powinieneś dokładnie to przetestować w Twoim przypadku, aby stwierdzić, że na pewno takie rozwiązanie poprawia wydajność. W następnym podrozdziale omawiam problemy występujące w zapytaniach czułych na parametry, czyli coś, co wielu użytkowników określa mianem problemu podsłuchiwania parametrów.

Dostępne są również podpowiedzi `PARAMETERIZATION SIMPLE` i `PARAMETERIZATION FORCED`, które pozwolą nadpisać aktualne ustawienie na poziomie bazy danych i zmienić je na poziomie zapytania. Na przykład, jeżeli zdefiniujesz zastosowanie `ALTER DATABASE AdventureWorks2012 SET PARAMETERIZATION FORCED`, zgodnie z wcześniejszymi informacjami, możesz wykorzystać podpowiedź `OPTION (PARAMETERIZATION SIMPLE)`, aby dla tego konkretnego zapytania zmienić zachowanie parametryzacji.

Nie zapomnij przywrócić konfiguracji do ustawień domyślnych za pomocą poniższego polecenia:

```
ALTER DATABASE AdventureWorks2012 SET PARAMETERIZATION SIMPLE
```

Procedury przechowywane

Jeżeli jednak chcesz bezpośrednio wykorzystać parametryzację, masz kilka opcji, między innymi zastosowanie procedur przechowywanych, funkcji skalarnych zdefiniowanych przez użytkownika i wielopoleceniowych funkcji zwracających wartości tabelaryczne. Wszystkie te obiekty są zaprojektowane tak, aby wielokrotnie wykorzystywać plany, i pokażą wartość `Proc` w kolumnie *objtype* widoku `sys.dm_exec_cached_plans`. Przyjrzyjmy się, co się stanie, jeżeli zapytanie, które nie zostało wcześniej sparametryzowane, wykorzystamy w procedurze przechowywanej przyjmującej parametr. Stwórz poniższą procedurę przechowywaną:

```
CREATE PROCEDURE test (@stateid int)
AS
SELECT * FROM Person.Address
WHERE StateProvinceID = @stateid
```

A następnie uruchom następujący kod:

```
DBCC FREEPROCCACHE
GO
exec test @stateid = 79
```

```
GO
exec test @stateid = 59
GO
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

Otrzymamy taki oto wynik:

usecounts	cacheobjtype	objtype	text
2	Compiled Plan	Proc	CREATE PROCEDURE test (@stateid int) AS ...

Podobnie jak w przypadku wymuszonej parametryzacji, gdzie ważne było, które zapytanie zostanie zoptymalizowane jako pierwsze, w przypadku procedury przechowywanej również należy zrozumieć, że pierwsza optymalizacja wykorzysta parametr przekazany w momencie tworzenia planu. W tym przypadku jako pierwszy został użyty parametr 79, oba plany są jednakowe i wykorzystują operator *Clustered Index Scan*. Możesz uruchomić poniższy kod, gdzie jako pierwszy jest użyty parametr 59, i w tym przypadku plan będzie wykorzystywał kombinację operatorów *Index Seek/Key Lookup*.

```
DBCC FREEPROCCACHE
GO
exec test @stateid = 59
GO
exec test @stateid = 79
GO
SELECT * FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE text like '%Person%'
```

Dlaczego jednak w obu przypadkach tworzone są różne plany? Wspomnieliśmy wcześniej, że w przypadku predykatu korzystającego z kolumny *AddressID* optymalizator może bezpiecznie wykorzystać ten sam plan, ponieważ kolumna jest częścią indeksu unikalnego, a predykat równości dla tej kolumny zawsze zwróci nie więcej niż jeden wiersz. Natomiast *StateProvinceID* nie jest częścią indeksu unikalnego i ma nierównomierne rozmieszczenie danych, co możesz zaobserwować, uruchamiając poniższe zapytanie:

```
SELECT StateProvinceID, COUNT(*) AS cnt
FROM Person.Address
GROUP BY StateProvinceID
ORDER BY cnt
```

Skrócony wynik pokazujący skrajne wartości w tabeli został przedstawiony poniżej. Widać, że dla niektórych wartości *StateProvinceID* znaleziony został tylko 1 rekord, podczas gdy dla innych istnieje aż kilka tysięcy rekordów.

Oczywiście w tym przypadku optymalizator zdecyduje, jakie operacje wykorzystać w planie zapytania, na podstawie szacunku kardynalności każdej operacji.

StateProvinceID	cnt
32	1
41	1
44	1
59	1
118	1
...	
50	1588
14	1954
79	2636
9	4564

Podśluchiwanie parametrów

W poprzednim podrozdziale omówiłem parametryzację i powtórne wykorzystanie planów. W tym podrozdziale dokładniej przyjrzymy się przypadkom, w których powtórne wykorzystanie planów może powodować problemy wydajnościowe. Jak widziałeś w rozdziale 6., SQL Server może korzystać z histogramu obiektu statystyk do oszacowania kardynalności zapytania, a następnie użyć tych informacji podczas próby określenia najbardziej optymalnego planu wykonania. Optymalizator osiąga to, analizując wartości parametrów zapytania. Zachowanie to nazywa się podśluchiowaniem parametrów i jest bardzo przydatne, gdyż dostosowanie planu do aktualnych parametrów zapytania naturalnie poprawia wydajność aplikacji. W tym rozdziale wyjaśniłem, że magazyn planów może przechowywać plany, dzięki czemu mogą być powtórnie wykorzystywane podczas kolejnego wykonania zapytania. Pozwala to zaoszczędzić czas optymalizacji i zasoby procesora.

Chociaż magazyn planów i optymalizator dobrze współpracują przez większość czasu, mogą się zdarzyć problemy wydajnościowe. Biorąc pod uwagę fakt, że optymalizator może tworzyć różne plany dla składniowo identycznych zapytań, niezależnie od ich parametrów, przechowywanie i wykorzystywanie tylko jednego z tych planów może powodować problemy wydajnościowe dla alternatywnych instancji zapytania, które mogłyby skorzystać z lepszego planu. Jest to znany problem T-SQL, korzystającego z bezpośredniej parametryzacji, jak w przypadku procedur przechowywanych. W tym podrozdziale dokładniej omówię ten problem, dam też kilka wskazówek, jak sobie z nim poradzić.

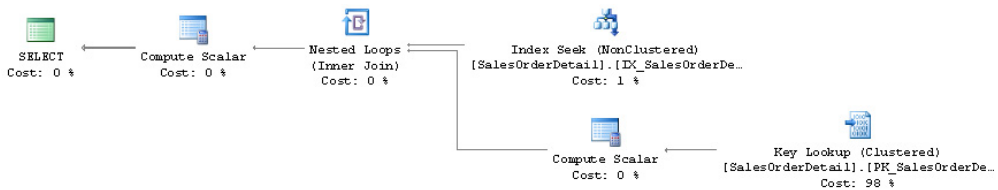
Aby zobaczyć przykład, stwórzmy prostą procedurę przechowywaną, która korzysta z tabeli `Sales.SalesOrderDetail` w bazie `AdventureWorks2012` (jeżeli istnieje poprzednia wersja procedury, należy ją usunąć):

```
CREATE PROCEDURE test (@pid int)
AS
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @pid
```

Uruchom poniższe polecenie, aby wykonać procedurę:

```
EXEC test @pid = 897
```

Optymalizator szacuje, że zwrócone zostanie tylko kilka rekordów, i tworzy plan wykonania, który korzysta z operatora *Index Seek*, aby szybko znaleźć rekordy w indeksie nieklastrowym, i operatora *Key Lookup*, aby przeszukać tabelę i pobrać brakujące kolumny (zobacz rysunek 8.2).



Rysunek 8.2. Plan wykorzystujący operatory *Index Seek* i *Key Lookup*

Ta kombinacja operatorów jest dobra, ponieważ mimo że jest to relatywnie kosztowna kombinacja, zapytanie jest wysoko selektywne. Co jednak w przypadku, kiedy wykorzystany zostanie inny parametr — taki, który będzie mniej selektywny? Wypróbuj na przykład poniższe zapytanie, które wykorzystuje polecenie `SET STATISTICS IO ON`, pozwalające wyświetlić aktywność dysku podczas generowania zapytania:

```
SET STATISTICS IO ON
GO
EXEC test @pid = 870
GO
```

Zakładka z komunikatami będzie zawierała poniższą informację:

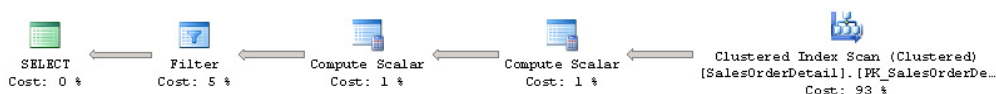
```
Table 'SalesOrderDetail'. Scan count 1, logical reads 18038, physical reads 57, read-ahead reads 447, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Jak widzisz, w przypadku tego wykonania `SQL Server` wykonał 18038 odczytów logicznych, a tabela ma tylko 1246. Oznacza to, że wykonywanych jest 14 razy więcej operacji I/O, niż miałyby to miejsce w przypadku skanowania całej tabeli. Jak widziałeś w rozdziale 4., wykonywanie operacji *Index Seek* i *Key Lookup* na tabeli bazowej, powodujące losowe operacje I/O, jest bardzo kosztowną operacją. Zauważ, że w swojej instancji `SQL Servera` możesz otrzymać odrobinę różniące się wyniki.

Wyczyść teraz magazyn planów, aby usunąć aktualnie przechowywany plan, a następnie jeszcze raz uruchom procedurę:

```
DBCC FREEPROCCACHE
GO
EXEC test @pid = 870
GO
```

Tym razem otrzymasz zupełnie inny plan wykonania. Informacje na temat I/O pokażą teraz tylko 1246 przeczytanych stron, a sam plan będzie zawierał operator *Clustered Index Scan* (zobacz rysunek 8.3). Ponieważ tym razem w magazynie nie było zoptymalizowanej wersji procedury przechowywanej, SQL Server zoptymalizował ją od początku, korzystając z nowego parametru, i stworzył nowy optymalny plan wykonania.



Rysunek 8.3. Plan wykorzystujący operator Clustered Index Scan

Oczywiście nie oznacza to, że nie powinieneś już ufać swoim procedurom przechowywanym ani że Twój kod jest niepoprawny. Jest to po prostu problem, który musisz przeanalizować, w szczególności jeżeli masz zapytania, dla których wydajność zmienia się drastycznie dla różnych parametrów. Jeżeli borykasz się z tym problemem, masz kilka dostępnych wyborów, które opiszę w dalszej części.

Kolejnym związanym z tym problemem jest to, że nie masz kontroli nad czasem życia planu w magazynie, dlatego za każdym razem, kiedy plan jest usuwany z magazynu, nowo stworzony plan wykonania będzie uzależniony od tego, który parametr akurat zostanie przekazany. Niektóre z poniższych rozwiązań pozwalają uzyskać pewien poziom stabilności planu poprzez poproszenie optymalizatora, aby tworzył plan w oparciu o typowy parametr lub w oparciu o średnią gęstość kolumny.

Optymalizacja pod typowy parametr

W pewnych przypadkach większość wykonania zapytania wykonywanych jest z wykorzystaniem tego samego planu wykonania i chciałbyś uniknąć ciągłego kosztu optymalizacji poprzez powtórne wykorzystanie planu. W takich przypadkach możesz skorzystać z wprowadzonej w SQL Serverze 2005 podpowiedzi o nazwie *OPTIMIZE FOR*, która jest użyteczna, jeżeli plan optymalny może zostać wygenerowany dla większości wartości używanych dla konkretnego parametru, i jednocześnie daje ona lepszą stabilność planu. W rezultacie tylko przypadki używające nietypowego parametru będą miały nieoptymalny plan.

Załóżmy, że prawie wszystkie wykonania naszej procedury przechowywanej skorzystałyby na użyciu planu stosującego operatory *Index Seek* i *Key Lookup*. Aby to wykorzystać, możesz napisać procedurę przechowywaną:

```
ALTER PROCEDURE test (@pid int)
AS
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @pid
OPTION (OPTIMIZE FOR (@pid = 897))
```

Kiedy uruchomisz procedurę przechowywaną po raz pierwszy, zostanie zoptymalizowana dla wartości 897, bez znaczenia, jaka wartość parametru została w rzeczywistości przekazana do procedury. Jeżeli chcesz to sprawdzić, uruchom poniższy kod:

```
EXEC test @pid = 870
```

Blisko końca planu XML (lub we właściwości *Parameter List* planu graficznego) możesz znaleźć następujący wpis:

```
<ParameterList>
  <ColumnReference Column="@pid" ParameterCompiledValue="(897)"
    ParameterRuntimeValue="(870)" />
</ParameterList>
```

Ten wpis jasno pokazuje, jaka wartość parametru została wykorzystana podczas optymalizacji, a jaka podczas wykonania. W tym przypadku procedura przechowywana jest optymalizowana tylko raz, a plan jest przechowywany w magazynie planów i może zostać ponownie wykorzystany. Zaletą tej odpowiedzi, oprócz uniknięcia kosztu optymalizacji, jest to, że masz całkowitą kontrolę nad tym, jaki plan jest tworzony podczas optymalizacji zapytania i zapisywany w magazynie. Odpowiedź *OPTIMIZE FOR* pozwala również skorzystać z większej liczby parametrów oddzielonych przecinakami.

Optymalizacja przy każdym wykonaniu

Jeżeli wykorzystanie różnych parametrów powoduje powstanie różnych planów zapytania i chcesz, aby wszystkie zapytania były wykonywane z możliwie największą wydajnością, rozwiązaniem może być optymalizacja przy każdym wykonaniu. Otrzymasz najlepszy możliwy plan za każdym razem, ale będziesz musiał zapłacić koszt optymalizacji, a zatem trzeba zdecydować, czy warto. Aby to zrobić, skorzystaj z odpowiedzi *RECOMPILE*:

```
ALTER PROCEDURE test (@pid int)
AS
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @pid
OPTION (RECOMPILE)
```

Wykorzystanie `OPTION (RECOMPILE)` pozwala również na podsłuchiwanie wartości zmiennych lokalnych, zgodnie z informacjami w następnym podrozdziale. Opcja ta spowoduje zwrócenie wartości `Option (Recompile) requested` w kolumnie `EventSubClass` dla zdarzeń śledzenia *SP:Recompile* i *SQL:StmtRecompile*.

Zmienne lokalne i odpowiedź `OPTIMIZE FOR UNKNOWN`

Innym często implementowanym w przeszłości rozwiązaniem jest wykorzystanie w zapytaniach zmiennych lokalnych zamiast parametrów. Jak wspomniałem w rozdziale 6., optymalizator zapytań nie jest w stanie zobaczyć wartości zmiennych lokalnych w czasie optymalizacji, ponieważ wartości te znane są dopiero w czasie wykonywania. Korzystając jednak ze zmiennych lokalnych, wyłączasz podsłuchiwanie parametrów, co oznacza, że optymalizator nie będzie w stanie uzyskać dostępu do histogramu statystyk, aby znaleźć dla zapytania optymalny plan. Zamiast tego będzie polegał wyłącznie na informacji o gęstości zawartej w obiekcie statystyk (ten temat również omówiłem w rozdziale 6.).

To rozwiązanie zignoruje po prostu wartości parametru i wykorzysta ten sam plan wykonania dla wszystkich wykonania, ale przynajmniej uzyskasz jednolity plan za każdym razem. Wariacją poprzednio pokazanej odpowiedzi `OPTIMIZE FOR` jest odpowiedź `OPTIMIZE FOR UNKNOWN`. Została ona wprowadzona w SQL Serverze 2008 i daje ten sam efekt co wykorzystanie zmiennych lokalnych. Zaletą odpowiedzi `OPTIMIZE FOR UNKNOWN` w porównaniu do `OPTIMIZE FOR` jest to, że nie wymaga podania wartości dla parametru. Nie musisz się też martwić tym, że wyspecyfikowana wartość z czasem przestanie być typowa.

Uruchomienie poniższych dwóch wersji naszej procedury przechowywanej da równoznaczne wyniki i stworzy ten sam plan wykonania. Pierwsza wersja używa zmiennych lokalnych, a druga odpowiedzi `OPTIMIZE FOR UNKNOWN`.

```
ALTER PROCEDURE test (@pid int)
AS
DECLARE @p int = @pid
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @p
```

```
ALTER PROCEDURE test (@pid int)
AS
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @pid
OPTION (OPTIMIZE FOR UNKNOWN)
```

W tym przypadku optymalizator stworzy plan z wykorzystaniem operatora *Clustered Index Scan*, niezależnie od tego, jaki parametr przekażesz do procedury. Zauważ, że odpowiedź `OPTIMIZE FOR UNKNOWN` będzie się odnosiła do wszystkich parametrów

wykorzystanych w zapytaniu, chyba że użyjesz poniższej składni, aby nakierować odpowiedź na konkretny parametr:

```
ALTER PROCEDURE test (@pid int)
AS
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @pid
OPTION (OPTIMIZE FOR (@pid UNKNOWN))
```

Pamiętaj również, że podsłuchiwanie parametrów jest pożądaną optymalizacją, więc jeśli chcesz ją wyłączyć, to tylko w razie napotkania problemów, o których pisałem w tym podrozdziale, i o ile ogólna wydajność zapytania zostanie podniesiona.

Warto zauważyć, że od SQL Servera 2005, w którym wprowadzona została kompilacja na poziomie polecenia pozwalająca na optymalizację pojedynczych poleceń, technicznie możliwe jest podsłuchiwanie wartości zmiennych lokalnych w ten sam sposób co parametrów. Ta funkcjonalność nie została jednak zaimplementowana, ponieważ istniało dużo kodu wykorzystującego zmienne lokalne właśnie do wyłączenia podsłuchiwania parametrów. Zmienne lokalne mogą być jednak podsłuchane z użyciem omawianej wcześniej odpowiedzi RECOMPILE. Na przykład uruchommy poniższy kod korzystający ze zmiennych lokalnych i odpowiedzi OPTION (RECOMPILE):

```
ALTER PROCEDURE test (@pid int)
AS
DECLARE @p int = @pid
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = @p
OPTION (RECOMPILE)
```

A następnie poniższy:

```
EXEC test @pid = 897
```

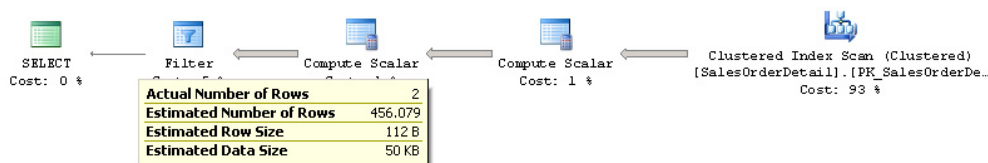
Optymalizator będzie w stanie zobaczyć wartość zmiennej lokalnej (w tym przypadku 897) i stworzyć plan optymalny dla tej konkretnej wartości (w tym przypadku plan z operacjami *Index Seek/Key Lookup* zamiast planu z *Clustered Index Scan* pokazanego wcześniej, kiedy wartość nie była podsłuchiwana).

Chociaż w rozdziale 6. wyjaśniłem, jak korzystać z histogramu i wektora gęstości obiektu statystyk do oszacowania kardynalności zapytania, omówmy ten temat jeszcze raz, ale z punktu widzenia wyłączania podsłuchiwania parametrów. Dowolna spośród procedur przechowywanych z początku tego podrozdziału — korzystająca ze zmiennych lokalnych lub odpowiedzi OPTIMIZE FOR UNKNOWN — zwróci plan przedstawiony na rysunku 8.4, z szacowaną kardynalnością 456,079.

Zobaczmy, jak SQL Server uzyskał wartość 456,079 i jakie są tego powody. W rozdziale 6. wyjaśniłem, że gęstość definiowana jest następująco:

1 / liczba unikalnych wartości

Tabela SalesOrderDetail ma 266 unikalnych wartości w kolumnie ProductID, dlatego gęstość to 1/266, czyli 0,003759399, co możesz zweryfikować, analizując obiekt statystyk



Rysunek 8.4. Szacowanie kardynalności z wyłączonym podsłuchiwanym parametrów

(na przykład za pomocą polecenia `DBCC SHOW_STATISTICS`). Jednym z założeń w modelu matematycznym statystyk jest jednolitość, a ponieważ w tym przypadku SQL Server nie może skorzystać z histogramu, założenie jednolitości mówi, że dla dowolnej wartości rozkład danych jest taki sam. Aby uzyskać szacowaną liczbę rekordów, SQL Server pomnoży gęstość przez aktualną całkowitą liczbę rekordów ($0,003759399 \cdot 121317$, czyli 456,079), co widać na planie. Ten sam wynik uzyskamy, dzieląc całkowitą liczbę rekordów przez liczbę unikalnych wartości ($121317/266$, czyli 456,079).

Zaletą używania podpowiedzi `OPTIMIZE FOR UNKNOWN` jest również to, że musisz zoptymalizować zapytanie raz, a powstały plan możesz wykorzystywać wielokrotnie. Nie ma również konieczności specyfikowania wartości jak w przypadku `OPTIMIZE FOR`.

Wyłączanie podsłuchiwania parametrów

Jak wspomniałem w poprzednim podrozdziale, kiedy w zapytaniu stosujesz zmienne lokalne do uniknięcia wykorzystania parametru procedury lub kiedy używasz podpowiedzi `OPTIMIZE FOR UNKNOWN`, włączasz podsłuchiwanie parametrów. Microsoft opublikował również flagę śledzenia 4136, pozwalającą wyłączyć podsłuchiwanie parametrów na poziomie instancji. Jak wyjaśniono w artykule 980653, znajdującym się w bazie wiedzy Microsoftu, flaga ta została wprowadzona w ramach aktualizacji dla starszych wersji SQL Servera, takich jak SQL Server 2005 SP3, SQL Server 2008 SP1 i SQL Server 2008 R2, i jest dostępna również w nowszych wersjach, włącznie z SQL Serverem 2014. Nadal istnieją trzy przypadki, w których ta flaga nie odniesie skutku:

- ▶ dla zapytań korzystających z podpowiedzi `OPTIMIZE FOR`;
- ▶ dla zapytań korzystających z podpowiedzi `OPTION (RECOMPILE)`;
- ▶ dla zapytań w procedurze przechowywanej używającej opcji `WITH RECOMPILE`.

Jak w przypadku wymuszonej parametryzacji na poziomie bazy danych, tę opcję powinno się rozważać tylko w ekstremalnych przypadkach i stosować ją z dużą ostrożnością, a taką aplikację należy dokładnie przetestować, aby się upewnić, czy rzeczywiście wydajność uległa poprawie. Możesz użyć tej flagi, jeżeli większość Twoich zapytań skorzysta na wyłączeniu podsłuchiwania parametrów, i korzystać z trzech podanych wyżej wyjątków dla zapytań, dla których podsłuchiwanie parametrów powinno być włączone. Microsoft zalecił, aby użytkownicy jego aplikacji Dynamics AX

rozważyli użycie tej flagi — zalecenie to zostało przedstawione w artykule dostępnym pod adresem <http://blogs.msdn.com/b/axperf/archive/2010/05/07/important-sql-server-change-parameter-sniffing-and-plan-caching.aspx>.

Podśluchiwanie parametrów i opcje SET wpływające na powtórne wykorzystanie planów

Jednym z bardziej interesujących problemów, z jakimi się spotkałem, było to, że procedura trwała zbyt długo lub nawet przekraczała dozwolony czas wykonania podczas uruchamiania jej z aplikacji, ale wykonywała się natychmiast, jeżeli uruchamiana była bezpośrednio w SQL Server Management Studio — nawet dla tych samych parametrów. Chociaż może być kilka przyczyn takiego problemu, włączając w to blokowanie, najbardziej powszechny powód związany jest z faktem, że stworzone zostały dwa różne plany dla różnych opcji SET i przynajmniej jeden z nich został zoptymalizowany przy użyciu kombinacji parametrów powodujących wygenerowaniu planu „złego” dla innej kombinacji parametrów. Chociaż możesz poczuć pokusę uruchomienia `sp_recompile`, aby wymusić nową optymalizację i pozwolić aplikacji na dalsze działanie, nie rozwiąże to problemu, który prędzej czy później powróci. Możesz również spotkać się z podobnym scenariuszem, w którym zaktualizowałeś statystyki, przebudowałeś indeks lub zmieniłeś coś innego, aby przekonać się, że nagle problem zniknął. Nie zniknął. Te zmiany prawdopodobnie wymusiły tylko ponowną optymalizację dla „dobrego” parametru, który testowałeś. Oczywiście najlepsze, co możesz zrobić w takiej sytuacji, to przechwycić „zły” plan do dalszej analizy, aby znaleźć trwałe rozwiązanie. W tym podrozdziale pokażę, jak to zrobić.

Pamiętaj, że zasadniczo optymalizacja to kosztowna operacja i aby uniknąć kosztów optymalizacji, magazyn danych spróbuje przechować plany w pamięci do ponownego wykorzystania. Jeżeli jednak nowe połączenie uruchamiające tę samą procedurę przechowywaną ma inne opcje SET, wygenerowany może zostać nowy plan zamiast pobrania już istniejącego w pamięci. Ten nowy plan może być powtórnie wykorzystywany, ale tylko wtedy, kiedy użyte zostaną te same ustawienia połączenia. Nowy plan jest potrzebny, ponieważ niektóre z opcji SET mogą wpływać na wybór planu wykonania przez fakt, że wpływają na rezultaty rozwiązywania wyrażeń stałych podczas procesu optymalizacji. Inny *ciąg połączeniowy* (*connection string*), `FORCEPLAN`, działa podobnie do podpowiedzi, żądając, aby optymalizator zachowywał kolejność złączeń zgodną ze składnią zapytania i aby korzystał wyłącznie z zagnieżdżonych pętli. Poniższe opcje SET wpłyną na powtórne wykorzystanie planów:

- ▶ `ANSI_NULL_DFLT_OFF`
- ▶ `ANSI_NULL_DFLT_ON`
- ▶ `ANSI_NULLS`
- ▶ `ANSI_PADDING`
- ▶ `ANSI_WARNINGS`

- ▶ ARITHABORT
- ▶ CONCAT_NULL_YIELDS_NULL
- ▶ DATEFIRST
- ▶ DATEFORMAT
- ▶ FORCEPLAN
- ▶ LANGUAGE
- ▶ NO_BROWSETABLE
- ▶ NUMERIC_ROUNDABORT
- ▶ QUOTED_IDENTIFIER



UWAGA

Polecenia SET i opcje bazy danych ANSI_NULLS OFF, ANSI_PADDING OFF i CONCAT_NULL_YIELDS_NULL OFF zostały zdeprecjonowane. W przyszłych wersjach SQL Servera polecenia te będą zawsze ustawione na ON.

Niestety narzędzia zarządzania i tworzenia takie jak SQL Server Management Studio, framework ADO.NET, a nawet narzędzie *sqlcmd*, mają w konfiguracji domyślnej różne opcje SET. Przekonasz się, że często problem polega na tym, iż jedna z opcji, ARITHABORT, jest domyślnie wyłączona w ADO.NET i włączona w Management Studio. Dlatego możliwe, że w naszym przykładzie Management Studio i aplikacja sieciowa korzystają z różnych planów wykonania, ale plan dla aplikacji sieciowej nie był dobry dla wywołań procedury z innymi parametrami.

Zobaczmy teraz, jak dowieść, że optymalizacja z różnymi parametrami jest powodem tego konkretnego problemu. Zobaczmy, jak wydobyć plany, aby sprawdzić parametry i opcje SET wykorzystane podczas optymalizacji. Ponieważ AdventureWorks2012 nie ma domyślnych opcji SET dla nowej bazy danych, stworzymy nową bazę, skopiujemy dane z AdventureWorks2012 i stworzymy nową procedurę. Aby to zrobić, uruchom poniższy kod:

```
CREATE DATABASE Test
GO
USE Test
GO
SELECT * INTO dbo.SalesOrderDetail
FROM AdventureWorks2012.Sales.SalesOrderDetail
GO
CREATE NONCLUSTERED INDEX IX_SalesOrderDetail_ProductID
ON dbo.SalesOrderDetail(ProductID)
GO
CREATE PROCEDURE test (@pid int)
AS
SELECT * FROM dbo.SalesOrderDetail
WHERE ProductID = @pid
```

Sprawdźmy dwie różne aplikacje, wykonując procedurę przechowywaną z SQL Server Management Studio i aplikacji .NET (kod i instrukcja budowy tej aplikacji znajdują się w ramce „Kod C# do testu opcji SET”). Na potrzeby tego testu chcemy założyć, że plan ze skanem tabeli jest złym planem, a plan wykorzystujący operatory *Index Seek/RID Lookup* jest planem optymalnym.

Zacznij od wyczyszczenia magazynu planów poprzez uruchomienie poniższego kodu:

```
DBCC FREEPROCCACHE
```

Uruchom aplikację .NET z linii komend i jako parametr przekaz wartość 870. Zauważ, że jedynym celem tej aplikacji jest uruchomienie stworzonej wcześniej procedury przechowywanej.

```
C:\TestApp\test
Enter ProductID: 870
```

Teraz możemy zacząć analizować magazyn planów, aby sprawdzić, jakie plany dostępne są w pamięci. Uruchom poniższy skrypt z bazy Test (będziemy go uruchamiać również w dalszej części tego ćwiczenia):

```
SELECT plan_handle, usecounts, pvt.set_options
FROM (
    SELECT plan_handle, usecounts, epa.attribute, epa.value
    FROM sys.dm_exec_cached_plans
    OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
    WHERE cacheobjtype = 'Compiled Plan') AS ecpa
PIVOT (MAX(ecpa.value) FOR ecpa.attribute IN ("set_options", "objectid")) AS pvt
WHERE pvt.objectid = OBJECT_ID('dbo.test')
```

Powinieneś uzyskać wynik podobny do poniższego:

plan_handle	usecounts	set_options
0x050007002255970F9042B8F8010000000100000000000000000000000000000000 ...	1	251

Wynik pokazuje, że mamy tylko jeden plan wykonania w magazynie, został on wykorzystany raz (o czym świadczy wartość *usecounts*), a wartość *set_options* (pobrana z funkcji DMF *sys.dm_exec_plan_attributes*) wynosi 251. Ponieważ było to pierwsze wykonanie procedury przechowywanej, została zoptymalizowana za pomocą parametru 870, który w tym przypadku stworzył plan korzystający ze skanu tabeli (tutaj uznawany za „zły” plan). Teraz uruchom aplikację jeszcze raz, stosując parametr, który zwróci tylko kilka rekordów i który skorzystałby na użyciu operatorów *Index Seek/RID Lookup*.

```
C:\TestApp\test
Enter ProductID: 898
```

Jeżeli jeszcze raz zajrzysz do magazynu planów, zobaczysz, że plan został użyty dwa razy, co zostało odnotowane w kolumnie *usecounts*; niestety dla tego parametru plan

nie jest odpowiedni. W prawdziwej bazie produkcyjnej drugie wykonanie może nie mieć oczekiwanej wydajności — wykonanie zapytania będzie trwało bardzo długo i może spowodować, że programista spróbuje przeanalizować problem, uruchamiając zapytanie w programie Management Studio za pomocą poniższego polecenia:

EXEC test @pid = 898

Programista może być zdziwiony faktem, że SQL Server zwraca dobry plan, a zapytanie zwraca wynik natychmiast. Powtórne sprawdzenie magazynu planów zwróci wynik podobny do poniższego:

<i>plan_handle</i>	<i>usecounts</i>	<i>set_options</i>
0x050007002255970FB049B8F801000000010000000000000000000000 ...	1	4347
0x050007002255970F9042B8F801000000010000000000000000000000 ...	2	251

Dla procedury uruchomionej za pomocą Management Studio dodany został nowy plan z inną wartością dla wartości *set options* (w tym przypadku 4347).

Co teraz? Czas przeanalizować plany i spojrzeć na opcje SET i parametry wykorzystane podczas optymalizacji. Wybierz `plan_handle` pierwszego planu (z wartością `set_options` równą 251 w Twoim przykładzie) i skorzystaj z niego, aby uruchomić poniższe zapytanie:

```
SELECT * FROM sys.dm_exec_query_plan(0x050007002255970F9042B8F801000000010000000000000000000000 ...)
```

Opcje SET znajdziesz na początku planu XML (a także w oknie właściwości planu graficznego):

```
<StatementSetOptions QUOTED_IDENTIFIER="true" ARITHABORT="false"
CONCAT_NULL_YIELDS_NULL="true" ANSI_NULLS="true" ANSI_PADDING="true"
ANSI_WARNINGS="true" NUMERIC_ROUNDABORT="false" />
```

A wykorzystane parametry znajdziesz na końcu (również dostępne w oknie właściwości planu graficznego):

```
<ParameterList>
  <ColumnReference Column="@pid" ParameterCompiledValue="(870)" />
</ParameterList>
```

Zrób to samo dla drugiego planu, a otrzymasz poniższe informacje dla opcji SET:

```
<StatementSetOptions QUOTED_IDENTIFIER="true" ARITHABORT="true"
CONCAT_NULL_YIELDS_NULL="true" ANSI_NULLS="true" ANSI_PADDING="true"
ANSI_WARNINGS="true" NUMERIC_ROUNDABORT="false" />
```

I następujące informacje o parametrach:

```
<ParameterList>
  <ColumnReference Column="@pid" ParameterCompiledValue="(898)" />
</ParameterList>
```

Informacje te pokazują, że opcja ARITHABORT ma różne wartości dla obu planów i że parametr wykorzystany do zoptymalizowania zapytania dla aplikacji sieciowej miał

wartość 870. Możesz również zweryfikować operatory wykorzystane w planie — pierwszy używał skanowania tabeli, a drugi kombinacji operatorów *Index Seek/RID Lookup*. Teraz, kiedy przechwyciłeś plany, możesz wymusić nową optymalizację, aby aplikacja mogła natychmiast zastosować lepszy plan (pamiętając, że nie jest to rozwiązanie trwałe). Spróbuj poniższego:

```
sp_recompile test
```

Możesz również wykorzystać poniższy skrypt, aby wyświetlić skonfigurowane opcje SET dla konkretnych wartości *set_options*:

```
DECLARE @set_options int = 4347
IF ((1 & @set_options) = 1) PRINT 'ANSI_PADDING'
IF ((4 & @set_options) = 4) PRINT 'FORCEPLAN'
IF ((8 & @set_options) = 8) PRINT 'CONCAT_NULL_YIELDS_NULL'
IF ((16 & @set_options) = 16) PRINT 'ANSI_WARNINGS'
IF ((32 & @set_options) = 32) PRINT 'ANSI_NULLS'
IF ((64 & @set_options) = 64) PRINT 'QUOTED_IDENTIFIER'
IF ((128 & @set_options) = 128) PRINT 'ANSI_NULL_DFLT_ON'
IF ((256 & @set_options) = 256) PRINT 'ANSI_NULL_DFLT_OFF'
IF ((512 & @set_options) = 512) PRINT 'NoBrowseTable'
IF ((4096 & @set_options) = 4096) PRINT 'ARITH_ABORT'
IF ((8192 & @set_options) = 8192) PRINT 'NUMERIC_ROUNDABORT'
IF ((16384 & @set_options) = 16384) PRINT 'DATEFIRST'
IF ((32768 & @set_options) = 32768) PRINT 'DATEFORMAT'
IF ((65536 & @set_options) = 65536) PRINT 'LanguageID'
```

Dla wartości *set_options* równej 4347 skrypt zwróci poniższy wynik:

```
ANSI_PADDING
CONCAT_NULL_YIELDS_NULL
ANSI_WARNINGS
ANSI_NULLS
QUOTED_IDENTIFIER
ANSI_NULL_DFLT_ON
ARITH_ABORT
```

Teraz, kiedy już zidentyfikowałeś, że jest to problem związany z podsłuchiwaniami parametrów, możesz zastosować którąś z technik pokazanych wcześniej.

KOD C# DO TESTU OPCJI SET

Do wykonania testu użyj poniższego kodu C#:

```
using System;
using System.Data;
using System.Data.SqlClient;

class Test
{
    static void Main()
    {
        SqlConnection cnn = null;
        SqlDataReader reader = null;
```

```

try
{
    Console.Write("Enter ProductID: ");
    string pid = Console.ReadLine();

    cnn = new SqlConnection("Data Source=(local);Initial Catalog=Test;Integrated
↪Security=SSPI");
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = cnn;
    cmd.CommandText = "dbo.test";
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add("@pid", SqlDbType.Int).Value = pid;
    cnn.Open();
    reader = cmd.ExecuteReader();
    while (reader.Read())
    {
        Console.WriteLine(reader[0]);
    }
    return;
}
catch (Exception e)
{
    throw e;
}
finally
{
    if (cnn != null)
    {
        if (cnn.State != ConnectionState.Closed)
            cnn.Close();
    }
}
}
}

```

Podobnie jak w przypadku kodu pokazanego w rozdziale 2., aby skompilować, musisz uruchomić poniższe polecenie w linii komend Windowsa, przy założeniu, że kod został zapisany w pliku *test.cs*:

```
csc test.cs
```

Do skompilowania tego kodu nie jest potrzebne Visual Studio — wystarczy biblioteka Microsoft .NET Framework, która jest niezbędna do zainstalowania SQL Servera. Dlatego będzie już zainstalowana w Twoim systemie. Jeżeli jednak plik wykonywalny *csc* nie został zawarty w zmiennej systemowej *PATH*, będziesz musiał znaleźć jego lokalizację (zazwyczaj znajduje się w katalogu *C:\Windows\Microsoft.NET*). Ciąg znaków połączenia w kodzie zakłada, że łączysz się z domyślną instancją SQL Servera z wykorzystaniem uwierzytelniania Windowsa — jeżeli jednak korzystasz z innej konfiguracji, będziesz musiał go zmienić.

Podsumowanie

W tym rozdziale omówiliśmy magazynowanie planów i skupiliśmy się na tym, jak w efektywny sposób wielokrotnie wykorzystywać plany. Optymalizacja zapytań to relatywnie kosztowna operacja, jeżeli więc zapytanie może być zoptymalizowane raz, a plan może być wykorzystany wielokrotnie, może to znacznie podnieść wydajność Twoich aplikacji. Dokładnie omówiliśmy proces kompilacji i rekompilacji zestawów i pokazałem, jak zidentyfikować problemy związane z nadmiarową kompilacją i rekompilacją zapytań.

Wielokrotne wykorzystywanie planów uzależnione jest od parametryzacji, dlatego również ten temat dokładnie omówiłem. Przyjrzelśmy się przypadkom, w których SQL Server decyduje o automatycznej parametryzacji zapytań, a także tym, w których należy takie zachowanie wymusić — albo poprzez użycie opcji konfiguracji dla wymuszonej parametryzacji, albo poprzez wykorzystanie obiektów takich jak procedury przechowywane, funkcje skalarne zdefiniowane przez użytkownika lub wielopoleceniowe funkcje zwracające wartości tabelaryczne.

Jednak mimo że analiza parametrów zapytania pomaga optymalizatorowi w tworzeniu lepszych planów wykonania, czasami powtórne wykorzystanie takich planów może powodować problemy, ponieważ plany mogą nie być optymalne dla niektórych parametrów. Pokazałem, że podsłuchiwanie parametrów jest optymalizacją, ale podałem też sposoby na radzenie sobie z sytuacjami, w których powtórne wykorzystanie planu może nie być odpowiednie.

Rozdział 9

Hurtownie danych

W tym rozdziale:

- ▶ Hurtownie danych
- ▶ Optymalizacja złączeń gwiazdzystych
- ▶ Indeksy magazynu kolumn
- ▶ Podsumowanie



Kiedy w książce lub artykule o SQL Serverze czytamy o wysoko wydajnych aplikacjach, zazwyczaj zakładamy, że mowa jest o aplikacjach transakcyjnych i bazach OLTP. W rzeczywistości, chociaż do tej pory nie wspomniałem o hurtowniach danych, większość takich tekstów odnosi się zarówno do baz OLTP, jak i do hurtowni danych. Optymalizator zapytań SQL Servera może automatycznie zidentyfikować i zoptymalizować zapytania hurtowni danych bez żadnej dodatkowej konfiguracji, dlatego koncepcje z tej książki dotyczące optymalizacji, operatorów, indeksów, statystyk i większości innych tematów mają zastosowanie również do hurtowni danych.

W tym rozdziale omówię tematy odnoszące się tylko do hurtowni danych. Wyjaśnienie tego, czym jest hurtownia danych, pozwala także zdefiniować systemy OLTP i wskazać różnice pomiędzy nimi. Opiszę sposób, w jaki optymalizator zapytań rozpoznaje zapytania hurtowni danych, i opowiem o optymalizacjach hurtowni danych wprowadzonych w kilku najnowszych wersjach SQL Servera.

Filtrowanie bitmapowe, czyli optymalizacja wykorzystywana w złączeniach gwiazdzystych od wersji 2008, może być stosowana do filtrowania rekordów z tabeli faktów na bardzo wczesnym etapie przetwarzania rekordów, a tym samym może znacznie poprawić wydajność zapytań hurtowni danych. Filtrowanie bitmapowe bazuje na filtrach Bloom, wymyślonych przez Burtona Blooma w roku 1970.

Indeksy magazynu kolumn, funkcja dostępna od wersji 2012, wprowadzają dwa nowe paradygmaty: przechowywanie oparte na kolumnach i nowe algorytmy przetwarzania partiami. W momencie jej wprowadzania funkcjonalność ta miała pewne ograniczenia — przede wszystkim to, że indeksów nie można było aktualizować i tworzone mogły być tylko indeksy nieklastrowe — w aktualnej wersji, SQL Serverze 2014, ograniczenia te zostały jednak usunięte. Optymalizacja złączenia gwiazdzystego i indeksy magazynu kolumn przechowywane w pamięci, xVelocity, to funkcjonalności dostępne tylko w wersji Enterprise Edition.

Wreszcie, chociaż wykracza to poza zakres tej książki, warto wspomnieć o tym, jak nowe trendy, takie jak wzrastające ilości danych oraz nowe źródła i rodzaje danych, zmieniają oblicze tradycyjnych hurtowni danych, a także jak koncepcje takie jak Big Data i rozwiązania w rodzaju Apache Hadoop zyskują popularność.

Hurtownie danych

Informacje w organizacji przechowywane są z reguły w jednej z dwóch form: w systemie operacyjnym lub w systemie analitycznym, a obie te formy mają bardzo różne przeznaczenie. Chociaż celem systemu operacyjnego, zwanego również systemem przetwarzania transakcji (OLTP — *online transaction processing*), jest wsparcie dla procesu biznesowego, celem systemu analitycznego lub hurtowni danych jest pomoc w mierzeniu procesu biznesowego i w podejmowaniu decyzji biznesowych. Systemy OLTP bazują na małych, wysoko wydajnych transakcjach składających się z poleceń INSERT, DELETE,

UPDATE i SELECT, natomiast hurtownia danych bazuje na dużych i skomplikowanych zapytaniach na zagregowanych danych. Poziom normalizacji to kolejna duża różnica pomiędzy tymi systemami: podczas gdy systemy OLTP z reguły zachowują trzecią formę normalną, dane w hurtowni będą z reguły korzystać ze zdenormalizowanego modelu opartego na wymiarach, nazywanego schematem gwiazdowym. Trzecia forma normalna używana w systemach OLTP pomaga w zachowaniu integralności danych i rozwiązywaniu problemów z nadmiarowością danych, ponieważ operacje aktualizacji muszą być wykonywane tylko w jednym miejscu. Model hurtowni danych oparty na wymiarach jest bardziej odpowiedni dla skomplikowanych zapytań *ad hoc*.

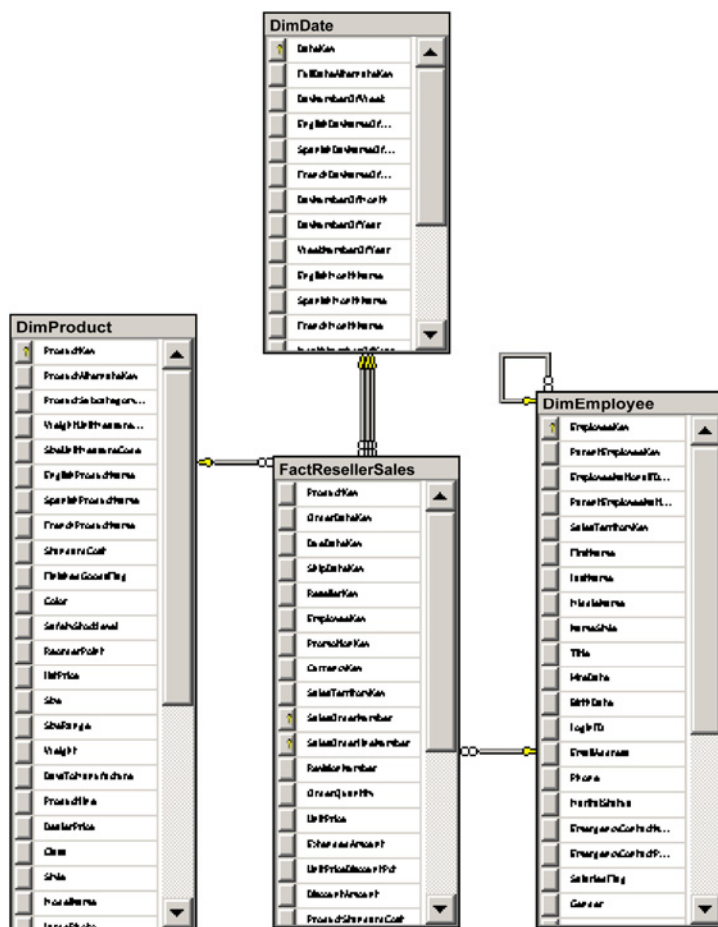
W książce *Star Schema: The Complete Reference* (McGraw-Hill Professional, 2010) Christopher Adamson przedstawia doskonałe podsumowanie różnic między systemem operacyjnym i systemem analitycznym — podsumowanie to przedstawia tabela 9.1.

Tabela 9.1. Porównanie systemu operacyjnego i systemu analitycznego

	System operacyjny	System analityczny
Przeznaczenie	Wsparcie procesu biznesowego	Mierzenie procesu biznesowego
Główny sposób interakcji	INSERT, UPDATE, DELETE, zapytania	Zapytania
Zakres interakcji	Indywidualne transakcje	Transakcje zagregowane
Wzorce zapytań	Przewidywalne i stałe	Nieprzewidywalne i zmienne
Czasowy obszar zainteresowania	Dane aktualne	Dane aktualne i historyczne
Zaprojektowany z myślą o optymalizacji	Współbieżności aktualizacji	Wysoko wydajnych zapytań
Zasady projektowe	Relacje encji (ER), trzecia forma normalna (3NF)	Projekt oparty na wymiarach (gwiazda lub kostka)
Inna nazwa	System transakcyjny, system OLTP	Hurtownia danych, magazyn danych

Projekt oparty na wymiarach, jeżeli zostanie zaimplementowany w bazie relacyjnej, na przykład w SQL Serverze, nazywa się schematem gwiazdowym. Hurtownie danych stosujące schemat gwiazdowy używają tabel faktów i tabel wymiarów, gdzie tabele faktów zawierają fakty biznesowe lub miary numeryczne, które mogą brać udział w kalkulacjach, a tabele wymiarów zawierają atrybuty lub opisy faktów. Nie wszystkie wartości numeryczne są faktami, jak w przypadku numerów telefonów, wieku i informacji o rozmiarze. Czasami wykonywana jest dodatkowa normalizacja tabel wymiarów, która nazywana jest schematem płatka śniegu (*snowflake*).

SQL Server zawiera przykładowe dane hurtowni danych, AdventureWorksDW2012, które zaprojektowane są w postaci płatka śniegu i będą wykorzystywane w przykładach z tego rozdziału. Rysunek 9.1 przedstawia diagram z niektórymi tabelami faktów i wymiarów bazy AdventureWorksDW2012.



Rysunek 9.1. Tabele faktów i wymiarów bazy AdventureWorksDW2012

Tabele faktów są zazwyczaj duże i mogą przechowywać miliony lub miliardy wierszy, natomiast tabele wymiarów są znacznie mniejsze. Rozmiary hurtowni danych wahają się od setek gigabajtów do terabajtów. Fakty przechowywane są jako kolumny w tabeli faktów, natomiast powiązane z nimi wymiary są przechowywane jako kolumny w tabelach wymiarów. Tabele faktów mają zazwyczaj klucze obce łączące je z kluczami głównymi w tabelach wymiarów. Kiedy zostaną użyte w zapytaniach, fakty są przeważnie agregowane lub sumowane, natomiast wymiary wykorzystywane są jako filtry lub predykaty zapytań. Tabela wymiarów ma również klucz sztuczny, będący zazwyczaj liczbą całkowitą, który jest też kluczem głównym tabeli. Na przykład w bazie AdventureWorksDW2012 tabela DimCustomer ma kolumnę *CustomerKey*, która jest kluczem głównym i zdefiniowana jest jako pole autoinkrementowane.



UWAGA

Więcej szczegółów na temat magazynów danych można znaleźć w książkach *The Data Warehouse Toolkit* Ralpha Kimballa i Margy Ross (Wiley, 2013) oraz *Star Schema: The Complete Reference* Christophera Adamsona.

Świat danych jednak bardzo gwałtownie ewoluuje i nowe trendy, takie jak wzrastające ilości danych oraz nowe źródła i typy danych, a także nowe wymagania wsparcia dla przetwarzania w czasie rzeczywistym, zmieniają oblicze tradycyjnych hurtowni danych. W niektórych przypadkach, w odpowiedzi na te trendy, proponowane są nowe podejścia.

Chociaż SQL Server Enterprise Edition umożliwia zarządzanie dużymi hurtowniami danych i zawiera omawiane w tym rozdziale optymalizacje połączeń gwiazdzystych i magazyny kolumn zoptymalizowane do działania w pamięci xVelocity, SQL Server jest również dostępny jako narzędzie pozwalające firmom obsługiwać bazy zawierające do 6 PB (petabajtów) danych. Zbudowany w oparciu o architekturę Multiple Parallel Processing (MPP — wielokrotne przetwarzanie współbieżne), SQL Server Parallel Data Warehouse udostępnia wysoko skalowalną architekturę pomagającą w pracy z tak dużymi hurtowniami.

Wraz ze wzrostem ilości danych, szybkości i różnorodności, z szerszym zakresem typów i źródeł danych, termin *Big Data* został przyjęty na potrzeby określania zbiorów danych tak dużych i złożonych, że nie mogą być zarządzane przez tradycyjne technologie zarządzania hurtowniami. Ponieważ tradycyjny magazyn danych nie przewidywał Big Data, aktualnie implementowane są technologie równoległe, a jedną z nich jest Apache Hadoop. Na przykład hurtownia danych Hadoop może funkcjonować równoległe z relacyjną hurtownią danych. W odpowiedzi na potrzebę integracji tradycyjnej relacyjnej hurtowni danych z Hadoopem Microsoft wypuścił PolyBase, dając tym samym możliwość odpytywania danych relacyjnych i nierelacyjnych w Hadoopie. Mechanizm PolyBase jest zintegrowany z SQL Server Parallel Data Warehouse.

W kwietniu 2014 roku Microsoft zapowiedział powstanie platformy Analytics Platform System (APS), będącej połączeniem komponentów sprzętowych i programowych, które zawiera SQL Server Parallel Data Warehouse i dystrybucję Hadoopa firmy Microsoft, HDInsight, która bazuje na Hortonworks Data Platform. Więcej informacji na temat tego rozwiązania znajdziesz pod adresem <http://blogs.technet.com/b/dataplatforminsider/archive/2014/05/20/architecture-of-the-microsoft-analytics-platform-system.aspx>.

Optymalizacja złączenia gwiazdzystego

Zapytania łączące tabelę faktów z tabelą wymiarów nazywane są zapytaniami ze złączeniami gwiazdzystymi, a SQL Server zawiera specjalne optymalizacje dla tego typu złączeń. Typowe zapytanie ze złączeniami gwiazdzystymi łączy tabelę faktów z jedną

lub więcej tabelami wymiarów, grupuje dane w oparciu o kolumny tabel wymiarów i wreszcie agreguje na jednej lub kilku kolumnach z tabeli faktów. Oprócz filtrów wykorzystywanych podczas złączania tabel można na tabeli faktów i wymiarów zastosować inne filtry. Oto przykład typowego zapytania tego rodzaju, które pokazuje powyższe charakterystyki dla bazy AdventureWorksDW2012:

```
SELECT TOP 10 p.ModelName, p.EnglishDescription,
    SUM(f.SalesAmount) AS SalesAmount
FROM FactResellerSales f JOIN DimProduct p
    ON f.ProductKey = p.ProductKey
JOIN DimEmployee e
    ON f.EmployeeKey = e.EmployeeKey
WHERE f.OrderDateKey >= 20030601
AND e.SalesTerritoryKey = 1
GROUP BY p.ModelName, p.EnglishDescription
ORDER BY SUM(f.SalesAmount) DESC
```

Ponieważ czasami implementacje magazynu danych nie specyfikują całkowicie relacji między tabelami faktów i wymiarów ani nie definiują bezpośrednio kluczy obcych po to, aby uniknąć narzutu związanego ze sprawdzeniem ograniczenia klucza głównego podczas aktualizacji danych, SQL Server wykorzystuje heurystykę do automatycznego wykrywania zapytań ze złączeniem gwiazdowym i może z dużym prawdopodobieństwem zidentyfikować tabele faktów i wymiarów. Jedną z takich heurystyk polega na uznaniu największej tabeli w złączeniu za tabelę faktów, która dodatkowo ma zdefiniowany rozmiar minimalny. Dodatkowo, aby zakwalifikować się do złączenia gwiazdowego, złączenia muszą być złączeniami wewnętrznymi, a wszystkie predykaty muszą być predykatami jednokolumnowymi z operatorem równości. Warto zauważyć, że nawet w przypadku, kiedy heurystyka nie zadziała poprawnie i jedna tabela wymiarów zostanie wybrana jako tabela faktów, i tak zwrócony zostanie poprawny plan zwracający poprawne dane, chociaż może on nie być najbardziej efektywny.

Na podstawie selektywności tabeli faktów SQL Server może zdefiniować trzy różne podejścia do optymalizacji zapytań ze złączeniem gwiazdowym. Dla wysoko selektywnych zapytań, które mogą zwracać do 10% wierszy z tabeli, SQL Server może stworzyć plan z operatorami *Nested Loops Join*, *Index Seek* i *Bookmark Lookup*. Dla zapytań ze średnią selektywnością, przetwarzających od 10 do 75% rekordów tabeli, optymalizator może wybrać operatory *Hash Join* z filtrami bitmapowymi w połączeniu ze skanowaniem tabeli faktów lub zakresowym skanowaniem tabeli faktów. Natomiast dla najmniej selektywnych zapytań, zwracających ponad 75% rekordów z tabeli, SQL Server może wybrać operatory *Hash Join* i skanowanie tabeli faktów. Wybór operatorów i planów nie jest zaskakujący dla najmniej i najbardziej selektywnych zapytań, ponieważ jest to zachowanie standardowe, zgodnie z rozdziałem 4. Nowością jednak jest wybór operatorów *Hash Join* i filtrów bitmapowych dla średnio selektywnych zapytań. Dlatego teraz przyjrzymy się tym optymalizacjom. Najpierw jednak krótkie wprowadzenie do filtrów bitmapowych.

Chociaż filtry bitmapowe są wykorzystywane podczas optymalizacji zapytań ze złączeniem gwiazdzystym omawianej w tym rozdziale, są stosowane od wersji 7.0. Mogą się również pokazywać na innych planach, a przykład takiego planu pokazałem w podrozdziale „Działania równoległe” w rozdziale 4. Filtry bitmapowe bazują na filtrach Blooma, oszczędzających miejsce probabilistycznych strukturach stworzonych przez Burtona Blooma w 1970 roku. Filtry bitmapowe są używane w planach równoległych i znacznie zwiększają wydajność tych planów poprzez wykonanie redukcji danych na wczesnym etapie, zanim rekordy zostaną przesłane do operatora współbieżności. Chociaż najbardziej powszechną kombinacją są filtry bitmapowe z operatorami *Hash Join*, mogą one być również stosowane z operatorami *Merge Join*.

Zgodnie z informacjami z rozdziału 4., operacja złączenia haszowego ma dwie fazy: budowania i sondowania. W fazie budowania wszystkie klucze złączenia tabeli budowania lub tabeli zewnętrznej są haszowane do tabeli haszy. Jako produkt uboczny tej operacji powstaje bitmapa, a bity w tablicy są ustawiane, aby odnosiły się do wartości haszy zawierających przynajmniej jeden wiersz. Bitmapa jest później analizowana podczas odczytu tabeli sondowanej lub wewnętrznej. Jeżeli bit nie jest ustawiony, rekord nie będzie miał rekordu odpowiadającego w tabeli zewnętrznej i nie będzie brany pod uwagę.

Filtry bitmapowe są wykorzystywane w zapytaniach ze złączeniem gwiazdzystym do odfiltrowania rekordów z tabeli faktów na bardzo wczesnym etapie przetwarzania zapytania. Optymalizacja ta została wprowadzona w SQL Serverze 2008, a istniejące zapytania mogą z niej korzystać bez żadnych zmian. Optymalizacja jest nazywana *zoptymalizowanym filtrowaniem bitmapowym*, aby odróżnić je od standardowego filtrowania bitmapowego znanego z poprzednich wersji SQL Servera. Chociaż standardowe filtrowanie bitmapowe może być stosowane w zapytaniach z operatorami *Merge Join* i *Hash Join*, zoptymalizowane filtrowanie bitmapowe może być wykorzystywane tylko ze złączeniami haszowymi. Prefix *Opt_* w nazwie operatora świadczy o tym, że zastosowane zostało zoptymalizowane filtrowanie bitmapowe.

Aby przefiltrować rekordy z tabeli faktów, SQL Server buduje tabele haszowe po stronie *wsadu budowania* złączenia haszowego (czyli w naszym przypadku po stronie tabeli wymiarów), tworząc mapę bitową, będącą tablicą bitów, która jest później używana do odfiltrowania z tabeli faktów rekordów niekwalifikujących się do złączenia. Mogą być błędne pozytywne wyniki, co oznacza, że przepuszczone zostaną wiersze niebiorące udziału w złączeniu, ale nie będzie błędnych wyników negatywnych. SQL Server przekazuje następnie mapę bitową do odpowiednich operatorów, aby pomogła w usunięciu rekordów niekwalifikujących się do złączenia na wczesnym etapie planu wykonania.

Dla jednego operatora może być zastosowanych wiele filtrów, a zoptymalizowane filtry bitmapowe mogą być stosowane dla operatorów wymiany, takich jak *Distribute Streams* i *Repartition Streams*, oraz operatorów filtrów. Dodatkową optymalizacją podczas korzystania z filtrów bitmapowych jest optymalizacja na poziomie wiersza, która pozwala na eliminację rekordów na jeszcze wcześniejszym etapie przetwarzania zapytania. Jeżeli

filtr bitmapowy bazuje na kolumnie typu `int` lub `bigint` nieprzyjmującej wartości `null`, filtr bitmapowy może zostać zaaplikowany bezpośrednio dla operacji na tabeli, a przykład takiego filtrowania znajduje się poniżej.

Uruchommy poniższe zapytanie:

```
SELECT TOP 10 p.ModelName, p.EnglishDescription,
    SUM(f.SalesAmount) AS SalesAmount
FROM FactResellerSales f JOIN DimProduct p
    ON f.ProductKey = p.ProductKey
JOIN DimEmployee e
    ON f.EmployeeKey = e.EmployeeKey
WHERE f.OrderDateKey >= 20030601
AND e.SalesTerritoryKey = 1
GROUP BY p.ModelName, p.EnglishDescription
ORDER BY SUM(f.SalesAmount) DESC
```

Ze względu na aktualny rozmiar bazy AdventureWorksDW2012 zapytanie nie jest wystarczająco kosztowne, aby wygenerować plan równoległy, ponieważ jego koszt to tylko 3,24627. Możemy jednak skorzystać z małej sztuczki, która zasymuluje więcej rekordów w bazie (zachęcam, abyś testował to lub podobne zapytania w środowisku testowym lub programistycznym).

Aby zasymulować większą tabelę, mającą 100000 rekordów i 10000 stron, uruchom poniższe polecenie:

```
UPDATE STATISTICS dbo.FactResellerSales WITH ROWCOUNT = 100000, PAGECOUNT = 10000
```



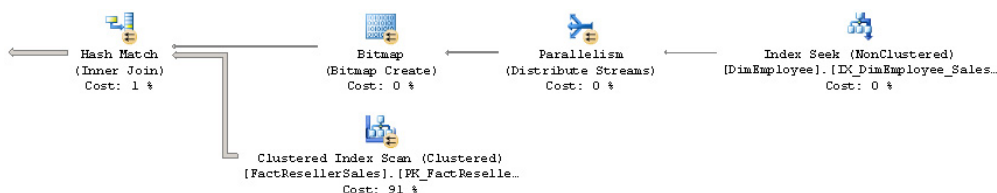
UWAGA

Nieudokumentowane opcje `ROWCOUNT` i `PAGECOUNT` polecenia `UPDATE STATISTICS` przedstawiam w rozdziale 6. Ponieważ są nieudokumentowane, nie powinny być uruchamiane w środowisku produkcyjnym. W tym ćwiczeniu zostaną wykorzystane wyłącznie do zasymulowania większej tabeli.

Wyczyść magazyn planów za pomocą poniższego polecenia:

```
DBCC FREEPROCCACHE
```

A teraz ponownie uruchom poprzednie zapytanie ze złączeniem gwiazdzystym. Tym razem otrzymamy bardziej kosztowne zapytanie i inny plan (zobacz rysunek 9.2).



Rysunek 9.2. Plan ze zoptymalizowanym filtrem bitmapowym

Jak tłumaczyłem wcześniej, filtr bitmapowy został stworzony na wsadzie budowania złączenia haszowego, który w tym przypadku czyta dane z tabeli faktów DimEmployee. Patrząc na wartość Defined Values (wartości zdefiniowane) w oknie właściwości operatora bitmapowego, zobaczysz, że nazwa operatora w tym wypadku to Opt_Bitmap1006. Spójrz teraz na właściwości operatora *Clustered Index Scan*, które zostały pokazane na rysunku 9.3. Widać, że poprzednio stworzony filtr bitmapowy, Opt_↪Bitmap1006, jest używany w sekcji *Predicate* do filtrowania rekordów z tabeli faktów FactResellerSales. Spójrz też na parametr IN ROW, który pokazuje, że wykorzystana została również optymalizacja na poziomie wiersza, filtrując dane najwcześniej jak to możliwe (w tym przypadku z operacji *Clustered Index Scan*).

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows	6208
Actual Number of Batches	0
Estimated I/O Cost	7.40979
Estimated Operator Cost	7.46487 (91%)
Estimated Subtree Cost	7.46487
Estimated CPU Cost	0.0550785
Estimated Number of Executions	1
Number of Executions	4
Estimated Number of Rows	10739.1
Estimated Row Size	27 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	17
Predicate	
[AdventureWorksDW2012].[dbo].[FactResellerSales]. [OrderDateKey] as [f].[OrderDateKey]>=(20030601) AND PROBE([Opt_Bitmap1006],[AdventureWorksDW2012].[dbo]. [FactResellerSales].[EmployeeKey] as [f].[EmployeeKey],N[IN ROW])	
Object	
[AdventureWorksDW2012].[dbo].[FactResellerSales]. [PK_FactResellerSales_SalesOrderNumber_SalesOrderLineNum ber] [f]	
Output List	
[AdventureWorksDW2012].[dbo]. [FactResellerSales].ProductKey, [AdventureWorksDW2012]. [dbo].[FactResellerSales].EmployeeKey, [AdventureWorksDW2012].[dbo]. [FactResellerSales].SalesAmount	

Rysunek 9.3. Właściwości operatora Clustered Index Scan

Uruchom teraz poniższe polecenie, aby poprawić liczbę rekordów i stron, którą zmieniliśmy wcześniej dla tabeli FactResellerSales:

```
DBCC UPDATEUSAGE (AdventureWorksDW2012, 'dbo.FactResellerSales') WITH COUNT_ROWS
```

Indeksy magazynu kolumn

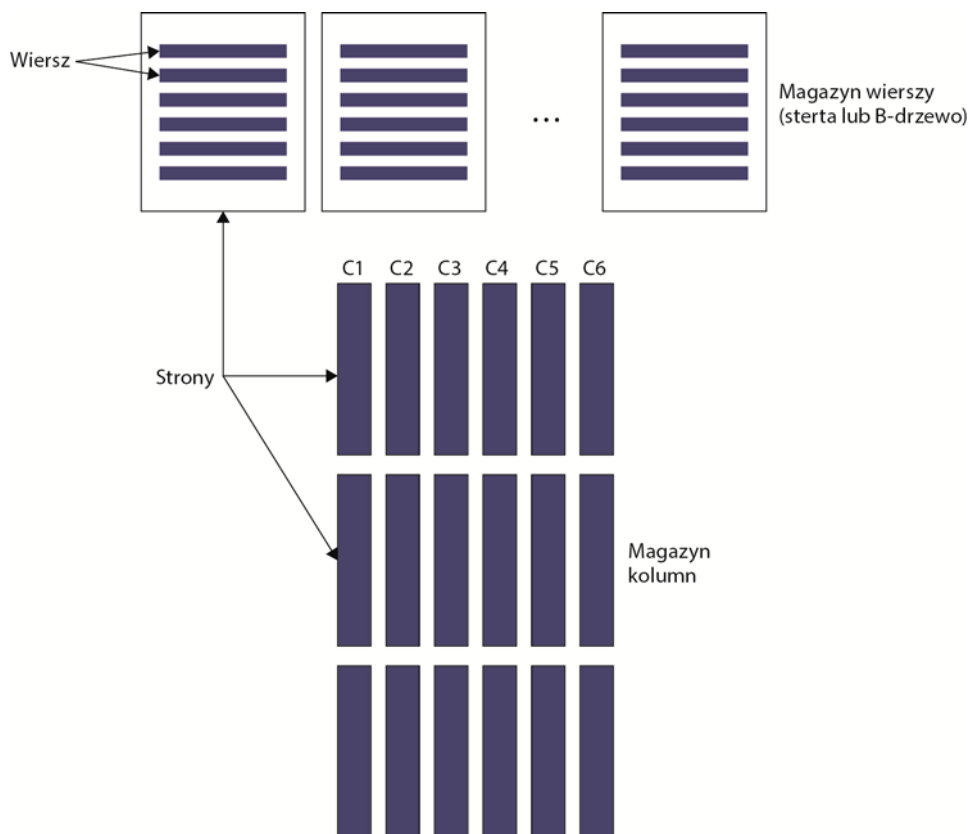
Jedną z głównych nowych funkcjonalności w SQL Serverze 2012 były indeksy magazynu kolumn. Korzystając z nowego podejścia do magazynowania danych opartego na kolumnach i nowych algorytmów przetwarzania zapytań, indeksy magazynu kolumn zoptymalizowane do działania w pamięci zostały zaprojektowane w celu poprawienia wydajności zapytań hurtowni danych o kilka rzędów wielkości. Chociaż podczas wprowadzania tej funkcjonalności brak możliwości aktualizacji danych był największą wadą, aktualna wersja, 2014, nie ma tego ograniczenia i obecnie można bezpośrednio aktualizować dane, a nawet tworzyć dla nich indeksy klastrowe magazynu kolumn. Fakt, że indeksy magazynu kolumn były pierwotnie ograniczone wyłącznie do indeksów nieklastrowych, również uważany był za wadę, ponieważ wymagał duplikację danych na i tak już bardzo dużym obiekcie takim jak tabela faktów.

Jak wspomniałem na początku tego rozdziału, w systemie OLTP transakcje zazwyczaj uzyskują dostęp do jednego lub kilku rekordów, natomiast typowe zapytania ze złączeniem gwiazdowym uzyskują dostęp do ogromnej liczby rekordów. Ponadto transakcja OLTP przeważnie pobierała wszystkie kolumny wiersza, a w przypadku zapytań ze złączeniem gwiazdowym zwykle wymagane jest tylko kilka kolumn. Ten wzorzec dostępu do danych pokazał, że podejście kolumnowe może być korzystne w przypadku hurtowni danych.

Tradycyjne podejście wykorzystywane przez SQL Server to przechowywanie danych na stronach danych — takie podejście nazywamy magazynem wierszy. Magazyny wierszy w SQL Serverze to sterty i struktury B-drzewa, takie jak standardowo indeksy klastrowe i nieklastrowe. Magazyn zorientowany na kolumny taki jak ten wykorzystywany przez indeksy magazynu kolumn dedykuje całe strony na przechowywanie danych z jednej kolumny. Oba typy magazynów zostały porównane na rysunku 9.4, gdzie magazyn wierszy zawiera strony z wierszami, każdy wiersz zawiera wszystkie kolumny, a magazyn kolumn zawiera strony z danymi tylko dla kolumn oznaczonych c1, c2 i tak dalej. Ponieważ dane przechowywane są w kolumnach, często zadawane pytanie dotyczy tego, jak z takiej struktury pobierany jest wiersz. To pozycja wartości w kolumnie wskazuje, do którego wiersza należy wartość. Na przykład pierwsza wartość na każdej stronie (c1, c2 i tak dalej) pokazana na rysunku 9.4 należy do pierwszego wiersza, druga wartość należy do drugiego wiersza i tak dalej.

Magazyn zorientowany na kolumny to nic nowego i był wykorzystywany wcześniej przez innych producentów baz danych. Indeksy magazynu kolumn opierają się na Microsoftowskiej technologii xVelocity, znanej wcześniej pod nazwą VertiPaq, która jest również stosowana w produktach SQL Server Analysis Services (SSAS), PowerPivot dla Excel i SharePoint. Podobnie jak w przypadku technologii Hekaton, omówionej w rozdziale 7., indeksy magazynu kolumn przechowywane są w pamięci.

Jak wspomniałem, indeksy magazynu kolumn dedykują całe strony bazy danych do przechowywania danych z jednej kolumny. Indeksy magazynu kolumn są również



Rysunek 9.4. Porównanie przechowywania danych w magazynie wierszy i w magazynie kolumn

podzielone na segmenty, które składają się z wielu stron, a każdy segment jest przechowywany w SQL Serverze jako odrębna wartość typu BLOB. W SQL Serverze 2014 aktualnie możliwe jest zdefiniowanie indeksu magazynu kolumn jako indeksu klastrowego, co jest korzystne, ponieważ niweluje potrzebę duplikowania danych.

Na przykład w SQL Serverze 2012 na stercie lub zwykłym indeksie klastrowym musiał być stworzony nieklastrowy indeks magazynu kolumn, co oznaczało duplikowanie danych i tak już dużej tabeli faktów.

Korzyści wydajnościowe

Indeksy magazynu kolumn zapewniają zwiększoną wydajność dzięki poniższym właściwościom:

- **Redukcja I/O.** Ponieważ strony magazynu wierszy zawierają wszystkie kolumny wiersza, w hurtowni danych bez indeksu magazynu kolumn, SQL Server musi przeczytać wszystkie kolumny, nawet te, które nie są wymagane przez zapytanie.

Wykorzystanie indeksu magazynu kolumn do odczytania tylko wymaganych kolumn może dać oszczędność od 85 do 90% operacji I/O.

- ▶ **Przetwarzanie partiami.** Nowe algorytmy przetwarzania zapytań zostały zaprojektowane do przetwarzania danych partiami i są bardzo efektywne w przetwarzaniu dużych ilości danych. Przetwarzanie partiami dokładniej omówię w następnym podrozdziale.
- ▶ **Kompresja.** Ponieważ dane z tej samej kolumny przechowywane są wspólnie na tych samych stronach, będą z reguły miały podobne lub powtarzające się wartości, które mogą być bardziej efektywnie kompresowane. Kompresja może poprawić wydajność, gdyż do przeczytania skompresowanych danych wymagane jest mniej operacji I/O, a w pamięci może zmieścić się więcej danych. Kompresja magazynu kolumn nie jest taka sama jak kompresja wierszy lub stron dostępna od wersji 2008 i wykorzystuje algorytmy kompresji VertiPaq. Różnica polega na tym, że w indeksach magazynu kolumn wartości kolumn są automatycznie kompresowane, a kompresja nie może być wyłączona, podczas gdy kompresja wierszy i stron jest opcjonalna i musi zostać jawnie skonfigurowana.
- ▶ **Eliminacja segmentów.** Indeksy magazynu kolumn są podzielone na segmenty, a SQL Server przechowuje wartości minimalne i maksymalne dla każdego segmentu w widoku `sys.column_store_segments`. Informacja ta może być wykorzystana przez SQL Server do porównania z filtrem zapytania, aby zidentyfikować, czy segment może zawierać żądane dane, a tym samym uniknąć odczytu niepotrzebnych segmentów, oszczędzając zasoby I/O i CPU.

Ponieważ zapytania hurtowni danych wykorzystujące indeksy magazynu kolumn mogą teraz być w wielu przypadkach wykonywane wielokrotnie szybciej niż w przypadku struktury magazynu wierszy, zapewniają również inne korzyści, na przykład ograniczenie lub wyeliminowanie konieczności wstępnie budowanych agregacji takich jak kostki OLAP, widoki indeksowane i tabele podsumowujące. Jedyna akcja konieczna, aby wykorzystać te zalety, to zdefiniowanie indeksu magazynu kolumn dla tabeli faktów. Nie ma potrzeby zmieniania zapytań ani stosowania nowej składni; optymalizator zapytań automatycznie weźmie pod uwagę indeks magazynu kolumn — chociaż to, czy taki indeks zostanie wykorzystany, będzie decyzją opartą na kosztach. Indeksy magazynu kolumn wykazują też większą elastyczność na zmiany niż wstępnie budowane agregacje. Jeżeli zapytanie ulegnie zmianie, indeks magazynu kolumn nadal będzie użyteczny, a optymalizator zapytań automatycznie zareaguje na zmiany w zapytaniu, natomiast wstępnie budowane agregacje mogą nie być w stanie wspierać zmienionych zapytań lub będą wymagać zmian.

Przetwarzanie partiami

W rozdziale 1., omawiając tradycyjny tryb przetwarzania zapytań, opisałem, jak operatory pobierają rekordy z innych operatorów wiersz po wierszu — na przykład za pomocą metody `GetRow()`. Taki tryb przetwarzania wierszy sprawdza się doskonale w tradycyjnych zapytaniach OLTP, gdzie zapytania powinny zwracać niewielkie liczby rekordów. Jednak dla zapytań hurtowni danych, które w procesie agregacji danych przetwarzają ogromne liczby rekordów, tryb przetwarzania wiersz po wierszu może być bardzo kosztowny.

W przypadku indeksów magazynu kolumn SQL Server nie tylko zapewnia magazyn oparty na kolumnach, ale wprowadza także tryb przetwarzania dużych ilości danych partia po partii. Partia przechowywana jest jako wektor w odrębnej części pamięci i zazwyczaj reprezentuje około 1000 rekordów. W SQL Serverze 2012 tylko kilka operatorów może działać w trybie partii lub rekordów, a są to: *Columnstore Index Scan*, *Hash Aggregate*, *Hash Join*, *Project* i *Filter*. Operator *Batch Hash Table Build* działa tylko w trybie partii; używany jest do budowy tabeli haszy partii dla indeksu magazynu kolumn w złączeniu haszowym. W wersji 2014 liczba operatorów mogących pracować w trybie partii została powiększona, a istniejące operatory, przede wszystkim *Hash Join*, zostały ulepszone.

Plan zawierający jednocześnie operatory przetwarzające wiersze i partie może spowodować degradację wydajności, wskutek tego, że konieczność komunikacji między trybem wierszy a trybem partii jest bardziej kosztowna niż komunikacja między operatorami działającymi w tym samym trybie. W rezultacie każde przejście między trybami ma przypisany koszt, który optymalizator wykorzystuje do minimalizacji liczby takich przejść w planie. Ponadto kiedy operatory przetwarzające wiersze i partie występują w tym samym planie, należy się upewnić, czy większa część planu, a przynajmniej najbardziej kosztowna część, jest wykonywana w trybie partii.

Plany będą pokazywały zarówno szacowany, jak i rzeczywisty tryb wykonywania, co zobaczysz w przykładach w dalszej części tego podrozdziału. Zazwyczaj szacowany i rzeczywisty tryb wykonywania powinny mieć tę samą wartość dla operacji. Możesz użyć tej informacji do szukania problemów z planem wykonania i aby się upewnić, czy przetwarzanie partiami faktycznie jest wykorzystywane. Plan pokazujący szacowany tryb wykonywania jako *batch* (partie) i rzeczywisty tryb jako *row* (wiersz) może świadczyć o problemie wydajnościowym.

W SQL Serverze 2012 ograniczenia pamięci lub liczby wątków mogą spowodować dynamiczną zmianę operacji z trybu partii do trybu wierszy, tym samym zmniejszając wydajność zapytania. Jeżeli szacowany plan pokazywał plan równoległy, a plan rzeczywisty wykonywany był seryjnie, możesz zgadywać, że nie było dostępnych wystarczająco dużo wątków. Jeżeli plan wykonywany był równoległy, możesz zakładać, że nie było dość pamięci. W tej wersji SQL Servera niektóre z nowych operatorów wykonujących operacje w partiach mogą korzystać z tabel haszowych, które muszą w całości zmieścić się w pamięci. Kiedy w czasie wykonywania nie jest dostępna wystarczająca

ilość pamięci, SQL Server dynamicznie przestawi operacje na tryb wiersza, gdzie będą mogły zostać wykorzystane standardowe tabele haszowe, które mogą zostać przeniesione na dysk w razie zbyt małej ilości dostępnej pamięci. Przetworzenie na tryb wierszy z powodu ograniczonej pamięci może być spowodowane złym szacunkiem kardynalności, więc możesz również rozważyć zweryfikowanie i zaktualizowanie statystyk.

Wymagania dotyczące pamięci zostały zlikwidowane w SQL Serverze 2014; tabela haszowa dla tabeli budowania złączenia *Hash Join* nie musi w całości mieścić się w pamięci i może zapisać część danych na dysku. Chociaż zapis na dysku nie jest idealnym rozwiązaniem, pozwala na kontynuowanie operacji w trybie partii bez konieczności zmiany na tryb wiersza jak w SQL Serverze 2012.

Od wersji 2012 operator *Hash Join* wspierał tylko złączenia wewnętrzne. Ta funkcjonalność została rozszerzona w SQL Serverze 2014 poprzez wprowadzenie pełnego spektrum złączeń, takich jak wewnętrzne, zewnętrzne, częściowe i antyczęściowe. *UNION ALL* i agregacje wartości skalarnych również wspierają teraz przetwarzanie partiami.

Tworzenie indeksów magazynu kolumn

Stworzenie nieklastrowego indeksu magazynu kolumn wymaga zdefiniowania listy kolumn, które mają być uwzględnione w indeksie, co nie jest konieczne, jeżeli stworzysz klastrowy indeks magazynu kolumn. Klastrowy indeks magazynu kolumn zawiera wszystkie kolumny w tabeli i przechowuje całą tabelę. Możesz stworzyć klastrowy indeks magazynu kolumn ze sterty lub ze zwykłego indeksu klastrowego. Chociaż dla tabeli możesz tworzyć klastrowe i nieklastrowe indeksy magazynu kolumn, w przeciwieństwie do magazynów wierszy, jednocześnie może istnieć tylko jeden taki indeks. Załóżmy na przykład, że stworzysz poniższy klastrowy indeks magazynu kolumn, korzystając z małej wersji tabeli *FactInternetSales2*:

```
CREATE TABLE dbo.FactInternetSales2 (
    ProductKey int NOT NULL,
    OrderDateKey int NOT NULL,
    DueDateKey int NOT NULL,
    ShipDateKey int NOT NULL)
GO
CREATE CLUSTERED COLUMNSTORE INDEX csi_FactInternetSales2
ON dbo.FactInternetSales2
```

Spróbuj teraz stworzyć nieklastrowy indeks:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX ncsl_FactInternetSales2
ON dbo.FactInternetSales2
(ProductKey, OrderDateKey)
```

Otrzymasz poniższy błąd:

```
Msg 35303, Level 16, State 1, Line 1
CREATE INDEX statement failed because a nonclustered index cannot be created on a table that
↳ has a clustered columnstore index. Consider replacing the clustered columnstore index with
```

- ↳ a nonclustered columnstore index. // Polecenie *CREATE INDEX* nie zostało wykonane, ponieważ indeks nieklastrowy nie może zostać stworzony dla tabeli mającej klastrowy indeks magazynu kolumn. Rozważ
- ↳ zamienienie klastrowego indeksu magazynu kolumn na nieklastrowy indeks magazynu kolumn.

Podobnie, jeżeli masz już nieklastrowy indeks magazynu kolumn i spróbujesz stworzyć indeks klastrowy, otrzymasz poniższy komunikat:

```
Msg 35304, Level 16, State 1, Line 1
CREATE INDEX statement failed because a clustered columnstore index cannot be created on
↳ a table that has a nonclustered index. Consider dropping all nonclustered indexes and trying
↳ again. // Polecenie CREATE INDEX nie zostało wykonane, ponieważ klastrowy indeks magazynu kolumn nie
↳ może zostać stworzony dla tabeli mającej indeks nieklastrowy. Rozważ usunięcie wszystkich indeksów
↳ nieklastrowych i ponowną próbę.
```

Zanim zaczniemy następne ćwiczenie, usuń stworzoną tabelę za pomocą poniższego polecenia:

```
DROP TABLE FactInternetSales2
```

Spójrzmy teraz na plany tworzone dla indeksów magazynu kolumn. Uruchom poniższy kod, aby stworzyć tabelę FactInternetSales2 w bazie AdventureWorksDW2012:

```
USE AdventureWorksDW2012
GO
CREATE TABLE dbo.FactInternetSales2 (
    ProductKey int NOT NULL,
    OrderDateKey int NOT NULL,
    DueDateKey int NOT NULL,
    ShipDateKey int NOT NULL,
    CustomerKey int NOT NULL,
    PromotionKey int NOT NULL,
    CurrencyKey int NOT NULL,
    SalesTerritoryKey int NOT NULL,
    SalesOrderNumber nvarchar(20) NOT NULL,
    SalesOrderLineNumber tinyint NOT NULL,
    RevisionNumber tinyint NOT NULL,
    OrderQuantity smallint NOT NULL,
    UnitPrice money NOT NULL,
    ExtendedAmount money NOT NULL,
    UnitPriceDiscountPct float NOT NULL,
    DiscountAmount float NOT NULL,
    ProductStandardCost money NOT NULL,
    TotalProductCost money NOT NULL,
    SalesAmount money NOT NULL,
    TaxAmt money NOT NULL,
    Freight money NOT NULL,
    CarrierTrackingNumber nvarchar(25) NULL,
    CustomerPONumber nvarchar(25) NULL,
    OrderDate datetime NULL,
    DueDate datetime NULL,
    ShipDate datetime NULL
)
```

Możemy teraz wstawić trochę rekordów, kopiując je z pierwotnej tabeli faktów:

```
INSERT INTO dbo.FactInternetSales2
SELECT * FROM AdventureWorksDW2012.dbo.FactInternetSales
WHERE SalesOrderNumber < 'S06'
```

Aby stworzyć klastrowy indeks magazynu kolumn, uruchom następujące polecenie:

```
CREATE CLUSTERED COLUMNSTORE INDEX csi_FactInternetSales2
ON dbo.FactInternetSales2
```

Poniższe polecenie pokaże, że od wersji 2014 indeksy magazynu kolumn mogą być aktualizowane i wspierają polecenia INSERT, DELETE, UPDATE i MERGE, a także inne standardowe metody, takie jak masowe narzędzie bcp i SQL Server Integration Services:

```
INSERT INTO dbo.FactInternetSales2
SELECT * FROM AdventureWorksDW2012.dbo.FactInternetSales
WHERE SalesOrderNumber > 'S06'
```

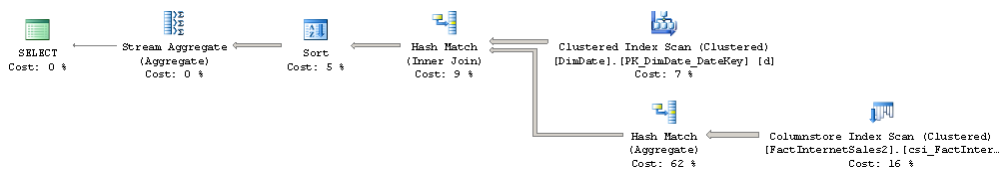
Próba uruchomienia poprzedniego zapytania dla indeksu magazynu kolumn na instancji SQL Servera 2012 zwróciłaby poniższy komunikat błędu, który opisuje również jedno z rozwiązań pozwalające wstawiać rekordy do indeksu magazynu kolumn w tej wersji SQL Servera:

```
Msg 35330, Level 15, State 1, Line 1
INSERT statement failed because data cannot be updated in a table with a columnstore index.
↳ Consider disabling the columnstore index before issuing the INSERT statement, then
↳ rebuilding the columnstore index after INSERT is complete. // Polecenie INSERT nie zostało
↳ wykonane, ponieważ dane w tabeli z indeksem magazynu kolumn nie mogą być aktualizowane. Rozważ wyłączenie
↳ indeksu magazynu kolumn przed ponownym uruchomieniem polecenia INSERT, a następnie przebudowę indeksu
↳ magazynu kolumn po zakończeniu operacji INSERT.
```

W tej chwili zapytanie ze złączeniem gwiazdzystym może nie mieć dość wierszy, aby wyprodukować plan wykorzystujący przetwarzanie partiami. Przetestuj to, uruchamiając poniższe zapytanie:

```
SELECT d.CalendarYear,
SUM(SalesAmount) AS SalesTotal
FROM dbo.FactInternetSales2 AS f
JOIN dbo.DimDate AS d
ON f.OrderDateKey = d.DateKey
GROUP BY d.CalendarYear
ORDER BY d.CalendarYear
```

Chociaż wyprodukowany plan zawiera operator *Columnstore Index Scan* (zobacz rysunek 9.5), sprawdzenie predykatów tego operatora pokaże, że rzeczywisty tryb wykonywania (*Actual Execution Mode*) to wiersze (*Row*), co oznacza, że wykorzystane zostało przetwarzanie wiersz po wierszu.

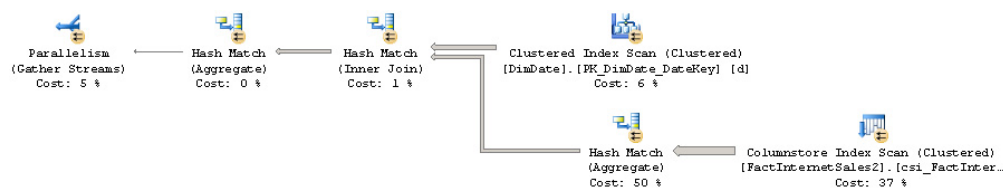


Rysunek 9.5. Plan wykorzystujący operator Columnstore Index Scan w trybie wiersz po wierszu

Dodajmy więcej rekordów, uruchamiając polecenie INSERT 20 razy:

```
INSERT INTO dbo.FactInternetSales2
SELECT * FROM AdventureWorksDW2012.dbo.FactInternetSales
GO 20
```

Chociaż skopiowanie tych samych danych 20 razy nie wpływa pozytywnie na jakość testu, będziesz przynajmniej mógł go wykonać bez dostępu do większej hurtowni danych. Tym razem dostaniemy bardziej kosztowny plan równoległy, wykorzystujący przetwarzanie w partiach, a jego część została pokazana na rysunku 9.6.



Rysunek 9.6. Plan wykorzystujący operator Columnstore Index Scan w trybie partii

Właściwości operatora *Columnstore Index Scan* pokazują teraz, że rzeczywisty tryb wykonania to partie (zobacz rysunek 9.7). Właściwość *Storage* (magazyn) pokazuje wartość *ColumnStore* (magazyn kolumn). Zauważ, że operator pokazuje również zoptymalizowaną mapę bitową, *Opt_Bitmap1006*, używaną w sekcji predykatów. Jest ona tworzona w górnym złączeniu haszowym używającym tabeli wymiarów *DimDate* jako wsadu budowania, ta część jednak nie jest pokazana na planie.

Podobnie jak w przypadku magazynu wierszy, możesz skorzystać z polecenia `ALTER INDEX REBUILD`, aby usunąć fragmentację indeksu:

```
ALTER INDEX csi_FactInternetSales2 on FactInternetSales2 REBUILD
```

Ponieważ tabela faktów jest bardzo duża, a operacja przebudowania indeksu magazynu kolumn jest operacją typu offline, możesz popartycjonować dane i zastosować te same zalecenia, które dotyczą dużych tabel magazynujących dane w wierszach. Oto ogólne podsumowanie procesu:

- ▶ Przebuduj tylko ostatnio wykorzystywaną partycję. Pojawienie się fragmentacji jest bardziej prawdopodobne dla partycji, które były ostatnio zmieniane.
- ▶ Przebuduj tylko partycje aktualizowane po załadowaniu danych albo po wykonaniu operacji DML.

Podobnie jak w przypadku usunięcia standardowego indeksu klastrowego, usunięcie klastrowego indeksu magazynu kolumn przekonwertuje tabelę z powrotem do sterty. Aby to sprawdzić, uruchom następujące zapytanie:

```
SELECT * FROM sys.indexes
WHERE object_id = OBJECT_ID('FactInternetSales2')
```

Columnstore Index Scan (Clustered)	
Scan a columnstore index, entirely or only a range.	
Physical Operation	Columnstore Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	ColumnStore
Actual Number of Rows	1268358
Actual Number of Batches	1509
Estimated I/O Cost	0.161644
Estimated Operator Cost	0.231411 (37%)
Estimated Subtree Cost	0.231411
Estimated CPU Cost	0.0697675
Estimated Number of Executions	1
Number of Executions	4
Estimated Number of Rows	1268360
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	7
Predicate	
PROBE([Opt_Bitmap1006],[AdventureWorksDW2012].[dbo].[FactInternetSales2].[OrderDateKey] as [f].[OrderDateKey])	
Object	
[AdventureWorksDW2012].[dbo].[FactInternetSales2].[csi_FactInternetSales2] [f]	
Output List	
[AdventureWorksDW2012].[dbo].[FactInternetSales2].[OrderDateKey], [AdventureWorksDW2012].[dbo].[FactInternetSales2].[SalesAmount]	

Rysunek 9.7. Właściwości operatora Columnstore Index Scan

Uzyskasz wynik podobny do poniższego:

object_id	name	index_id	type	type_desc
1762105318	csi_FactInternetSales2	1	5	CLUSTERED COLUMNSTORE

Usunięcie indeksu za pomocą polecenia DROP INDEX i powtórne uruchomienie powyższego zapytania zmieni wartość index_id na 0 i wartość type_desc na HEAP:

```
DROP INDEX FactInternetSales2.csi_FactInternetSales2
```

Jeżeli korzystasz z indeksów magazynu kolumn w SQL Serverze 2012, możesz posłużyć się jednym z poniższych sposobów aktualizacji danych:

- **Wykorzystanie przełączania pomiędzy partycjami** — wymaga, aby do ładowania danych do tabeli faktów użyć tabeli przejściowej, którą można aktualizować. Kiedy dane zostaną załadowane, możesz stworzyć indeks magazynu kolumn na tabeli przejściowej, a następnie podmienić tabelę przejściową do pustej partycji w tabeli głównej. Podobna procedura może pomóc w aktualizacji istniejących danych, ale najpierw musisz zmienić partycję tabeli faktów na tabelę przejściową, usunąć jej indeks, a potem ją zaktualizować. Po aktualizacji

danych możesz stworzyć indeks magazynu kolumn na tabeli przejściowej, a następnie podmienić tabelę przejściową do pustej partycji tabeli głównej, jak podczas wstawiania.

- ▶ **Wykorzystanie UNION ALL** — wymaga tabeli faktów z indeksem magazynu kolumn i zwykłej tabeli o tym samym schemacie, która będzie zawierała najbardziej aktualne dane. Będziesz mógł wtedy odpytywać obie tabele jednocześnie za pomocą UNION ALL. Wykorzystując ten sposób, należy zadbać, aby przetwarzanie partiami było faktycznie stosowane dla najcięższych operacji.
- ▶ **Powtórne stworzenie indeksu** — wiąże się z usunięciem lub wyłączeniem indeksu, zaktualizowaniem lub wstawieniem niezbędnych danych i powtórnym stworzeniem lub włączeniem indeksu magazynu kolumn. Jest to zalecenie przedstawione w pokazanym wcześniej komunikacie błędu 35330.

Podpowiedzi

W przypadkach, kiedy optymalizator nie buduje odpowiedniego planu podczas pracy z indeksami magazynu kolumn, użyteczne może być kilka podpowiedzi. Jeden z takich przypadków ma miejsce, gdy optymalizator ignoruje ten indeks. Możesz użyć podpowiedzi INDEX, aby wymusić wykorzystanie istniejącego nieklastrowego indeksu magazynu kolumn. Aby przetestować tę podpowiedź, stwórz poniższy indeks na istniejącej tabeli faktów FactInternetSales:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX csi_FactInternetSales
ON dbo.FactInternetSales (
    ProductKey,
    OrderDateKey,
    DueDateKey,
    ShipDateKey,
    CustomerKey,
    PromotionKey,
    CurrencyKey,
    SalesTerritoryKey,
    SalesOrderNumber,
    SalesOrderLineNumber,
    RevisionNumber,
    OrderQuantity,
    UnitPrice,
    ExtendedAmount,
    UnitPriceDiscountPct,
    DiscountAmount,
    ProductStandardCost,
    TotalProductCost,
    SalesAmount,
    TaxAmt,
    Freight,
    CarrierTrackingNumber,
    CustomerPONumber,
    OrderDate,
```

```

    DueDate,
    ShipDate
)

```

Poniższe zapytanie przedstawia sposób wykorzystania podpowiedzi INDEX:

```

SELECT d.CalendarYear,
SUM(SalesAmount) AS SalesTotal
FROM dbo.FactInternetSales AS f
    WITH (INDEX(csi_FactInternetSales))
    JOIN dbo.DimDate AS d
        ON f.OrderDateKey = d.DateKey
GROUP BY d.CalendarYear
ORDER BY d.CalendarYear

```

Innym przypadkiem jest sytuacja, w której z jakiegoś powodu nie chcesz wykorzystywać istniejącego indeksu magazynu kolumn. Nowa podpowiedź IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX pozwala wymusić takie zachowanie:

```

SELECT d.CalendarYear,
SUM(SalesAmount) AS SalesTotal
FROM dbo.FactInternetSales AS f
    JOIN dbo.DimDate AS d
        ON f.OrderDateKey = d.DateKey
GROUP BY d.CalendarYear
ORDER BY d.CalendarYear
OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX)

```

Możesz teraz usunąć stworzony indeks za pomocą poniższego polecenia:

```
DROP INDEX FactInternetSales.csi_FactInternetSales
```

Podsumowanie

W tym rozdziale omówiłem hurtownie danych i wyjaśniłem, w jaki sposób systemy operacyjne i systemy analityczne odpowiadają na bardzo różne potrzeby — pierwsze pomagają w procesie biznesowym, a drugie wspomagają mierzenie tego procesu. Fakt, że hurtownia danych jest wykorzystywana w inny sposób niż system OLTP, stworzył okazję do wdrożenia nowego modelu dla silnika bazy danych, który został zaimplementowany z zastosowaniem indeksów magazynu kolumn, wprowadzonych pierwotnie w wersji 2012.

Typowe zapytanie ze złączeniem gwiazdowym wykorzystuje tylko 10 – 15% kolumn tabeli faktów, dlatego zaimplementowane zostało podejście oparte na kolumnach, które wydaje się bardziej odpowiednie niż podejście tradycyjne, oparte na wierszach. Ponieważ przetwarzanie wiersz po wierszu również nie było wydajną metodą w przypadku dużych ilości wierszy przetwarzanych przez zapytania ze złączeniem gwiazdowym, zaimplementowany został nowy tryb przetwarzania w partiach. Wreszcie, podążając za trendami w sprzęcie, indeksy magazynu kolumn, podobnie jak w przypadku mechanizmu Hekaton, zostały przeniesione do pamięci.

SQL Server jest w stanie automatycznie wykryć i zoptymalizować zapytania magazynu danych. Jedną z objaśnionych optymalizacji, używającą filtrowania bitmapowego, oparta jest na koncepcji z 1970 roku. Filtrowanie bitmapowe może być użyte do filtrowania wierszy z tabeli faktów na bardzo wczesnym etapie przetwarzania zapytań, poprawiając tym samym wydajność zapytań ze złączeniem gwiazdowym.

Rozdział 10

Ograniczenia i podpowiedzi procesora zapytań

W tym rozdziale:

- ▶ Badania dotyczące optymalizacji zapytań
- ▶ Kolejność złączeń
- ▶ Rozbijanie skomplikowanych zapytań
- ▶ Podpowiedzi
- ▶ Podsumowanie



Optymalizacja zapytań to bardzo skomplikowany problem, nie tylko dla SQL Servera, ale także dla każdego innego systemu relacyjnego. Pomimo faktu, że badania dotyczące optymalizacji zapytań sięgają początku lat siedemdziesiątych XX wieku, wyzwania w niektórych fundamentalnych aspektach wciąż są rozwiązywane. Pierwszym dużym problem w znalezieniu optymalnego planu jest to, że dla wielu zapytań zbadanie całego obszaru przeszukiwania nie jest możliwe. Efekt znany jako *eksplozja kombinacji* sprawia, że liczba możliwych kombinacji rośnie gwałtownie wraz ze wzrostem liczby tabel w złączeniach. Aby uczynić możliwym wykonanie tego procesu, do ograniczenia obszaru przeszukiwania (czyli liczby możliwych planów do rozważenia) stosowana jest heurystyka, zgodnie z informacjami z rozdziału 3. Jeżeli jednak optymalizator nie będzie w stanie zbadać całego obszaru poszukiwań, nie ma możliwości, aby udowodnić, że możesz uzyskać absolutnie optymalny plan, a nawet że najlepszy plan jest wśród kandydatów do rozpatrzenia, niezależnie od tego, czy zostanie wybrany, czy nie. Dlatego bardzo ważne jest, aby zestaw planów, które rozważa optymalizator zapytań, składał się z planów o jak najniższym koszcie.

To prowadzi nas do kolejnego dużego problemu technicznego: trafne szacowanie kosztu i kardynalności. Ponieważ optymalizator oparty na kosztach wybiera plan wykonania z najniższym szacowanym kosztem, jakość wybranego planu jest tylko na tyle dobra, na ile trafne są szacunki kosztu i kardynalności. Nawet zakładając, że czas nie jest problemem i że optymalizator z łatwością może przeanalizować całą przestrzeń poszukiwań, błędy w szacowaniu kardynalności i kosztów sprawią, że wybrany zostanie nieoptymalny plan. Modele szacowania kosztów są oczywiście niedokładne, ponieważ nie uwzględniają wszystkich warunków sprzętowych i muszą bazować na pewnych założeniach dotyczących środowiska. Na przykład model kosztów zakłada, że każde zapytanie zaczyna się z pustą pamięcią podręczną (czyli że dane czytane są z dysku, a nie z pamięci), a to założenie może w pewnych przypadkach prowadzić do błędnych szacunków kosztów. Ponadto szacunek kosztów zależy od szacunku kardynalności, który również jest niedokładny i ograniczony, szczególnie jeżeli chodzi o szacowanie wyników pośrednich w planie. Błędy w wynikach pośrednich w efekcie nawarstwiają się wraz z kolejnymi złączeniami tabel i w kalkulacje wkrada się coraz więcej błędów. Na dodatek niektóre operacje nie są uwzględniane w modelu matematycznym komponentu szacującego kardynalność, co oznacza, że optymalizator musi zgadywać lub odwoływać się do heurystyki, aby radzić sobie z tymi sytuacjami.



UWAGA

Jak wspomniałem we wcześniejszej części książki, w SQL Serverze 2014 wprowadzony został nowy mechanizm szacowania kardynalności mający poprawić jakość i wsparcie procesu szacowania kardynalności. Więcej szczegółów uzyskasz w rozdziale 6.

Badania nad optymalizacją zapytań

Badania nad optymalizacją zapytań rozpoczęto we wczesnych latach siedemdziesiątych. Jedną z pierwszych prac opisujących optymalizację opartą na kosztach to opublikowany w 1979 roku artykuł *Access Path Selection in a Relational Database Management System* autorstwa między innymi Pata Selingera. Autorzy opisują w nim optymalizator zapytań dla eksperymentalnego systemu zarządzania bazą danych stworzonego w 1975 roku w placówce znanej teraz jako IBM Almaden Research Center. System ten, nazywany *System R*, wzbogacił pole optymalizacji zapytań poprzez wprowadzenie optymalizacji opartej na kosztach, wykorzystanie statystyk i efektywnej metody determinowania kolejności złączeń, a także dodanie kosztu CPU do formuły szacowania kosztu.

Jednak mimo ogromnego wpływu na badania nad optymalizacją zapytań system ten miał ogromną wadę: nie mógł być rozbudowany o dodatkowe transformacje. To doprowadziło do stworzenia bardziej elastycznych architektur optymalizacyjnych, które pozwalały na stopniowe dodawanie nowych funkcjonalności do optymalizatorów. Pionierami w tej dziedzinie były: Exodus Optimizer Generator zdefiniowany przez Goetza Graefego i Davida DeWitta oraz późniejszy Volcano Optimizer Generator Goetza Graefego i Williama McKenny. Goetz Graefe skupił się później na pracy nad systemem Cascades Framework, rozwiązując problemy z poprzednich dwóch systemów.

Najważniejsze w tych badaniach dla nas jest to, że SQL Server w roku 1999, kiedy system został przebudowany dla wersji 7.0, zaimplementował nowy mechanizm szacowania kosztów oparty na Cascades. Elastyczna architektura tego frameworku sprawiła, że dodawanie nowych funkcjonalności, takich jak nowe reguły transformacji czy operatory fizyczne, jest znacznie łatwiejsze.

Kolejność złączeń

Kolejność złączeń to jeden z bardziej skomplikowanych problemów z dziedziny optymalizacji zapytań i jest tematem dogłębnych badań od lat siedemdziesiątych. Odnosi się do procesu obliczania optymalnej kolejności złączeń (czyli kolejności, w jakiej łączone są tabele biorące udział w zapytaniu). Ponieważ kolejność złączeń to kluczowy czynnik pozwalający kontrolować przepływ danych pomiędzy operatorami w planie, optymalizator musi zwracać na nią baczną uwagę. Jak już wspomniałem, kolejność złączeń jest bezpośrednio związana z rozmiarem obszaru poszukiwań, gdyż liczba możliwych planów rośnie gwałtownie wraz ze wzrostem liczby łączonych tabel.

Operacja złączenia łączy rekordy z dwóch tabel na podstawie jakiejś wspólnej informacji, a predykat definiujący, które kolumny mają być wykorzystane do połączenia tabel, nazywa się *predykatem złączenia*. Złączenie jednocześnie działa tylko dla dwóch tabel, dlatego złączenie żądające danych z n tabel musi zostać wykonane jako sekwencja

$n - 1$ złączeń, chociaż należy zaznaczyć, że złączenie nie musi być zakończone (tzn. nie muszą zostać połączone wszystkie żądane rekordy z obu tabel), aby rozpoczęło się następne złączenie.

Optymalizator musi w kwestii złączeń podjąć dwie ważne decyzje: wybrać kolejność złączeń i wybrać algorytm złączenia. Wybór algorytmu złączenia opisałem w rozdziale 4., więc w tym podrozdziale opowiem tylko o kolejności złączeń. Jak wspomniałem, kolejność, w jakiej łączone są tabele, może znacznie wpłynąć na koszt i wydajność zapytania. Chociaż wyniki zapytania są takie same niezależnie od kolejności złączeń, koszt różnych kombinacji kolejności może się znacznie różnić.

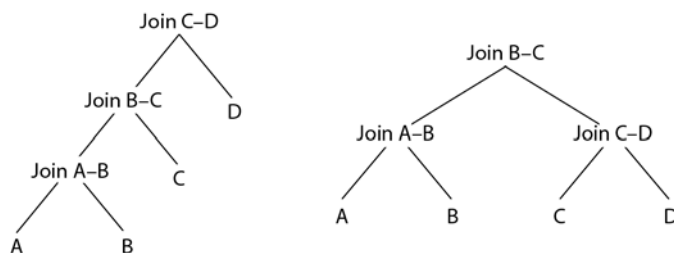
W rezultacie komutatywnych i asocjacyjnych właściwości złączeń nawet proste zapytania mają wiele możliwych kolejności złączeń, a liczba ta rośnie wykładniczo ze wzrostem liczby tabel, które trzeba połączyć. Zadaniem optymalizatora zapytań jest znalezienie optymalnej sekwencji złączeń pomiędzy tabelami użytymi w zapytaniu.

Komutatywna właściwość złączenia pomiędzy tabelami A i B oznacza, że *A JOIN B* jest logicznym ekwiwalentem *B JOIN A*. To definiuje, która tabela zostanie odczytana jako pierwsza lub, mówiąc inaczej, jaką rolę w złączeniu odegra każda z tabel. Na przykład w przypadku operatora *Nested Loops Join* pierwsza jest odczytywana tabela zewnętrzna, a druga — wewnętrzna. W złączeniu *Hash Join* pierwszą odczytywaną tabelą jest wsad budowania, a drugą — wsad sondowany. Jak widziałeś w rozdziale 4., poprawne zdefiniowanie, która tabela odegra rolę tabeli wewnętrznej, a która zewnętrznej w złączeniu *Nested Loops Join*, a także która tabela będzie wsadem budowania, a która wsadem sondowanym w złączeniu *Hash Join*, ma ogromne znaczenie dla wydajności i kosztu i jest to wybór podejmowany przez optymalizator zapytań.

Właściwość asocjacyjna złączenia pomiędzy tabelami A, B i C oznacza, że *(A JOIN B) JOIN C* jest logicznym ekwiwalentem *A JOIN (B JOIN C)*. Na przykład *(A JOIN B) JOIN C* specyfikuje, że tabela A musi najpierw zostać połączona z tabelą B, a następnie rezultat musi zostać połączony z tabelą C. *A JOIN (B JOIN C)* oznacza, że tabela B musi najpierw zostać połączona z tabelą C, a następnie rezultat musi zostać złączony z tabelą A. Każda możliwa permutacja może mieć inny koszt i inną wydajność, w zależności, na przykład, od rozmiaru wyników tymczasowych.

Liczba możliwych kolejności złączeń w zapytaniu wzrasta wykładniczo wraz z liczbą łączonych tabel. W istocie, gdy mamy zaledwie kilka tabel, liczba możliwości może iść w tysiące lub nawet miliony, chociaż dokładna liczba kombinacji uzależniona jest od ogólnego kształtu drzewa zapytania. Oczywiście dla optymalizatora niemożliwe jest przejrzanie wszystkich tych kombinacji: zajęłoby to zbyt wiele czasu. SQL Server musi więc wykorzystywać heurystykę do ograniczenia przestrzeni przeszukiwania.

Jak już wspomniałem, zapytania w procesorze zapytań są reprezentowane jako drzewa, a kształt drzewa zapytania, podyktowany przez złączenia, jest tak ważny, że niektóre z tych drzew mają swoje nazwy, takie jak lewostronnie rozgałęzione, prawostronnie rozgałęzione czy krzaczaste. Rysunek 10.1 przedstawia drzewo lewostronnie



Rysunek 10.1. Drzewo lewostronnie rozgałęzione i drzewo krzaczaste

rozgałęzione i krzaczaste dla złączenia czterech tabel. Na przykład drzewo lewostronnie rozgałęzione mogłoby odzwierciedlać złączenie $JOIN(JOIN(JOIN(A, B), C), D)$, a drzewo krzaczaste — złączenie $JOIN(JOIN(A, B), JOIN(C, D))$.

Drzewa lewostronnie rozgałęzione nazywane są również drzewami linearnymi lub drzewami przetwarzania linearnego i jak widzisz, ich wygląd pasuje do opisu. Drzewa krzaczaste mogą natomiast przyjmować dowolny kształt, więc zbiór drzew krzaczastych zawiera zarówno drzewa lewostronnie rozgałęzione, jak i –prawostronnie rozgałęzione.

Liczba drzew lewostronnie rozgałęzionych obliczana jest jako $n!$ (n silnia), gdzie n to liczba tabel w relacji. *Silnia* to wynik mnożenia wszystkich możliwych wartości całkowitych mniejszych lub równych n . Na przykład dla złączeń pięciu tabel liczba możliwych kombinacji kolejności wynosi $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Liczba możliwych kombinacji kolejności złączeń dla drzewa krzaczastego jest bardziej skomplikowana i można ją obliczyć jako $(2n - 2)! / (n - 1)!$.

Należy zapamiętać, że liczba możliwych kombinacji kolejności złączeń rośnie bardzo szybko wraz ze wzrostem liczby łączonych tabel, zgodnie z tabelą 10.1. Na przykład, teoretycznie, gdybyśmy mieli złączenie sześciu tabel, optymalizator musiałby rozpatrzyć 30240 możliwych kombinacji. Oczywiście jest to tylko liczba możliwych permutacji. Oprócz tego optymalizator musi określić operatory fizyczne i metody dostępu do danych, a także zoptymalizować inne części zapytania, takie jak agregacje, podzapytania i tak dalej.

Jak więc optymalizator zapytań SQL Servera analizuje wszystkie te możliwe kombinacje? Odpowiedź brzmi: nie robi tego. Wykonanie pełnej ewaluacji wszystkich kombinacji dla większości zapytań trwałoby za długo, dlatego optymalizator musi znaleźć równowagę między czasem optymalizacji i jakością planu wynikowego. Celem optymalizatora jest znalezienie wystarczająco dobrego planu w jak najkrótszym czasie. Zamiast analizować każdą z kombinacji optymalizator, korzystając z heurystyki, próbuje zawęzić możliwości do najbardziej prawdopodobnych kandydatów. Ten proces wyjaśniłem w rozdziale 3.

Tabela 10.1. Możliwe kombinacje kolejności złączeń dla drzew lewostronnie rozgałęzionego i krzaczastego

Liczba tabel	Liczba drzew lewostronnie rozgałęzionych	Liczba drzew krzaczastych
2	2	2
3	6	12
4	24	120
5	120	1680
6	720	30 240
7	5040	665 280
8	40 320	17 297 280
9	362 880	518 918 400
10	3 628 800	17 643 225 600
11	39 916 800	670 442 572 800
12	479 001 600	28 158 588 057 600

Rozbijanie skomplikowanych zapytań

Jak widziałeś w poprzednim podrozdziale, w niektórych przypadkach optymalizator SQL Servera może nie być w stanie wyprodukować wystarczająco dobrego planu dla zapytań z dużą liczbą złączeń. To samo dotyczy skomplikowanych zapytań zawierających złączenia i agregacje. Ponieważ jednak żądanie wszystkich danych w jednym zapytaniu zazwyczaj nie jest konieczne, dobrym rozwiązaniem w takich przypadkach jest rozbić skomplikowanego zapytania na dwa lub więcej prostszych zapytań i przechowywanie wyników pośrednich w tabelach tymczasowych. Rozbić skomplikowanych zapytań daje różnorakie korzyści:

- ▶ **Lepsze plany.** Wydajność zapytań jest lepsza, ponieważ optymalizator dla prostszych zapytań jest w stanie tworzyć wydajniejsze plany.
- ▶ **Lepsze statystyki.** Ponieważ jednym z problemów bardziej skomplikowanych planów jest degradacja statystyk pośrednich, dzięki rozbić zapytań i przechowywaniu wyników pośrednich w tabelach tymczasowych SQL Server może tworzyć nowe statystyki, co bardzo poprawi jakość szacunku kardynalności dla pozostałych zapytań. Warto zauważyć, że należy korzystać z tabel tymczasowych, a nie ze zmiennych tabelarycznych, gdyż te drugie nie mają wsparcia dla statystyk.
- ▶ W wersji SQL Server 2012 Service Pack 2 wprowadzona została nowa flaga, która oferuje lepsze szacowanie kardynalności dla tabel tymczasowych. Flaga ta ma również zostać zaimplementowana w SQL Serverze 2014. Więcej szczegółów znajdziesz pod adresem <http://support.microsoft.com/kb/2952444>.

- ▶ **Brak konieczności wykorzystania podpowiedzi.** Ponieważ wykorzystanie podpowiedzi to powszechna praktyka w naprawianiu problemów ze skomplikowanymi planami, rozbicie zapytania pozwala uzyskać wydajny plan bez konieczności ich stosowania. Dodatkowym atutem jest to, że optymalizator może automatycznie reagować na przyszłe zmiany danych lub schematu. Zapytanie korzystające z podpowiedzi wymagałoby interakcji, gdyż wykorzystana podpowiedź może przestać być użyteczna lub nawet może negatywnie wpływać na wydajność zapytania po zmianach. Podpowiedzi, które powinny być wykorzystywane tylko w ostateczności, omówię w dalszej części tego rozdziału.

W artykule *When to Break Down Complex Queries* autorzy opisują kilka problematycznych wzorców zapytań, dla których optymalizator SQL Servera nie jest w stanie stworzyć dobrego planu. Chociaż artykuł został opublikowany w roku 2011 i odnosi się do wersji od SQL Server 2005 do wersji z nazwą kodową Denali, to samo zachowanie zaobserwowałem w SQL Serverze 2014. Oto kilka z tych wzorców, które po krótko omówię:

- ▶ logika OR w klauzuli WHERE;
- ▶ złączenia i zagregowane zbiory danych;
- ▶ zapytania z dużą liczbą skomplikowanych złączeń.

Logika OR w klauzuli WHERE

SQL Server jest w stanie zidentyfikować i stworzyć efektywne plany z użyciem logiki OR w klauzuli WHERE w następujących przypadkach:

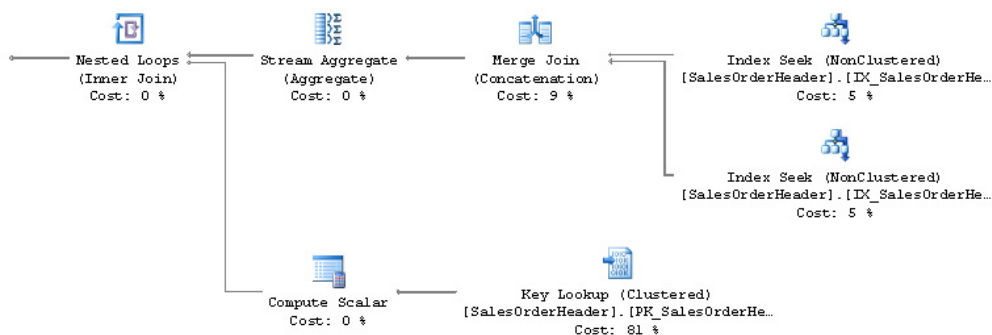
- ▶ WHERE a.col1 = @val1 OR a.col1 = @val2, co jest równoważne z WHERE a.col1 IN (@val1, @val2)
- ▶ WHERE a.col1 = @val1 OR a.col2 = @val2
- ▶ WHERE a.col1 = @val1 OR a.col2 IN (SELECT col2 FROM tab2)

Wszystkie powyższe zapytania wykorzystują te same lub różne kolumny, ale w tej samej tabeli. Oto przykład zapytania obrazujący pierwszy przypadek:

```
SELECT * FROM Sales.SalesOrderHeader
WHERE CustomerID = 11020 OR SalesPersonID = 285
```

W tym zapytaniu SQL Server jest w stanie skorzystać z operacji przeszukania indeksu na dwóch indeksach, IX_SalesOrderHeader_CustomerID i IX_SalesOrderHeader_SalesPersonID, i skorzystać z unii indeksów do rozwiązania obu predykatów — ten wydajny plan jest przedstawiony na rysunku 10.2.

Złe plany mogą powstać, jeżeli filtry dla operatora OR dotyczą różnych tabel, czyli zapytanie jest zgodne z wzorcem WHERE a.col1 = @val1 OR b.col2 = @val2. Uruchomienie dwóch poniższych zapytań z wykorzystaniem bardzo selektywnego predykatu



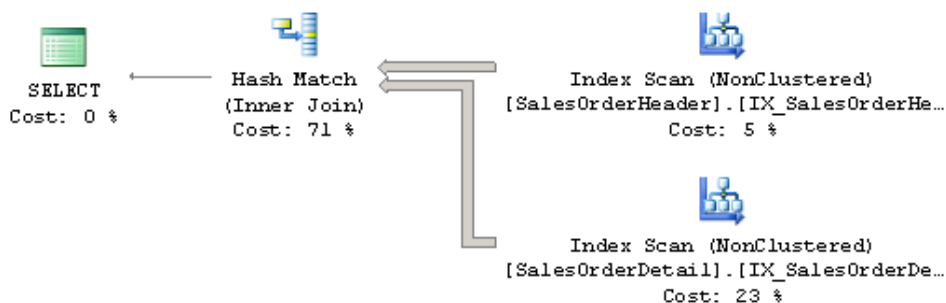
Rysunek 10.2. Plan dla WHERE a.col1 = @val1 OR a.col2 = @val2

stworzy dwa plany używające efektywnych operatorów *Index Seek* dla indeksów IX_SalesOrderDetail_ProductID i IX_SalesOrderHeader_CustomerID, zwracające odpowiednio dwa i trzy rekordy:

```
SELECT SalesOrderID FROM Sales.SalesOrderDetail
WHERE ProductID = 897
SELECT SalesOrderID FROM Sales.SalesOrderHeader
WHERE CustomerID = 11020
```

Jeżeli jednak złączymy obie tabele za pomocą tych samych selektywnych predykatów (tylko dla celów testowych), SQL Server, zamiast skorzystać z bardziej wydajnych operacji wyszukiwania, zwróci bardzo kosztowny plan skanujący wspomniane indeksy (zobacz rysunek 10.3):

```
SELECT sod.SalesOrderID FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
ON soh.SalesOrderID = sod.SalesOrderID
WHERE sod.ProductID = 897 OR soh.CustomerID = 11020
```

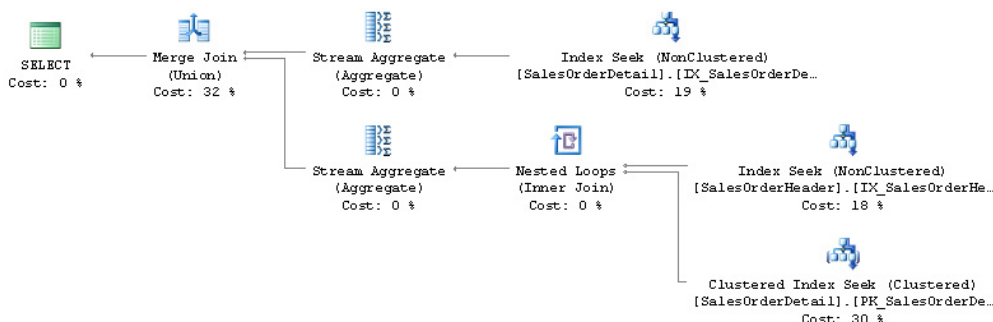


Rysunek 10.3. Plan dla WHERE a.col1 = @val1 OR b.col2 = @val2

Tego typu problem można rozwiązać, stosując klauzulę UNION zamiast warunku OR, zgodnie z poniższym zapytaniem. Zwrócone zostaną te same rekordy, lecz teraz zostanie stworzony bardziej wydajny plan z wykorzystaniem przeszukiwania indeksów.

```
SELECT sod.SalesOrderID FROM Sales.SalesOrderHeader soh
      JOIN Sales.SalesOrderDetail sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE sod.ProductID = 897
UNION
SELECT sod.SalesOrderID FROM Sales.SalesOrderHeader soh
      JOIN Sales.SalesOrderDetail sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = 11020
```

Chociaż zapytanie wygląda na bardziej skomplikowane i nadmiarowe, jego plan, przedstawiony na rysunku 10.4, będzie bardzo efektywny.



Rysunek 10.4. Plan z wykorzystaniem klauzuli UNION

Złączenia i zagregowane zbiory danych

Drugi wzorzec zapytań ma miejsce, kiedy rezultaty agregacji są łączone w dużych i skomplikowanych zapytaniach. Jak pamiętasz z rozdziału 6., statystyki mogą zapewnić dobry szacunek kardynalności dla operacji wykonywanych na tabeli, której dotyczy statystyka (jak na przykład podczas szacowania liczby rekordów zwracanych przez operację filtra). Optymalizator zapytań musi jednak wykorzystać ten szacunek w operacji na wczesnym etapie planu, wykonać szacunek dla kolejnej operacji i otrzymać nowy wynik, i tak dalej. W niektórych skomplikowanych planach jakość tych szacunków może ulec szybkiej degradacji.

Spoglądając na taki plan, możesz zauważyć, że na początku przepływu danych szacowane i rzeczywiste liczby rekordów są bardzo zbliżone, ale po złączeniu pośrednich wyników agregacji z innymi pośrednimi wynikami agregacji jakość szacunku kardynalności ulega zmniejszeniu. Oczywiście, kiedy optymalizator nie jest w stanie uzyskać dobrych szacunków na temat rozmiaru zestawu danych, może podejmować nieoptymalne decyzje dotyczące złączeń, kolejności złączeń czy też innych operacji w planie.

Rozwiązaniem problemu jest rozbicie tego typu zapytań. Możesz podzielić zapytania na podstawie planu, znajdując miejsca, w których różnice w szacowanej i rzeczywistej liczbie rekordów są duże. Ponadto każda agregacja w skomplikowanym zapytaniu może zostać zapisana w tabeli tymczasowej, co pozwoli stworzyć lepsze statystyki

niż te dla pozostałych części zapytania. Zapytanie z dużą liczbą bardzo skomplikowanych złączeń to kolejny wzorzec, który może skorzystać na rozbiciu go na dwa lub więcej prostszych zapytań i zapisaniu wyników pośrednich w tabeli tymczasowej. Problem dotyczący zapytań z dużą liczbą złączeń wyjaśniam dokładniej w podrozdziale „Kolejność złączeń”.

Podpowiedzi

SQL to język deklaratywny — definiuje tylko, jakie dane pobrać z bazy danych, i nie opisuje sposobu pozyskania tych danych. To, jak wiemy, jest zadanie optymalizatora zapytań, który analizuje pewną liczbę potencjalnych planów wykonania dla danego zapytania, szacuje koszt każdego z tych planów i na podstawie szacowanych kosztów wybiera najbardziej efektywny spośród analizowanych planów.

Mogą się jednak zdarzyć sytuacje, w których wybrany plan nie jest tak wydajny, jak można by się tego spodziewać, i w ramach procesu wyszukiwania rozwiązań możesz sam spróbować znaleźć lepszy plan. Ale zanim zaczniesz go szukać, pamiętaj, że znalezienie lepszego planu nie zawsze jest możliwe. Plan może być wydajny, ale zapytanie może być bardzo kosztowne lub system może mieć problemy wydajnościowe, które wpływają na wykonanie zadania.

Chociaż optymalizator doskonale wywiązuje się ze swoich obowiązków, czasami, jak widziałeś w tej książce, nie udaje mu się wyprodukować wydajnego planu. Nawet jednak w przypadkach, kiedy nie otrzymujesz wydajnego planu, powinieneś i tak rozróżnić sytuacje, gdy pojawiają się problemy, ponieważ nie zapewniasz optymalizatorowi wszystkich informacji, których wymaga do wykonania zadania, i sytuacje, gdy problemy spowodowane są ograniczeniami optymalizatora. Dotychczas w tej książce starałem się skupić na sposobach zapewnienia optymalizatorowi informacji potrzebnych do określenia wydajnego planu, takich jak odpowiednie indeksy i dobra jakość statystyk, oraz na tym, jak szukać problemów w sytuacjach, kiedy nie otrzymujesz dobrego planu. W tym rozdziale opisuję, co zrobić, kiedy natrafisz na ograniczenie optymalizatora zapytań.

Mogą zdarzyć się sytuacje, w których optymalizator po prostu się myli i z tego powodu możemy być zmuszeni zastosować podpowiedzi. Podpowiedzi to dyrektywy optymalizatora, które pozwolą nam bezpośrednio kontrolować plan wykonania dla danego zapytania w celu poprawy jego wydajności. Sięgając po podpowiedź, rezygnujemy z deklaratywnego charakteru języka SQL i przekazujemy bezpośrednie polecenia do optymalizatora. Wymuszanie określonego zachowania optymalizatora jest ryzykowne; podpowiedzi należy stosować ostrożnie i tylko jako ostateczność, kiedy wyczerpiemy już wszystkie inne opcje w dążeniu do właściwego planu.

Mając to ostrzeżenie na uwadze, omówimy teraz kilka spośród podpowiedzi dostępnych w SQL Serverze, dzięki czemu jeżeli pojawi się taka konieczność, będziesz wiedzieć, jak i kiedy stosować podpowiedzi. Skupimy się tylko na tych podpowiedziach,

które w mojej praktyce najczęściej dawały pozytywne wyniki w pewnych sytuacjach. Kilka innych podpowiedzi, takich jak `OPTIMIZE FOR`, `OPTIMIZE FOR UNKNOWN` i `RECOMPILE`, zostało już omówione w rozdziale 8. i nie będę do nich wracał.

Kiedy korzystać z podpowiedzi?

Podpowiedzi to potężne narzędzie pozwalające wymusić konkretne zachowanie optymalizatora zapytań. Korzystając z nich, powinienś bardzo uważać, ponieważ ograniczają wybór optymalizatora. Sprawiają również, że kod jest mniej elastyczny i wymaga dodatkowej konserwacji. Korzystaj z podpowiedzi tylko wówczas, gdy masz pewność, że nie masz innego wyboru. Zanim sięgniesz po podpowiedź, rozważ przynajmniej poniższe potencjalne problemy:

- ▶ **Problemy z systemem.** Upewnij się, że Twoje problemy nie są spowodowane problemami z systemem, takimi jak blokady czy wąskie gardła w zasobach serwera, np. I/O, pamięci czy CPU.
- ▶ **Błędy w szacowaniu kardynalności.** Optymalizator często nie wybiera poprawnego planu z powodu błędów w szacowaniu kardynalności. Błędy te czasami można naprawić na przykład poprzez aktualizację statystyk czy wykorzystanie większej próbki dla statystyk (lub skanowanie całej tabeli), kolumn wyliczeniowych, statystyk wielokolumnowych lub statystyk filtrowanych. W niektórych przypadkach błędy w szacunku kardynalności mogą być spowodowane wykorzystaniem funkcjonalności, dla których statystyki nie są wspierane, na przykład zmiennych tabelarycznych lub funkcji użytkownika zwracających zmienne tabelaryczne. W takich przypadkach, jeżeli nie otrzymujesz wydajnego planu, rozważ zastosowanie standardowych lub tymczasowych tabel. Statystyki i błędy w szacowaniu kardynalności omówiłem dokładnie w rozdziale 6.
- ▶ **Dodatkowa kontrola.** Być może będziesz musiał skontrolować jeszcze kilka innych mechanizmów, zanim wykorzystasz podpowiedzi. Jedną z oczywistych rzeczy podczas poprawiania wydajności zapytań jest zapewnienie optymalizatorowi odpowiednich indeksów. To, jak upewnić się, czy optymalizator używa indeksów, omówiłem w rozdziale 5. Możesz również rozważyć inne, mniej oczywiste procesy, takie jak rozbięcie zapytania na kroki lub mniejsze części i przechowywanie wyników pośrednich w tabelach tymczasowych, co pokazałem we wcześniejszej części tego rozdziału. Możesz wykorzystać tę metodę także w procesie szukania problemów, na przykład aby znaleźć najbardziej kosztowną część oryginalnego pierwotnego planu i skupić się na niej. Jeżeli jednak zapytanie w wersji podzielonej będzie wykazywało lepszą wydajność, możesz też zachować je jako wersję ostateczną.

Jak już pisałem we wcześniejszej części tego rozdziału, optymalizatory zapytań poprawiły się znacznie w ciągu ponad 30 lat badań, ale ciągle zmagają się z pewnymi

problemami. Optymalizator zapytań SQL Servera stworzy efektywny plan wykonania dla większości zapytań, ale będzie to coraz trudniejsze wraz ze wzrostem skomplikowania zapytań i liczby złączanych tabel, a także wykorzystaniem agregacji i innych funkcjonalności SQL Server. Jeżeli po sprawdzeniu przedstawionych powyżej i w całej książce zaleceń optymalizator wciąż nie znajduje dla Twojego zapytania odpowiedniego planu, możesz rozważyć użycie podpowiedzi, aby pokierować optymalizatorem w odpowiednią stronę.

Pamiętaj, że korzystając z podpowiedzi, efektywnie włączasz niektóre z dostępnych reguł transformacji, a tym samym ograniczasz dostępny obszar poszukiwań. Wykonane zostaną tylko reguły transformacji, które mogą pomóc w osiągnięciu zamierzonego planu. Na przykład, jeżeli wykorzystasz podpowiedzi, aby wymusić konkretną kolejność złączeń, optymalizator wyłączy reguły, które zmieniają kolejność. Zawsze staraj się stosować najmniej restrykcyjne podpowiedzi, ponieważ pozwolą one zachować większą elastyczność, co z kolei ułatwi późniejszą konserwację. Ponadto podpowiedzi nie można wykorzystać do generowania niepoprawnego planu lub planu, który nie byłby brany pod uwagę podczas normalnego procesu optymalizacji.

Co więcej, podpowiedź, która pierwotnie sprawdza się doskonale, może znacznie pogorszyć wydajność w późniejszym czasie, jeżeli pewne warunki ulegną zmianie — na przykład w rezultacie aktualizacji schematu, aktualizacji i nowych wersji SQL Servera lub nawet zmian w danych. Podpowiedzi mogą nie pozwolić optymalizatorowi dostosować planu do zmian, co spowoduje obniżoną wydajność zapytania. Twoim zadaniem jest monitorowanie i sprawdzanie zapytań z podpowiedziami, aby mieć pewność, że po zmianach również spełnią swoją rolę.

Pamiętaj również, że jeżeli zdecydujesz się na wykorzystanie podpowiedzi, aby zmienić część planu lub operator w planie, to po zastosowaniu podpowiedzi optymalizator wykona optymalizację od nowa. Będzie posłuszny podpowiedziom podczas procesu optymalizacji, ale wciąż będzie w stanie zmieniać pozostałe elementy planu, więc w rezultacie mogą zostać wprowadzone niepożądane zmiany w pozostałych częściach planu. Zauważ też, że to, iż zapytanie nie jest tak wydajne, jak byś chciał, nie oznacza wcale, że otrzymałeś zły plan. Jeżeli wykonywana przez Ciebie operacja jest kosztowna i pochłania wiele zasobów, możliwe jest, że żadne zabiegi nie pomogą w osiągnięciu pożądanej wydajności.



UWAGA

Od wersji SQL Server 2005 SP1 dostępna jest flaga 2301, którą możesz wykorzystać do włączenia zaawansowanych optymalizacji odpowiednich dla zapytań wspierających podejmowanie decyzji dla dużych zbiorów danych. Więcej szczegółów oraz listę scenariuszy, w których flaga może być użyteczna, znajdziesz pod adresem <http://blogs.msdn.com/b/ianjo/archive/2006/04/24/582219.aspx>.

Rodzaje podpowiedzi

SQL Server udostępnia szerokie spektrum podpowiedzi, które można sklasyfikować następująco:

- ▶ Podpowiedzi zapytań mówią optymalizatorowi, aby zaaplikował podpowiedź dla całego zapytania. Wykorzystują klauzulę `OPTION`, która jest umiejscawiana na końcu zapytania.
- ▶ Podpowiedzi złączeń odnoszą się do konkretnego złączenia i mogą być specyfikowane z użyciem podpowiedzi złączeń w stylu ANSI.
- ▶ Podpowiedzi tabel odnoszą się do pojedynczej tabeli i są zazwyczaj specyfikowane za pomocą słowa kluczowego `WITH` w klauzuli `WHERE`.

Kolejną użyteczną klasyfikacją jest podzielenie podpowiedzi na podpowiedzi operatorów fizycznych i podpowiedzi zorientowane na cel:

- ▶ Podpowiedzi operatorów fizycznych, jak sugeruje nazwa, pozwalają zażądać wykorzystania konkretnego operatora, kolejności złączeń lub umiejscowienia agregacji. Większość podpowiedzi omawianych w tym rozdziale to podpowiedzi operatorów fizycznych.
- ▶ Podpowiedź zorientowana na cel nie specyfikuje, jak zbudować plan, ale zamiast tego specyfikuje, jaki cel ma zostać osiągnięty, pozostawiając w gestii optymalizatora znalezienie najlepszych operatorów fizycznych do osiągnięcia tego celu. Podpowiedzi zorientowane na cel są zazwyczaj bezpieczniejsze i wymagają mniej wiedzy na temat wewnętrznej budowy optymalizatora zapytań. Przykłady podpowiedzi zorientowanych na cel to `OPTIMIZE FOR` i `FAST N`.

Podpowiedzi blokujące nie wpływają na wybór planu, dlatego nie będą tutaj omawiane. W dalszej części opisałem też wskazówki planu, które pozwalają zaaplikować podpowiedź bez konieczności zmiany kodu aplikacji, i podpowiedź `USE PLAN`, która daje możliwość zmuszenia optymalizatora do wykorzystania konkretnego planu wykonania dla zapytania.

W następnych kilku podrozdziałach omawiam podpowiedzi wpływające na złączenia, kolejność złączeń, agregacje, skanowanie i przeszukiwanie indeksów, widoki i tak dalej.



UWAGA

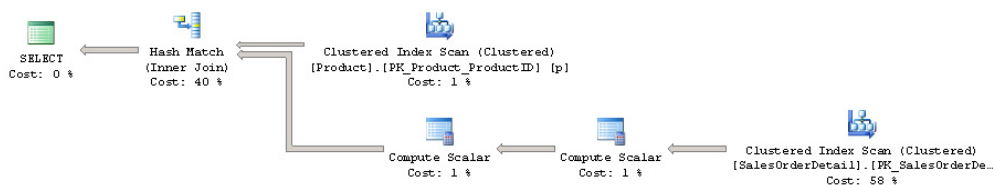
W przypadku prostej i niewielkiej bazy danych, takiej jak `AdventureWorks2012`, SQL Server dla przykładów z tego rozdziału stworzy odpowiedni plan najprawdopodobniej bez konieczności stosowania podpowiedzi. Na potrzeby demonstracji będziemy jednak sprawdzać plany alternatywne, przy czym niektóre z nich mogą być bardziej kosztowne niż wersje bez podpowiedzi.

Złączenia

Możemy bezpośrednio zażądać, aby optymalizator wykorzystał jeden z dostępnych algorytmów złączeń: *Nested Loops Join*, *Merge Join* lub *Hash Join*. Możemy to zrobić na poziomie zapytania i w tym przypadku wszystkie złączenia zapytania zostaną podporządkowane lub na poziomie pojedynczego złączenia, wpływając tylko na to jedno złączenie (choć, jak zobaczymy w dalszej części, druga opcja ma również wpływ na kolejność złączeń).

Najpierw skupmy się na podpowiedziach złączeń na poziomie zapytania, w tej wersji algorytm specyfikowany jest za pomocą klauzuli `OPTION`. Możesz też wyspecyfikować dwa z trzech dostępnych złączeń, co w efekcie spowoduje wykluczenie trzeciego algorytmu z puli dostępnej dla optymalizatora. Decyzja dotycząca wyboru jednego z pozostałych dwóch operatorów zostanie podjęta na podstawie kosztów. Na przykład poniższe zapytanie bez podpowiedzi stworzy plan z rysunku 10.5, który korzysta ze złączenia *Hash Join*:

```
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
```

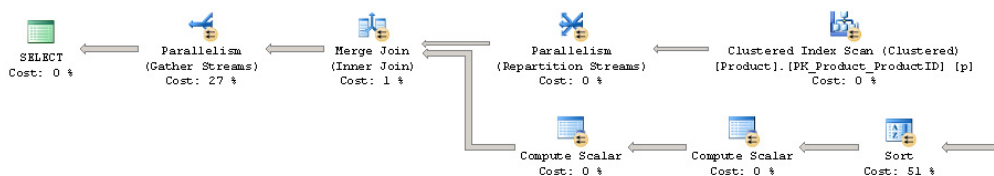


Rysunek 10.5. Plan wykorzystujący operator Hash Join

Natomiast poniższe zapytanie zażąda, aby optymalizator wyłączył operator *Hash Join* i wykorzystał *Nested Loops Join* lub *Merge Join*. W tym przypadku SQL Server wybierze bardziej kosztowny plan z operatorem *Merge Join*, co pokazuje również, że pierwotna decyzja optymalizatora była poprawna. Część planu jest widoczna na rysunku 10.6.

```
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
OPTION (LOOP JOIN, MERGE JOIN)
```

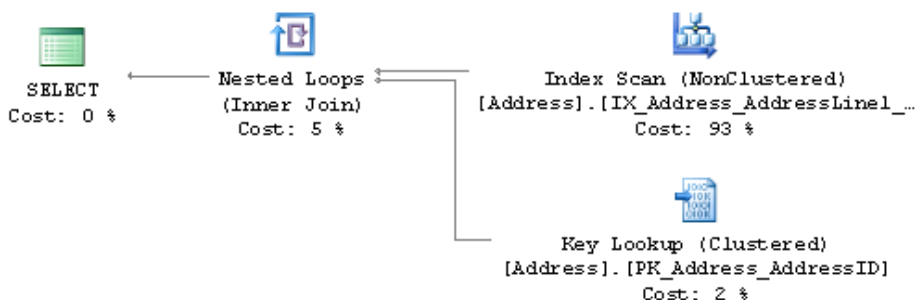
Podpowiedzi złączeń nie tylko mogą wymusić złączenia, które jawnie zdefiniujemy w tekście zapytania, mogą również wpływać na większość złączeń wprowadzanych przez optymalizator, na przykład na walidację kluczy obcych czy walidację kaskadową. Inne złączenia, takie jak *Nested Loops* używane w wyszukiwaniu zaznaczeń, nie mogą być zmieniane, ponieważ byłoby to wbrew pierwotnemu celowi wyszukiwania zaznaczeń.



Rysunek 10.6. Plan wykonania z wyłączonym operatorem Hash Join

Na przykład poniższa podpowiedź, aby wykorzystany został operator *Merge Join*, zostanie zignorowana, zgodnie z planem z rysunku 10.7:

```
SELECT AddressID, City, StateProvinceID, ModifiedDate
FROM Person.Address
WHERE City = 'Santa Fe'
OPTION (MERGE JOIN)
```



Rysunek 10.7. Podpowiedź zignorowana w planie z wyszukiwaniem zaznaceń

Jak już wspomniałem, podpowiedzi nie mogą zmusić optymalizatora, aby wygenerował niepoprawny plan, dlatego poniższe zapytanie nie zostanie skompilowane, ponieważ złączenia *Merge* i *Hash Join* wymagają operatora równości w predykcji złączenia:

```
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON sod.ProductID > p.ProductID
WHERE p.ProductID > 900
OPTION (HASH JOIN)
```

Próba uruchomienia tego zapytania zakończy się zwróceniem poniższego komunikatu:

Msg 8622, Level 16, State 1, Line 1

Query processor could not produce a query plan because of the hints defined in this query.

↳ Resubmit the query without specifying any hints and without using SET FORCEPLAN. //Procesor

↳ zapytań nie mógł stworzyć planu zapytania z powodu podpowiedzi zdefiniowanych w zapytaniu. Spróbuj ponownie

↳ wykonać zapytanie bez żadnych podpowiedzi i bez użycia SET FORCEPLAN.

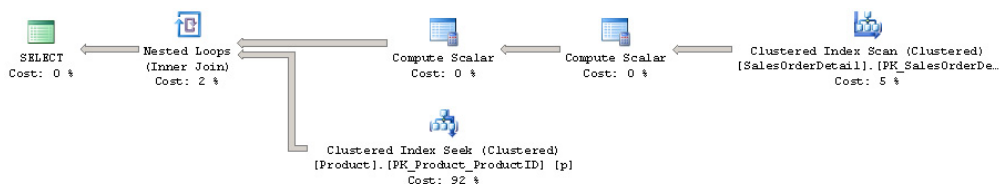
Pamiętaj, że zastosowanie podpowiedzi zapytania będzie oddziaływało na całe zapytanie. Jeżeli potrzebujesz kontroli nad pojedynczym złączeniem, możesz zastosować podpowiedzi złączeń w stylu ANSI, których zaletą jest to, że dla każdego złączenia w zapytaniu możemy indywidualnie wybrać operator złączenia. Złączenia w stylu ANSI dodadzą jednak automatycznie zachowanie podpowiedzi *FORCE ORDER*, która wymusza utrzymanie kolejności złączeń i umiejscowienia agregacji zgodnej ze składnią zapytania. Zachowanie to zostanie wyjaśnione w podrozdziale „*FORCE ORDER*” w dalszej części tego rozdziału.

Tymczasem pozwól, że pokażę przykład. Pierwsze zapytanie przedstawione w tym podrozdziale, bez żadnych podpowiedzi, wykorzystuje operator *Hash Join* uzyskujący dostęp do tabeli *Product*, co oznacza, że tabela *Product* będzie tabelą budowania, a tabela *SalesOrderDetail* będzie wsadem sondowanym. Ten plan został pokazany na rysunku 10.5. Poniższe zapytanie skorzysta z podpowiedzi wymuszającej wykorzystanie operatora *Nested Loops Join*. Zauważ, że tym razem słowo kluczowe *INNER* jest wymagane:

```
SELECT *
FROM Production.Product AS p
INNER LOOP JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
```

Jak jednak wspomniałem wcześniej, podpowieź będzie miała również wpływ na kolejność złączeń: tabela *Product* zawsze będzie tabelą zewnętrzną, a *SalesOrderDetail* wewnętrzną. Gdybyśmy chcieli odwrócić role, musielibyśmy zmienić zapytanie zgodnie z poniższym, w którym *SalesOrderDetail* będzie wsadem zewnętrznym, a *Product* wewnętrznym, co widać na planie z rysunku 10.8.

```
SELECT *
FROM Sales.SalesOrderDetail AS sod
INNER LOOP JOIN Production.Product AS p
ON p.ProductID = sod.ProductID
```



Rysunek 10.8. Wykonanie planu ze złączeniami w stylu ANSI

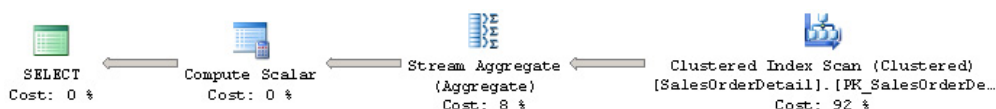
Ponadto podczas wykonywania kodu w Management Studio w zakładce *Messages* pojawi się poniższe ostrzeżenie, wskazujące, że wymuszony został nie tylko algorytm złączenia, ale również kolejność złączenia (czyli tabele zostały złączone dokładnie w takiej kolejności, w jakiej zostały zapisane w zapytaniu):

Warning: The join order has been enforced because a local join hint is used.

Agregacje

Podobnie jak w przypadku algorytmów złączeń, algorytmy agregacji także można wymusić za pomocą podpowiedzi GROUP. Konkretnie podpowiedź ORDER GROUP pozwala zażądać, aby optymalizator wykorzystał operator *Stream Aggregate*, a podpowiedź HASH GROUP żąda wykorzystania operatora *Hash Aggregate*. Aby zobaczyć efekty tych podpowiedzi, spójrz na poniższe zapytanie bez podpowiedzi, które stworzy plan z rysunku 10.9 wykorzystujący operator *Stream Aggregate*:

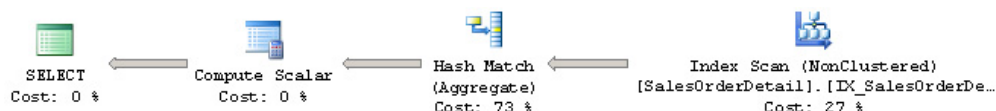
```
SELECT SalesOrderID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
```



Rysunek 10.9. Zapytanie wykorzystujące operator Stream Aggregate

Ponieważ tabela `SalesOrderDetail` ma indeks klastrowy na kolumnie `SalesOrderID` i tym samym dane są już posortowane po kolumnie z klauzuli GROUP BY, wykorzystanie operatora *Stream Aggregate* jest oczywistym wyborem. Jeżeli jednak do poprzedniego zapytania dodamy podpowiedź HASH GROUP, zgodnie z poniższym, wymuszone zostanie użycie operatora *Hash Aggregate* i powstanie pokazany na rysunku 10.10 plan, który jest bardziej kosztowny niż to konieczne:

```
SELECT SalesOrderID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
OPTION (HASH GROUP)
```



Rysunek 10.10. Plan wykonania z podpowiedzią HASH GROUP

Natomiast agregacja skalarna zawsze zastosuje operator *Stream Aggregate*. Podpowiedź wymuszająca operator *Hash Aggregate* dla agregacji skalarnej, jak w poniższym zapytaniu, zostanie zignorowana w SQL Serverze 2014 i SQL Serverze 2012. W wersji SQL Server 2008 R2 i wcześniejszych jednak zwrócony zostanie wcześniej błąd kompilacji 8622:

```
SELECT COUNT(*) FROM Sales.SalesOrderDetail
OPTION (HASH GROUP)
```

FORCE ORDER

Podpowiedź `FORCE ORDER` daje użytkownikowi pełną kontrolę nad umiejscowieniem złączeń i agregacji w planie. Prosi ona optymalizator, aby zachował kolejność złączeń i umiejscowienie agregacji zgodne ze składnią zapytania. Zauważ, że pokazane wcześniej podpowiedzi w stylu ANSI, oprócz kontroli nad algorytmem złączenia, dają również kontrolę nad kolejnością złączeń. Zarówno `FORCE ORDER`, jak i podpowiedzi złączeń w stylu ANSI są potężnymi narzędziami i z tego powodu należy z nich korzystać bardzo ostrożnie. Jak już wyjaśniałem w tym rozdziale, znalezienie optymalnej kolejności złączeń jest kluczową i trudną częścią procesu optymalizacji zapytań, ponieważ liczba możliwych kombinacji może być ogromna, nawet dla zapytań korzystających z zaledwie kilku tabel. W ostatecznym rozrachunku, korzystając z podpowiedzi `FORCE ORDER`, próbujesz samodzielnie zoptymalizować kolejność złączeń.

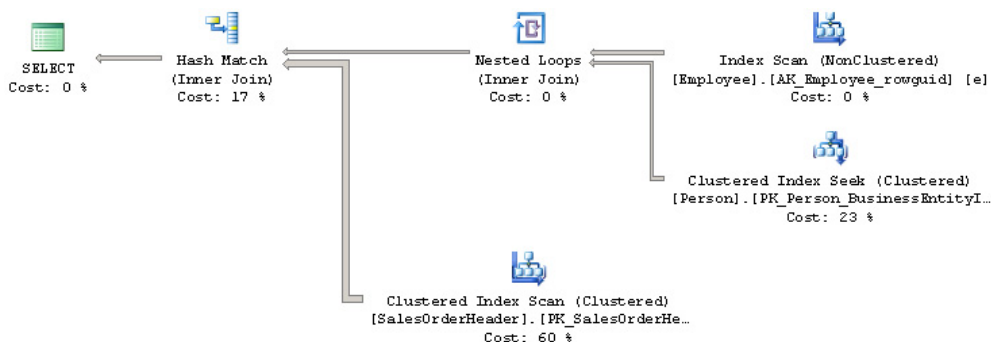
Możesz skorzystać z `FORCE ORDER`, aby uzyskać dowolną formę zapytania, czyli drzewo lewostronnie rozgałęzione, prawostronnie rozgałęzione lub krzaczaste. Optymalizator SQL Servera tworzy zazwyczaj drzewo lewostronnie rozgałęzione, możesz jednak wymusić stworzenie drzewa krzaczastego lub prawostronnie rozgałęzionego, zmieniając lokalizację klauzuli `ON` w predykcji złączenia, korzystając z podzapytań lub nawiasów i tak dalej. Wymuszanie kolejności złączeń nie wpływa na fazę upraszczania podczas optymalizacji, a niektóre złączenia i tak mogą zostać usunięte (zobacz rozdział 3.).

Jeżeli z jakiegoś powodu musisz zmienić kolejność złączeń w zapytaniu, możesz zacząć od kolejności rekomendowanej przez optymalizator i zmienić tylko tę część, która według Ciebie powoduje problemy takie jak błędy szacowania kardynalności. Możesz również postępować zgodnie z zasadami, według których działa optymalizator (zobacz rozdziałem 4.). Na przykład jeżeli wymuszasz złączenie *Hash Join*, to jako wsad budowania wybierz najmniejszą tabelę, a jeżeli wymuszasz złączenie *Nested Loops*, używaj małych tabel jako wsad zewnętrzny, a tabel z indeksami jako wsad wewnętrzny. Możesz też rozpocząć od łączenia małych tabel lub tabel, które pomogą odfiltrować jak największą liczbę rekordów.

Oto przykład. Poniższe zapytanie bez podpowiedzi pokaże plan z rysunku 10.11:

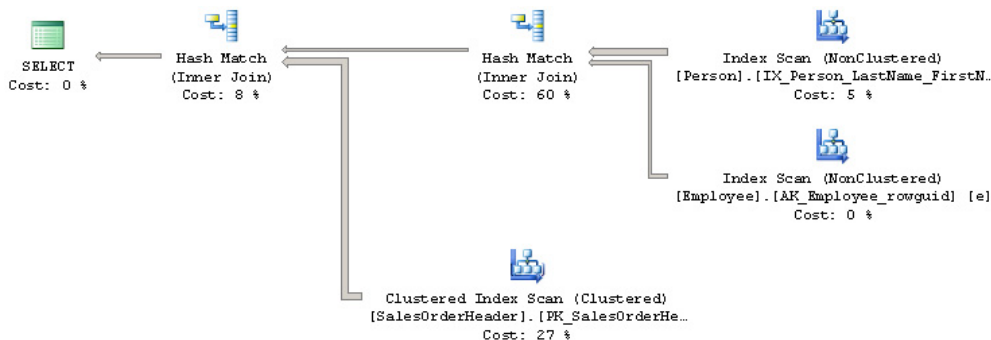
```
SELECT LastName, FirstName, soh.SalesOrderID
FROM Person.Person p JOIN HumanResources.Employee e
  ON e.BusinessEntityID = p.BusinessEntityID
JOIN Sales.SalesOrderHeader soh
  ON p.BusinessEntityID = soh.SalesPersonID
WHERE ShipDate > '2008-01-01'
```

Jak widzisz, optymalizator nie zachowuje kolejności złączeń przedstawionej w składni zapytania (na przykład tabela `Person` nie jest pierwszą czytaną tabelą). Na podstawie kosztów optymalizator znalazł bardziej wydajną kolejność złączeń. Zobaczmy teraz, co się stanie, jeżeli dodamy do zapytania podpowiedź `FORCE ORDER`. Stworzony zostanie plan z rysunku 10.12.



Rysunek 10.11. Plan wykonania bez podpowiedzi

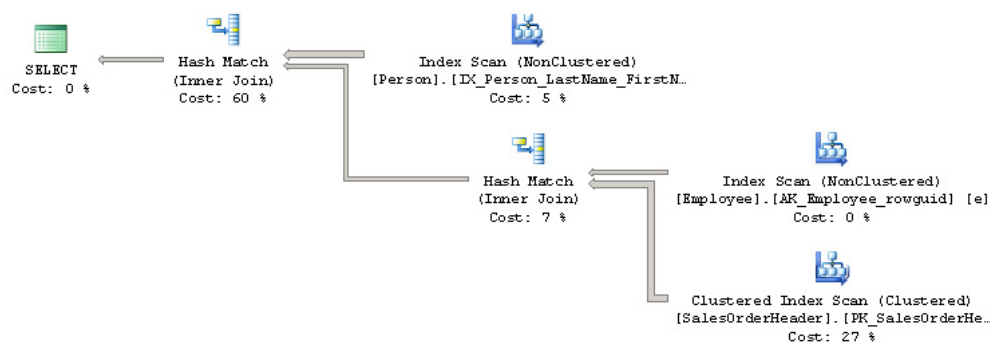
```
SELECT LastName, FirstName, soh.SalesOrderID
FROM Person.Person p JOIN HumanResources.Employee e
  ON e.BusinessEntityID = p.BusinessEntityID
JOIN Sales.SalesOrderHeader soh
  ON p.BusinessEntityID = soh.SalesPersonID
WHERE ShipDate > '2008-01-01'
OPTION (FORCE ORDER)
```



Rysunek 10.12. Plan wykonania z podpowiedzią FORCE ORDER

W tym zapytaniu wykorzystującym podpowiedź `FORCE ORDER` tabele zostaną złączone w kolejności, w jakiej zostały zapisane w zapytaniu, i domyślnie stworzone zostanie drzewo lewostronnie rozgałęzione. Gdybyś jednak użył podpowiedzi `FORCE ORDER` w zapytaniu wykorzystującym złączenia ANSI, SQL Server wykorzystałby położenie klauzuli `ON` do zdefiniowania położenia złączeń. Możesz skorzystać z tej techniki do stworzenia drzewa krzaczastego, jak w poniższym przykładzie, który pokaże plan z rysunku 10.13:

```
SELECT LastName, FirstName, soh.SalesOrderID
FROM Person.Person p JOIN HumanResources.Employee e
JOIN Sales.SalesOrderHeader soh
  ON e.BusinessEntityID = soh.SalesPersonID
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE ShipDate > '2008-01-01'
OPTION (FORCE ORDER)
```

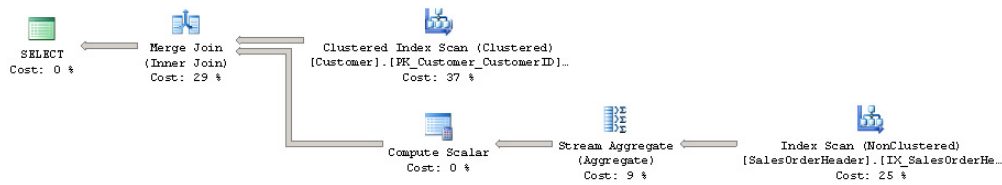


Rysunek 10.13. Plan wymuszający zastosowania drzewa krzaczastego

Chociaż kolejność złączeń tabel została wyspecyfikowana jako Person, Employee i SalesOrderHeader, umiejscowienie klauzuli ON definiuje, że Employee i SalesOrderHeader powinny być połączone jako pierwsze, tworząc tym samym drzewo prawostronnie rozgałęzione.

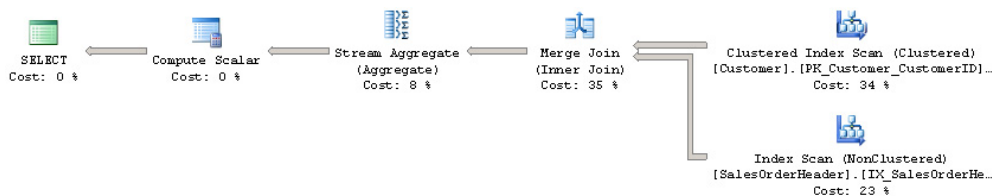
Oprócz kontroli nad kolejnością złączeń, jak wspomniałem we wstępie do tego podrozdziału, podpowiedź `FORCE ORDER` można zastosować do wymuszenia kolejności agregacji. Rozważmy poniższy przykład bez podpowiedzi, który stworzy plan z rysunku 10.14:

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c
JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
```



Rysunek 10.14. Plan z agregacją przed złączeniem

Jak widzisz, w tym przypadku optymalizator zdecydował się wykonać agregację przed złączeniem. Pamiętaj, że jak wspomniałem w rozdziałach 3. i 4., optymalizator może wykonać agregację przed złączeniem lub po złączeniu, jeżeli pozwoli to poprawić wydajność zapytania. Dodając podpowiedź `FORCE ORDER`, jak w poniższym zapytaniu, możesz spowodować, że agregacja zostanie wykonana po złączeniu (zobacz rysunek 10.15):



Rysunek 10.15. Podpowiedź FORCE PLAN wykorzystana dla podpowiedzi

```
SELECT c.CustomerID, COUNT(*)
FROM Sales.Customer c
JOIN Sales.SalesOrderHeader o
ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID
OPTION (FORCE ORDER)
```

Podobne polecenie, `SET FORCEPLAN`, również można wykorzystać do zachowania kolejności złączeń zgodnej z klauzulą `FROM` zapytania. To polecenie spowoduje jednak zastosowanie tylko operatorów *Nested Loops Joins*, chyba że do stworzenia planu będą niezbędne inne typy złączeń lub zapytanie zawiera inne podpowiedzi zapytań bądź złączeń. Różnicą między tym poleceniem a podpowiedziami pokazanymi do tej pory jest to, że `SET FORCEPLAN` musi zostać włączone i będzie efektywne, dopóki nie zostanie wyłączone. Więcej informacji na ten temat znajdziesz w dokumentacji Books Online.

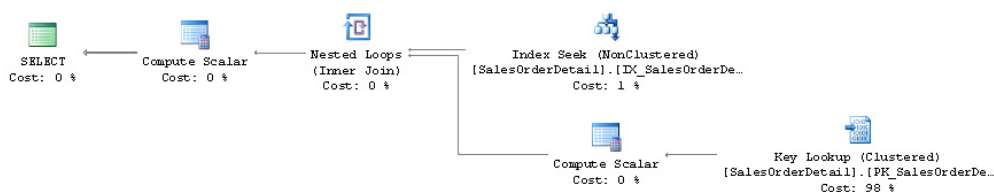
INDEX, FORCESCAN i FORCESEEK

Podpowiedzi `INDEX`, `FORCESCAN` i `FORCESEEK` to podpowiedzi tabel i omówimy je po kolei. Podpowiedzi `INDEX` można użyć do zażądania, aby optymalizator wykorzystał konkretny indeks lub indeksy (przykład jego zastosowania został pokazany w dyskusji na temat indeksów magazynu kolumn w rozdziale 9.). Można użyć identyfikatora lub nazwy indeksu, zalecane jest jednak użycie nazwy, ponieważ nie mamy wpływu na identyfikatory indeksów nieklastrowych. Jeżeli jednak chcesz korzystać z identyfikatorów, znajdziesz je w kolumnie `index_id` widoku `sys.indexes`, gdzie 0 oznacza stertę, 1 oznacza indeks klastrowy, a wartość większa od 1 to indeks nieklastrowy. Dla zapytania korzystającego ze sterty użycie podpowiedzi `INDEX(0)` spowoduje zastosowanie operatora *Table Scan*, natomiast `INDEX(1)` zwróci komunikat o błędzie informujący, że taki indeks nie istnieje. Tabela z indeksem klastrowym zaś pozwoli na obie wartości: `INDEX(0)` wymusi *Clustered Index Scan*, a `INDEX(1)` — *Clustered Index Scan* lub *Clustered Index Seek*. Podpowiedź `FORCESCAN` wymaga, aby optymalizator używał tylko operacji skanowania indeksów jako metody dostępu do danej tabeli lub widoku, i może być stosowana z podpowiedzią `INDEX` lub bez niej. Natomiast podpowiedź `FORCESEEK` może być użyta do wymuszenia na optymalizatorze wykorzystania operacji *Index Seek* i można ją stosować dla indeksów klastrowych i nieklastrowych. Jak zobaczysz w dalszej części, może również działać w połączeniu z podpowiedzią `INDEX`.

Oprócz pomocy w poprawianiu wydajności zapytań możesz w niektórych przypadkach rozważyć zastosowanie podpowiedzi dla indeksów, aby obniżyć częstotliwość występowania blokad. Zauważ, że kiedy korzystasz z podpowiedzi INDEX, zapytanie staje się zależne od wyspecyfikowanego indeksu i przestanie działać, jeżeli ten indeks zostanie usunięty. Użycie FORCESEEK bez dostępnego indeksu spowoduje powstanie błędu, co pokażę w dalszej części tego podrozdziału.

Możesz również skorzystać z podpowiedzi INDEX, aby uniknąć operacji wyszukiwania zaznaczeń, jak w poniższym przykładzie. Ponieważ optymalizator szacuje, że w następnym zapytaniu zwrócone zostanie tylko kilka rekordów, zdecyduje o zastosowaniu kombinacji operatorów *Index Seek* i *Key Lookup* (zobacz rysunek 10.16).

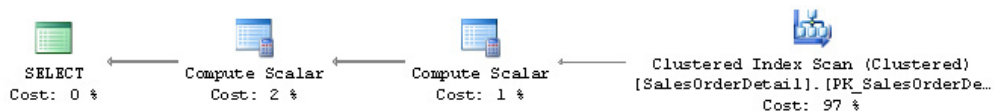
```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 897
```



Rysunek 10.16. Plan ze złączeniem Nested Loops Join

Załóżmy jednak, że chcesz uniknąć operacji wyszukiwania zaznaczeń; możesz wykorzystać podpowiedź INDEX, aby wymusić skanowanie tabeli, co spowoduje wykorzystanie sterty lub indeksu klastrowego. Poniższe zapytanie wymusza użycie operatora *Clustered Index Scan* (zobacz rysunek 10.17):

```
SELECT * FROM Sales.SalesOrderDetail WITH (INDEX(0))
WHERE ProductID = 897
```



Rysunek 10.17. Plan z podpowiedzią INDEX

To samo zachowanie można uzyskać, używając podpowiedzi FORCESCAN, jak w poniższym zapytaniu:

```
SELECT * FROM Sales.SalesOrderDetail WITH (FORCESCAN)
WHERE ProductID = 897
```

W tym przykładzie zastosowanie podpowiedzi INDEX(1) da podobny efekt, ponieważ SQL Server nie może używać indeksu klastrowego do wykonania operacji *Index Seek*; klucz klastrowy jest założony na kolumnach *SalesOrderID* i *SalesOrderDetailID*, dlatego jedynym poprawnym wyborem jest skanowanie indeksu klastrowego. Oczywiście

możesz też wymusić sytuację odwrotną. W poniższym przykładzie optymalizator szacuje, że zwrócona zostanie większa liczba rekordów, dlatego wybiera plan z operatorem *Clustered Index Scan*:

```
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 870
```

Ponieważ na kolumnie *ProductID* mamy indeks (*IX_SalesOrderDetail_ProductID*), możemy wymusić, aby plan skorzystał z tego indeksu, jak w poniższym zapytaniu. Plan wynikowy będzie wykorzystywał operator *Index Seek* dla indeksu *IX_SalesOrderDetail_ProductID* i operację *Key Lookup* na tabeli bazowej, która w tym przypadku jest indeksem klastrowym:

```
SELECT * FROM Sales.SalesOrderDetail WITH (INDEX (IX_SalesOrderDetail_ProductID))
WHERE ProductID = 870
```

Podobny efekt możesz osiągnąć za pomocą podpowiedzi *FORCESEEK*, która została wprowadzona w SQL Serverze 2008. Poniższe zapytanie stworzy plan podobny do poprzedniego:

```
SELECT * FROM Sales.SalesOrderDetail WITH (FORCESEEK)
WHERE ProductID = 870
```

Możesz nawet połączyć obie podpowiedzi, aby uzyskać ten sam plan, jak w poniższym zapytaniu. Pamiętaj, że użycie podpowiedzi *INDEX* nie oznacza, że wykonana zostanie operacja *Index Seek*, dlatego podpowiedź *FORCESEEK* może pomóc osiągnąć ten cel.

```
SELECT * FROM Sales.SalesOrderDetail
WITH (INDEX (IX_SalesOrderDetail_ProductID), FORCESEEK)
WHERE ProductID = 870
```

Wykorzystanie *FORCESEEK* w sytuacjach, w których SQL Server nie będzie w stanie wykonać operacji *Index Seek*, spowoduje błąd, a zapytanie nie zostanie skompilowane, zgodnie z poniższym zapytaniem, które zwraca pokazywany wcześniej błąd 8622, informujący o nieprawidłowej podpowiedzi:

```
SELECT * FROM Sales.SalesOrderDetail WITH (FORCESEEK)
WHERE OrderQty = 1
```

FAST N

FAST N to jedna z tak zwanych *podpowiedzi zorientowanych na cel*. Nie wskazuje, jakie operatory fizyczne mają zostać wykorzystane, lecz specyfikuje, jaki cel ma osiągnąć plan. Podpowiedź ta jest stosowana do zoptymalizowania zapytania, aby pobrało pierwsze *n* wierszy rezultatu najszybciej jak to możliwe. Może pomóc w sytuacjach, w których istotne jest tylko kilka pierwszych wierszy zwracanych przez zapytanie i być może w ogóle nie będziesz używał pozostałych wierszy. Minusem tej podpowiedzi jest to, że pobranie pozostałych rekordów może prawdopodobnie zająć więcej

czasu, niż gdybyś nie wykorzystał podpowiedzi. Innymi słowy: ponieważ zapytanie jest zoptymalizowane do pobrania tylko n rekordów, zwrócenie wszystkich rekordów może być bardzo kosztowne.

Optymalizator zazwyczaj osiąga cel FAST N , unikając operatorów blokujących takich jak *Sort*, *Hash Join* i *Hash Aggregation*, dlatego klient wykonujący zapytanie nie musi czekać na zwrócenie pierwszych rekordów. Spójrzmy na przykład. Uruchom poniższe zapytanie, zwracające plan z rysunku 10.18:

```
SELECT * FROM Sales.SalesOrderDetail
ORDER BY ProductID
```

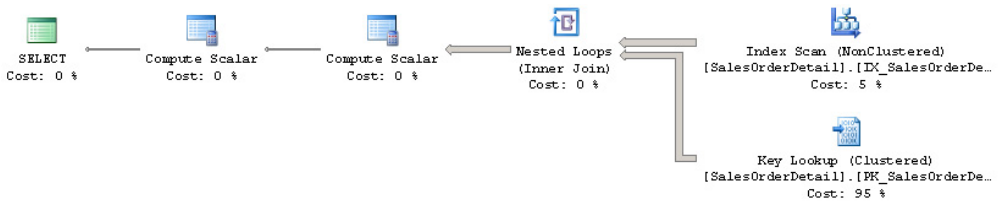


Rysunek 10.18. Plan wykorzystujący operację Sort

W tym przypadku operator *Sort* to najbardziej efektywny sposób na posortowanie rekordów po kolumnie *ProductID*, jeżeli chcesz zobaczyć cały wynik. Ponieważ jednak *Sort* jest operatorem blokującym, SQL Server nie będzie w stanie zwrócić żadnych rekordów, dopóki cała operacja sortowania zostanie nie zakończona. Załóżmy teraz, że Twoja aplikacja chce wyświetlić stronę z 20 rekordami naraz — możesz skorzystać z podpowiedzi FAST, aby otrzymać te 20 rekordów najszybciej jak to możliwe:

```
SELECT * FROM Sales.SalesOrderDetail
ORDER BY ProductID
OPTION (FAST 20)
```

Tym razem nowy plan (zobacz rysunek 10.19) skanuje dostępny indeks nieklastrowy, wykonując jednocześnie operacje *Key Lookup* na tabeli klastrowej. Ponieważ plan ten wykorzystuje losowe operacje I/O, byłby bardzo kosztowny dla całego zapytania, 20 rekordów zwróci jednak bardzo szybko.



Rysunek 10.19. Plan korzystający z podpowiedzi FAST N

Istnieje również podpowiedź FASTFIRSTROW, nie jest jednak tak elastyczna jak FAST N , ponieważ jako N możesz wyspecyfikować dowolną liczbę. FASTFIRSTROW to ekwiwalent podpowiedzi FAST 1.

NOEXPAND i EXPAND VIEWS

Zanim pomyślisz o użyciu podpowiedzi NOEXPAND i EXPAND VIEWS, omówmy domyślne zachowanie zapytań wykorzystujących widoki indeksowane, abyś mógł zobaczyć, w jaki sposób te podpowiedzi zmieniają to zachowanie. Jak wyjaśniłem w rozdziale 3., SQL Server w początkowych etapach optymalizacji, podczas łączenia, rozwiązuje widoki do ich definicji (na przykład aby bezpośrednio wykorzystać tabele z widoku). To zachowanie jest jednakowe dla wszystkich wersji SQL Servera. W dalszej części procesu optymalizacji, ale tylko w wersji Enterprise Edition, SQL Server może dopasować zapytanie do istniejącego widoku indeksowanego. Czyli na początku widok został rozwinięty, ale potem został dopasowany do istniejącego widoku indeksowanego. Podpowiedź EXPAND VIEWS usuwa krok dopasowania, a tym samym sprawia, że widoki są rozwijane, ale nie dopasowywane na końcu procesu optymalizacji. Oznacza to, że podpowiedź ma sens tylko w wersji Enterprise Edition.

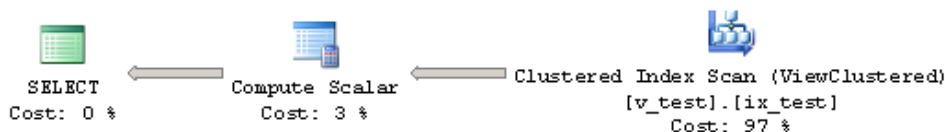
Natomiast podpowiedź NOEXPAND żąda, aby SQL Server nie rozwiązywał widoków i zamiast tego wykorzystał podany widok indeksowany. Ta podpowiedź działa we wszystkich edycjach SQL Servera i jest jedynym sposobem (dla wersji innych niż Enterprise Edition) na dopasowanie zapytania do istniejącego widoku indeksowanego. Oto przykład. Za pomocą poniższego kodu stwórz widok indeksowany w bazie AdventureWorks2012:

```
CREATE VIEW v_test
WITH SCHEMABINDING AS
SELECT SalesOrderID, COUNT_BIG(*) as cnt
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
GO
CREATE UNIQUE CLUSTERED INDEX ix_test ON v_test(SalesOrderID)
```

A następnie uruchom poniższe zapytanie:

```
SELECT SalesOrderID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
```

Jeżeli korzystasz z SQL Server Enterprise Edition (lub wersji próbnej bądź wersji Developer Edition, które współdzielą ten sam silnik), otrzymasz plan z rysunku 10.20, który dopasowuje istniejący widok indeksowany.



Rysunek 10.20. Plan wykorzystujący istniejący widok indeksowany

Alternatywnie możesz wykorzystać odpowiedź `EXPAND VIEWS`, jak w poniższym zapytaniu, aby uniknąć dopasowania widoku indeksowanego. Otrzymasz plan przedstawiony na rysunku 10.21.

```
SELECT SalesOrderID, COUNT(*)
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
OPTION (EXPAND VIEWS)
```



Rysunek 10.21. Plan z odpowiedzią `EXPAND VIEWS`

W przeciwieństwie do `EXPAND VIEWS`, która jest odpowiedzią zapytania, `NOEXPAND` jest odpowiedzią tabeli, a poniższe zapytanie przedstawia, jak z niej korzystać i uzyskać wynik podobny do poprzedniego zapytania. Zwróć uwagę, że nazwa widoku indeksowanego została podana bezpośrednio.

```
SELECT * FROM v_test WITH (NOEXPAND)
```

Wybrany plan przeskanuje indeks `v_test` w podobny sposób jak w przypadku planu z rysunku 10.20.

Usuń teraz stworzony wcześniej widok indeksowany:

```
DROP VIEW v_test
```

Wskazówki planów

W niektórych sytuacjach może zaistnieć potrzeba wykorzystania odpowiedzi dla zapytania, którego tekstu nie możesz zmienić w aplikacji. Taka sytuacja ma miejsce podczas pracy z kodem zewnętrznym lub z aplikacjami, których nie możesz zmienić.

Wskazówki planów, funkcjonalność wprowadzona w SQL Serverze 2005, mogą być w takim przypadku pomocne. Działają poprzez przechowywanie listy zapytań na serwerze, wraz z odpowiedziami, które mają być dla nich zastosowane, odseparowując w ten sposób specyfikację odpowiedzi od samego zapytania. Aby wykorzystać wskazówki planów, będziesz musiał podać zapytanie, które chcesz zoptymalizować, i odpowiedź z klauzulą `OPTION` lub plan XML za pomocą odpowiedzi `USE PLAN`, co omówię w następnym podrozdziale. Podczas optymalizacji zapytania SQL Server wykorzysta odpowiedzi zapisane w definicjach wskazówek planu. Możesz również wyspecyfikować `NULL` jako wskazówkę, a wtedy optymalizator usunie wszystkie istniejące odpowiedzi z zapytania. Wskazówki mogą także dopasowywać zapytania w różnych kontekstach — na przykład w procedurze przechowywanej, funkcji skalarnej zdefiniowanej przez użytkownika lub samodzielnym poleceniu niebędącym częścią żadnego obiektu w bazie danych.

Do stworzenia wskazówki możesz wykorzystać procedurę przechowywaną `sp_create_plan_guide` oraz procedurę `sp_control_plan_guide` do usunięcia, włączenia i wyłączenia wskazówki. Możesz również sprawdzić, jakie wskazówki są zdefiniowane w Twojej bazie w widoku `sys.plan_guides`.

Aby upewnić się, że zapytanie w definicji wskazówki zgadza się z faktycznie wykonywanym zapytaniem, szczególnie w przypadku poleceń samodzielnych, możesz skorzystać z klasy zdarzeń *Plan Guide Successful* mechanizmu Profiler, która pokaże, czy SQL Server był w stanie stworzyć plan z użyciem wskazówki planu. Natomiast zdarzenie *Plan Guide Unsuccessful* pokazuje, że SQL Server nie był w stanie wykorzystać wskazówek, czyli zapytanie zostało zoptymalizowane bez podpowiedzi. Zdarzenie *Plan Guide Unsuccessful* wystąpi na przykład wtedy, kiedy spróbujesz wymusić złączenie *Merge* lub *Hash Join* dla operatora nierówności w warunku złączenia, zgodnie z wcześniejszymi informacjami z tego rozdziału.

Spójrzmy na przykłady tych zdarzeń. Załóżmy, że chcemy użyć wskazówki do uniknięcia zastosowania operatora *Merge* lub *Hash Join* w poprzednim zapytaniu, aby uniknąć wysokiego wykorzystania pamięci. Zanim uruchomisz kod, uruchom sesję Profiler, połącz ją ze swoją instancją SQL Servera, wybierz pusty szablon, aby rozpocząć nową definicję śledzenia, wybierz oba zdarzenia *Plan Guide Successful* i *Plan Guide Unsuccessful* w sekcji *Performance* z zakładki *Events*, a następnie rozpocznij śledzenie.

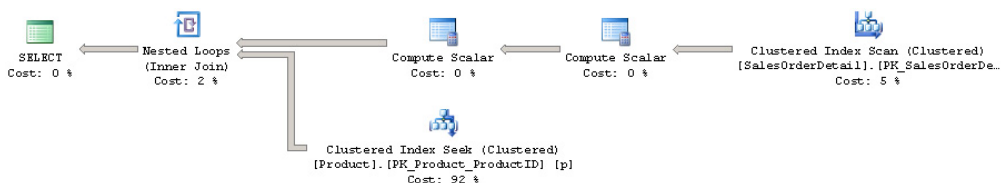
Następnie stwórz poniższą procedurę przechowywaną:

```
CREATE PROCEDURE test
AS
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
```

Zanim stworzymy wskazówkę planu, wykonaj procedurę wykonywaną i wyświetl jej plan, aby sprawdzić, czy rzeczywiście wykorzystany został operator *Hash Join*. Następnie stwórz wskazówkę wymuszającą użycie operatora *Nested Loops Join*.

```
EXEC sp_create_plan_guide
@name = N'plan_guide_test',
@stmt = N'SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID',
@type = N'OBJECT',
@module_or_batch = N'test',
@params = NULL,
@hints = N'OPTION (LOOP JOIN)'
```

Jeżeli teraz powtórnie wykonasz procedurę przechowywaną, przekonasz się, że faktycznie wykorzystuje operator *Nested Loops Join* (zobacz rysunek 10.22). Zauważ jednak, że jest to bardziej kosztowny plan, ponieważ przed zdefiniowaniem wskazówki SQL Server wybiera poprawny rodzaj złączenia.



Rysunek 10.22. Zapytanie wykorzystujące wskazówkę planu

Ponadto podczas wykonywania procedury Profiler powinien przechwycić zdarzenie *Plan Guide Successful*, świadczące o tym, że SQL Server był w stanie skorzystać ze zdefiniowanej wskazówki planu, która w tym przypadku nosi nazwę `plan_guide_test`. Możesz również zweryfikować wykorzystanie wskazówki we właściwości *PlanGuideName*, która w tym przypadku będzie miała wartość `plan_guide_test`.

Wskazówki planu są automatycznie włączane podczas ich tworzenia, można je jednak włączać i wyłączać w dowolnym momencie. Na przykład poniższe polecenie wyłączy poprzednią wskazówkę, a procedura znów będzie wykorzystywała operator *Hash Join*:

```
EXEC sp_control_plan_guide N'DISABLE', N'plan_guide_test'
```

Aby powtórnie włączyć wskazówkę, użyj poniższego polecenia:

```
EXEC sp_control_plan_guide N'ENABLE', N'plan_guide_test'
```

Aby posprzątać, usuń wskazówkę i procedurę przechowywaną. Zauważ, że najpierw musisz usunąć wskazówkę, ponieważ usunięcie planu, do którego odnosi się jakaś wskazówka, nie jest możliwe.

```
EXEC sp_control_plan_guide N'DROP', N'plan_guide_test'
DROP PROCEDURE test
```

USE PLAN

Spójrzmy jeszcze na odpowiedź `USE PLAN`, która również została wprowadzona w SQL Serverze 2005. Jest to ekstremalny przypadek odpowiedzi, ponieważ pozwala wyspecyfikować cały plan wykonania, który ma zostać wykorzystany podczas optymalizacji zapytania. Odpowiedź ta jest użyteczna, jeżeli wiesz, że istnieje lepszy plan wykonania niż proponowany przez optymalizator. Ma to miejsce na przykład wtedy, kiedy w przeszłości stworzony został bardziej wydajny plan albo kiedy taki plan został stworzony w innym systemie czy nawet w innej wersji SQL Servera. Plan należy podać w formacie XML i do jego wygenerowania wykorzystuje się SQL Server, ponieważ samodzielne napisanie takiego planu może być ekstremalnie trudne.

Odpowiedź `USE PLAN` może wymusić większość właściwości zapytania — włączając w to strukturę drzewa, kolejność złączeń, algorytmy złączeń, agregacje, sortowanie i unie oraz operacje na indeksach takie jak skanowanie, przeszukiwanie i przecięcia — tak aby wykonywane były tylko reguły transformacji wykorzystywane do znalezienia

żądanego planu. Ponadto od wersji 2008 USE PLAN wspiera polecenia aktualizacji (INSERT, UPDATE, DELETE i MERGE). Niektóre polecenia wciąż nie są wspierane, na przykład zapytania pełnotekstowe i rozproszone oraz zapytania z kursorami dynamicznymi, napędzanymi zestawami kluczy i jednokierunkowymi.

Załóżmy, że mamy to samo zapytanie, z którym pracowaliśmy w sekcji „Wskazówki planów”, wykorzystujące operator *Hash Join*:

```
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
```

Załóżmy również, że chcesz, aby SQL Server wykorzystał inny plan wykonania, który możemy wygenerować z użyciem odpowiedzi:

```
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
OPTION (LOOP JOIN)
```

Możesz wymusić, aby nowy plan wykorzystał operator *Nested Loops Join* zamiast *Hash Join*. Aby to osiągnąć, wyświetl plan XML (klikając prawym przyciskiem myszy na planie graficznym i wybierając opcję *Show Execution Plan XML...*), skopiuj go do edytora, zamień wszystkie istniejące apostrofy (') na cudzysłowy ("), a następnie skopiuj plan do zapytania:

```
SELECT *
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID
OPTION (USE PLAN N'<?xml version="1.0" encoding="utf-16"?>
...
</ShowPlanXML>')
```

Oczywiście plan XML jest zbyt długi, aby go tu wstawić, dlatego pokazałem tylko początek i koniec. Upewnij się, że zapytanie jest zakończone znakami ') po planie XML. Uruchomienie tego polecenia zażąda, aby SQL Server spróbował wykorzystać identyczny plan, a zapytanie zostanie wykonane z operatorem *Nested Loops Join*.

Możesz połączyć wskazówki planów i odpowiedź USE PLAN, aby wymusić konkretny plan wykonania w sytuacji, w której nie chcesz zmieniać tekstu oryginalnego zapytania. Następne (i ostatnie) zapytanie korzysta z tej samej procedury testowej co zapytanie z podrozdziału „Wskazówki planów” oraz wcześniej wygenerowanego planu XML (być może będziesz musiał ponownie stworzyć procedurę, jeżeli już ją usunąłeś). Zwróć uwagę na użycie dwóch apostrofów przed podaniem planu XML, co oznacza, że tym razem teks zapytania musi kończyć się znakami ') ').

```
EXEC sp_create_plan_guide
@name = N'plan_guide_test',
@stmt = N'SELECT *
FROM Production.Product AS p
```

```

JOIN Sales.SalesOrderDetail AS sod
ON p.ProductID = sod.ProductID',
@type = N'OBJECT',
@module_or_batch = N'test',
@params = NULL,
@hints = N'OPTION (USE PLAN N'<?xml version="1.0" encoding="utf-16"?>
...
</ShowPlanXML>'')'

```

Pamiętaj, że kiedy odpowiedź `USE PLAN` jest używana bezpośrednio w zapytaniu, niepoprawny plan spowoduje błąd zapytania. Jeżeli jednak `USE PLAN` zostanie zastosowana w ramach wskazówki planu, niepoprawny plan spowoduje skompilowanie bez żądanej odpowiedzi.

Podsumowanie

Procesor zapytań SQL Servera zazwyczaj wybiera odpowiedni plan dla zapytań, mogą się jednak zdarzyć sytuacje, kiedy po długotrwałym szukaniu problemu nie otrzymujesz wystarczająco wydajnego planu wykonania. Optymalizacja zapytań to bardzo skomplikowany problem i pomimo kilku dekad badań rozwiązania pewnych fundamentalnych aspektów są wciąż poprawiane.

W tym rozdziale przedstawiłem zalecenia na temat tego, co robić, jeżeli SQL Server nie zwraca dobrego planu. Jedną z takich metodologii, z których możesz skorzystać w przypadku problemów ze skomplikowanymi zapytaniami, jest rozbitcie zapytania na dwa lub więcej prostszych zapytań i przechowywanie wyników pośrednich w tabelach tymczasowych.

Chociaż do poprawiania wydajności w takich przypadkach można wykorzystać odpowiedzi, bezpośrednio oddziałując na wybór planu wykonania, należy ich używać ostrożnie i tylko w ostateczności. Powinieneś również być świadomy tego, że kod z podpowiedziami będzie wymagał dodatkowej obsługi i jest znacznie mniej podatny na zmiany wprowadzane w bazie danych lub aplikacji, a także na zmiany spowodowane aktualizacjami oprogramowania.

Mam ogromną nadzieję, że ta książka zapewniła Ci wiedzę niezbędną do pisania lepszych zapytań oraz przekazywania procesorowi zapytań informacji potrzebnych do tworzenia wydajnych planów zapytań. Jednocześnie mam nadzieję, że dowiedziałeś się więcej na temat tego, jak zdobywać informacje potrzebne do diagnozy przypadków, w których pomimo wysiłków nie otrzymujesz dobrego planu. Dzięki zapoznaniu się ze sposobem działania procesora zapytań i poznaniu problemów, z którymi do dzisiaj boryka się to skomplikowane oprogramowanie, będziesz lepiej przygotowany do podjęcia decyzji, kiedy i jak użyć odpowiedzi do poprawienia wydajności zapytań.

Dodatek

Źródła



Ten dodatek zawiera listę opracowań technicznych, artykułów, prac naukowych i książek, z których możesz skorzystać, aby dowiedzieć się więcej na temat optymalizacji zapytań w SQL Serverze lub aby pogłębić wiedzę na temat konkretnych zagadnień. Chociaż większość z nas doskonale zna zawartość opracowań technicznych, artykułów i książek, dla niektórych prace naukowe mogą być nowością. Czytanie prac naukowych i akademickich wymaga zazwyczaj głębszej wiedzy technicznej niż dokumentacja, książki i blogi, które czytamy na co dzień. Skupiają się przeważnie na konkretnym obszarze badań lub na konkretnym problemie i zawsze odnoszą się do innych prac, których listę znajdziesz na końcu każdej z nich. Podążając za tymi odniesieniami, które również będą miały swoje odniesienia, znajdziesz niemalże niewyczerpane źródło informacji.

Większość wyszczególnionych tutaj publikacji jest ogólnie dostępnych i podaje adresy, pod którymi je znajdziesz.

Opracowania techniczne

- ▶ *Clustered Indexes and Heaps*,
Burzin Patel, Sanjay Mishra,
<http://technet.microsoft.com/en-us/library/cc917672.aspx>
- ▶ *Columnstore Indexes for Fast Data Warehouse Query Processing in SQL Server 11.0*,
Eric N. Hanson,
<http://download.microsoft.com/download/8/C/1/8C1CE06B-DE2F-40D1-9C5C-3EE521C25CE9/Columnstore%20Indexes%20for%20Fast%20DW%20QP%20SQL%20Server%2011.pdf>
- ▶ *In-Memory OLTP-Common Workload Patterns and Migration Considerations*,
Mike Weiner, Ami Levin,
<http://msdn.microsoft.com/en-us/library/dn673538.aspx>
- ▶ *Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator*,
Joseph Sack,
<http://msdn.microsoft.com/en-us/library/dn673537.aspx>
- ▶ *Plan Caching and Recompilation in SQL Server 2012*,
Greg Low,
<http://msdn.microsoft.com/en-us/library/dn148262.aspx>
- ▶ *SQL Server 2005 Waits and Queues*,
Tom Davidson, Danny Tambs,
<http://technet.microsoft.com/en-US/library/cc966413.aspx>

- ▶ *SQL Server In-Memory OLTP Internals Overview*,
Kalen Delaney,
<http://msdn.microsoft.com/en-us/library/dn720242.aspx>
- ▶ *Statistics Used by the Query Optimizer in Microsoft SQL Server 2008*,
Eric N. Hanson, Yavor Angelov,
[http://msdn.microsoft.com/en-us/library/dd535534\(v=sql.100\).aspx](http://msdn.microsoft.com/en-us/library/dd535534(v=sql.100).aspx)
- ▶ *Troubleshooting Performance Problems in SQL Server 2008*,
Sunil Agarwal, Boris Baryshnikov, Keith Elmore, Juergen Thomas,
Kun Cheng, Burzin Patel,
[http://technet.microsoft.com/en-us/library/dd672789\(v=sql.100\).aspx](http://technet.microsoft.com/en-us/library/dd672789(v=sql.100).aspx)
- ▶ *Using Star Join and Few-Outer-Row Optimizations to Improve Data Warehousing Queries*,
Dayong Gu, Ashit Gosalia, Wei Liu, Vinay Kulkarni,
<http://technet.microsoft.com/en-us/library/gg567299.aspx>

Artykuły

- ▶ *Ascending Keys and Auto Quick Corrected Statistics*,
Ian Jose,
<http://blogs.msdn.com/b/ianjo/archive/2006/04/24/582227.aspx>
- ▶ *Changes to automatic update statistics in SQL Server — traceflag 2371*,
Juergen Thomas,
<http://blogs.msdn.com/b/saponsqlserver/archive/2011/09/07/changes-to-automatic-update-statistics-in-sql-server-traceflag-2371.aspx>
- ▶ *Controlling Autostat (AUTO_UPDATE_STATISTICS) behavior in, SQL Server*,
<http://support.microsoft.com/kb/2754171>
- ▶ *Enable plan-affecting SQL Server query optimizer behavior that can be controlled by different trace flags on a specific-query level*,
<http://support.microsoft.com/kb/2801413/en-us>
- ▶ *FIX: Poor performance when you run a query that contains correlated AND predicates in SQL Server 2008 or in SQL Server 2008 R2 or in SQL Server 2012*,
<http://support.microsoft.com/kb/2658214>
- ▶ *FIX: Poor performance when you use table variables in SQL Server 2012*,
<http://support.microsoft.com/kb/2952444>
- ▶ *Intro to Query Execution Bitmap Filters*,
<http://blogs.msdn.com/b/sqlqueryprocessing/archive/2006/10/27/query-execution-bitmaps.aspx>

- ▶ *Query Fingerprints and Plan Fingerprints (The Best SQL 2008 Feature That You've Never Heard Of)*,
Bart Duncan,
http://blogs.msdn.com/b/bartd/archive/2008/09/03/query-fingerprints-and-plan-fingerprints_3a00_-the-best-new-sql-2008-feature-you_2700_-ve-never-heard-of.aspx
- ▶ *Query Processor Modelling Extensions in SQL Server 2005 SP1*,
Ian Jose,
<http://blogs.msdn.com/b/ianjo/archive/2006/04/24/582219.aspx>
- ▶ *Sql_Handle and Plan_Handle Explained*,
<http://blogs.msdn.com/b/sqlprogrammability/archive/2007/01/09/2-0-sql-handle-and-plan-handle-explained.aspx>
- ▶ *Statistical maintenance functionality (autostats) in SQL Server*,
<http://support.microsoft.com/kb/195565>
- ▶ *The coming in-memory database tipping point*,
Dave Campbell,
<http://blogs.technet.com/b/dataplatforminsider/archive/2012/04/09/the-coming-in-memory-database-tipping-point.aspx>
- ▶ *When to Break Down Complex Queries*,
Steve Howard,
<http://blogs.msdn.com/b/sqlcat/archive/2013/09/09/when-to-break-down-complex-queries.aspx>

Prace naukowe

- ▶ *Access Path Selection in a Relational Database Management System*,
P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R.A. Lorie,
T. G. Price,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.5879>
- ▶ *An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server*,
Surajit Chaudhuri, Vivek Narasayya,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.114.771>
- ▶ *An Overview of Cost-based Optimization of Queries with Aggregates*,
Surajit Chaudhuri,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.137.3356>
- ▶ *An Overview of Data Warehousing and OLAP Technology*,
Surajit Chaudhuri, Umeshwar Dayal,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.133.6667>

- ▶ *An Overview of Query Optimization in Relational Systems*,
Surajit Chaudhuri,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.137.3356>
- ▶ *C-Store: A Column-oriented DBMS*,
Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.4717>
- ▶ *Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer*,
Florian Waas, Cesar Galindo-Legaria,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.2115>
- ▶ *Do Query Optimizers Need to be SSD-aware?*,
Steven Pelley, Thomas F. Wenisch, Kristen Lefevre,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.363.7595>
- ▶ *Optimizing Join Orders*,
Michael Steinbrunn, Guido Moerkotte, Alfons Kemper,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.8155>
- ▶ *Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server*,
Cesar A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, Shin Zhang,
<http://dl.acm.org/citation.cfm?id=1546682.1547293>
- ▶ *Query Evaluation Techniques for Large Databases*,
Goetz Graefe,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.108.3178>
- ▶ *Query Optimization*,
Yannis E. Ioannidis,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.4154>
- ▶ *Query Optimization in Microsoft SQL Server PDW*,
Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David DeWitt, Cesar Galindo-Legaria,
<http://dl.acm.org/citation.cfm?id=2213953>
- ▶ *Query Processing Techniques for Solid State Drives*,
Dimitris Tsirogiannis, Janet L. Wiener, Stavros Harizopoulos, Goetz Graefe, Mehul A. Shah,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.7568>
- ▶ *Self-Tuning Database Systems: A Decade of Progress*,
Surajit Chaudhuri, Vivek Narasayya,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.8842>

- ▶ *Testing Cardinality Estimation Models in SQL Server*,
Campbell Fraser, Leo Giakoumakis, Vikas Hamine,
Katherine F. Moore-Smith,
<http://dl.acm.org/citation.cfm?id=2304526>
- ▶ *Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences*,
Leo Giakoumakis, Cesar Galindo-Legaria,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.3767>
- ▶ *The Cascades Framework for Query Optimization*,
Goetz Graefe,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.98.9460>
- ▶ *The History of Histograms*,
Yannis Ioannidis,
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.4839>

Książki

- ▶ *Database System Concepts*,
Abraham Silberschatz, Henry F. Korth, S. Sudarshan,
McGraw-Hill, 2010
- ▶ *Inside the SQL Server Query Optimizer*,
Benjamin Nevarez,
Red Gate Books, 2011
- ▶ *Microsoft SQL Server 2012 Internals*,
Kalen Delaney, Bob Beauchemin, Conor Cunningham, Jonathan Kehayias,
Benjamin Nevarez, Paul S. Randal,
Microsoft Press, 2013
- ▶ *SQL Server 2005 Practical Troubleshooting: The Database Engine*,
Ken Henderson,
Addison-Wesley Professional, 2006
- ▶ *Star Schema: The Complete Reference*,
Christopher Adamson,
McGraw-Hill Professional, 2010
- ▶ *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*,
Ralph Kimball, Margy Ross,
Wiley, 2002

Skorowidz

A

- agregacja, 118, 147, 355
 - haszowa, 150
 - przed złączeniem, 358
- aktualizacja, 167, 172, 173
 - statystyk, 210
- Algebrizer, 101
- algorytmy złączeń, 352
- analiza DTA, 195
- architektura, 19
 - Hekatona, 255, 277
- asocjacja, 113
- atomowość, 256
- atrybut
 - CardinalityEstimationModelVersion, 38
 - scan count, 51
 - StatementOptmEarlyAbortReason, 37
- autoparametryzacja, 296

B

- baza danych AdventureWorksDW2012, 320
- błędy szacunku kardynalności, 227, 349
- brakujące indeksy, 202

C

- czas, 59
 - CLR, 65
 - procesora, 59
 - wykonywania, 65

D

- Data Collector, 82, 87
 - konfiguracja, 83
 - tabele, 88
 - wykorzystanie, 87

- Database Engine Tuning Advisor, 193
- DISTINCT, 151
- Distinct Sort, 151
- DLL, 276
- DMF, Dynamic Management Function, 44, 55
- DMV, Dynamic Management View, 44, 55
- dokumentacja Books Online, 181
- dostęp do danych, 138
- drzewo, 103
 - algebrizera, 101
 - krzaczaste, 343, 358
 - lewostronnie rozgałęzione, 343
 - logiczne, 106
 - parsowania, 101
- DTA, 195, 197, 200
- dynamiczne
 - funkcje zarządzania, 44, 55
 - widoki zarządzania, 44, 55
- działania równoległe, 158

E

- edytor XML, 32
- ekran
 - Complete the Wizard, 85
 - Map Logins and Users, 84
 - Setup Data Collection Sets, 85
- eliminacja
 - blokad, 255
 - segmentów, 328
 - złączenia, 108

F

- fazy search, 131, 133
- filtr
 - bitmapowy, 324
 - Blooma, 323

flaga
2389, 238, 241
2390, 241
4137, 227
format XML, 32
fragmentacja indeksów, 204
funkcjonalność
Data Collector, 83
Index Tuning Wizard, 194
podglądania danych, 48

G

generowanie planów zapytań, 22
gęstość, 216

H

Halloween, 172, 173
Hash Aggregate, 150
Hash Join, 157
haszowanie, 147
Hekaton, 253
indeksy, 257, 264
kompiler, 277
rekordy, 264
tabele, 257, 258
histogram, 218
hurtownie danych, 317, 318

I

IAM, Index Allocation Map, 140
identyfikator RID, 182
ikona ostrzeżenia, 42
indeksy, 151, 175
brakujące, 202
filtrowane, 187, 188
fragmentacja, 204
główne, 178
haszowe, 264, 268, 271
klastrowe, 177, 181, 185
klucz, 185
koszt, 195
kreator dostosowywania, 194
magazynu kolumn, 326, 330
nieklastrowe, 169, 177, 180

nieużywane, 206
operacje, 189
pokrywające, 186
tworzenie, 177
unikalne, 178
usuwanie, 180
zakresowe, 268, 272
Index Tuning Wizard, 193
informacje o operatorze, 29
inkluzja, 222
interfejs graficzny, 47
izolacja, 256

J

jednolitość, 222
język
C#, 313
SQL, 19
T-SQL, 19

K

kardynalność, 217
klasy zdarzeń, 290, 365
klucz
indeksu klastrowego, 185
obcy, 107, 108
klucze rosnące, 236
kolejność złączeń, 341
kolekcja
Query Hash Statistics, 90
Query Statistics, 89
kolumny
sys.dm_exec_query_stats, 59, 60
wyliczeniowe, 232, 280
kombinacje kolejności złączeń, 344
kompilacja, 25
zestawów zapytań, 288
kompilator Hekaton, 256
kompresja, 328
komutacja, 113
konfiguracja
Data Collector, 83
narzędzia AMR, 281
śledzenia, 66
zdarzenia, 75

konserwacja statystyk, 246
 kontrola dodatkowa, 349
 korzystanie z podpowiedzi, 349
 korzyści wydajnościowe, 327
 koszt, 249
 posiadania, 194
 zapytań, 21, 23, 64
 kreator
 Data Collection Wizard, 281
 dostosowywania indeksów, 194

L

liczba
 odczytów, 59
 optymalizacji, 95–98
 planów, 96
 planów trywialnych, 95
 stron LOB, 51
 uruchomień, 96
 zapisów, 59
 liczniki
 nieudokumentowane, 95
 udokumentowane, 95
 lista sprawdzeń, 286
 LOB, large object, 51
 logika OR, 345

M

magazyn
 CACHESTORE_OBJCP, 294
 CACHESTORE_PHDR, 294
 CACHESTORE_SQLCP, 294
 CACHESTORE_XPROC, 294
 kolumn, 326, 327
 OBJCP, 62
 planów, 44, 49, 196, 287, 292
 SQLCP, 62
 wierszy, 327
 mapa alokacji indeksu, 140
 mapowanie zdarzeń SQL Trace, 71
 MAT, Mixed Abstract Tree, 277
 mechanizm
 Native Compilation Advisor, 285
 szacowania kardynalności, 222

Memo, 122, 127
 Merge Join, 119, 156
 metoda
 Close(), 28
 GetRow(), 28
 Open(), 28
 module_end, 72
 modyfikacje ciągów znaków, 263

N

narzędzie
 AMR, 280, 281
 Index Tuning Wizard, 193
 narzut optymalizacji, 197
 natywnie kompilowane
 procedury przechowywane, 273
 New Session Wizard, 74
 niezależność, 222

O

oddalanie planu zapytania, 32
 ograniczenia, 166, 279
 CHECK, 280
 DEFAULT, 280
 procesora zapytań, 339
 okno
 Configure Management Data
 Warehouse Storage, 84
 nowej sesji
 strona Advanced, 77
 strona Data Storage, 76
 strona Events, 75
 strona General, 74
 właściwości zapytania, 30
 OLTP, 253
 opcja
 ANSI_NULLS OFF, 310
 ANSI_PADDING OFF, 310
 CONCAT_NULL_YIELDS_NULL
 OFF, 310
 PAGECOUNT, 242, 324
 ROWCOUNT, 242, 324
 SET, 309

- opcje
 - konfiguracji zdarzenia, 75
 - New Session Wizard, 74
 - optymalizacji dla zapytań, 297
 - tworzenia skryptu, 200
- operacja
 - DELETE, 169, 183
 - Index Scan, 270
 - Index Seek, 270, 271
 - INSERT, 168
 - Key Lookup, 186
 - SELECT, 184
 - SET SHOWPLAN_ALL, 35
 - SET STATISTICS PROFILE, 36
 - Sort, 271
- operacje na indeksach, 189
- operator, 127
 - Clustered Index Delete, 171
 - Clustered Index Scan, 105, 139, 250, 304
 - Clustered Index Seek, 141–143, 153, 154
 - Columnstore Index Scan, 332, 333, 334
 - Compute Scalar, 148
 - Distinct Sort, 151
 - Distribute Streams, 161, 165
 - Gather Streams, 161
 - Hash Aggregate, 27, 150
 - Hash Join, 157, 322, 352, 367
 - Index Scan, 27, 141, 191, 273
 - Index Seek, 142, 190, 303
 - Key Lookup, 303
 - Nested Loops Join, 153
 - Repartition Streams, 164
 - Sort, 149
 - Stream Aggregate, 147, 151, 160, 355
 - Table Scan, 139
- operatorzy
 - dostępu do danych, 138
 - współbieżności, 163
 - wymiany, 160
 - zapytań, 137
- optymalizacja, 129, 197
 - pod typowy parametr, 304
 - przy każdym wykonaniu, 305
 - zapytań, 17, 21, 196, 297, 340
 - złączenia gwiazdowego, 321
- OptimizerHardwareDependentProperties, 39

- optymalizator zapytań, 19, 91
- ostrzeżenie, 39
 - ColumnsWithNoStatistics, 40
 - NoJoinPredicate, 41
 - PlanAffectingConvert, 42
 - SpillToTempDb, 42
 - UnmatchedIndexes, 43

P

- parametry typowe, 304
- parametryzacja, 295
 - wymuszona, 299
- parsowanie, 20, 92, 101
- partycjonowanie, 162, 230
 - danych, 160
 - haszowe, 164
 - z rozgłaszaniem, 165
- pełna optymalizacja, 129
- plan
 - graficzny, 26
 - per indeks, 169
 - per wiersz, 170
 - równoległy, 159
 - tekstowy, 35
 - SET SHOWPLAN_ALL, 35
 - SET SHOWPLAN_TEXT, 35
 - SET STATISTICS PROFILE, 35
 - trywialny, 109
 - współbieżny, 158
 - wykonania, 21, 25, 39, 118
 - rzeczywisty, 33
 - szacowany, 33
 - w formacie XML, 34
 - zapytania, 21
 - z ostrzeżeniem
 - ColumnsWithNoStatistics, 40
 - NoJoinPredicate, 41
 - PlanAffectingConvert, 42
 - SpillToTempDb, 42
 - UnmatchedIndexes, 43
- plan_handle, 61
- plan_hash, 62
- pliki
 - .sqlplan, 33
 - XML, 78

- pobieranie planów, 44
- podgląd
 - danych na żywo, 48
 - magazynu planów, 57
- podpowiedzi, 335, 348, 351
 - procesora zapytań, 339
 - zignorowane, 353
 - zorientowane na cel, 361
- podpowiedź
 - EXPAND VIEWS, 363, 364
 - FAST N, 361
 - FORCE ORDER, 356, 357
 - FORCE PLAN, 359
 - FORCESCAN, 359
 - FORCESEEK, 359
 - HASH GROUP, 355
 - INDEX, 359, 360
 - NOEXPAND, 363
 - OPTIMIZE FOR UNKNOWN, 306
 - QUERYRULEOFF, 120
 - USE PLAN, 366, 368
- podsluchiwanie parametrów, 302, 308, 309
- polecenia
 - DDL, 73
 - SET, 310
- polecenie
 - CREATE INDEX, 179, 194
 - DBCC FREEPROCCACHE, 49, 99, 142
 - DBCC RULEOFF, 120
 - DBCC RULEON, 120
 - DBCC SHOW_STATISTICS, 214, 276
 - DBCC TRACESTATUS, 241
 - SET STATISTICS IO, 50
 - SET STATISTICS TIME, 50
 - UPDATE STATISTICS, 201, 242, 243
- porównania, 263
- powtórne
 - stworzenie indeksu, 335
 - wykorzystanie planów, 309
- poziomy izolacji, 275
- predykat
 - StateProvinceID, 145, 146
 - VacationHours, 106
- predykaty z AND, 225
- problem
 - Halloween, 172, 173
 - z systemem, 349

- procedury
 - kompilowane natywnie, 274
 - przechowywane, 273, 300
- proces
 - kompilacji i rekompilacji, 25, 289
 - przetwarzania zapytania, 20, 92, 93
 - optymalizacji, 92, 130
- program
 - Memory Optimization Advisor, 286
 - SQL Profiler, 45, 66
- przechowywanie
 - danych, 327
 - planów, 21, 23
- przechwytywanie
 - planów zapytania, 47
 - zdarzeń, 78, 291
- przeglądanie magazynu planów, 292
- przekroczenie dozwolonego czasu, 135
- przełączanie pomiędzy partycjami, 334
- przepływ danych, 28
- przesunięcie predykatów, 104
- przeszukiwanie, 141
- przetwarzanie zapytania, 20
 - partiami, 328, 329
- przypisywanie, 20, 21, 92, 101

Q

- Query Hash Statistics, 90
- Query Statistics, 89
- query_hash, 62

R

- raport
 - Query Statistics History, 87, 88
 - Recommended Stored Procedures Based on Usage, 284
 - Recommended Tables Based on Contention, 283
 - Recommended Tables Based on Usage, 283
 - Server Activity History, 87
 - statystyk wydajności tabeli, 284
 - statystyk procedury przechowywanej, 285
 - Transaction Performance Analysis Overview, 282

redukcja I/O, 327

reguła

 GbAggBeforeJoin, 118

 GbAggToStrm, 121

 JNtoSM, 119, 120

reguły

 komutacji i asocjacji, 113

 transformacji, 112, 117

rekompilacja, 25

 zestawów zapytań, 288

repartycjonowanie strumieni, 158

reprezentacja zapytania, 103

rodzaje odpowiedzi, 351

rozbijanie zapytań, 344

rozproszenie strumieni, 158

rozszerzenie .sqlplan, 33

rpc_completed, 73

S

schemat XSD, 36, 39

sekwencja zdarzeń, 290

selektywność, 210

serwery

 testowe, 197

 połączone, 245

sesja, 73

silnik

 bazodanowy Hekaton, 253, 256

 OLTP, 254

 przechowywania Hekaton, 255

 wykonywania, 138

skanowanie, 139

skrypt

 AdventureWorks2012, 100

 do utworzenia sesji, 76

sortowanie, 139, 147, 263, 271

sp_statement_com, 73

spójność, 256

sprawdzanie obiektów statystyk, 213

sprzeczności, 105

SQL Server 2000, 213

SQL Server Profiler, 66

SQL Trace, 47, 52, 65

sql_batch_completed, 73

sql_handle, 61

sql_statement_completed, 73

SSAS, SQL Server Analysis Services, 326

statement_end_offset, 60

statement_start_offset, 60

statystyki, 127, 209

 dla kluczy rosnących, 236

 dla kolumn wyliczeniowych, 232

 filtrowane, 234

 inkrementacyjne, 229

 na serwerach połączonych, 245

 wydajności tabeli, 284

 wydajnościowe, 58

sterta, 177, 181

Stream Aggregate, 147

struktura

 Bw-drzewa, 269

 indeksu klastrowego, 182

 Memo, 122, 127

strumień, 158

sys.dm_exec_query_optimizer_info, 94–100,
110, 131

sys.dm_exec_query_plan, 44

sys.dm_exec_query_stats, 57, 59, 60, 196

sys.dm_exec_query_transformation_stats, 115

sys.dm_exec_requests, 55

sys.dm_exec_sessions, 55

sys.fn_trace_geteventinfo, 71

sys.fn_xe_file_target_read_file, 80

sys.trace_xe_event_map, 71

system

 analityczny, 319

 operacyjny, 319

 wykonywania Hekaton, 256

szacowanie

 kardynalności, 157, 218–222, 232, 243, 308

 kosztów, 249

szacunek kardynalności, 23, 210, 235

Ś

śledzenie, 46, 66

 zdarzenia, 47

T

tabele

 Hekatona, 257

 faktów i wymiarów, 320

TCO, total cost of ownership, 194

total_clr_time, 59

total_logical_reads, 59

total_logical_writes, 59

total_physical_reads, 59

total_worker_time, 59

transformacja, 112, 117

GbAggToStrm, 127

trwałość, 256

tryb

partii, 333

wiersz po wierszu, 332

tworzenie

indeksów, 177

indeksów magazynu kolumn, 330

procedur przechowywanych, 273

sesji, 48, 73

sesji zdarzeń, 76

skryptu, 200

statystyk, 210

tabel Hekatona, 258

typy

danych, 280

partycjonowania, 162

U

uchwyt do planu, 57

upraszczanie, 104

usuwanie

planów, 49, 294

tabeli, 208

złączeń, 107

W

wartość

AVG_RANGE_ROWS, 221

RANGE_HI_KEY, 220

plan_handle, 61

plan_hash, 62

query_hash, 62

sql_handle, 61

statement_end_offset, 60

statement_start_offset, 60

zgadywana, 218

wektor gęstości, 216

węzeł nadrzędny, 27

widok

sys.dm_exec_query_optimizer_info, 94

sys.dm_exec_query_stats, 57, 59

sys.dm_exec_query_transformation_stats,
115

sys.dm_exec_requests, 55, 56

sys.dm_exec_sessions, 55, 56

widoki

DMV, 57

z predykatem, 106

właściwości

asocjacyjne złączenia, 342

predykatów, 192

wyszukiwania, 192

operatora, 30

Clustered Index Delete, 171

Clustered Index Scan, 140, 154, 250, 325

Clustered Index Seek, 154

Columnstore Index Scan, 334

Distribute Streams, 165

Index Scan, 191, 273

Index Seek, 190

Repartition Streams, 164

współbieżności, 163

optymalizatora, 39

planów, 36

planu trywialnego, 110

zapytania, 30

właściwość

Defined Values, 148

Optimization Level, 111

PlanGuideName, 366

StatementOptLevel, 111

wskazówki planów, 364, 366

współbieżność, 158

wycofanie wykładnicze, 235

wydajność, 59, 60

czas, 59

czas procesora, 59

liczba odczytów, 59

wykonywanie zapytań, 21, 23

wykorzystanie

Data Collector, 87

przełączania pomiędzy partycjami, 334

- serwera testowego, 198
- UNION ALL, 335
- wykrywanie sprzeczności, 105
- wyłączanie podsłuchiwanie parametrów, 308
- wyszukiwanie
 - RID, 146
 - zaznaczeń, 143, 353
- wyświetlanie
 - planów zapytań, 33
 - planu tekstowego, 35
 - struktury Memo, 125
- wyzwalacze DML, 280

X

- XML, 32
- XQuery, 78

Z

- zabezpieczenie, 172
 - przed Halloween, 173
- zagregowane zbiory danych, 347
- zapytania
 - ad hoc, 297
 - na tabelach Data Collector, 88
 - skomplikowane, 344
- zapytanie
 - Parallel Scan, 161
 - QUERYTRACEON, 37

- zastosowanie drzewa krzaczastego, 358
- zawieranie, 222
- zaznaczenia, 143
- zbieranie
 - danych, 115
 - strumieni, 158
- zbiory danych, 347
- zdarzenia
 - Performance, 45
 - rozszerzone, 47, 48, 69, 71
 - SQL Trace, 71
- zdarzenie
 - Completed, 291
 - Recompile, 291
 - Showplan XML, 46
 - Starting, 290
 - StmtCompleted, 290, 291
 - StmtRecompile, 291
 - StmtStarting, 290, 291
- złączenia, 152, 347, 352
 - w stylu ANSI, 354
- złączenie
 - dwóch tabel, 107
 - gwiazdziste, 321
 - Hash Join, 157
 - Merge Join, 155
 - Nested Loops Join, 360
- zmienne lokalne, 306

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA