Microsoft

Itzik Ben-Gan
Dejan Sarka
Adam Machanic
Kevin Farlee

Zapytania w języku T-SQL

w Microsoft SQL Server 2014 i SQL Server 2012

Przekład: Natalia Chounlamany
Marek Włodarz

APN Promise, Warszawa 2015

Zapytania w języku T-SQL w Microsoft SQL Server 2014 i SQL Server 2012

Authorized Polish translation of the English language edition entitled: T-SQL Querying, ISBN: 978-0-7356-8504-8, by Itzik Ben-Gan, Dejan Sarka, Adam Machanic, and Kevin Farlee, published by Pearson Education, Inc, publishing as Microsoft Press, A Division Of Microsoft Corporation.

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: T-SQL Querying, ISBN: 978-0-7356-8504-8, by Itzik Ben-Gan, Dejan Sarka, Adam Machanic, and Kevin Farlee, opublikowanego przez Pearson Education, Inc, publikującego jako Microsoft Press, oddział Microsoft Corporation.

APN PROMISE SA, ul. Kryniczna 2, 03-934 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

ISBN: 978-83-7541-158-4

Przekład: Natalia Chounlamany, Marek Włodarz
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Dla Lilach, za to, że nadaje sens wszystkiemu, co robię.
– Itzik Ben-Gan

Spis treści

Przedmowa

Od roku 1993 pracuję w zespole Microsoft SQL Server. To kawał czasu i wspaniałe było obserwowanie od środka, jak produkt ten się rozwija i dojrzewa, aby stać się tym, czym jest obecnie. Wspaniałe było też przyglądanie się, jak coraz więcej i więcej klientów wykorzystuje SQL Server do prowadzenia swoich przedsiębiorstw i biznesów. Ale przede wszystkim miałem ten zaszczyt, że mogłem wspierać najbardziej błyskotliwą i zaangażowaną społeczność techniczną, jaką kiedykolwiek widziałem.

Społeczność Microsoft SQL Server jest pełna prawdziwie zadziwiających, inteligentnych ludzi. Są oni dumni z tego, że mogą dzielić się swoją wiedzą z innymi, wszystko po to, by społeczność była jeszcze silniejsza. Każdy na świecie może wejść do Twittera i zadać dowolne pytanie na kanale #sqlhelp, a po kilku sekundach otrzyma odpowiedź jednego z najbystrzejszych światowych ekspertów. Jeśli szukamy prawdziwej wiedzy w dziedzinie wydajności, magazynowania danych, optymalizacji zapytań, projektowania wielkoskalowego, modelowania lub dowolnego innego zagadnienia związanego z bazami danych, eksperci należący do społeczności chętnie podzielą się swoim doświadczeniem. Warto poznać ich nie tylko ze względu na wiedzę, ale również niezwykłe, przyjazne osobowości. Członkowie społeczności SQL Server lubią, aby nazywać ich rodziną - rodziną SQL.

Każdy członek społeczności zna najważniejszych udziałowców dzięki ich wiedzy w określonych dziedzinach. Jeśli ktoś zapyta, kto jest najlepszym specjalistą w dziedzinie wydajności baz danych, każdy członek społeczności poda te same nazwiska - cztery lub pięć. Jeśli pytanie będzie dotyczyło magazynowania danych, również od każdego usłyszymy tę samą odpowiedź. W każdej dziedzinie znajdziemy kilku ekspertów, którzy są najlepszymi fachowcami w wybranym obszarze domeny bazodanowej. Jest tylko jeden wyjątek, który znam, i dotyczy on języka T-SQL. Oczywiście, istnieje wielu utalentowanych programistów T-SQL, ale jeśli zapytamy kogokolwiek, kto jest najlepszy, zawsze usłyszymy tylko jedno nazwisko: Itzik Ben-Gan.

Itzik poprosił mnie o napisanie tej przedmowy do jego nowej książki i czuję się zaszczycony, że mogę to zrobić. Jego wcześniejsze książki - *Inside Microsoft SQL Server: T-SQL Querying* (Microsoft Press, 2009*), *Inside Microsoft SQL Server: T-SQL Programming*

* Wydanie polskie: *Microsoft SQL Server 2008 od środka: Zapytania w języku T-SQL*, APN Promise, Warszawa 2009.

(Microsoft Press, 2009*) oraz *Microsoft SQL Server High-Performance T-SQL Using Window Functions* (Microsoft Press, 2012**) - można znaleźć na półkach każdego administratora czy projektanta baz danych, jakiego znam. Książki te to przeszło 2000 stron najwyższej jakości wiedzy technicznej na temat języka T-SQL i określają one standard wysokiej jakości treści w dziedzinie baz danych.

Teraz dołączyła do nich nowa książka, zatytułowana po prostu *Zapytania w języku T-SQL*. Nie tylko łączy ona zagadnienia omawiane w trzech wcześniejszych pozycjach, ale również, a może przede wszystkim, dodaje informacje dotyczące SQL Server 2012 i 2014, w tym funkcje okna, nowy estymator kardynalności, sekwencje, magazyn kolumnowy, In-Memory OLTP i wiele więcej. Itzik znalazł również nowych współautorów: są to Kevin Farlee, Adam Machanic i Dejan Sarka. Kevin jest członkiem zespołu projektowego Microsoft SQL Server i kimś, z kim pracowałem przez wiele lat. Adam to jedno z tych nielicznych nazwisk, o których wspominałem wcześniej - jeden z najlepszych ekspertów w dziedzinie wydajności baz danych na świecie. Dejan wreszcie jest szeroko znany przede wszystkim dzięki swojej wiedzy na temat BI i modelowania danych.

Mogę się spodziewać, że książka ta stanie się kolejną standardową pozycją na temat T-SQL dla wszystkich członków społeczności Microsoft SQL Server.

Mark Souza
General Manager, Cloud and Enterprise Engineering
Microsoft

* Wydanie Polskie: *Microsoft SQL Server 2008 od środka: Programowanie w języku T-SQL*, APN Promise, Warszawa 2010.

** Wydanie polskie: *Microsoft SQL Server 2012. Optymalizacja kwerend T-SQL przy użyciu funkcji okna*, APN Promise, Warszawa 2012

Wstęp

Książka ta, będąc aktualizacją zarówno pozycji *Microsoft SQL Server 2008 od środka: Zapytania w języku T-SQL* (APN Promise, 2009), jak i części książki *Microsoft SQL Server 2008 od środka: Programowanie w języku T-SQL* (APN Promise, 2010), zapewnia projektantom i administratorom baz danych szczegółowy przegląd wewnętrznej architektury T-SQL i wyczerpujące źródło informacji. Zawiera omówienie nowych funkcjonalności wprowadzonych w wydaniach SQL Server 2012 i 2014, ale w wielu przypadkach dotyczy obszarów, które nie są zależne od wersji oprogramowania i zapewne będą aktualne również w przyszłych wydaniach SQL Server. Zaprezentowane zostały rozwiązania najtrudniejszych zagadnień dotyczących zapytań opartych na zbiorach i problemów dostrajania zapytań, wsparte głęboką wiedzą i doświadczeniem zespołu autorskiego. Czytelnik będzie mógł pogłębić swoje zrozumienie architektury i wewnętrznych mechanizmów i opanować praktyczne podejścia i zaawansowane techniki optymalizowania wydajności kodu. Książka przedstawia wiele unikatowych technik, które zostały opracowane, ulepszone i doszlifowane przez autorów w ciągu wielu lat pracy, zapewniając wysoko wydajne rozwiązania dla często spotykanych wyzwań. Większość przedstawianych rozwiązań i technik koncentruje się na wydajności i skuteczności tworzonego kodu. Autorzy podkreślają również potrzebę posiadania pełnego zrozumienia języka i jego podstaw matematycznych.

Kto powinien przeczytać tę książkę

Książka ta ma na celu pomóc doświadczonym praktykom T-SQL w osiągnięciu lepszego zrozumienia i wydajności. Adresatami tej publikacji są programiści T-SQL, administratorzy baz danych, specjaliści BI, analitycy danych i oraz wszyscy, którzy poważnie zajmują się językiem T-SQL. Główny cel to przygotować Czytelnika na rzeczywiste wymagania, w których potrzebne jest posługiwanie się językiem T-SQL. Treści nie skupiają się na przygotowaniu do jakiegoś egzaminu certyfikacyjnego, ale można zauważyć, że książka wyczerpuje wiele zagadnień sprawdzanych na egzaminach 70-461 oraz 70-464. Tak więc, choć książki tej nie można traktować jako jedynej pomocy naukowej w przygotowaniu do tych egzaminów, bez wątpienia będzie to bardzo przydatna lektura uzupełniająca.

Założenia

Autorzy zakładają, że Czytelnik ma przynajmniej rok solidnego doświadczenia w pracy w SQL Server, pisaniu i dostrajaniu kodu T-SQL. Przyjmują, że czytelnik ma już wprawę w tworzeniu kodu T-SQL, zna podstawy dostrajania zapytań i jest gotów zmierzyć się z bardziej wymagającymi wyzwaniami. Książka może być też użyteczna dla osób, które mają podobne doświadczenia z inną platformą bazodanową i dialektem języka SQL, ale preferowana jest rzeczywista wiedza i doświadczenie dotyczące SQL Server oraz T-SQL.

Kto nie powinien czytać tej książki

Książka ta raczej nie nadaje się dla nowicjuszy w dziedzinie baz danych i języka SQL.

Struktura ksiażki

Książka rozpoczyna się dwoma rozdziałami, w których przedstawiane są podstawy logicznego i fizycznego przetwarzania zapytań, niezbędne do zrozumienia większości pozostałych rozdziałów.

Rozdział pierwszy przedstawia logiczne przetwarzania zapytań. Omówione zostały szczegółowo logiczne fazy procesu przetwarzania, unikatowe aspekty zapytań SQL oraz szczególny zestaw reguł i koncepcji, które trzeba opanować, aby móc programować w relacyjnym, zorientowanym na zbiory środowisku.

Rozdział drugi zawiera omówienie dostrajania zapytań i fizyczną warstwę mechanizmu bazodanowego. Przedstawione zostały wewnętrzne struktury danych, narzędzia do pomiaru wydajności zapytań, metody dostępowe, oszacowania liczebności, indeksy, szeregowanie zapytań przy użyciu zdarzeń rozszerzonych, technologia magazynu kolumnowego, stosowanie tabel tymczasowych i zmiennych tablicowych, porównanie podejścia opartego na zbiorach do rozwiązań wykorzystujących kursory, dostrajanie zapytań oraz równoległe wykonywanie zapytań. (Fragment o równoległym przetwarzaniu zapytań został napisany przez Adama Machanica).

Kolejnych pięć rozdziałów zajmuje się różnymi zagadnieniami związanymi z manipulowaniem danymi. Oprócz przedstawienia i objaśnienia poszczególnych funkcjonalności, autorzy skupiają się głównie na wydajności kodu i wykorzystania omawianych funkcjonalności do rozwiązywania często spotykanych zadań. W rozdziale 3 przedstawione są zapytania wielotabelowe wykorzystujące podzapytania, operator APPLY, złączenia oraz operatory relacyjne UNION, INTERSECT i EXCEPT. Rozdział 4 omawia problematykę analizy danych przy użyciu grupowania, przestawiania (*pivoting*) oraz funkcji okna. Rozdział 5 zawiera omówienie filtrów TOP oraz OFFSET-FETCH i rozwiązanie problemów typu pierwszych *N* z grupy. W rozdziale 6 omówione zostały takie zagadnienia modyfikacji danych, jak minimalnie rejestrowane operacje, wydajne

stosowanie obiektów sekwencji, scalanie danych oraz klauzula OUTPUT. Wreszcie w rozdziale 7 omówione są zagadnienia dotyczące danych typu data i czas, włącznie z obsługą interwałów (przedziałów) czasowych.

Rozdział 8 omawia wykorzystanie języka T-SQL w praktyce BI; jego autorem jest Dejan Sarka. Omówione zostały techniki przygotowywania danych do analizy i wykorzystanie T-SQL do realizacji zadań statystycznej analizy. Przedstawione zostały zagadnienia związane z częstościami występowania, statystyki opisowe dla zmiennych ciągłych, zależności liniowe, średnie ruchome oraz entropia zbioru danych.

Rozdział 9 zawiera omówienie konstruktów programistycznych wspieranych przez T-SQL. Należą do nich dynamiczny kod SQL, funkcje definiowane przez użytkownika, procedury składowane, wyzwalacze, programowanie SQLCLR (ten fragment autorstwa Adama Machanica), transakcje i współbieżność oraz obsługa błędów. Wcześniej zagadnienia te zostały przedstawione w książce *Inside Microsoft SQL Server: T-SQL Programming*.

Rozdział 10 przedstawia najważniejsze udoskonalenie dostępne w wydaniu SQL Server 2014 – silnik In-Memory OLTP. Autorem tego rozdziału jest Kevin Farlee z firmy Microsoft, który brał udział w projektowaniu tej funkcjonalności.

Rozdział 11 zawiera omówienie grafów i zapytań rekurencyjnych. Pokazuje, jak obsługiwać struktury grafów, takie jak hierarchie pracowników, zestawienia magazynowe czy mapy w SQL Server przy użyciu języka T-SQL. Pokazuje implementację takich zagadnień, jak model wyliczanej ścieżki (przy użyciu własnego rozwiązania lub za pomocą typu danych HIERARCHYID) oraz model zagnieżdżonych zbiorów. Pokazane zostało również wykorzystanie zapytań rekurencyjnych do manipulowania danymi w grafach.

Wymagania systemowe

Do wykonania przykładów kodu zawartych w książce potrzebne są następujące składniki programowe:

- Microsoft SQL Server 2014:
  - Wydanie 64-bitowe Enterprise, Developer lub Evaluation; inne wydania nie wspierają mechanizmów In-Memory OLTP ani technologii magazynu kolumnowego, omówionych w książce. Wersję próbną można pobrać ze strony *http://www.microsoft.com/sql*.
  - Aktualne wymagania sprzętowe i programowe można sprawdzić pod adresem *http://msdn.microsoft.com/en-us/library/ms143506(v=sql.120).aspx*.
  - W oknie dialogowym Feature Selection (wybór funkcjonalności) programu instalacyjnego SQL Server 2014 należy wybrać następujące komponenty: Database Engine Services, Client Tools Connectivity, Documentation Components, Management Tools – Basic, Management Tools – Complete.

- Microsoft Visual Studio 2013 z rozszerzeniem Microsoft SQL Server Data Tools (SSDT):
  - Aktualne wymagania sprzętowe i kompatybilność platformy dla Visual Studio 2013 można znaleźć pod adresem *http://www.visualstudio.com/products/visual-studio-2013-compatibility-vs*.
  - Informacje na temat instalowania SSDT zawiera strona *http://msdn.microsoft.com/en-us/data/tools.aspx*.

Zależnie od konfiguracji systemu Windows do zainstalowania i konfiguracji SQL Server 2014 oraz Visual Studio 2013 konieczne mogą być uprawnienia lokalnego administratora.

Pobieranie przykładów kodu

Książka zawiera bardzo wiele przykładów kodu. Całość kodu źródłowego można pobrać z witryny autorów *http://tsql.solidq.com/books/tq3*.

Kod źródłowy jest udostępniony jako plik skompresowany o nazwie *T-SQL Querying – YYYYMMDD.zip*, gdzie *YYYYMMDD* wskazuje ostatnią aktualizację zawartości. Po pobraniu pliku należy wykonać instrukcje zawarte w pliku Readme.txt dołączonym do archiwum, aby zainstalować przykłady kodu.

Errata, aktualizacje i wsparcie dla książki

Dołożyliśmy wszelkich starań, aby zapewnić dokładność i poprawność tej książki, jak i towarzyszącego jej kodu przykładowego. W razie wykrycia błędu można przesłać go do wydawcy pod adresem *mspinput@microsoft.com*. Pod tym samym adresem można również skontaktować się z zespołem Microsoft Press Book Support, jeśli potrzebna jest inna pomoc. Proszę zauważyć, że pod tym adresem nie jest oferowana pomoc techniczna dla oprogramowania Microsoft. Pomoc dotycząca oprogramowania i sprzętu firmy Microsoft jest dostępna poprzez witrynę *http://support.microsoft.com*.

Bezpłatne ebooki z Microsoft Press

Począwszy od wstępnych informacji technicznej, po pogłębione omówienie szczególnych zagadnień, bezpłatne ebooki wydawnictwa Microsoft Press obejmują szeroki zakres tematyki. Publikacje te są dostępne w formatach PDF, EPUB oraz Mobi for Kindle, gotowe do pobrania z witryny:

http://aka.ms/mspressfree

Warto zaglądać tam często, aby dowiedzieć się, co jest nowego!

Podziękowania

Wiele osób przyczyniło się do tego, aby książka ta ujrzała światło dzienne, zarówno bezpośrednio, jak i pośrednio, i zasługuje na podziękowania i wyróżnienie. Bardzo możliwe, że niezamierzenie pominąłem jakieś nazwiska i z góry za to przepraszam.

Lilach: jesteś tym kimś, kto sprawia, że chcę być dobry w tym, co robię. Niezależnie od tego, że inspirujesz mnie nieustannie, miałaś też nieoficjalną rolę w powstaniu tej książki – byłaś pierwszą czytelniczką! Spędziliśmy tak wiele godzin, sczytując tekst i wyszukując błędy, zanim wysłałem go do redaktorów. Mam wrażenie, że przynajmniej niektóre aspekty T-SQL znasz lepiej, niż ludzie, którzy zajmują się nim zawodowo.

Dziękuję moim rodzicom, Mila i Gabi, i moim dzieciom, Mickey i Ina, za stałe wsparcie i zaakceptowanie mojej czasowej nieobecności. Ostatnich kilka lat było bardzo intensywnych i mam nadzieję, że nadchodzące lata będą zdrowe i szczęśliwe.

Współautorzy książki, Dejan Sarka, Adam Machanic i Kevin Farlee: było prawdziwym zaszczytem stać się częścią tak doświadczonej grupy ludzi. Każdy z was jest takim ekspertem w swojej dziedzinie, że zdecydowałem, iż odpowiednie tematy najlepiej będzie powierzyć właśnie wam. Dejan stworzył rozdział o praktycznym stosowaniu języka T-SQL w BI, Adam napisał fragmenty o równoległym przetwarzaniu zapytań i programowaniu SQL CLR, zaś Kevin rozdział o mechanizmie In-Memory OLTP. Dzięki za wzięcie udziału w tworzeniu tej książki.

Do technicznego recenzenta książki, Alejandro Mesa: czytałeś i (nieoficjalnie) recenzowałeś moje wcześniejsze książki. Podszedłeś do tematu z taką pasją, że byłem szczęśliwy, że zdecydowałeś się przyjąć bardziej oficjalną rolę recenzenta. Chcę również podziękować recenzentowi wcześniejszego wydania, Steve Kass: wykonałeś tak wspaniałą i dokładną robotę, że jej echo jest ciągle słyszalne w tej najnowszej.

Mark Souza: brałeś udział w niemal całym procesie powstawania książki, od pierwszego pomysłu, biorąc odpowiedzialność za aspekty techniczne, organizacyjne i społecznościowe. Jeśli ktokolwiek naprawdę czuje puls społeczności SQL Server, to właśnie ty. Wszyscy jesteśmy wdzięczni za to, co robisz, i zaszczytem jest to, że to ty napisałeś przedmowę.

Podziękowania należą się też wielu redaktorom w wydawnictwie Microsoft Press. Devon Musgrave, który pełnił funkcję zarówno redaktora koordynującego, jak i projektowego: to dzięki tobie książka stała się realnym bytem. Zdaję sobie sprawę, że ta książka była tylko jedną z wielu, za które jesteś odpowiedzialny i chcę podziękować za czas i wysiłki, które przeznaczyłeś właśnie na nią. Jest jeszcze kilka nazwisk, które wymienię (i ponownie przepraszam, jeśli kogoś pominąłem): Carol Dillingham, redaktor projektu, Curtis Philips, menedżer projektu z Publishing.com, Roger LeBlanc, redaktor językowy, który pracował również przy moich wcześniejszych książkach, oraz Andrea Fox, korektorka. To była prawdziwa przyjemność pracować wspólnie z wami.

Co do SolidQ, mojej firmy przez ostatnią dekadę: wspaniałe jest być częścią tak świetnej firmy, która rozwinęła się do tego, czym jest obecnie. Członkowie tej firmy

są dla mnie znacznie więcej, niż kolegami; są partnerami, przyjaciółmi i rodziną. Oto niepełna z pewnością lista tych, którym chcę podziękować w tym miejscu: Fernando G. Guerrero, Douglas McDowell, Herbert Albert, Dejan Sarka, Gianluca Hotz, Antonio Soto, Jeanne Reeves, Glenn McCoin, Fritz Lechnitz, Eric Van Soldt, Berry Walker, Marilyn Templeton, Joelle Budd, Gwen White, Jan Taylor, Judy Dyess, Alberto Martin, Lorena Jimenez, Ron Talmage, Andy Kelly, Rushabh Mehta, Joe Chang, Mark Tabladillo, Eladio Rincón, Miguel Egea, Alejandro J. Rocchi, Daniel A. Seara, Javier Loria, Paco González, Enrique Catalá, Esther Nolasco Andreu, Rocío Guerrero, Javier Torrenteras, Rubén Garrigós, Victor Vale Diaz, Davide Mauri, Danilo Dominici, Erik Veerman, Jay Hackney, Grega Jerkič, Matija Lah, Richard Waymire, Carl Rabeler, Chris Randall, Tony Rogerson, Christian Rise, Raoul Illyés, Johan Åhlén, Peter Larsson, Paul Turley, Bill Haenlin, Blythe Gietz, Nigel Semmi, Paras Doshi i tak wielu innych.

Członkowie zespołu projektowego Microsoft SQL Server, w przeszłości i obecnie: Tobias Ternstrom, Lubor Kollar, Umachandar Jayachandran (UC), Boris Baryshnikov, Conor Cunningham, Kevin Farlee, Marc Friedman, Milan Stojic, Craig Freedman, Campbell Fraser, Mark Souza, T. K. Rengarajan, Dave Campbell, César Galindo-Legaria – i niewątpliwie wielu innych. Wiem, że starania o dodanie funkcji okna do SQL Server nie były łatwe ani trywialne. Dzięki za wielki wysiłek i za cały czas, który spędziliście ze mną i na odpowiadanie na moje maile, odpowiadanie na moje pytania i udzielanie wyjaśnień.

Członkowie zespołu redakcyjnego *SQL Server Pro*, byli i obecni: Megan Keller, Lavon Peters, Michele Crockett, Mike Otey, Jayleen Heft i wielu innych. Od przeszło piętnastu lat pisuję do waszego magazynu i jestem wdzięczny za możliwość dzielenia się wiedzą z waszymi czytelnikami.

MPV dla SQL Server, w przeszłości i obecnie: Paul White, Alejandro Mesa, Erland Sommarskog, Aaron Bertrand, Tibor Karaszi, Benjamin Nevarez, Simon Sabin, Darren Green, Allan Mitchell, Tony Rogerson i wielu innych – a przede wszystkim lider MVP, Simon Tien. To wspaniały program i jestem dumny z tego, że stałem się jego częścią. Poziom wiedzy i doświadczenia waszej grupy jest zadziwiający i jestem zawsze podekscytowany, gdy dochodzi do spotkania, zarówno po to, by dzielić się pomysłami, jak i by pogadać nad kuflem piwa. Szczególne podziękowania należą się Paulowi White. Nauczyłem się od ciebie tak wiele i zawsze sprawia mi radość czytanie twoich prac. Bezpiecznie mogę powiedzieć, że jesteś moim ulubionym autorem. Kto wie, może któregoś dnia siądziemy razem do pracy nad czymś wspólnym.

Na koniec chcę podziękować moim studentom: nauczanie T-SQL jest tym, co lubię najbardziej. To moja pasja. Dzięki za to, że chcecie mnie słuchać, a przede wszystkim za wszystkie celne pytania, które skłaniają mnie do szukania większej wiedzy.

Pozdrawiam, Itzik

ROZDZIAŁ 1

Logiczne przetwarzanie zapytań

Obserwując prawdziwych ekspertów z różnych dziedzin, można zauważyć, że ich wspólną cechą jest solidna znajomość podstaw. Wszyscy profesjonaliści niezależnie od specjalizacji zajmują się w gruncie rzeczy rozwiązywaniem problemów. A wszystkie problemy, niezależnie od stopnia złożoności, wiążą się ze stosowaniem kombinacji podstawowych technik. Aby zostać ekspertem w danej dziedzinie, trzeba budować wiedzę na solidnych podstawach. Opanowanie podstaw i wkładanie dużego wysiłku w doskonalenie warsztatu pozwala na zdobycie umiejętności rozwiazywania każdego problemu.

Ta książka poświęcona jest zapytaniom Transact-SQL (T-SQL) – podstawowym technikom i ich zastosowaniu do rozwiązywania problemów. Moim zdaniem, najlepiej rozpocząć ją od rozdziału przedstawiającego podstawy logicznego przetwarzania zapytań. Rozdział ten ma kluczowe znaczenie – nie tylko dlatego, że wprowadza najważniejsze zagadnienia związane z przetwarzaniem zapytań, ale również dlatego, że programowanie w języku SQL zasadniczo różni się od programowania w innych językach.

Uwaga To trzecia edycja książki. Poprzednie wydanie składało się z dwóch tomów, które zostały połączone w jeden tom. Szczegółowe informacje o zmianach oraz instrukcję pobrania kodu źródłowego i przykładowych danych znaleźć można we Wstępie.

T-SQL to stosowany w systemie Microsoft SQL Server dialekt (lub rozszerzenie) standardów ANSI oraz ISO języka SQL. Na łamach tej książki będę zamiennie używał terminów *SQL* oraz *T-SQL*. Omawiając aspekty języka wywodzące się ze standardu ANSI/ISO SQL i odnoszące się do większości dialektów, będę zazwyczaj używać terminu *SQL* lub *standardowy język SQL*. Natomiast przy omawianiu aspektów języka związanych z implementacją systemu SQL Server, zwykle będę korzystać z terminu *T-SQL*. Warto pamiętać, że formalna nazwa języka to *Transact-SQL*, choć zwykle używa się skrótu *T-SQL*. Większość programistów uważa skrót T-SQL za wygodniejszy w użyciu, w związku z tym zdecydowałem się na stosowanie tego terminu.

Źródła wymowy skrótu SQL

Wielu angielskojęzycznych profesjonalistów zajmujących się bazami danych wymawia skrót *SQL* jako *sequel* („s-ikłel”), pomimo tego że poprawna wymowa to *S-Q-L* („es kju el”). Można postawić pewną hipotezę co do źródeł tej nieprawidłowej wymowy. Podejrzewam, że wynika ona zarówno z aspektów historycznych, jak i językowych.

Odnośnie aspektów historycznych, w latach 70-tych firma IBM opracowała język o nazwie SEQUEL, która była akronimem słów Structured English QUEry Language (Strukturalny język zapytań w języku angielskim). Język ten został zaprojektowany z myślą o manipulowaniu danymi zapisanymi w systemie bazodanowym o nazwie System R, który był oparty na opracowanym przez dr. Edgara F. Coddiego modelu RDBMS (Relational Database Management Systems – Systemy zarządzania relacyjnymi bazami danych). W późniejszym okresie z powodu sporu o zastrzeżony znak towarowy, akronim SEQUEL został skrócony do SQL. Instytut ANSI przyjął język SQL jako standard w roku 1986, a organizacja ISO w roku 1987. Instytut ANSI zadeklarował, że oficjalna wymowa skrótu SQL to „es kju el”, ale ewidentnie nie wszyscy zdają sobie z tego sprawę.

Ze względów językowych – słowo *sequel* („sikłel”) jest po prostu wygodniejsze w wymowie, w szczególności dla osób angielskojęzycznych i to wpływa na powszechne użycie tej formy.

Dodatkowe informacje Zamieszczone w tym rozdziale informacje o historii języka SQL zostały zaczerpnięte z artykułu wolnej encyklopedii Wikipedia, który dostępny jest (w języku angielskim) pod adresem *http://en.wikipedia.org/wiki/SQL*.

Programowanie w języku SQL składa się w wielu różnych aspektów, takich jak myślenie w kategoriach zbiorów, kolejność logicznego przetwarzania elementów zapytania oraz logika trójwartościowa. Próby programowania w języku SQL bez tej wiedzy prowadzą do powstawania kodu, który jest rozwlekły, mało efektywny i trudny w utrzymywaniu. Celem tego rozdziału jest zaprezentowanie języka SQL w sposób zgodny z intencjami jego twórców. Rozpocznę od zbudowania solidnych podstaw, na których opierać się będą pozostałe umiejętności. Omawiając elementy charakterystyczne dla języka T-SQL, podkreślę ten fakt.

Na łamach tej książki omawiać będę złożone problemy i zaawansowane techniki. Jednak w tym rozdziale ograniczę się do podstaw tworzenia zapytań. Wydajnością zajmę się w dalszej części książki, na razie skoncentruję się na logicznych aspektach przetwarzania zapytań. W związku z tym proszę o kompletne zignorowanie problemu wydajności podczas lektury tego rozdziału. Problematyka ta zostanie szczegółowo omówiona w dalszej

części książki. Niektóre z faz przetwarzania logicznego zapytań mogą wydawać się bardzo nieefektywne. Jednak warto mieć świadomość, że rzeczywisty sposób fizycznego przetwarzania zapytania może bardzo różnić się od przetwarzania logicznego.

W systemie SQL Server za generowanie fizycznego planu wykonania zapytania odpowiada *optymalizator zapytań*. W ramach planu optymalizator definiuje m.in. kolejność uzyskiwania dostępu do tabel i wykorzystywane do tego celu metody, zastosowane indeksy czy algorytmy złączania. Optymalizator generuje wiele poprawnych planów wykonania i wybiera spośród nich ten o najniższym koszcie. Fazy logicznego przetwarzania zapytania mają ściśle określoną kolejność. Natomiast optymalizator może stosować pewne skróty w generowanym fizycznym planie wykonywania. Oczywiście tego rodzaju skróty stosowane są tylko wówczas, gdy istnieje gwarancja, że otrzymany zbiór wynikowy będzie zbiorem poprawnym – innymi słowy, że będzie to taki sam zbiór, jaki zostałby uzyskany w wyniku ścisłego realizowania kolejnych faz logicznego przetwarzania zapytania. Na przykład, aby umożliwić wykorzystanie określonego indeksu, optymalizator może zadecydować o zastosowaniu filtra znacznie wcześniej, niż to wynika z przetwarzania logicznego. Kolejność przetwarzania elementów zgodnie z planem fizycznego wykonania zapytania nazywać będziemy *kolejnością fizycznego przetwarzania*.

Z wspomnianych powodów należy dokonać wyraźnego rozgraniczenia pomiędzy logicznym i fizycznym przetwarzaniem zapytań.

Nadeszła pora, aby zająć się szczegółowym omówieniem faz logicznego przetwarzania zapytań.

Fazy logicznego przetwarzania zapytań

W tym podrozdziale przedstawię kolejne fazy logicznego przetwarzania zapytań. Na początku pokrótce opiszę wszystkie kroki tego procesu, a następnie w kolejnych podrozdziałach omówię je bardziej szczegółowo, demonstrując ich przebieg na przykładzie konkretnego zapytania. Ten podrozdział stanowi w pewnym sensie streszczenie, do którego można się odwołać, aby szybko sprawdzić kolejność i ogólne znaczenie poszczególnych faz.

Listing 1-1 zawiera ogólną postać zapytania, wraz z numerami wskazującymi kolejność logicznego przetwarzania poszczególnych klauzul.

Listing 1-1 Ponumerowane kroki procesu logicznego przetwarzania zapytania

```
(5) SELECT (5-2) DISTINCT (7) TOP(<specyfikacja_top>) (5-1) <lista_select>
(1) FROM (1-J) <lewa_tabela> <typ_złączenia> JOIN <prawa_tabela> ON <predykat_ON>
       | (1-A) <lewa_tabela> <typ_operatora_apply> APPLY <prawa_tabela_wejściowa>
                                                          AS <alias>
       | (1-P) <lewa_tabela> PIVOT(<specyfikacja_pivot>) AS <alias>
       | (1-U) <lewa_tabela> UNPIVOT(<specyfikacja_unpivot >) AS <alias>
(2) WHERE <predykat_where >
```

```
(3) GROUP BY <specyfikacja_group_by>
(4) HAVING <predykat_having >
(6) ORDER BY <lista_order_by>
(7) OFFSET <specyfikacja_offset> ROWS FETCH NEXT <specyfikacja_fetch> ROWS ONLY;
```

Pierwszym zauważalnym aspektem języka SQL, odróżniającym go od innych języków programowania, jest kolejność przetwarzania kodu. W większości języków programowania przetwarzanie instrukcji odbywa się w kolejności zgodnej z kierunkiem tekstu (nazywanej *kolejnością wpisania*). W języku SQL proces przetwarzania rozpoczyna się od klauzuli FROM, natomiast klauzula SELECT, która została wpisana na początku, zostaje przetworzona prawie na samym końcu. Kolejność tę nazywać będę *kolejnością logicznego przetwarzania* i nie należy mylić jej z kolejnością wpisania lub kolejnością fizycznego przetwarzania.

Nie bez powodu kolejność logicznego przetwarzania różni się od kolejności wpisania. Jak wspomniałem wcześniej, SQL wywodzi się z języka SEQUEL, a pierwsza litera *E* w nazwie SEQUEL reprezentuje słowo *English* (Angielski). Projektanci języka SQL dążyli do uzyskania języka deklaratywnego, w którym instrukcje mogą być wpisywane w sposób przypominający język naturalny. Analizując sposób wydawania poleceń, np. „Przynieś mi książkę z biura", można zauważyć, że obiekt (książka) został wskazany przed określeniem jego lokalizacji (biuro). Mimo iż osoba realizująca polecenie będzie musiała najpierw dotrzeć do biura, a dopiero później wybrać książkę. Analogicznie wpisując polecenia w języku SQL, rozpoczynamy od klauzuli SELECT z wybranymi kolumnami, a dopiero później dodajemy klauzulę FROM z tabelami źródłowymi. Jednak proces logicznego przetwarzania zapytania rozpoczyna się od klauzuli FROM. Świadomość tego faktu pomaga w zrozumieniu wielu aspektów języka SQL, które wcześniej mogły wydawać się niejasne.

W każdym kroku logicznego przetwarzania zapytania generowana jest tabela wirtualna, która pełni rolę danych wejściowych kolejnego kroku. Te tabele wirtualne nie są dostępne dla obiektu wywołującego (aplikacji klienckiej lub zapytania zewnętrznego). Zwrócona zostaje tylko tabela wygenerowana w ostatnim kroku procesu. Jeśli pewna klauzula nie została użyta w zapytaniu, odpowiadający jej krok jest po prostu pomijany. W kolejnej części rozdziału pokrótce omówię poszczególne fazy przetwarzania logicznego.

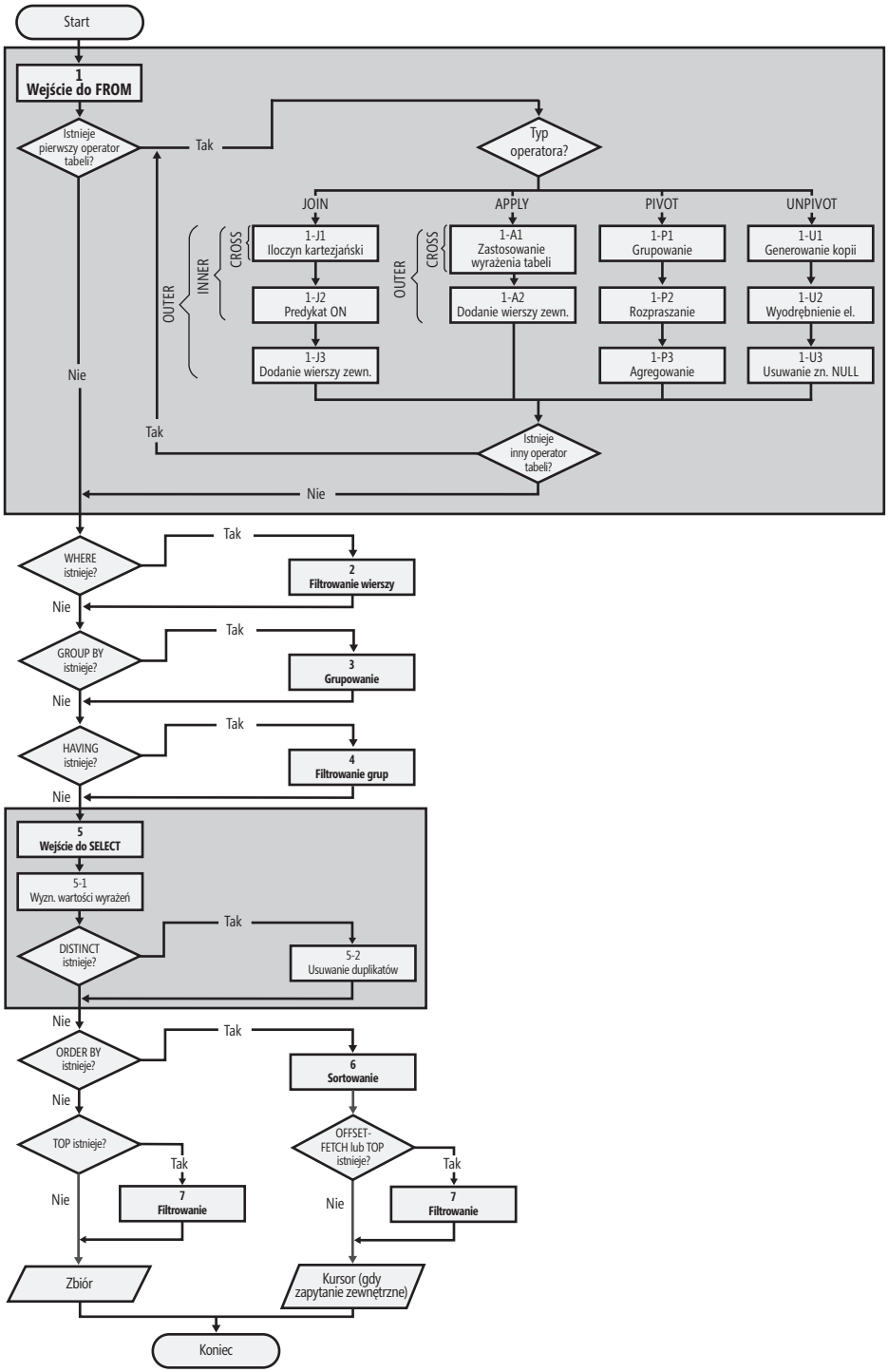
Krótkie omówienie faz logicznego przetwarzania zapytania

Opis niektórych faz może wydać się na razie mało zrozumiały, lecz nie należy się tym zbytnio przejmować. Poniższe opisy mają jednie charakter referencyjny. Poszczególne fazy zostaną omówione dużo bardziej szczegółowo po przedstawieniu przykładowego scenariusza. Opiszę jedynie operator tabel JOIN, ponieważ jest on najczęściej stosowany i powszechnie znany. Pozostałe operatory tabel (APPLY, PIVOT oraz UNPIVOT)

zostaną omówione w dalszej części rozdziału zatytułowanej „Pozostałe aspekty logicznego przetwarzania zapytań".

Na rysunku 1-1 zaprezentowany został diagram przepływu ilustrujący poszczególne fazy logicznego przetwarzania zapytania. W tym rozdziale będę wielokrotnie odwoływać się do numerów przypisanych krokom procesu na tym diagramie.

- **(1) FROM** W tej fazie następuje zidentyfikowanie źródłowych tabel zapytania i przetworzenie operatorów tabel. Realizacja każdego z operatorów tabel przebiega w serii faz podrzędnych. Np. operacja złączania JOIN składa się z faz: (1-J1) Iloczyn kartezjański, (1-J2) Predykat ON oraz (1-J3) Dodanie wierszy zewnętrznych. W tej fazie generowana jest tabela wirtualna VT1.
- **(1-J1) Iloczyn kartezjański** Ta faza polega na wykonaniu iloczynu kartezjańskiego (CROSS JOIN) pomiędzy dwiema tabelami połączonymi operatorem tabel, co skutkuje wygenerowaniem tabeli wirtualnej VT1-J1.
- **(1-J2) Predykat ON** Ta faza polega na odfiltrowaniu wierszy z tabeli VT1-J1 na podstawie predykatu zastosowanego w klauzuli ON (<*predykat_ON*>). Do generowanej tabeli VT1-J2 wstawione zostają tylko te wiersze, dla których podany predykat ma wartość TRUE.
- **(1-J3) Dodanie wierszy zewnętrznych** Jeśli użyta została klauzula OUTER JOIN (zamiast CROSS JOIN lub INNER JOIN), wiersze z zachowanej tabeli lub tabel, dla których nie został spełniony warunek dopasowania, zostaną dodane jako wiersze zewnętrzne do tabeli VT1-J2. W ten sposób wygenerowana zostanie tabela wirtualna VT1-J3.
- **(2) WHERE** W tej fazie następuje odfiltrowanie wierszy z tabeli VT1 na podstawie predykatu zastosowanego w klauzuli WHERE (<*predykat_where*>). Do tabeli VT2 wstawione zostają tylko te wiersze, dla których podany predykat ma wartość TRUE.
- **(3) GROUP BY** Ta faza polega na zorganizowaniu wierszy tabeli VT2 w grupy, na podstawie zbioru wyrażeń zdefiniowanego w klauzuli GROUP BY i nazywanego *zbiorem grupowania*. W ten sposób wygenerowana zostaje tabela VT3. Ostateczny efekt polega na utworzeniu po jednym wierszu wynikowym dla każdej grupy spełniającej określone warunki.
- **(4) HAVING** Ta faza polega na odfiltrowaniu grup z tabeli VT3 na podstawie predykatu zastosowanego w klauzuli HAVING (<*predykat_having*>). Do tabeli VT4 wstawione zostają tylko te grupy, dla których podany predykat ma wartość TRUE.
- **(5) SELECT** W tej fazie następuje przetworzenie elementów klauzuli SELECT, co skutkuje wygenerowaniem tabeli VT5.
- **(5-1) Wyznaczenie wartości wyrażeń** Ta faza polega na przetworzeniu wyrażeń zdefiniowanych na liście klauzuli SELECT, czego rezultatem jest wygenerowanie tabeli VT5-1.



Rysunek 1-1 Diagram przepływu ilustrujący logiczne przetwarzanie zapytania.

- **(5-2) DISTINCT** W tej fazie następuje usunięcie powtarzających się wierszy z tabeli VT5-1 i wygenerowanie tabeli VT5-2.
- **(6) ORDER BY** W tej fazie wiersze tabeli wirtualnej VT5-2 zostają posortowane według listy określonej w klauzuli ORDER BY, co skutkuje wygenerowaniem kursora wirtualnego VC6. W przypadku braku klauzuli ORDER BY tabela VT5-2 staje się tabelą wirtualną VT6.
- **(7) TOP | OFFSET-FETCH** Ta faza polega na odfiltrowaniu wierszy z kursora VC6 lub tabeli VT6 na podstawie specyfikacji TOP lub OFFSET-FETCH, co skutkuje wygenerowaniem odpowiednio kursora VC7 lub tabeli VT7. Jeśli zastosowana została specyfikacja TOP, w tej fazie odfiltrowana zostaje określona liczba wierszy w kolejności zdefiniowanej w klauzuli ORDER BY lub w przypadku jej braku w kolejności przypadkowej. Jeśli zastosowana została specyfikacja OFFSET-FETCH, w tej fazie pominięta zostaje określona liczba wierszy, a następnie wybrana zostaje wskazana liczba wierszy w kolejności zdefiniowanej w klauzuli ORDER BY. Filtr OFFSET-FETCH został wprowadzony w wersji SQL Server 2012.

Przykładowe zapytanie oparte na scenariuszu z użyciem tabeli klientów i zamówień

Przeanalizujemy teraz przykładowe zapytanie, które pomoże nam w zilustrowaniu poszczególnych faz procesu logicznego przetwarzania zapytań. Zaczniemy od uruchomienia pokazanego poniżej fragmentu kodu, który tworzy tabele dbo.Customers (dane klientów) i dbo.Orders (dane zamówień), wypełnia je przykładowymi danymi, a następnie wykona zapytania wyświetlające zawartość obu tabel:

```
SET NOCOUNT ON;
USE tempdb;

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

CREATE TABLE dbo.Customers
(
  custid  CHAR(5)     NOT NULL,
  city    VARCHAR(10) NOT NULL,
  CONSTRAINT PK_Customers PRIMARY KEY(custid)
);

CREATE TABLE dbo.Orders
(
  orderid INT     NOT NULL,
  custid  CHAR(5)     NULL,
  CONSTRAINT PK_Orders PRIMARY KEY(orderid),
  CONSTRAINT FK_Orders_Customers FOREIGN KEY(custid)
    REFERENCES dbo.Customers(custid)
);
GO
```

```
INSERT INTO dbo.Customers(custid, city) VALUES
  ('FISSA', 'Madrid'),
  ('FRNDO', 'Madrid'),
  ('KRLOS', 'Madrid'),
  ('MRPHS', 'Zion'  );

INSERT INTO dbo.Orders(orderid, custid) VALUES
  (1, 'FRNDO'),
  (2, 'FRNDO'),
  (3, 'KRLOS'),
  (4, 'KRLOS'),
  (5, 'KRLOS'),
  (6, 'MRPHS'),
  (7, NULL   );

SELECT * FROM dbo.Customers;
SELECT * FROM dbo.Orders;
```

W wyniku wykonania kodu wygenerowane zostaną następujące dane wyjściowe:

```
custid city
------ ----------
FISSA  Madrid
FRNDO  Madrid
KRLOS  Madrid
MRPHS  Zion

orderid     custid
----------- ------
1           FRNDO
2           FRNDO
3           KRLOS
4           KRLOS
5           KRLOS
6           MRPHS
7           NULL
```

Za przykład niech posłuży zapytanie przedstawione na Listingu 1-2. Zapytanie to zwraca listę tych klientów z miasta Madrid, którzy złożyli mniej niż trzy zamówienia (w tym również zero zamówień), wraz z informacją o liczbie złożonych przez nich zamówień. Wynik zostanie posortowany rosnąco według liczby złożonych zamówień.

Listing 1-2 Zapytanie: Klienci z miasta Madrid, którzy złożyli mniej niż trzy zamówienia

```
SELECT C.custid, COUNT(O.orderid) AS numorders
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.custid = O.custid
WHERE C.city = 'Madrid'
GROUP BY C.custid
HAVING COUNT(O.orderid) < 3
ORDER BY numorders;
```

Powyższe zapytanie zwróci następujące wiersze:

```
custid numorders
------ -----------
FISSA  0
FRNDO  2
```

FISSA i FRNDO są klientami z miasta Madrid i złożyli mniej niż trzy zamówienia. Warto przestudiować tekst przykładowego zapytania i spróbować odczytać go zgodnie z fazami przedstawionymi na Listingu 1-1, na Rysunku 1-1 oraz w podrozdziale „Krótkie omówienie faz logicznego przetwarzania zapytania". Osoby, które po raz pierwszy przeprowadzają tego typu analizę zapytania, mogą mieć z nią pewne problemy. Kolejny podrozdział powinien jednak pomóc w zrozumieniu tajników tego procesu.

Szczegółowe omówienie faz logicznego przetwarzania zapytania

Ten podrozdział zawiera szczegółowy opis faz procesu logicznego przetwarzania zapytania zilustrowany na przykładzie przedstawionego uprzednio zapytania.

Krok 1: Faza FROM

W fazie FROM identyfikowana jest tabela lub tabele, z których muszą zostać pobrane dane. Jeśli zostały zastosowane operatory tabel, są one przetwarzane w tej fazie w kolejności od lewej do prawej. Każdy operator tabeli działa na jednej lub dwóch tabelach wejściowych i zwraca jedną tabelę wyjściową. Wynik zastosowania operatora tabeli zostaje użyty w roli lewej tabeli wejściowej następnego operatora, jeśli takowy istnieje, lub jako tabela wejściowa dla następnej fazy przetwarzania logicznego. Każdy operator tabeli charakteryzuje się własnym zestawem podrzędnych faz przetwarzania. Np. operacja JOIN składa się z faz: (1-J1) Iloczyn kartezjański, (1-J2) Predykat ON oraz (1-J3) Dodanie wierszy zewnętrznych. Jak wspomniałem wcześniej, pozostałe operatory tabel opiszę w dalszej części tego rozdziału. Faza FROM prowadzi do wygenerowania wirtualnej tabeli VT1.

Krok 1-J1: Wyznaczenie iloczynu kartezjańskiego (złączenie krzyżowe)

Jest to pierwsza z trzech faz podrzędnych, występujących w przypadku użycia operatora tabeli JOIN. W tej fazie podrzędnej wyznaczony zostaje iloczyn kartezjański (złączenie krzyżowe) dwóch tabel, na których wykonywana jest operacja złączania. W rezultacie wygenerowana zostaje tabela wirtualna VT1-J1. Ta tabela zawiera po jednym wierszu dla każdej możliwej kombinacji wiersza z lewej tabeli i wiersza z prawej tabeli. Jeśli lewa tabela zawiera *n* wierszy, a prawa *m* wierszy, tabela VT1-J1 zawierać

będzie *n*×*m* wierszy. Kolumny tabeli VT1-J1 zostają zakwalifikowane (opatrzone przedrostkiem) przy użyciu nazw tabel źródłowych (lub aliasów tabel, jeśli zostały one zdefiniowane w zapytaniu). W kolejnych krokach (począwszy od kroku 1-J2) odwołania do nazwy kolumny, która nie jest jednoznaczna (tj. występuje w więcej niż jednej tabeli wejściowej), muszą zawierać nazwę tabeli (np. *C.custid*). Dodawanie nazw tabel do nazw kolumn, które pojawiają się w tylko jednej tabeli wejściowej, nie jest obowiązkowe (np. można użyć wyrażenia *O.orderid* lub po prostu *orderid*).

A teraz zastosujemy krok 1-J1 na przykładowym zapytaniu (przedstawionym na Listingu 1-2):

```
FROM dbo.Customers AS C ... JOIN dbo.Orders AS O
```

W rezultacie otrzymamy tabelę wirtualną VT1-J1 zawierającą 28 wierszy (4×7) (przedstawioną poniżej jako Tabela 1-1).

Tabela 1-1 Tabela wirtualna VT1-J1 wygenerowana w kroku 1-J1

C.custid	C.city	O.orderid	O.custid
FISSA	Madrid	1	FRNDO
FISSA	Madrid	2	FRNDO
FISSA	Madrid	3	KRLOS
FISSA	Madrid	4	KRLOS
FISSA	Madrid	5	KRLOS
FISSA	Madrid	6	MRPHS
FISSA	Madrid	7	NULL
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
FRNDO	Madrid	3	KRLOS
FRNDO	Madrid	4	KRLOS
FRNDO	Madrid	5	KRLOS
FRNDO	Madrid	6	MRPHS
FRNDO	Madrid	7	NULL
KRLOS	Madrid	1	FRNDO
KRLOS	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
KRLOS	Madrid	6	MRPHS
KRLOS	Madrid	7	NULL
MRPHS	Zion	1	FRNDO

Tabela 1-1 Tabela wirtualna VT1-J1 wygenerowana w kroku 1-J1

C.custid	C.city	O.orderid	O.custid
MRPHS	Zion	2	FRNDO
MRPHS	Zion	3	KRLOS
MRPHS	Zion	4	KRLOS
MRPHS	Zion	5	KRLOS
MRPHS	Zion	6	MRPHS
MRPHS	Zion	7	NULL

Krok 1-J2: Zastosowanie predykatu ON (warunku złączania)

Klauzula ON stanowi pierwszą z trzech klauzul zapytania (ON, WHERE lub HAVING) służących do filtrowania wierszy w oparciu o predykat. Predykat w klauzuli ON zostaje zastosowany na wszystkich wierszach tabeli wirtualnej zwróconej w poprzednim kroku (VT1-J1). Tylko wiersze, dla których wyrażenie <*predykat_ON*> ma wartość TRUE, zostają dodane do tabeli wirtualnej generowanej w tym kroku (VT1-J2).

A teraz wykonajmy krok 1-J2 dla przykładowego zapytania:

```
ON C.custid = O.custid
```

Pierwsza kolumna Tabeli 1-2 zawiera wartości wyrażenia logicznego zawartego w klauzuli ON dla wierszy tabeli wirtualnej VT1-J1.

Tabela 1-2 Wartości logiczne predykatu ON dla wierszy tabeli VT1-J1

Wartość logiczna	C.custid	C.city	O.orderid	O.custid
FALSE	FISSA	Madrid	1	FRNDO
FALSE	FISSA	Madrid	2	FRNDO
FALSE	FISSA	Madrid	3	KRLOS
FALSE	FISSA	Madrid	4	KRLOS
FALSE	FISSA	Madrid	5	KRLOS
FALSE	FISSA	Madrid	6	MRPHS
UNKNOWN	FISSA	Madrid	7	NULL
TRUE	FRNDO	Madrid	1	FRNDO
TRUE	FRNDO	Madrid	2	FRNDO
FALSE	FRNDO	Madrid	3	KRLOS
FALSE	FRNDO	Madrid	4	KRLOS
FALSE	FRNDO	Madrid	5	KRLOS
FALSE	FRNDO	Madrid	6	MRPHS

Tabela 1-2 Wartości logiczne predykatu ON dla wierszy tabeli VT1-J1

Wartość logiczna	C.custid	C.city	O.orderid	O.custid
UNKNOWN	FRNDO	Madrid	7	NULL
FALSE	KRLOS	Madrid	1	FRNDO
FALSE	KRLOS	Madrid	2	FRNDO
TRUE	KRLOS	Madrid	3	KRLOS
TRUE	KRLOS	Madrid	4	KRLOS
TRUE	KRLOS	Madrid	5	KRLOS
FALSE	KRLOS	Madrid	6	MRPHS
UNKNOWN	KRLOS	Madrid	7	NULL
FALSE	MRPHS	Zion	1	FRNDO
FALSE	MRPHS	Zion	2	FRNDO
FALSE	MRPHS	Zion	3	KRLOS
FALSE	MRPHS	Zion	4	KRLOS
FALSE	MRPHS	Zion	5	KRLOS
TRUE	MRPHS	Zion	6	MRPHS
UNKNOWN	MRPHS	Zion	7	NULL

Do tabeli wirtualnej VT1-J2 wstawione zostają tylko te wiersze, dla których <*predykat_ON*> ma wartość TRUE, jak pokazano w tabeli 1-3.

Tabela 1-3 Tabela wirtualna VT1-J2 otrzymana w kroku 1-J2

Wartość logiczna	C.custid	C.city	O.orderid	O.custid
TRUE	FRNDO	Madrid	1	FRNDO
TRUE	FRNDO	Madrid	2	FRNDO
TRUE	KRLOS	Madrid	3	KRLOS
TRUE	KRLOS	Madrid	4	KRLOS
TRUE	KRLOS	Madrid	5	KRLOS
TRUE	MRPHS	Zion	6	MRPHS

Krok 1-J3: Dodanie wierszy zewnętrznych

Ten krok wykonywany jest tylko w przypadku zastosowania złączeń zewnętrznych. Przeprowadzając złączenie zewnętrzne, można oznaczyć jedną lub obie tabele jako *zachowane*, wybierając typ złączenia zewnętrznego (LEFT, RIGHT lub FULL). Oznaczenie tabeli jako zachowanej oznacza, że zwrócone zostają wszystkie jej wiersze, nawet te niespełniające warunku <*predykat_ON*>.

Znaczniki NULL oraz logika trójwartościowa

Pozwolę sobie teraz na małą dygresję, aby zwrócić uwagę na pewne istotne aspekty języka SQL związane z tą fazą. Klasyczny model relacyjny definiuje dwa znaczniki reprezentujące brakującą wartość: znacznik A (brakująca i odnosząca się) oraz znacznik I (brakująca i nieodnosząca się). Znacznik A reprezentuje sytuację, gdy wartość odnosi się do danej relacji, ale z jakiegoś powodu jej brakuje. Weźmy pod uwagę na przykład atrybut *dataurodzenia* relacji Osoba. Mimo iż informacja ta jest znana dla większości osób, istnieją rzadkie wyjątki. Przykład może stanowić dziadek mojej żony, który oczywiście urodził się określonego dnia, ale nikt nie zna dokładnej daty. Natomiast znacznik I reprezentuje sytuację, gdy wartość nie odnosi się do danej relacji. Za przykład może posłużyć atrybut *idklienta* relacji Zamówienia. Załóżmy, że jeśli w wyniku przeprowadzenia inwentaryzacji odkryjemy niezgodność między oczekiwanym stanem magazynu a stanem faktycznym, dodamy fikcyjne zamówienie uzupełniające wykryty brak. Identyfikator klienta nie odnosi się do tego typu transakcji.

W odróżnieniu od modelu relacyjnego, standardowy język SQL nie wprowadza rozróżnienia między wspomnianymi typami brakujących wartości i oferuje tylko jeden znacznik ogólnego użytku – znacznik NULL. Znaczniki NULL bywają źródłem wielu niejasności i problemów w języku SQL, a zatem warto dobrze je zrozumieć w celu uniknięcia pomyłek. Jednym z najczęściej popełnianych błędów jest używanie określenia „wartość NULL" – NULL nie jest wartością, lecz znacznikiem reprezentującym brakującą wartość. W związku z tym prawidłowym określeniem jest „znacznik NULL" lub po prostu „NULL". Ponieważ w języku SQL brakujące wartości są reprezentowane przy pomocy tylko jednego znacznika NULL, system SQL nie wie, czy ma do czynienia z brakującą i odnoszącą się, czy też brakującą i nieodnoszącą się wartością. Na przykład w wykorzystywanej w przykładzie tabeli jedno z zamówień zawiera znacznik NULL w kolumnie *custid*. Załóżmy, że jest to fikcyjne zamówienie niezwiązane z żadnym klientem (jak w opisanym poprzednio scenariuszu). Przypisany znacznik NULL sygnalizuje brak wartości, ale nie informuje o tym, że dany atrybut nie odnosi się do tego typu zamówienia.

Stosowanie znaczników NULL w języku SQL prowadzi często do nieporozumień związanych z predykatami definiowanymi m.in. w filtrach zapytań czy ograniczeniach CHECK. W języku SQL predykat może przyjąć wartość TRUE (prawda), FALSE (fałsz) lub UNKNOWN (nieznana). Jest to tak zwana *logika trójwartościowa*, która jest charakterystyczna dla języka SQL. Wyrażenia logiczne w większości języków programowania mogą jedynie przyjmować wartość TRUE lub FALSE. Wartość logiczna UNKNOWN w języku SQL pojawia się zwykle w wyrażeniach powiązanych ze znacznikiem *NULL*. Wartość logiczną UNKNOWN

mają na przykład następujące trzy wyrażenia: *NULL* > 1759; *NULL* = *NULL*; *X* + *NULL* > *Y*. Należy pamiętać, że znacznik NULL reprezentuje brak wartości. Zgodnie ze standardem języka SQL, logiczny wynik porównania brakującej wartości z inną wartością (nawet innym znacznikiem NULL) zawsze jest nieznany (UNKNOWN).

Występowanie wyników logicznych o wartości UNKNOWN oraz znaczników NULL może prowadzić do nieporozumień. Choć wyrażenie *NOT* TRUE ma wartość FALSE, a wyrażenie NOT FALSE ma wartość TRUE, to zaprzeczenie wartości UNKNOWN (NOT UNKNOWN) jest nadal wartością UNKNOWN.

Wartości logiczne UNKNOWN oraz znaczniki NULL są traktowane w różny sposób w różnych elementach języka. Na przykład, wszystkie filtry zapytania (ON, WHERE oraz HAVING) traktują wartość UNKNOWN tak samo jak wartość FALSE. Wiersz, dla którego filtr przyjmuje wartość UNKNOWN, zostaje wyeliminowany ze zbioru wyników. Innymi słowy, filtry zapytania zwracają jedynie wystąpienia TRUE. Natomiast w ograniczeniu CHECK wartość UNKNOWN jest traktowana tak samo jak wartość TRUE. Załóżmy, że dla określonej tabeli zdefiniowaliśmy ograniczenie CHECK, zgodnie z którym wartość kolumny reprezentującej wynagrodzenie musi być większa niż zero. Wiersz ze znacznikiem NULL będzie mógł zostać wstawiony do tego typu tabeli, ponieważ wyrażenie (*NULL* > *0*) ma wartość UNKNOWN, która w ograniczeniu CHECK jest traktowana jak wartość TRUE. Innymi słowy, ograniczenie CHECK odrzuca wystąpienia FALSE.

Porównanie w filtrze dwóch znaczników NULL prowadzi do uzyskania wartości UNKNOWN, która, jak wspomnieliśmy wcześniej, jest traktowana jak wartość FALSE, jak gdyby porównywane znaczniki NULL różniły się od siebie.

Natomiast w ograniczeniach UNIQUE, niektórych operatorach relacyjnych oraz operacjach sortowania i grupowania znaczniki NULL są traktowane jak równe wartości:

- Nie można wstawić do tabeli dwóch wierszy ze znacznikiem NULL w kolumnie, dla której zdefiniowane zostało ograniczenie UNIQUE. Pod tym względem język T-SQL wyłamuje się ze standardu.
- Klauzula GROUP BY umieszcza wszystkie znaczniki NULL w jednej grupie.
- Klauzula ORDER BY sortuje wszystkie znaczniki NULL razem.
- Operatory UNION, EXCEPT oraz INTERSECT traktują znaczniki NULL jak równe wartości, porównując wiersze obu zbiorów.

Co ciekawe, w standardzie SQL proces porównywania wartości przy pomocy operatorów relacyjnych UNION, EXCEPT oraz INTERSECT został opisany nie na zasadzie porównania (= oraz <>), lecz przy pomocy relacji unikatowości (odpowiednio *NOT DISTINCT* oraz *IS DISTINCT*). Co więcej, standard definiuje jawny *predykat unikatowości* postaci: IS [NOT] DISTINCT FROM. Natomiast

wspomniane operatory relacyjne wykorzystują operator unikatowości w sposób niejawny, który różni się od predykatów bazujących na operatorach = oraz <> tym, że realizuje logikę dwuwartościową. Następujące wyrażenia są prawdziwe: *NULL IS NOT DISTINCT FROM NULL*, *NULL IS DISTINCT FROM 1759*. W wersji SQL Server 2014 język T-SQL nie wspiera jawnego predykatu unikatowości, choć mamy nadzieję, że taka możliwość zostanie dodana w przyszłości (prośba o zaimplementowanie tej funkcji została przedstawiona na stronie: *http://connect.microsoft.com/SQLServer/feedback/details/286422/*).

Podsumowując, aby uniknąć przyszłych komplikacji, warto wiedzieć, jak wartości logiczne UNKNOWN oraz znaczniki NULL są traktowane w różnych elementach języka.

Lewe złączenie zewnętrzne (LEFT OUTER JOIN) oznacza lewą tabelę jako zachowaną, prawe złączenie zewnętrzne (RIGHT OUTER JOIN) oznacza prawą tabelę, natomiast (LEFT OUTER JOIN) oznacza obie tabele. W kroku 1-J3 otrzymujemy wiersze z tabeli wirtualnej VT1-J2 wraz z wierszami zachowanej tabeli lub tabel, dla których nie odnaleziono dopasowania w kroku 1-J2. Te dodane wiersze są nazywane *wierszami zewnętrznymi*. Atrybutom niezachowanej tabeli w wierszach zewnętrznych przypisane zostają znaczniki NULL. W efekcie wygenerowana zostaje tabela wirtualna VT1-J3.

W naszym przykładzie tabela Customers została oznaczona jako zachowana:

```
Customers AS C LEFT OUTER JOIN Orders AS O
```

Tylko dla wiersza klienta FISSA nie odnaleziono żadnego dopasowania w tabeli Orders i dlatego nie został on umieszczony w tabeli wirtualnej VT1-J2. W związku z tym wiersz odpowiadający klientowi FISSA zostaje dodany do zawartości tabeli VT1-J2 ze znacznikami NULL przypisanymi atrybutom z tabeli Orders. Efekt stanowi tabela wirtualna VT1-J3 (zilustrowana przy użyciu Tabeli 1-4). Ponieważ klauzula FROM przykładowego zapytania nie zawiera więcej operatorów tabel, tabela wirtualna VT1-J3 stanowi równocześnie tabelę wirtualną VT1 zwracaną w fazie FROM.

Tabela 1-4 Tabela wirtualna VT1-J3 (a także VT1) otrzymana w kroku 1-J3

C.custid	C.city	O.orderid	O.custid
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
MRPHS	Zion	6	MRPHS
FISSA	Madrid	NULL	NULL

Uwaga Jeśli klauzula FROM zawiera więcej niż jeden operator tabeli, proces przetwarzania logicznego rozpoczyna się od lewej strony. Wynik zastosowania pierwszego operatora tabeli zostaje przekazany w postaci lewej tabeli wejściowej do kolejnego operatora tabeli. Ostatnia otrzymana w ten sposób tabela wirtualna staje się tabelą wejściową kolejnego kroku.

Krok 2: Faza WHERE

Filtr WHERE zostaje zastosowany na wszystkich wierszach tabeli wirtualnej otrzymanej w poprzednim kroku. Te wiersze, dla których *<predykat_where>* ma wartość TRUE, wchodzą w skład zwracanej w tym kroku tabeli wirtualnej (VT2).

Ostrzeżenie W tym miejscu nie można odwoływać się do aliasów kolumn zdefiniowanych na liście SELECT, ponieważ lista ta nie została jeszcze przetworzona. W związku z tym nie można napisać np. *SELECT YEAR(orderdate) AS orderyear ... WHERE orderyear > 2014*. To ograniczenie może z pozoru wydawać się usterką programu SQL Server, ale staje się jasne po zrozumieniu procesu logicznego przetwarzania zapytania. Co więcej, w tej fazie nie można również użyć agregacji, ponieważ dane nie zostały jeszcze pogrupowane – nie można napisać np. *WHERE orderdate = MAX(orderdate)*.

Zastosujemy teraz filtr z przykładowego zapytania:

```
WHERE C.city = 'Madrid'
```

Wiersz klienta o identyfikatorze MRPHS zostanie usunięty z tabeli wirtualnej VT1, ponieważ jego atrybut *city* nie ma wartości Madrid. W efekcie wygenerowana zostaje tabela wirtualna VT2 przedstawiona w tabeli 1-5.

Tabela 1-5 Tabela wirtualna VT2 otrzymana w kroku 2

C.custid	C.city	O.orderid	O.custid
FRNDO	Madrid	1	FRNDO
FRNDO	Madrid	2	FRNDO
KRLOS	Madrid	3	KRLOS
KRLOS	Madrid	4	KRLOS
KRLOS	Madrid	5	KRLOS
FISSA	Madrid	NULL	NULL

Przy definiowaniu złączenia OUTER JOIN w zapytaniu często pojawia się wątpliwość, czy predykat powinien zostać umieszczony w klauzuli ON, czy też w klauzuli WHERE. Główna różnica między tymi dwoma rozwiązaniami polega na tym,

że klauzula ON zostaje zastosowana przed dodaniem wierszy zewnętrznych (krok 1-J3), natomiast klauzula WHERE po ich dodaniu. Przeprowadzona przez klauzulę ON eliminacja wiersza z zachowanej tabeli nie jest ostateczna, ponieważ zostanie on dodany ponownie w kroku 1-J3, natomiast eliminacja wiersza dokonana przez klauzulę WHERE jest ostateczna. Można w uproszczeniu stwierdzić, że predykat ON służy do dopasowywania wierszy, natomiast predykat WHERE do ich odfiltrowywania. Wiersze pochodzące z zachowanej tabeli w operacji złączania nie mogą zostać usunięte przez klauzulę ON, ponieważ służy ona jedynie do dopasowywania wierszy z niezachowanej tabeli. Natomiast klauzula WHERE może eliminować wiersze również z zachowanej tabeli. Zapamiętanie tej głównej różnicy może pomóc w dokonywaniu trafnego wyboru.

Załóżmy na przykład, że chcemy zwrócić informacje o wybranych klientach i ich zamówieniach z tabel Customers (Klienci) oraz Orders (Zamówienia). Chcemy zwrócić tylko dane klientów powiązanych z miastem Madrid – niezależnie od tego, czy złożyli oni jakiekolwiek zamówienia. Właśnie z myślą o tego typu sytuacjach zaprojektowana została operacja złączenia zewnętrznego. Zastosujemy lewe złączenie zewnętrzne (LEFT OUTER JOIN) między tabelą Customers, która zostanie oznaczona jako zachowana oraz tabelą Orders. Aby zwrócone zostały dane także tych klientów, którzy nie złożyli żadnych zamówień, korelacja między tabelami Customers oraz Orders musi zostać zdefiniowana w klauzuli ON (*ON C.custid = O.custid*). Wiersze klientów bez zamówień zostaną wyeliminowane w kroku 1-J2, ale dodane z powrotem w postaci wierszy zewnętrznych w kroku 1-J3. Jednocześnie, ponieważ interesują nas jedynie klienci z miasta Madrid, predykat określający miasto umieszczamy w klauzuli WHERE (*WHERE C.city = 'Madrid'*). Umieszczenie go w klauzuli ON spowodowałoby, że dane klientów z miast innych niż Madrid zostałyby z powrotem dodane do zbioru wynikowego w kroku 1-J3.

Wskazówka Ta różnica logiczna między klauzulami ON oraz WHERE występuje tylko wtedy, gdy używamy złączenia zewnętrznego. W przypadku zastosowania złączenia wewnętrznego miejsce zdefiniowania predykatu nie ma znaczenia, ponieważ krok 1-J3 zostaje pominięty. Predykaty zostają wykonane bezpośrednio po sobie i oba służą do filtrowania.

Krok 3: Faza GROUP BY

W fazie GROUP BY wiersze tabeli zwróconej w poprzednim kroku zostają powiązane z grupami zgodnie z zestawem lub zestawami grupowania zdefiniowanymi w klauzuli GROUP BY. Dla uproszczenia przyjmijmy założenie, że mamy do czynienia z jednym zestawem wyrażeń określających sposób grupowania danych. Zestaw ten nazywać będziemy *zestawem grupowania*.

W tej fazie wiersze zwróconej w poprzednim kroku tabeli zostają poukładane w grupy. Dla każdej unikatowej kombinacji wartości wyrażeń z zestawu grupowania utworzona zostaje osobna grupa. Każdy zwrócony wiersz zostaje powiązany z tylko jedną grupą. Tabela wirtualna VT3 składa się z wierszy tabeli VT2 poukładanych w grupy (nazywanych *surowymi* danymi) wraz z identyfikatorami grup (nazywanymi *informacjami o grupie*).

Wykonajmy teraz krok 3 dla przykładowego zapytania:

```
GROUP BY C.custid
```

Otrzymamy tabelę wirtualną VT3 przedstawioną w tabeli 1-6.

Tabela 1-6 Tabela wirtualna VT3 otrzymana w kroku 3

Informacje o grupach	Surowe dane			
C.custid	C.custid	C.city	O.orderid	O.custid
FRNDO	FRNDO	Madrid	1	FRNDO
	FRNDO	Madrid	2	FRNDO
KRLOS	KRLOS	Madrid	3	KRLOS
	KRLOS	Madrid	4	KRLOS
	KRLOS	Madrid	5	KRLOS
FISSA	FISSA	Madrid	NULL	NULL

Ostatecznie zapytanie grupujące wygeneruje po jednym wierszu dla każdej grupy (o ile niektóre grupy nie zostaną odfiltrowane). W związku z tym, gdy zapytanie zawiera klauzulę GROUP BY, wszystkie kolejne kroki (HAVING, SELECT itd.) mogą zawierać tylko wyrażenia, które zwracają dla każdej z grup wartość skalarną (pojedynczą). Te wyrażenia mogą odwoływać się do kolumn lub wyrażeń z listy GROUP BY (jak *C.custid* w przedstawionym przykładowym zapytaniu) bądź do funkcji agregujących np. *COUNT(O.orderid)*.

Przyjrzyjmy się wirtualnej tabeli VT3 przedstawionej w postaci Tabeli 1-6 i zastanówmy się, co powinno zwrócić zapytanie dla grupy klienta FRNDO, gdyby klauzula SELECT miała postać *SELECT C.custid, O.orderid.* Grupa zawiera dwie różne wartości *orderid*, a zatem zwracane dane nie miałyby charakteru skalarnego. System SQL nie zezwala na budowanie tego typu zapytań. Jednak gdybyśmy zastosowali wyrażenie *SELECT C.custid, COUNT(O.orderid) AS numorders,* odpowiedź dla klienta FRNDO byłaby skalarna i byłaby nią liczba 2.

W tej fazie znaczniki NULL są traktowane jak równe sobie, w związku z tym wszystkie znaczniki NULL zostają umieszczone w tej samej grupie, podobnie jak znane wartości.

Krok 4: Faza HAVING

Filtr HAVING zostaje zastosowany na grupach w tabeli zwróconej w poprzednim kroku. W generowanej w tym kroku tabeli wirtualnej (VT4) umieszczone zostają tylko te grupy, dla których <*predykat_having*> przyjmuje wartość TRUE. Filtr HAVING stanowi jedyny filtr stosowany na pogrupowanych danych.

Wykonajmy ten krok dla przykładowego zapytania:

```
HAVING COUNT(O.orderid) < 3
```

Grupa związana z identyfikatorem klienta KRLOS zostaje usunięta, ponieważ zawiera trzy zamówienia. Wygenerowana zostaje tabela wirtualna VT4, przedstawiona w tabeli 1-7.

Tabela 1-7 Tabela wirtualna VT4 otrzymana w kroku 4

C.custid	C.custid	C.city	O.orderid	O.custid
FRNDO	FRNDO	Madrid	1	FRNDO
	FRNDO	Madrid	2	FRNDO
FISSA	FISSA	Madrid	NULL	NULL

Uwaga W tym przypadku musieliśmy użyć wyrażenia *COUNT(O.orderid)* zamiast *COUNT(*)*. Ponieważ zastosowaliśmy złączenie zewnętrzne, dodane zostały wiersze zewnętrzne reprezentujące klientów bez zamówień. Wyrażenie *COUNT(*)* skutkowałoby zliczeniem również tych wierszy zewnętrznych i wygenerowaniem niepożądanej wartości 1 dla klienta o identyfikatorze FISSA. Wyrażenie *COUNT(O.orderid)* prawidłowo zlicza liczbę zamówień dla każdego z klientów i skutkuje wygenerowaniem pożądanej wartości 0 dla klienta o identyfikatorze FISSA. Warto pamiętać, że wyrażenie *COUNT(<wyrażenie>)* podobnie jak inne funkcje agregujące ignoruje znaczniki NULL.

Uwaga Parametrem wejściowym funkcji agregującej nie może być zapytanie podrzędne np. *HAVING SUM((SELECT ...)) > 10*.

Krok 5: Faza SELECT

Mimo iż klauzula SELECT znajduje się na samym początku zapytania, zostaje przetworzona dopiero w piątym kroku. W fazie SELECT konstruowana jest tabela, która finalnie zostanie zwrócona do procesu wywołującego. Faza ta składa się z dwóch faz podrzędnych: (5-1) Wyznaczenie wartości wyrażeń oraz (5-2) Zastosowanie klauzuli DISTINCT.

Krok 5-1: Wyznaczenie wartości wyrażeń

Wyrażenia znajdujące się na liście SELECT mogą zwracać kolumny lub wynik przetwarzania kolumn tabeli wirtualnej, która została wygenerowana w poprzednim kroku. Należy pamiętać, że w zapytaniu agregującym po kroku 3 można odwoływać się do kolumn z poprzedniego kroku tylko wtedy, gdy zostały one umieszczone w zbiorze informacji o grupie (na liście GROUP BY). Do kolumn należących do surowych danych można odwołać się tylko wtedy, gdy zostały one zagregowane. Kolumny pobrane z tabeli wygenerowanej w poprzednim kroku zachowują oryginalne nazwy, o ile nie użyto aliasu (np. *col1 AS c1*). Wyrażenia przetwarzające kolumny powinny zostać opatrzone aliasem decydującym o tym, jaka nazwa kolumny zostanie zastosowana w tabeli wynikowej np. *YEAR(orderdate) AS orderyear.*

Ważne Jak wspomniałem wcześniej, aliasów utworzonych przez listę SELECT nie można wykorzystywać we wcześniejszych krokach np. w fazie WHERE. Powody stają się całkiem oczywiste po zrozumieniu kolejności logicznego przetwarzania klauzul zapytania. Co więcej, aliasy wyrażeń nie mogą być stosowane nawet w innych wyrażeniach z tej samej listy SELECT. To ograniczenie wynika z innej charakterystycznej cechy języka SQL, a mianowicie równoczesnego wykonywania wielu operacji. Na przykład, kolejność logicznego przetwarzania wyrażeń z następującej listy SELECT nie powinna mieć znaczenia i nie jest gwarantowana: *SELECT c1 + 1 AS e1, c2 + 1 AS e2*. W związku z tym nie można użyć następującego wyrażenia: *SELECT c1 + 1 AS e1, e1 + 1 AS e2*. Aliasy kolumn mogą być stosowane wyłącznie w krokach realizowanych po kroku, w którym zostały one utworzone. Jeśli zdefiniujemy alias kolumny w fazie SELECT, możemy odwołać się do niego w fazie ORDER BY np. *SELECT YEAR(orderdate) AS orderyear ... ORDER BY orderyear.*

Zrozumienie koncepcji równoczesnego wykonania operacji może być trudne. W większości języków programowania do zamiany wartości między zmiennymi wykorzystywana jest zmienna tymczasowa. Natomiast do zamiany wartości kolumn w języku SQL wystarczy poniższe zapytanie:

```
UPDATE dbo.T1 SET c1 = c2, c2 = c1;
```

Z logicznego punktu widzenia należy przyjąć założenie, że cała operacja odbywa się w jednej chwili, tak jakby tabela źródłowa nie była modyfikowana do czasu zakończenia całej operacji, kiedy to wynik zastępuje dane źródłowe. Analogiczny mechanizm powoduje, że następujące zapytanie UPDATE służy do modyfikacji wszystkich wierszy tabeli T1 w taki sposób, że do wartości kolumny c1 dodana zostaje maksymalna wartość kolumny c1 tabeli T1 z momentu rozpoczęcia procesu modyfikacji:

```
UPDATE dbo.T1 SET c1 = c1 + (SELECT MAX(c1) FROM dbo.T1);
```

Nie trzeba się obawiać, że maksymalna wartość c1 ulegnie zmianie w czasie wykonywania operacji. Nie ma takiego ryzyka, ponieważ cała operacja zostaje wykonana w jednej chwili.

Zrealizujmy ten krok dla przykładowego zapytania:

```
SELECT C.custid, COUNT(O.orderid) AS numorders
```

Otrzymamy tabelę wirtualną VT5-1 przedstawioną w tabeli 1-8. Ponieważ przykładowe zapytanie nie wymaga wykonania dodatkowej fazy podrzędnej (DISTINCT) fazy SELECT, tabela wirtualna VT5-1 wygenerowana w tej fazie podrzędnej stanowi jednocześnie tabelę wirtualną VT5 zwróconą z fazy SELECT.

Tabela 1-8 Tabela wirtualna VT5-1 (a także VT5) otrzymana w kroku 5

C.custid	numorders
FRNDO	2
FISSA	0

Krok 5-2: Zastosowanie klauzuli DISTINCT

Jeśli w zapytaniu określona została klauzula DISTINCT, z wygenerowanej w poprzednim kroku tabeli wirtualnej usunięte zostają duplikaty wierszy i w ten sposób utworzona zostaje tabela wirtualna VT5-2.

Uwaga Język SQL, w odróżnieniu od modelu relacyjnego, zezwala na występowanie powtarzających się wierszy w tabeli (o ile nie zdefiniowano klucza głównego bądź ograniczenia UNIQUE) lub zbiorze wyników zapytania. Natomiast w modelu relacyjnym treścią relacji jest zbiór *krotek* (w języku SQL nazywanych *wierszami*), a zgodnie z teorią mnogości *zbiór* (w odróżnieniu od *wielozbioru*) nie może zawierać dwóch (lub więcej) identycznych elementów. Zastosowanie klauzuli DISTINCT gwarantuje, że zapytanie zwróci jedynie unikatowe wiersze i pod tym względem spełni założenia modelu relacyjnego.

Warto mieć świadomość, że faza SELECT rozpoczyna się od wyznaczenia wartości wyrażeń (faza podrzędna 5-1) i dopiero później usuwa zduplikowane wiersze (faza podrzędna 5-2). Brak tej wiedzy utrudnia przewidywanie, jaki wynik zwrócą pewnego rodzaju zapytania. Na przykład poniższe zapytanie służy do pobrania unikatowych identyfikatorów klientów, którzy złożyli przynajmniej jedno zamówienie:

```
SELECT DISTINCT custid
FROM dbo.Orders
WHERE custid IS NOT NULL;
```

Jak można się spodziewać, zapytanie zwróci trzy wiersze:

```
custid
------
FRNDO
KRLOS
MRPHS
```

A teraz załóżmy, że chcemy dodać do wynikowych wierszy numerację bazującą na kolejności wartości w kolumnie *custid*. Nie zdając sobie sprawy z problemu, dodajemy do zapytania wyrażenie bazujące na funkcji ROW_NUMBER:

```
SELECT DISTINCT custid, ROW_NUMBER() OVER(ORDER BY custid) AS rownum
FROM dbo.Orders
WHERE custid IS NOT NULL;
```

Zachęcam czytelników, aby przed uruchomieniem zapytania spróbowali zgadnąć, ile wierszy zawierać będzie zbiór wyników. Z pozoru mogłoby się wydawać, że prawidłowa odpowiedź to 3, jednak w praktyce jest inaczej. Faza 5-1 wyznacza wartość wszystkich wyrażeń, także tego bazującego na funkcji ROW_NUMBER, jeszcze przed usunięciem duplikatów. W tym przykładzie w fazie SELECT przetworzona zostaje tabela VT4 zawierająca sześć wierszy (po zastosowaniu filtra WHERE). Natomiast w fazie 5-1 wygenerowana zostaje tabela VT5-1 z sześcioma wierszami o numerach od 1 do 6. Dodanie numerów wierszy sprawia, że wszystkie wiersze są unikatowe. W związku z tym w fazie 5-2 nie zostają znalezione żadne duplikaty do usunięcia. Oto wynik wykonania tego zapytania:

```
custid rownum
------ --------------------
FRNDO  1
FRNDO  2
KRLOS  3
KRLOS  4
KRLOS  5
MRPHS  6
```

Aby zastosować numerację na wierszach o unikatowym identyfikatorze klienta, trzeba pozbyć się duplikatów przed przypisaniem numeru wiersza. Cel ten można osiągnąć przy pomocy wyrażenia tabeli, takiego jak wspólne wyrażenie tabeli (Common Table Expression – CTE), na przykład:

```
WITH C AS
(
  SELECT DISTINCT custid
  FROM dbo.Orders
  WHERE custid IS NOT NULL
)
SELECT custid, ROW_NUMBER() OVER(ORDER BY custid) AS rownum
FROM C;
```

Tym razem uzyskujemy pożądany efekt:

```
custid rownum
------ --------------------
FRNDO  1
KRLOS  2
MRPHS  3
```

W procesie przetwarzania pierwotnego przykładowego zapytania (przedstawionego na listingu 1-2) krok 5-2 został pominięty ze względu na brak klauzuli DISTINCT. Nawet gdyby została ona zastosowana, w tym konkretnym przykładzie i tak nie spowodowałaby usunięcia żadnych wierszy.

Krok 6: Faza ORDER BY

Wiersze otrzymane w poprzednim kroku zostają posortowane zgodnie z listą kolumn określoną w klauzuli ORDER BY, co skutkuje zwróceniem kursora VC6. Jest to jedyna faza, w której można skorzystać z aliasów kolumn utworzonych w fazie SELECT.

Jeśli zdefiniowana została klauzula DISTINCT, wyrażenia w klauzuli ORDER BY mają dostęp jedynie do tabeli wirtualnej wygenerowanej w fazie SELECT (VT5). W przeciwnym przypadku wyrażenia w klauzuli ORDER BY mogą uzyskać dostęp zarówno do wyjściowej, jak i wejściowej tabeli wirtualnej fazy SELECT (VT4 oraz VT5). Co więcej, brak klauzuli DISTINCT oznacza, że można umieścić w klauzuli ORDER BY dowolne wyrażenie, które mogłoby wystąpić w klauzuli SELECT. Innymi słowy, można przeprowadzać sortowanie według wyrażeń, które nie zostają zwrócone w finalnym zbiorze wyników.

Nie bez powodu dostęp do wyrażeń, które nie są zwracane, nie jest dozwolony w przypadku określenia klauzuli DISTINCT. W przypadku dodania wyrażeń do listy SELECT klauzula DISTINCT może potencjalnie zmienić liczbę zwracanych wierszy. Natomiast brak klauzuli DISTINCT sprawia, że lista SELECT nie wpływa na liczbę zwracanych wierszy.

W naszym przykładzie klauzula DISTINCT nie została zastosowana, w związku z tym klauzula ORDER BY ma dostęp zarówno do tabeli VT4 (przedstawionej w tabeli 1-7), jak i tabeli VT5 (przedstawionej w tabeli 1-8).

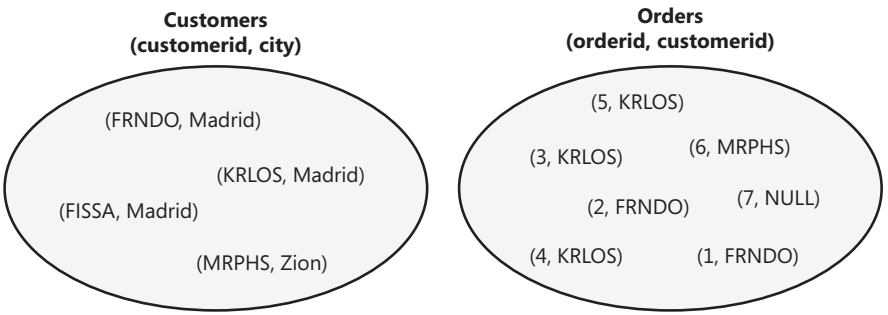
Ważne Ten krok różni się od pozostałych tym, że zamiast relacyjnego zbioru wyników zwrócony zostaje kursor. Należy pamiętać, że model relacyjny bazuje na teorii zbiorów, a treścią relacji (nazywanej w języku SQL *tabelą*) jest zbiór krotek (wierszy). Kolejność elementów zbioru nie jest określona. Natomiast wynik zapytania z klauzulą prezentacji ORDER BY zawiera wiersze uporządkowane w określonej kolejności. W standardzie języka SQL tego typu wynik nazywany jest *kursorem*. Zrozumienie tej różnicy stanowi klucz do właściwego zrozumienia języka SQL.

W klauzuli ORDER BY można również użyć numeru porządkowego kolumny wynikowej na liście SELECT. Na przykład, następujące zapytanie sortuje wiersze tabeli Orders według kolumny *custid* i dodatkowo *orderid*:

```
SELECT orderid, custid FROM dbo.Orders ORDER BY 2, 1;
```

Jednak praktyka ta nie jest zalecana ze względu na ryzyko, że osoba modyfikująca listę SELECT zapomni o odpowiednim dostosowaniu listy ORDER BY. Ponadto długi ciąg zapytania utrudnia sprawdzanie, który element na liście ORDER BY odpowiada któremu elementowi na liście SELECT.

Opisując zawartość tabeli, zwykle przedstawiamy listę jej wierszy. Jednak taki sposób prezentacji może być mylący, ponieważ sygnalizuje istnienie pewnego porządku, podczas gdy tabela stanowi zbiór wierszy o nieokreślonej kolejności. Rysunek 1-2 demonstruje przykład bardziej prawidłowego sposobu prezentowania zawartości tabel bez sugerowania kolejności.



Rysunek 1-2 Zbiory Customers oraz Orders.

Uwaga Język SQL nie określa kolejności *wierszy* tabeli, aczkolwiek definiuje pozycje porządkowe *kolumn* wynikające z kolejności ich utworzenia. Użycie wyrażenia *SELECT ** (niezalecanego ze względów, które zostaną wyjaśnione w dalszej części książki) sprawia, że kolumny zostają zwrócone w kolejności ich utworzenia. Pod tym względem standard SQL odbiega od modelu relacyjnego.

Ponieważ w tym kroku zamiast tabeli zostaje zwrócony kursor, zapytanie, które zawiera klauzulę ORDER BY służącą wyłącznie do określania porządku prezentacji, nie może zostać wykorzystane do definiowania wyrażenia tzn. widoku, wbudowanej funkcji zwracającej tabelę, tabeli pochodnej lub wspólnego wyrażenia tabeli. Wynik zapytania musi zostać przekazany do procesu wywołującego, który wspiera przetwarzanie kolejnych rekordów po jednym na raz. Tego typu procesem może być aplikacja kliencka lub kod SQL wykorzystujący obiekt CURSOR. Gdy spróbujemy użyć kursora w procesie, który spodziewa się relacyjnych danych wejściowych, wykonanie zakończy się niepowodzeniem. Na przykład poniższa próba zdefiniowania tabeli pochodnej skutkować będzie wygenerowaniem błędu:

```
SELECT orderid, custid
FROM ( SELECT orderid, custid
       FROM dbo.Orders
       ORDER BY orderid DESC  ) AS D;
```

Nieprawidłowy jest również poniższy kod SQL stanowiący próbę zdefiniowania widoku:

```
CREATE VIEW dbo.MyOrders
AS

SELECT orderid, custid
FROM dbo.Orders
ORDER BY orderid DESC;
GO
```

W języku T-SQL wyjątek od reguły, która zabrania definiowania wyrażeń tabeli bazujących na zapytaniu z klauzulą ORDER BY, stanowią zapytania z filtrem TOP lub OFFSET-FETCH. Omówieniem tego wyjątku zajmę się po przedstawieniu wspomnianych filtrów.

Klauzula ORDER BY traktuje znaczniki NULL jak równe sobie, co oznacza, że zajmują one tę samą pozycję w porządku sortowania. Standard SQL pozostawia swobodę w decydowaniu, czy mechanizm sortowania umieszcza znaczniki NULL po lub przed znanymi wartościami, ale działanie to musi być spójne. W języku T-SQL znaczniki NULL są uznawane za mniejsze od znanych wartości (czyli poprzedzają je).

Wykonajmy ten krok dla przykładowego zapytania z listingu 1-2:

```
ORDER BY numorders
```

Otrzymamy kursor VC6 zaprezentowany w tabeli 1-9.

Tabela 1-9 Kursor VC6 otrzymany w kroku 6

C.custid	numorders
FISSA	0
FRNDO	2

Krok 7: Zastosowanie filtra TOP lub OFFSET-FETCH

TOP oraz OFFSET-FETCH stanowią filtry zapytania oparte na liczbie wierszy i kolejności. To odróżnia je od bardziej tradycyjnych filtrów zapytania (ON, WHERE oraz HAVING), które bazują na predykacie. Filtr TOP jest charakterystyczny dla języka T-SQL, natomiast filtr OFFSET-FETCH został zdefiniowany w standardzie. Jak wspomniano wcześniej, filtr OFFSET-FETCH został wprowadzony w wersji SQL Server 2012. W tej części rozdziału filtry te zostaną omówione w kontekście logicznego przetwarzania zapytań.

Specyfikacja filtra TOP składa się z dwóch elementów. Pierwszy z nich (wymagany) to liczba lub procent zwracanych wierszy (zaokrąglany w górę). Drugi z nich (opcjonalny) definiuje kolejność decydującą o tym, które wiersze zostaną zwrócone. Niestety filtr TOP (a także OFFSET-FETCH) bazuje na kolejności prezentacji określonej przy

pomocy klauzuli ORDER BY zapytania, zamiast na niezależnej definicji kolejności. Za chwilę wyjaśnię, dlaczego takie rozwiązanie projektowe przysparza problemów i prowadzi do nieporozumień. Jeśli porządek nie został zdefiniowany, należy przyjąć założenie, że jest on przypadkowy, co prowadzi do uzyskiwania niedeterministycznego wyniku. W związku z tym, uruchamiając zapytanie ponownie, nie mamy gwarancji otrzymania tego samego wyniku, nawet jeśli dane nie zostały zmodyfikowane w międzyczasie. W tym kroku wygenerowana zostaje tabela wirtualna VT7. Jeśli zdefiniowany został porządek, wygenerowany zostaje kursor VC7. Na przykład następujące zapytanie zwraca trzy zamówienia o najwyższych wartościach *orderid*:

```
SELECT TOP (3) orderid, custid
FROM dbo.Orders
ORDER BY orderid DESC;
```

Zapytanie spowoduje wygenerowanie poniższego wyniku:

```
orderid     custid
----------- ------
7           NULL
6           MRPHS
5           KRLOS
```

Nawet zastosowanie klauzuli ORDER BY nie gwarantuje determinizmu. Jeśli sortowane wartości nie są unikatowe (np. gdybyśmy w poprzednim zapytaniu zastąpili atrybut *orderid* atrybutem *custid*), kolejność wierszy z taką samą wartością sortowania powinna być traktowana jako przypadkowa. Istnieją dwie techniki zapewniania deterministycznego działania filtra TOP. Pierwsza z nich polega na zastosowaniu listy ORDER BY gwarantującej unikatowość (np. *custid, orderid*). Natomiast druga polega na dodaniu do nieunikatowej listy ORDER BY opcji WITH TIES. Ta opcja sprawia, że filtr dołącza wszystkie wiersze z wygenerowanego zbioru wyników zapytania, w których wartości sortowania są takie same jak w ostatnim wierszu zwracanym bez tej opcji. Użycie tej opcji sprawia, że wybór zwracanych wierszy jest deterministyczny, w odróżnieniu od kolejności ich prezentowania, która nadal może się zmieniać w przypadku istnienia powtarzających się wartości sortowania.

Filtr OFFSET-FETCH przypomina filtr TOP, lecz zawiera dodatkową możliwość określenia, ile wierszy ma zostać pominiętych (OFFSET), poprzedzającą opcję definiującą liczbę wierszy do pobrania (FETCH). Klauzule OFFSET oraz FETCH muszą zostać umieszczone po klauzuli ORDER BY. Na przykład, następujące zapytanie definiuje sortowanie w kolejności malejącej według atrybutu *orderid*, pomija cztery pierwsze wiersze i zwraca dwa kolejne wiersze:

```
SELECT orderid, custid
FROM dbo.Orders
ORDER BY orderid DESC
OFFSET 4 ROWS FETCH NEXT 2 ROWS ONLY;
```

Powyższe zapytanie zwróci następujący wynik:

```
orderid     custid
----------- ------
3           KRLOS
2           FRNDO
```

W wersji SQL Server 2014 implementacja filtra OFFSET-FETCH w języku T-SQL nie wspiera jeszcze opcji PERCENT oraz WITH TIES, mimo iż zostały one udokumentowane w standardzie języka SQL. W tym kroku generowany jest kursor VC7.

Krok 7 został w naszym przykładzie pominięty, ponieważ kwerenda nie zawiera filtra TOP ani OFFSET-FETCH.

Jak wspomniałem wcześniej, do definiowania porządku filtra TOP oraz OFFSET-FETCH służy ta sama klauzula ORDER BY, która zwykle jest wykorzystywana do określania porządku prezentacji, co może prowadzić do nieporozumień. Przeanalizujmy następujące przykładowe zapytanie:

```
SELECT TOP (3) orderid, custid
FROM dbo.Orders
ORDER BY orderid DESC;
```

W tym przypadku klauzula ORDER BY służy do dwóch różnych celów. Po pierwsze, do definiowania porządku prezentacji (czyli prezentowania zwracanych wierszy w kolejności malejącej według atrybutu *orderid*). Po drugie, do określenia, które trzy wiersze mają zostać wybrane przez filtr TOP (trzy wiersze o najwyższej wartości *orderid*). Problem pojawia się, gdy chcemy zdefiniować wyrażenie tabeli bazujące na tego typu zapytaniu. Jak wspomniałem w części zatytułowanej „Krok 6: Faza ORDER BY", zazwyczaj nie można definiować wyrażenia tabeli bazującego na zapytaniu z klauzulą ORDER BY, ponieważ dla tego typu zapytania generowany jest kursor. Pokazałem, że próby zdefiniowania tabeli pochodnej lub widoku zakończą się niepowodzeniem, gdy wewnętrzne zapytanie zawiera klauzulę ORDER BY.

Istnieje wyjątek od tej reguły, ale, aby uniknąć przyszłych rozczarowań, trzeba go dobrze zrozumieć. Wyjątek zezwala na zastosowanie w wewnętrznym zapytaniu klauzuli ORDER BY, wspierającej filtr TOP lub OFFSET-FETCH. Należy pamiętać, że jeśli w takiej sytuacji zapytanie zewnętrzne nie zawiera klauzuli ORDER BY, kolejność prezentowania wierszy w zbiorze wyników nie jest gwarantowana. Zdefiniowana kolejność odnosi się jedynie do zapytania zawierającego klauzulę ORDER BY. Aby określić porządek prezentacji wierszy w zbiorze wyników, trzeba umieścić klauzulę ORDER BY w najbardziej zewnętrznym zapytaniu. Oto odnoszący się do tego faktu cytat z dokumentacji standardu ISO/IEC 9075-2:2011(E), sekcja 4.15.3 zatytułowana Derived Tables:

> *<Wyrażenie zapytania> może zawierać opcjonalną <klauzulę order by>. Kolejność wierszy tabeli określona przez <wyrażenie zapytania> jest gwarantowana tylko dla <wyrażenia zapytania>, które bezpośrednio zawiera <klauzulę order by>.*

Pomimo iż z pozoru może wydawać się inaczej, w poniższym przykładzie kolejność prezentowania wierszy w zbiorze wyników nie jest gwarantowana, ponieważ zewnętrzne zapytanie nie zawiera klauzuli ORDER BY:

```
SELECT orderid, custid
FROM ( SELECT TOP (3) orderid, custid
       FROM dbo.Orders
       ORDER BY orderid DESC          ) AS D;
```

Może się zdarzyć, że po uruchomieniu powyższego zapytania otrzymamy wynik, którego wiersze wydają się być posortowane zgodnie ze specyfikacją porządku zdefiniowaną w wewnętrznym zapytaniu. Jednak nie ma gwarancji, że sytuacja ta się powtórzy. Być może wynika ona ze sposobu optymalizacji zapytania, który może ulec zmianie. Jednym z błędów najczęściej popełnianych przez programistów SQL jest wyciąganie wniosków i budowanie oczekiwań na podstawie obserwowanego działania, zamiast wiedzy teoretycznej.

Inny częsty przykład niezrozumienia teorii stanowi próba utworzenia „posortowanego widoku". Jak wspomniałem, użycie opcji TOP lub OFFSET-FETCH umożliwia dołączenie klauzuli ORDER BY do wewnętrznego zapytania. Niektórzy programiści próbują wykorzystać tę możliwość, stosując filtr, który nie eliminuje żadnych wierszy, za pomocą opcji TOP (100) PERCENT:

```
-- Próba stworzenia posortowanego widoku
IF OBJECT_ID(N'dbo.MyOrders', N'V') IS NOT NULL DROP VIEW dbo.MyOrders;
GO
--Uwaga: kod nie tworzy posortowanego widoku!
CREATE VIEW dbo.MyOrders
AS

SELECT TOP (100) PERCENT orderid, custid
FROM dbo.Orders
ORDER BY orderid DESC;
GO
```

Wykonajmy na utworzonym widoku następujące zapytanie:

```
SELECT orderid, custid FROM dbo.MyOrders;
```

Jak wyjaśniłem, jeśli zapytanie wykonywane na takim widoku nie zawiera klauzuli ORDER BY, żadna kolejność prezentacji nie jest gwarantowana. Po uruchomieniu tego zapytania w moim systemie uzyskałem następującą kolejność wierszy:

```
orderid     custid
----------- ------
1           FRNDO
2           FRNDO
3           KRLOS
4           KRLOS
5           KRLOS
```

```
6           MRPHS
7           NULL
```

Jak widać, wiersze zbioru wyników nie zostały posortowane malejąco według atrybutu *orderid*. W tym przypadku optymalizator zapytań w systemie SQL Server zauważył, że może zignorować filtr wraz ze specyfikacją porządku, ponieważ zapytanie zewnętrzne nie zawiera klauzuli ORDER BY, a wyrażenie *TOP (100) PERCENT* nie odfiltrowuje żadnych wierszy.

Niektóre osoby próbują utworzyć posortowany widok, stosując klauzulę OFFSET z pominięciem 0 wierszy oraz bez klauzuli FETCH w następujący sposób:

```
-- Próba utworzenia posortowanego widoku
IF OBJECT_ID(N'dbo.MyOrders', N'V') IS NOT NULL DROP VIEW dbo.MyOrders;
GO
-- Uwaga: kod nie tworzy posortowanego widoku!
CREATE VIEW dbo.MyOrders
AS

SELECT orderid, custid
FROM dbo.Orders
ORDER BY orderid DESC
OFFSET 0 ROWS;
GO
--Wykonanie zapytania na widoku
SELECT orderid, custid FROM dbo.MyOrders;
```

Po uruchomieniu tego zapytania w moim systemie uzyskałem następującą kolejność wierszy:

```
orderid     custid
----------- ------
7           NULL
6           MRPHS
5           KRLOS
4           KRLOS
3           KRLOS
2           FRNDO
1           FRNDO
```

Wynik wydaje się być posortowany w kolejności malejącej według *orderid*, ale należy pamiętać, że takie działanie nie jest gwarantowane. Wynika ono jedynie z faktu, że w tym przypadku optymalizator nie zawiera jeszcze zasady, która pozwoliłaby mu zignorować filtr o tej specyfikacji.

Tego typu nieporozumień można byłoby uniknąć, gdyby filtry zostały zaprojektowane w taki sposób, aby umożliwiać stosowanie osobnej specyfikacji porządku, niezależnej od klauzuli kolejności prezentacji ORDER BY. Przykład dobrego projektu zapewniającego taką separację stanowią funkcje okna np. ROW_NUMBER. Funkcje okna pozwalają na definiowanie porządku służącego jedynie do wyznaczania rankingu – niezależnie od porządku prezentacji określonego w zapytaniu. Jednak

w przypadku filtrów TOP oraz OFFSET-FETCH jest już za późno. W związku z tym to na nas, programistach, spoczywa odpowiedzialność za dobre zrozumienie istniejącego projektu i odpowiednie dostosowanie oczekiwań.

Na zakończenie uruchomimy następujący kod w celu usunięcia niepotrzebnych obiektów:

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

Pozostałe aspekty logicznego przetwarzania zapytań

W tym podrozdziale omówię pozostałe aspekty logicznego przetwarzania zapytań, takie jak operatory tabel (JOIN, APPLY, PIVOT i UNPIVOT), funkcje okna oraz dodatkowe operatory relacyjne (UNION, EXCEPT i INTERSECT). Omawiane elementy języka stanowią złożone zagadnienia, lecz w tym rozdziale skoncentruję się na aspektach związanych z logicznym przetwarzaniem zapytań. W związku z tym czytelnicy, którzy nie mają doświadczenia w stosowaniu omawianych elementów języka (np. PIVOT, UNPIVOT lub APPLY), mogą mieć problemy ze zrozumieniem prezentowanych informacji. W dalszej części książki przedstawię bardziej szczegółowe omówienie, z uwzględnieniem m.in. zastosowań oraz kwestii wydajnościowych. Po zapoznaniu się z tymi informacjami warto powrócić do tego rozdziału, aby lepiej zrozumieć proces logicznego przetwarzania zapytań zawierających dany element języka.

Operatory tabeli

Język T-SQL zezwala na umieszczanie w klauzuli FROM zapytania czterech operatorów tabeli: JOIN, APPLY, PIVOT oraz UNPIVOT.

Uwaga Operatory tabeli APPLY, PIVOT oraz UNPIVOT nie wchodzą w skład standardu, lecz stanowią rozszerzenia charakterystyczne dla języka T-SQL. Tym niemniej, operator APPLY, w odróżnieniu od operatorów PIVOT oraz UNPIVOT, ma w standardzie odpowiednik noszący nazwę LATERAL.

W poprzedniej części rozdziału omówiłem fazy logicznego przetwarzania związane z operatorem JOIN. W tej części pokrótce przedstawię trzy pozostałe operatory oraz ich miejsce w modelu logicznego przetwarzania zapytań.

Operatory tabel otrzymują na wejściu jedną lub dwie tabele. W zależności od położenia tabeli względem słowa kluczowego operatora (*JOIN*, *APPLY*, *PIVOT*, *UNPIVOT*) jest ona nazywana *lewą tabelą wejściową* lub *prawą tabelą wejściową*. Rolę tabel wejściowych mogą odgrywać m.in.: zwykła tabela, tabela tymczasowa, zmienna tabeli, tabela

pochodna, wspólne wyrażenie tabeli, widok lub funkcja zwracająca tabelę. W procesie logicznego przetwarzania zapytań operatory tabeli są stosowane od lewej do prawej strony; tabela wirtualna zwrócona przez jeden operator tabeli staje się lewą tabelą wejściową kolejnego operatora tabeli.

Każdy operator tabeli wymaga zrealizowania innego zestawu kroków. Dla wygody oraz zwiększenia czytelności numer każdego z kroków poprzedziłem pierwszą literą operatora tabeli (J dla JOIN, A dla APPLY, P dla PIVOT oraz U dla UNPIVOT).

Oto cztery operatory tabeli i ich składowe:

```
(J) <lewa_tabela_wejściowa>
      {CROSS | INNER | OUTER} JOIN <prawa_tabela_wejściowa>
       ON <predykat_ON>

(A) <lewa_tabela_wejściowa>
      {CROSS | OUTER} APPLY <prawa_tabela_wejściowa>

(P) <lewa_tabela_wejściowa>
      PIVOT (<funkcja_agregująca(<element_agregacji>)> FOR
        <element_rozpraszania> IN(<lista_kolumn_docelowych>))
        AS <alias_tabeli_wyników>

(U) <lewa_tabela_wejściowa>
      UNPIVOT (<kolumna_wartości_docelowych> FOR
        <kolumna_nazw_docelowych> IN(<lista_kolumn_źródłowych>))
        AS <alias_tabeli_wyników>
```

Dla przypomnienia, operacja złączania wymaga wykonania podzbioru następujących kroków (w zależności od typu złączenia):

- J1: Wyznaczenie iloczynu kartezjańskiego
- J2: Zastosowanie predykatu ON
- J3: Dodanie wierszy zewnętrznych

APPLY

Operator APPLY (w zależności od typu) wymaga zrealizowania jednego lub obu poniższych kroków:

1. A1: Zastosowanie prawego wyrażenia tabeli na wierszach lewej tabeli
2. A2: Dodanie wierszy zewnętrznych

Operator APPLY stosuje prawe wyrażenie tabeli na każdym wierszu lewej tabeli wejściowej. Prawe wyrażenie tabeli może odwoływać się do kolumn lewej tabeli wejściowej. Wartość prawego wyrażenia tabeli zostaje wyznaczona jednokrotnie dla każdego wiersza lewej tabeli. Krok ten prowadzi do ujednolicenia zbiorów poprzez dopasowanie każdego wiersza lewej tabeli do odpowiadających mu wierszy prawego wyrażenia tabeli i zwrócenie łącznego wyniku.

Krok A1 zostaje wykonany niezależnie od tego, czy zastosowany został operator CROSS APPLY, czy OUTER APPLY. Natomiast krok A2 odnosi się tylko do operatora OUTER APPLY. Operator CROSS APPLY nie zwraca zewnętrznego wiersza (z lewej tabeli), jeśli wewnętrzne (prawe) wyrażenie tabeli zwróci dla niego pusty zbiór. Natomiast operator OUTER APPLY zwraca tego typu wiersze, wstawiając znaczniki NULL w miejscu atrybutów wewnętrznego wyrażenia tabeli.

Na przykład następujące zapytanie zwraca dla każdego klienta po dwa zamówienia o najwyższej wartości *orderid*:

```
SELECT C.custid, C.city, A.orderid
FROM dbo.Customers AS C
  CROSS APPLY
    ( SELECT TOP (2) O.orderid, O.custid
      FROM dbo.Orders AS O
      WHERE O.custid = C.custid
      ORDER BY orderid DESC ) AS A;
```

Przedstawione zapytanie wygeneruje następujący wynik:

```
custid     city       orderid
---------- ---------- -----------
FRNDO      Madrid     2
FRNDO      Madrid     1
KRLOS      Madrid     5
KRLOS      Madrid     4
MRPHS      Zion       6
```

Jak widać, zbiór wyników nie zawiera danych klienta o identyfikatorze FISSA, ponieważ wyrażenie tabeli A zwróciło dla niego pusty zbiór. Aby uwzględnić także dane tych klientów, którzy nie złożyli żadnych zamówień, należy zastosować operator OUTER APPLY w następujący sposób:

```
SELECT C.custid, C.city, A.orderid
FROM dbo.Customers AS C
  OUTER APPLY
    ( SELECT TOP (2) O.orderid, O.custid
      FROM dbo.Orders AS O
      WHERE O.custid = C.custid
      ORDER BY orderid DESC ) AS A;
```

Przedstawione zapytanie wygeneruje poniższy wynik:

```
custid     city       orderid
---------- ---------- -----------
FISSA      Madrid     NULL
FRNDO      Madrid     2
FRNDO      Madrid     1
KRLOS      Madrid     5
KRLOS      Madrid     4
MRPHS      Zion       6
```

PIVOT

Operator PIVOT służy do obracania lub, innymi słowy, przestawiania (ang. *pivot*) danych z wierszy do kolumn z jednoczesnym wykonaniem agregacji.

Załóżmy, że chcemy wykonać zapytanie na widoku Sales.OrderValues z przykładowej bazy danych TSQLV3 (Wstęp do książki zawiera szczegółowe informacje o przykładowej bazie danych) i zwrócić całkowitą wartość zamówień obsługiwanych przez każdego z pracowników w poszczególnych latach. Wynik ma zawierać po jednym wierszu dla każdego pracownika z kolumnami dla każdego roku działalności i całkowitą wartością na przecięciu wiersza pracownika z kolumną roku. Cel ten można osiągnąć przy pomocy zapytania z operatorem PIVOT:

```
USE TSQLV3;

SELECT empid, [2013], [2014], [2015]
FROM ( SELECT empid, YEAR(orderdate) AS orderyear, val
       FROM Sales.OrderValues ) AS D
  PIVOT( SUM(val) FOR orderyear IN([2013],[2014],[2015]) ) AS P;
```

Przedstawione zapytanie wygeneruje następujący wynik:

```
empid  2013      2014       2015
------ --------- ---------- ---------
9      9894.52   26310.39   41103.17
3      18223.96  108026.17  76562.75
6      16642.61  43126.38   14144.16
7      15232.16  60471.19   48864.89
1      35764.52  93148.11   63195.02
4      49945.12  128809.81  54135.94
5      18383.92  30716.48   19691.90
2      21757.06  70444.14   74336.56
8      22240.12  56032.63   48589.54
```

Zapytanie podrzędne służy do wygenerowania tabeli pochodnej D i nie należy się nim przejmować. Wystarczy mieć świadomość, że operator PIVOT otrzymuje w roli lewej tabeli wejściowej wyrażenie tabeli o nazwie D, zawierające po jednym wierszu dla każdego zamówienia, przy czym każdy wiersz zawiera identyfikator pracownika (*empid*), rok zamówienia (*orderyear*) oraz wartość zamówienia (*val*). Działanie operatora PIVOT można podzielić na następujące fazy logiczne:

1. P1: Grupowanie
2. P2: Rozpraszanie
3. P3: Agregowanie

Pierwsza faza (P1) jest dość skomplikowana. Jak widać na podstawie zapytania, operator PIVOT wykorzystuje dwie kolumny tabeli D w roli argumentów wejściowych (*val* oraz *orderyear*). W pierwszej fazie wiersze z tabeli D zostają w niejawny sposób pogrupowane według wszystkich kolumn, które nie zostały zastosowane w roli argumentów

wejściowych operatora PIVOT, jak gdyby zapytanie zawierało ukrytą klauzulę GROUP BY. W tym przypadku tylko kolumna *empid* nie pełni roli argumentu wejściowego operatora PIVOT, w związku z tym otrzymujemy grupy dla każdego pracownika.

Uwaga Faza niejawnego grupowania przez operator PIVOT nie ma wpływu na jawne deklaracje klauzuli GROUP BY w zapytaniu. Operacja PIVOT prowadzi do utworzenia wirtualnej tabeli wyników przekazywanej do kolejnej fazy logicznej, którą może być inny operator tabeli lub faza WHERE. Jak wspomniałem wcześniej, faza GROUP BY może następować po fazie WHERE. W związku z tym, gdy zapytanie zawiera zarówno operator PIVOT, jak i klauzulę GROUP BY, następują dwie osobne fazy grupowania – wcześniejsza stanowi pierwszą fazę przetwarzania operatora PIVOT (P1), a późniejsza fazę przetwarzania klauzuli GROUP BY zapytania.

Druga faza przetwarzania operatora PIVOT (P2) wiąże się z rozproszeniem wartości *<kolumny_rozpraszania>* do odpowiadających jej kolumn docelowych. Z logicznego punktu widzenia polega ona na zastosowaniu następującego wyrażenia CASE na każdej kolumnie docelowej określonej w klauzuli IN:

```
CASE WHEN <kolumna_rozpraszania> = <element_kolumny_docelowej> THEN <wyrażenie> END
```

Omawiany przykład wiąże się z logicznym zastosowaniem trzech wyrażeń:

```
CASE WHEN orderyear = 2013 THEN val END,
CASE WHEN orderyear = 2014 THEN val END,
CASE WHEN orderyear = 2015 THEN val END
```

Uwaga Wyrażenie CASE bez klauzuli ELSE zawiera niejawną klauzulę ELSE NULL.

Dla każdej kolumny docelowej wyrażenie CASE zwróci wartość (kolumnę *val*), gdy wiersz docelowy zawiera odpowiadający jej rok zamówienia oraz znacznik NULL w przeciwnym przypadku.

Trzecia faza przetwarzania operatora PIVOT (P3) polega na zastosowaniu określonej funkcji agregacji na każdym wyrażeniu CASE, co prowadzi do wygenerowania kolumn wynikowych. W naszym przykładzie wyrażenia przyjmują następującą postać:

```
SUM(CASE WHEN orderyear = 2013 THEN val END) AS [2013],
SUM(CASE WHEN orderyear = 2014 THEN val END) AS [2014],
SUM(CASE WHEN orderyear = 2015 THEN val END) AS [2015]
```

Podsumowując, przedstawione uprzednio zapytanie z operatorem PIVOT stanowi logiczny odpowiednik poniższego zapytania:

```
SELECT empid,
  SUM(CASE WHEN orderyear = 2013 THEN val END) AS [2013],
```

```
  SUM(CASE WHEN orderyear = 2014 THEN val END) AS [2014],
  SUM(CASE WHEN orderyear = 2015 THEN val END) AS [2015]
FROM ( SELECT empid, YEAR(orderdate) AS orderyear, val
       FROM Sales.OrderValues ) AS D
GROUP BY empid;
```

UNPIVOT

Jak można się było przekonać, operator PIVOT przestawia dane z wierszy do kolumn. Dla kontrastu, operator UNPIVOT przestawia dane z kolumn do wierszy (operacja ta nazywana jest odwrotnym przestawianiem).

Zanim zademonstruję fazy logicznego przetwarzania operatora UNPIVOT, uruchomimy następujący kod w celu utworzenia i wypełnienia tabeli dbo.EmpYearValues, a następnie wyświetlenia jej zawartości przy użyciu zapytania:

```
SELECT empid, [2013], [2014], [2015]
INTO dbo.EmpYearValues
FROM ( SELECT empid, YEAR(orderdate) AS orderyear, val
       FROM Sales.OrderValues ) AS D
  PIVOT( SUM(val) FOR orderyear IN([2013],[2014],[2015]) ) AS P;

UPDATE dbo.EmpYearValues
  SET [2013] = NULL
WHERE empid IN(1, 2);

SELECT empid, [2013], [2014], [2015] FROM dbo.EmpYearValues;
```

Kod zwróci następujący wynik:

```
empid       2013       2014       2015
----------- ---------- ---------- ----------
3           18223.96   108026.17  76562.75
6           16642.61   43126.38   14144.16
9           9894.52    26310.39   41103.17
7           15232.16   60471.19   48864.89
1           NULL       93148.11   63195.02
4           49945.12   128809.81  54135.94
2           NULL       70444.14   74336.56
5           18383.92   30716.48   19691.90
8           22240.12   56032.63   48589.54
```

Poniższe zapytanie posłuży jako przykład podczas opisywania faz przetwarzania logicznego związanych z operatorem UNPIVOT:

```
SELECT empid, orderyear, val
FROM dbo.EmpYearValues
  UNPIVOT( val FOR orderyear IN([2013],[2014],[2015]) ) AS U;
```

Zapytanie dokonuje odwrotnego przestawienia (innymi słowy rozdzielenia) wartości rocznych dla pracowników z każdego wiersza źródłowego do osobnego wiersza dla każdego roku. W efekcie osiągniemy następujący rezultat:

```
empid       orderyear  val
----------- ---------- -----------
3           2013       18223.96
3           2014       108026.17
3           2015       76562.75
6           2013       16642.61
6           2014       43126.38
6           2015       14144.16
9           2013       9894.52
9           2014       26310.39
9           2015       41103.17
7           2013       15232.16
7           2014       60471.19
7           2015       48864.89
1           2014       93148.11
1           2015       63195.02
4           2013       49945.12
4           2014       128809.81
4           2015       54135.94
2           2014       70444.14
2           2015       74336.56
5           2013       18383.92
5           2014       30716.48
5           2015       19691.90
8           2013       22240.12
8           2014       56032.63
8           2015       48589.54
```

Operacja UNPIVOT wiąże się z następującymi trzema fazami przetwarzania logicznego:

1. U1: Generowanie kopii
2. U2: Wyodrębnianie elementu
3. U3: Usuwanie wierszy ze znacznikami NULL

W pierwszym kroku generowane są kopie wierszy (U1) z lewego wejściowego wyrażenia tabeli operatora UNPIVOT (w tym przypadku EmpYearValues). Kopia zostaje wygenerowana dla każdej kolumny, która ma zostać poddana odwrotnemu przestawieniu (została wymieniona w klauzuli IN operatora UNPIVOT). Ponieważ klauzula IN zawiera trzy nazwy kolumn, każdy wiersz źródłowy posłuży do utworzenia trzech kopii. Wynikowa tabela wirtualna będzie zawierała nową kolumnę z nazwami kolumn źródłowych w postaci ciągów znaków. Nazwa kolumny zostaje zdefiniowana tuż przed klauzulą IN (w naszym przykładzie *orderyear*). Tabela wirtualna generowana w pierwszym kroku naszego przykładu została przedstawiona w tabeli 1-10.

Tabela 1-10 Tabela wirtualna otrzymana w pierwszym kroku operacji UNPIVOT

empid	2013	2014	2015	orderyear
3	18223.96	108026.17	76562.75	2013
3	18223.96	108026.17	76562.75	2014

Tabela 1-10 Tabela wirtualna otrzymana w pierwszym kroku operacji UNPIVOT

empid	2013	2014	2015	orderyear
3	18223.96	108026.17	76562. 75	2015
6	16642.61	43126.38	14144.16	2013
6	16642.61	43126.38	14144.16	2014
6	16642.61	43126.38	14144.16	2015
9	9894.52	26310.39	41103.17	2013
9	9894.52	26310.39	41103.17	2014
9	9894.52	26310.39	41103.17	2015
7	15232.16	60471.19	48864.89	2013
7	15232.16	60471.19	48864.89	2014
7	15232.16	60471.19	48864.89	2015
1	NULL	93148.11	63195.02	2013
1	NULL	93148.11	63195.02	2014
1	NULL	93148.11	63195.02	2015
4	49945.12	128809.81	54135.94	2013
4	49945.12	128809.81	54135.94	2014
4	49945.12	128809.81	54135.94	2015
2	NULL	70444.14	74336.56	2013
2	NULL	70444.14	74336.56	2014
2	NULL	70444.14	74336.56	2015
5	18383.92	30716.48	19691.90	2013
5	18383.92	30716.48	19691.90	2014
5	18383.92	30716.48	19691.90	2015
8	22240.12	56032.63	48589.54	2013
8	22240.12	56032.63	48589.54	2014
8	22240.12	56032.63	48589.54	2015

Drugi krok (U2) polega na wyodrębnieniu z kolumny źródłowej wartości, która odpowiada przestawianemu odwrotnie elementowi reprezentowanemu przez bieżącą kopię wiersza. Nazwa kolumny docelowej, w której umieszczane są wartości, została zdefiniowana przed klauzulą FOR (w naszym przypadku *val*). Kolumna docelowa będzie zawierała wartość z kolumny źródłowej odpowiadającą wartości *orderyear* bieżącego wiersza tabeli wirtualnej. Tabela wirtualna generowana w tym kroku dla omawianego przykładu została zilustrowana w tabeli 1-11.

Tabela 1-11 Tabela wirtualna otrzymana w drugim kroku operacji UNPIVOT

empid	val	orderyear
3	18223.96	2013
3	108026.17	2014
3	76562.75	2015
6	16642.61	2013
6	43126.38	2014
6	14144.16	2015
9	9894.52	2013
9	26310.39	2014
9	41103.17	2015
7	15232.16	2013
7	60471.19	2014
7	48864.89	2015
1	NULL	2013
1	93148.11	2014
1	63195.02	2015
4	49945.12	2013
4	128809.81	2014
4	54135.94	2015
2	NULL	2013
2	70444.14	2014
2	74336.56	2015
5	18383.92	2013
5	30716.48	2014
5	19691.90	2015
8	22240.12	2013
8	56032.63	2014
8	48589.54	2015

Trzeci i ostatni krok przetwarzania logicznego operatora UNPIVOT (U3) polega na usunięciu wierszy ze znacznikami NULL w kolumnie wartości wynikowych (w naszym przykładzie *val*). Tabela wirtualna otrzymana po zrealizowaniu tego kroku dla naszego przykładu została przedstawiona w tabeli 1-12.

Tabela 1-12 Tabela wirtualna otrzymana w trzecim kroku operacji UNPIVOT

empid	val	orderyear
3	18223.96	2013
3	108026.17	2014
3	76562.75	2015
6	16642.61	2013
6	43126.38	2014
6	14144.16	2015
9	9894.52	2013
9	26310.39	2014
9	41103.17	2015
7	15232.16	2013
7	60471.19	2014
7	48864.89	2015
1	93148.11	2014
1	63195.02	2015
4	49945.12	2013
4	128809.81	2014
4	54135.94	2015
2	70444.14	2014
2	74336.56	2015
5	18383.92	2013
5	30716.48	2014
5	19691.90	2015
8	22240.12	2013
8	56032.63	2014
8	48589.54	2015

Na zakończenie uruchamimy poniższy kod w celu usunięcia niepotrzebnych obiektów:

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

Funkcje okna

Funkcje okna służą do wykonywania dla każdego wiersza analizy danych na odpowiednim podzbiorze (*oknie*) powiązanego zbioru wyników zapytania. Klauzula OVER służy do definiowania specyfikacji okna. Język T-SQL wspiera wiele różnych typów funkcji okna: agregacje, rankingi, funkcje przesunięcia oraz funkcje statystyczne. Podobnie jak w przypadku pozostałych funkcji omawianych w tym rozdziale, skoncentruję się na aspektach związanych z przetwarzaniem logicznym.

Jak wspomniałem, podzbiór, na którym stosowana jest funkcja okna, pochodzi z powiązanego zbioru wyników zapytania. Z perspektywy logicznego przetwarzania zapytania zbiór wyników zostaje wygenerowany dopiero po osiągnięciu fazy SELECT (5), wcześniej ulega on ciągłym zmianom. Z tego względu funkcje okna są ograniczone do faz SELECT (5) oraz ORDER BY (6), jak widać na listingu 1-3.

Listing 1-3 Funkcje okna w procesie logicznego przetwarzania zapytania

```
(5) SELECT (5-2) DISTINCT (7) TOP(<specyfikacja_top>) (5-1) <select_list>
(1) FROM (1-J) <lewa_tabela> <typ_złączenia> JOIN <prawa_tabela> ON <predykat_ON>
       | (1-A) <lewa_tabela> <typ_operatora_apply> APPLY <prawa_tabela_wejściowa>
                                                    AS <alias>
       | (1-P) <lewa_tabela> PIVOT(<specyfikacja_pivot>) AS <alias>
       | (1-U) <lewa_tabela> UNPIVOT(<specyfikacja_unpivot>) AS <alias>
(2) WHERE <predykat_where>
(3) GROUP BY <specyfikacja_group_by>
(4) HAVING <predykat_having>
(6) ORDER BY <lista_order_by>
(7) OFFSET <specyfikacja_offset> ROWS FETCH NEXT <specyfikacja_fetch> ROWS ONLY;
```

Choć nie omówiłem jeszcze działania funkcji okna, chciałbym zademonstrować ich zastosowanie w obu dozwolonych fazach. Lista SELECT w poniższym przykładowym zapytaniu zawiera funkcję agregacji okna o nazwie *COUNT*:

```
USE TSQLV3;

SELECT orderid, custid,
  COUNT(*) OVER(PARTITION BY custid) AS numordersforcust
FROM Sales.Orders
WHERE shipcountry = N'Spain';
```

Zapytanie wygeneruje następujący wynik:

```
orderid     custid      numordersforcust
----------- ----------- ----------------
10326       8           3
10801       8           3
10970       8           3
10928       29          5
10568       29          5
```

```
10887       29          5
10366       29          5
10426       29          5
10550       30          10
10303       30          10
10888       30          10
10911       30          10
10629       30          10
10872       30          10
10874       30          10
10948       30          10
11009       30          10
11037       30          10
11013       69          5
10917       69          5
10306       69          5
10281       69          5
10282       69          5
```

Przed zastosowaniem jakichkolwiek ograniczeń zbiór początkowy, na którym operuje funkcja okna, stanowi tabela wirtualna przekazana w postaci tabeli wejściowej do fazy SELECT (a dokładniej VT4). Klauzula partycji oka ogranicza wiersze ze zbioru początkowego do tych, które mają taką samą wartość atrybutu *custid* jak bieżący wiersz. Funkcja *COUNT(*)* wyznacza liczbę wierszy w tym zbiorze. Należy pamiętać, że tabela wirtualna przetwarzana w fazie SELECT przeszła już fazę filtra WHERE, w związku z tym wybrane zostały tylko zamówienia z atrybutem *shipcountry* o wartości Spain.

Specyfikację funkcji okna można również umieszczać na liście ORDER BY. Na przykład następujące zapytanie sortuje wiersze według całkowitej liczby wierszy wyjściowych powiązanych z klientem, czyli po prostu liczby zamówień złożonych przez tego klienta (w kolejności malejącej):

```
SELECT orderid, custid,
  COUNT(*) OVER(PARTITION BY custid) AS numordersforcust
FROM Sales.Orders
WHERE shipcountry = N'Spain'
ORDER BY COUNT(*) OVER(PARTITION BY custid) DESC;
```

Wykonanie przedstawionego zapytania przyniesie następujący efekt:

```
orderid     custid      numordersforcust
----------- ----------- ----------------
10550       30          10
10303       30          10
10888       30          10
10911       30          10
10629       30          10
10872       30          10
10874       30          10
10948       30          10
11009       30          10
11037       30          10
```

```
11013       69          5
10917       69          5
10306       69          5
10281       69          5
10282       69          5
10928       29          5
10568       29          5
10887       29          5
10366       29          5
10426       29          5
10326       8           3
10801       8           3
10970       8           3
```

Operatory UNION, EXCEPT oraz INTERSECT

Ta część rozdziału koncentruje się na aspektach związanych z logicznym przetwarzaniem operatorów UNION (w wersji ALL oraz w wersji z niejawnym wymogiem unikatowości), EXCEPT oraz INTERSECT. Te operatory przypominają analogiczne operatory zdefiniowane w matematycznej teorii mnogości (odpowiednio sumę, różnicę i część wspólną zbiorów), jednak w kontekście języka SQL odnoszą się do relacji, które stanowią szczególny typ zbioru. Listing 1-4 demonstruje ogólną postać zapytania zawierającego jeden ze wspomnianych operatorów. Dodana numeracja wskazuje, w jakiej kolejności dokonywane jest logiczne przetwarzanie poszczególnych elementów kodu.

Listing 1-4 Ogólna postać zapytania z operatorem UNION, EXCEPT lub INTERSECT

```
(1) zapytanie1
(2) <operator>
(1) zapytanie2
(3) [ORDER BY <lista_order_by>]
```

Operatory UNION, EXCEPT oraz INTERSECT porównują pełne wiersze dwóch tabel wejściowych. Język T-SQL wspiera dwa rodzaje operatorów UNION. Operator UNION ALL zwraca jeden zbiór wyników zawierający wszystkie wiersze z obu tabel wejściowych. Natomiast operator UNION zwraca jeden zbiór wyników zawierający wszystkie niepowtarzające się wiersze z obu tabel wejściowych (bez duplikatów). Operator EXCEPT zwraca unikatowe wiersze należące do pierwszej tabeli wejściowej, ale niepojawiające się w drugiej tabeli wejściowej. Natomiast operator INTERSECT zwraca unikatowe wiersze występujące w obu tabelach wejściowych.

Cząstkowe zapytania nie mogą zawierać klauzuli ORDER BY, ponieważ powinny zwracać zbiory (nieuporządkowane). Tym niemniej można umieścić klauzulę ORDER BY na końcu zapytania, co spowoduje, że zostanie ona zastosowana na wyniku działania operatora.

Z perspektywy logicznego przetwarzania każde zapytanie wejściowe jest przetwarzane osobno (wraz ze wszystkimi odpowiednimi fazami, oprócz niedozwolonej fazy ORDER BY). Następnie zastosowany zostaje operator, a na wygenerowanym przez niego zbiorze wyników wykonana zostaje klauzula ORDER BY, o ile została ona zdefiniowana.

Przeanalizujmy następujące przykładowe zapytanie:

```
USE TSQLV3;

SELECT region, city
FROM Sales.Customers
WHERE country = N'USA'

INTERSECT

SELECT region, city
FROM HR.Employees
WHERE country = N'USA'

ORDER BY region, city;
```

Przedstawione zapytanie wygeneruje następujący wynik:

```
region          city
--------------- ---------------
WA              Kirkland
WA              Seattle
```

Na początku każde wejściowe zapytanie zostaje osobno przetworzone, przechodząc przez wszystkie niezbędne fazy logicznego przetwarzania. Pierwsze zapytanie zwróci informacje o regionie i mieście (*region*, *city*) dla klientów z USA. Natomiast drugie zwróci te same informacje dla pracowników z USA. Operator INTERSECT zwróci unikatowe wiersze, które występują w obu tabelach wejściowych – w naszym przykładzie nazwy regionów i miast, w których zamieszkują zarówno klienci, jak i pracownicy. Na zakończenie klauzula ORDER BY posortuje wiersze według atrybutów *region* oraz *city*.

Inny przykład faz logicznego przetwarzania stanowi poniższe zapytanie z operatorem EXCEPT służącym do wybrania klientów, którzy nie złożyli żadnych zamówień:

```
SELECT custid FROM Sales.Customers
EXCEPT
SELECT custid FROM Sales.Orders;
```

Pierwsze zapytanie zwraca zbiór identyfikatorów klientów z tabeli Customers, natomiast drugie zbiór identyfikatorów klientów z tabeli Orders. Operator EXCEPT zwraca podzbiór tych wierszy pierwszego zbioru, które nie występują w drugim zbiorze. Warto pamiętać, że zbiór nie zawiera duplikatów, w związku z tym operator EXCEPT zwraca tylko unikatowe wystąpienia wierszy pierwszego zbioru, które nie należą do drugiego zbioru.

Nazwy kolumn zbioru wyników zostają wyznaczone na podstawie pierwszej tabeli wejściowej operatora. Kolumny znajdujące się na tych samych pozycjach muszą mieć zgodne typy danych. Porównując wiersze, operatory UNION, EXCEPT oraz INTERSECT w niejawny sposób wykorzystują predykat unikatowości (znacznik NULL nie różni się od innego znacznika NULL, ale różni się od wartości innej niż NULL).

Podsumowanie

Każdy specjalista SQL powinien rozumieć proces logicznego przetwarzania zapytań oraz cechy charakterystyczne języka SQL. Znajomość tych aspektów języka pozwala na tworzenie właściwych rozwiązań i uzasadnianie dokonywanych wyborów. Należy pamiętać, że klucz do mistrzostwa leży w opanowaniu podstaw.

ROZDZIAŁ 2

Optymalizowanie zapytań

Rozdział 1 „Logiczne przetwarzanie zapytań” poświęcony był logicznemu projektowi języka SQL. Natomiast ten rozdział koncentruje się na jego fizycznej implementacji na platformie Microsoft SQL Server. Ponieważ język SQL oparty jest na międzynarodowym standardzie, podstawowe elementy języka wyglądają tak samo w różnych systemach bazodanowych. Jednak warstwa fizyczna nie bazuje na standardzie, dlatego to w niej można zaobserwować większe różnice między poszczególnymi platformami. W związku z tym, aby opanować sztukę optymalizowania zapytań, trzeba dokładnie poznać warstwę fizyczną wykorzystywanego systemu. W tym rozdziale omówię różne aspekty warstwy fizycznej programu SQL Server, których znajomość stanowi klucz do optymalizowania zapytań.

Optymalizowanie zapytań wiąże się z analizowaniem ich planów wykonania oraz dokonywaniem pomiarów wydajności przy użyciu różnych narzędzi. Kluczowy etap tego procesu stanowi próba zidentyfikowania najbardziej pracochłonnych aktywności w planie oraz objęcia ich pomiarem wydajności. Po rozwiązaniu tego problemu można podjąć działania w celu wyeliminowania lub zawężenia aktywności, których wykonanie wymaga najwięcej pracy.

Aby zrozumieć generowany plan wykonania zapytania, trzeba bardzo dobrze znać metody dostępu. Z kolei kluczem do zrozumienia metod dostępu jest bardzo dobra znajomość wewnętrznych struktur danych wykorzystywanych przez SQL Server.

Na początku rozdziału zaprezentuję omówienie wewnętrznych struktur danych. Następnie opiszę narzędzia służące do mierzenia wydajności zapytania. Po omówieniu tych zagadnień przedstawię szczegółowy przegląd metod dostępu oraz analizę powiązanych z nimi planów wykonania zapytania. Inne poruszone w tym rozdziale tematy to szacowanie liczebności, funkcje indeksowania, wybieranie zapytań wymagających optymalizacji, statystyki i informacje o indeksie oraz zapytaniu, tabele tymczasowe, porównanie rozwiązań iteracyjnych z tymi bazującymi na zbiorach, optymalizowanie zapytań poprzez ich korygowanie oraz równoległe wykonywanie zapytań.

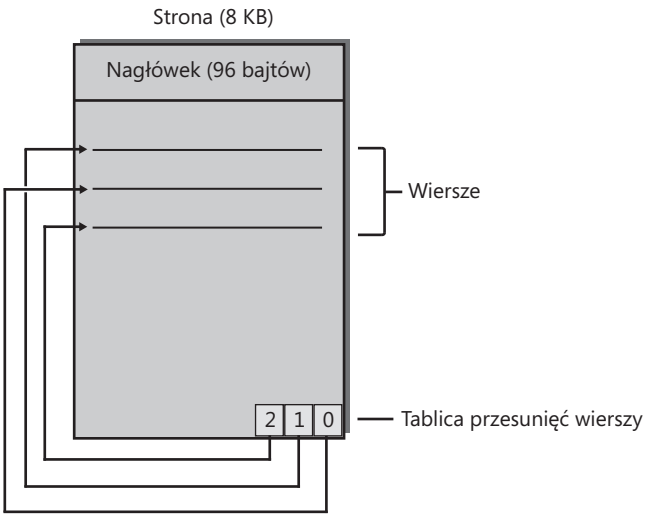
Niniejszy rozdział koncentruje się na tradycyjnych tabelach przechowywanych na dysku. Tabele zoptymalizowane pamięciowo zostaną omówione w rozdziale 10, „In-Memory OLTP”.

Struktury wewnętrzne

Ten podrozdział koncentruje się na wewnętrznych strukturach danych w programie SQL Server. Zacznę od pojęć stron oraz fragmentów (ang. *extent*). Następnie przedstawię porównanie tabel o strukturze sterty z tabelami o strukturze B-drzewa. Na koniec omówię również indeksy nieklastrowe.

Strony i fragmenty

Strona stanowi podstawową jednostkę przechowywania danych (o wielkości 8 KB) w platformie SQL Server. W przypadku tabel przechowywanych na dysku strona stanowi najmniejszą jednostkę we/wy, którą SQL Server może odczytywać lub zapisywać. Sytuacja wygląda inaczej w przypadku tabel zoptymalizowanych pamięciowo, ale jak wspomniałem wcześniej, to zagadnienie zostanie omówione osobno w rozdziale 10. Struktura strony została zilustrowana na rysunku 2-1.



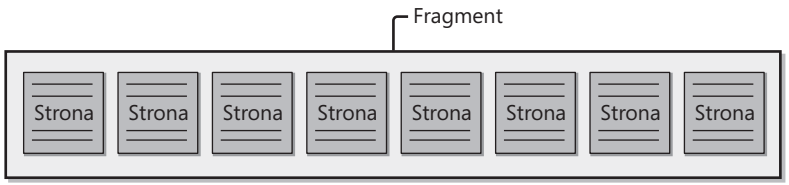
Rysunek 2-1 Diagram strony

96-bajtowy nagłówek strony zawiera m.in. takie informacje jak jednostka alokacji, do której należy strona, wskaźniki do poprzedniej i kolejnej strony na liście (w przypadku indeksu) czy ilość wolnego miejsca na stronie. Na podstawie jednostki alokacji można wydedukować, do którego obiektu należy strona, korzystając z widoków sys.allocation_units oraz sys.partitions. Strony danych i indeksów zawierają rekordy danych lub indeksów. Na końcu strony znajduje się tablica przesunięć wierszy, która stanowi tablicę 2-bajtowych wskaźników do poszczególnych rekordów na stronie. Tablica przesunięć wierszy wraz ze znajdującymi się w nagłówku strony wskaźnikami do poprzedniej i kolejnej strony służy do wymuszania kolejności klucza indeksu.

Aby strona mogła zostać odczytana lub zapisana przez SQL Server, musi zostać umieszczona w puli buforów (pamięci podręcznej danych). SQL Server nie może przetwarzać stron bezpośrednio w pliku danych zapisanym na dysku. Jeśli SQL Server chce dokonać odczytu strony, która znajduje się już w pamięci w związku z podejmowanymi niedawno aktywnościami, wykona jedynie operację odczytu logicznego (odczytu z pamięci). Jeśli strona nie znajduje się jeszcze w pamięci, SQL Server zacznie od wykonania operacji odczytu fizycznego, skutkującej pobraniem strony z pliku danych do pamięci, a następnie wykona operację odczytu logicznego. Analogicznie, jeśli SQL Server chce zapisać stronę, która znajduje się już w pamięci, przeprowadzi operację zapisu logicznego. Dodatkowo ustawi flagę „zanieczyszczona" w nagłówku strony w pamięci, aby zasygnalizować, że stan w pamięci jest bardziej aktualny niż stan w pliku danych.

Co pewien czas SQL Server uruchamia procesy o nazwie *lazywriter* (zapis z opóźnieniem) oraz *checkpoint* (punkt kontrolny), aby zapisać na dysku zanieczyszczone strony przechowywane w pamięci podręcznej, wykorzystując niekiedy dodatkowe wątki w celu usprawnienia realizacji tego zadania. SQL Server używa algorytmu o nazwie *LRU-K2* do wyznaczania stron do zwolnienia z pamięci podręcznej w oparciu o dwie ostatnie wizyty (zarejestrowane w nagłówku strony w pamięci). SQL Server wykorzystuje zwykle proces zapisu z opóźnieniem do oznaczania stron jako zwolnionych w oparciu o ten algorytm.

Fragment (ang. *extent*) to jednostka obejmująca osiem sąsiadujących ze sobą stron, jak pokazano na rysunku 2-2.



Rysunek 2-2 Diagram ilustrujący fragment

SQL Server wspiera dwa typy fragmentów: *mieszane* oraz *jednolite*. Przy tworzeniu nowych obiektów (tabel bądź indeksów) SQL Server przydziela im pojedyncze strony z mieszanych fragmentów, dopóki nie osiągną one rozmiaru ośmiu stron. W związku z tym różne strony tego samego mieszanego fragmentu mogą należeć do różnych obiektów. Do śledzenia, które fragmenty są mieszane i zawierają wolne strony do przydzielenia, służą strony mapy bitowej nazywane *udostępnionymi globalnymi mapami alokacji* (Shared Global Allocation Map – SGAM). Gdy obiekt osiągnie rozmiar ośmiu stron, SQL Server zaczyna przydzielać mu strony pochodzące z ujednoliconych fragmentów. Ujednolicony fragment należy w całości do tego samego obiektu. SQL Server wykorzystuje strony mapy bitowej nazywane *globalnymi mapami alokacji* (Global Allocation Map – GAM) do śledzenia, które fragmenty są wolne i mogą zostać przydzielone.

Struktura tabel

Istnieją dwa sposoby organizowania tabel: sterta lub B-drzewo. Zasadniczo, tabela ma strukturę B-drzewa, gdy został dla niej zdefiniowany indeks klastrowy, a sterty w przeciwnym przypadku. Można utworzyć indeks klastrowy bezpośrednio przy użyciu polecenia CREATE CLUSTERED INDEX lub, począwszy od wersji SQL Server 2014, przy użyciu wbudowanej definicji indeksu klastrowego *INDEX <nazwa_indeksu> CLUSTERED*. Można również utworzyć indeks klastrowy w sposób pośredni poprzez zdefiniowanie klucza głównego lub ograniczenia UNIQUE. Gdy do tabeli dodany zostaje klucz główny, SQL Server wymusza go przy użyciu indeksu klastrowego, z wyjątkiem sytuacji, gdy zastosowane zostało słowo kluczowe NONCLUSTERED lub tabela posiada już indeks klastrowy. W przypadku dodania do tabeli ograniczenia UNIQUE, SQL Server wymusza je przy użyciu indeksu nieklastrowego, o ile nie zostało zastosowane słowo kluczowe CLUSTERED. Podsumowując, tabela z indeksem klastrowym jest zorganizowana jako B-drzewo (ang. *B-tree*), a bez niego jako sterta (ang. *heap*). Ponieważ tabela musi mieć jedną z tych dwóch postaci, struktura tabeli jest w skrócie nazywana HOBT (ang. *heap or B-tree*).

W systemie SQL Server tabele znacznie częściej mają strukturę B-drzewa. Wynika to przede wszystkim z faktu, iż SQL Server domyślnie wymusza klucz główny przy użyciu indeksu klastrowego. A zatem, gdy twórca tabeli nie myśli o fizycznej strukturze i po prostu definiuje klucz główny z wykorzystaniem domyślnych ustawień, efekt końcowy stanowi B-drzewo. W konsekwencji użytkownicy programu SQL Server mają większe doświadczenie w pracy z B-drzewami, niż ze stertami. Ponadto indeksy klastrowe zapewniają większą kontrolę pod względem wyboru wzorców wstawiania i odczytywania. Z tych względów w większości sytuacji struktura B-drzewa stanowi lepsze rozwiązanie.

Czasami zastosowanie stery okazuje się bardziej korzystne. Na przykład w rozdziale 6 zatytułowanym „Modyfikowanie danych" omówię narzędzia do importu zbiorczego i warunki, jakie trzeba spełnić, aby operacja importowania była rejestrowana w minimalnym zakresie. Jeśli tabela docelowa ma strukturę B-drzewa, tego typu rejestrowanie można osiągnąć tylko wtedy, gdy jest ona pusta w momencie rozpoczęcia operacji importu. Natomiast w przypadku sterty można skonfigurować rejestrowanie w minimalnym zakresie nawet wtedy, gdy tabela zawiera już jakieś dane. W związku z tym, dodając do tabeli docelowej dużo więcej danych niż zawierała ona dotychczas, można zwiększyć wydajność operacji importu, usuwając indeksy, importując dane na stertę, a następnie odtwarzając indeksy. W tym przykładzie sterta pełni funkcję stanu tymczasowego.

Niezależnie od struktury tabeli można dla niej zdefiniować zero lub więcej indeksów nieklastrowych. Indeksy nieklastrowe mają zawsze strukturę B-drzewa. Struktury HOBT, jak również indeksy nieklastrowe mogą być zbudowane z jednej lub wielu jednostek zwanych *partycjami*. Z technicznego punktu widzenia, struktura HOBT oraz każdy z indeksów nieklastrowych może być podzielony na partycje w inny sposób.

Każda partycja każdej struktury HOBT oraz indeksu nieklastrowego przechowuje dane w kolekcjach stron nazywanych *jednostkami alokacji*. Istnieją trzy typy jednostek alokacji: IN_ROW_DATA, ROW_OVERFLOW_DATA oraz LOB_DATA.

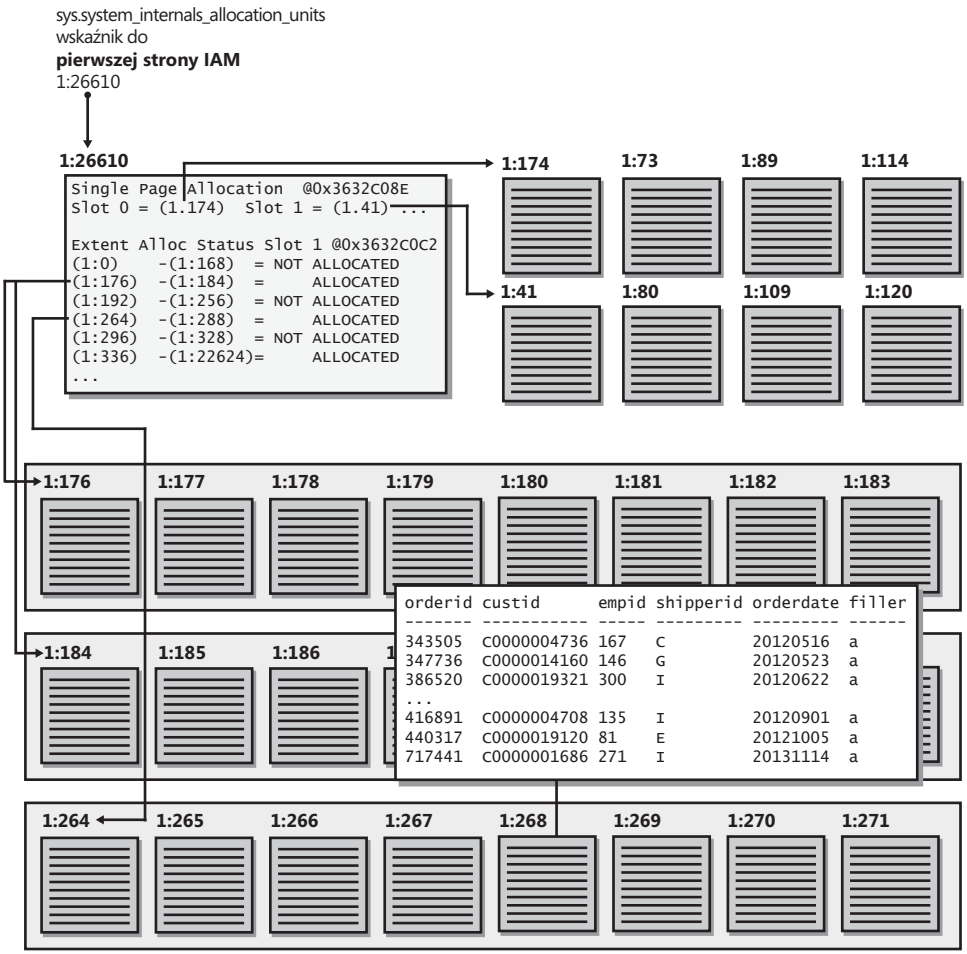
Jednostki IN_ROW_DATA przechowują wszystkie kolumny o stałej długości, a także kolumny o zmiennej długości, o ile rozmiar wiersza nie przekracza 8060 bajtów. Jednostki ROW_OVERFLOW_DATA służą do przechowywania danych typu VARCHAR, NVARCHAR, VARBINARY, SQL_VARIANT lub definiowanego przez użytkownika typu CLR, których rozmiar nie przekracza 8000 bajtów, ale które zostały przeniesione z pierwotnego wiersza, ponieważ przekroczyły limit rozmiaru wiersza wynoszący 8060 bajtów. Jednostki LOB_DATA zawierają dane dużych obiektów: VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX) o rozmiarze przekraczającym 8000 bajtów, XML lub definiowanego przez użytkownika typu CLR. Widok systemowy sys.system_internals_allocation_units zawiera wskaźniki do kolekcji stron przechowywanych w jednostkach alokacji.

W kolejnych częściach rozdziału omówię struktury sterty, indeksu klastrowego oraz indeksu nieklastrowego. Dla uproszczenia przyjąłem założenie, że dane nie są podzielone na partycje, choć nawet w takiej sytuacji przedstawione informacje byłyby prawdziwe, aczkolwiek odnosiłyby się do pojedynczej partycji.

Sterta

Sterta to tabela, która nie zawiera indeksu klastrowego. Struktura ta jest nazywana *stertą*, ponieważ dane nie są w żaden sposób uporządkowane – stanowią po prostu stertę stron i fragmentów. Tabela Orders w przykładowej bazie danych PerformanceV3 (instrukcję instalacji przykładowej bazy danych znaleźć można we Wstępie do książki) ma w rzeczywistości strukturę B-drzewa, ale rysunek 2-3 ilustruje, jak wyglądałaby, gdyby miała strukturę sterty.

SQL Server mapuje dane należące do sterty, wykorzystując jedną lub wiele stron mapy bitowej, nazywanej *mapą alokacji indeksu* (Index Allocation Map – IAM). Nagłówek strony IAM zawiera wskaźniki do pierwszych ośmiu stron, które SQL Server alokował dla stery z mieszanego fragmentu. W nagłówku znajduje się również wskaźnik do lokalizacji początkowej zakresu 4 GB, który strona IAM mapuje w pliku danych. W treści strony IAM znaleźć można po jednym bicie reprezentującym każdy fragment w tym zakresie. Bit ma wartość 0, jeśli reprezentowany przez niego zakres nie należy do obiektu, do którego należy dana strona IAM oraz 1 w przeciwnym przypadku. Gdy jedna strona IAM nie wystarcza, aby pokryć wszystkie dane obiektu, SQL Server będzie utrzymywał łańcuch stron IAM. Skanując stertę, SQL Server potrzebuje informacji ze stron IAM do określenia, które strony i fragmenty muszą zostać odczytane. Tego typu skanowanie będziemy nazywać *skanowaniem w kolejności alokacji* (ang. *allocation order scan*). Jest ono przeprowadzane zgodnie z porządkiem pliku i w związku z tym, gdy dane nie znajdują się w pamięci podręcznej, zwykle przeprowadzane są odczyty sekwencyjne.



Rysunek 2-3 Diagram sterty

Jak widać na rysunku 2-3, SQL Server utrzymuje wewnętrzne wskaźniki do pierwszej strony IAM oraz pierwszej strony danych sterty. Te wskaźniki znaleźć można w widoku systemowym sys.system_internals_allocation_units.

Ponieważ dane na stercie nie są w żaden sposób uporządkowane, nowe wiersze dodawane do tabeli mogą być umieszczane w dowolnym miejscu. SQL Server wykorzystuje strony mapy bitowej *Page Free Space* (w skrócie PFS) do śledzenia wolnego miejsca na stronach. Takie rozwiązanie umożliwia szybkie odnajdowanie strony z wystarczającą ilością wolnego miejsca, by mogła pomieścić wiersz lub, jeśli taka strona nie jest dostępna, alokowanie nowej strony.

Gdy wiersz rozszerza się w wyniku modyfikacji kolumny o zmiennej długości i strona nie zawiera wystarczającej ilości wolnego miejsca, aby umożliwić przeprowadzenie tej operacji, SQL Server przenosi wiersz na stronę z wystarczającą ilością

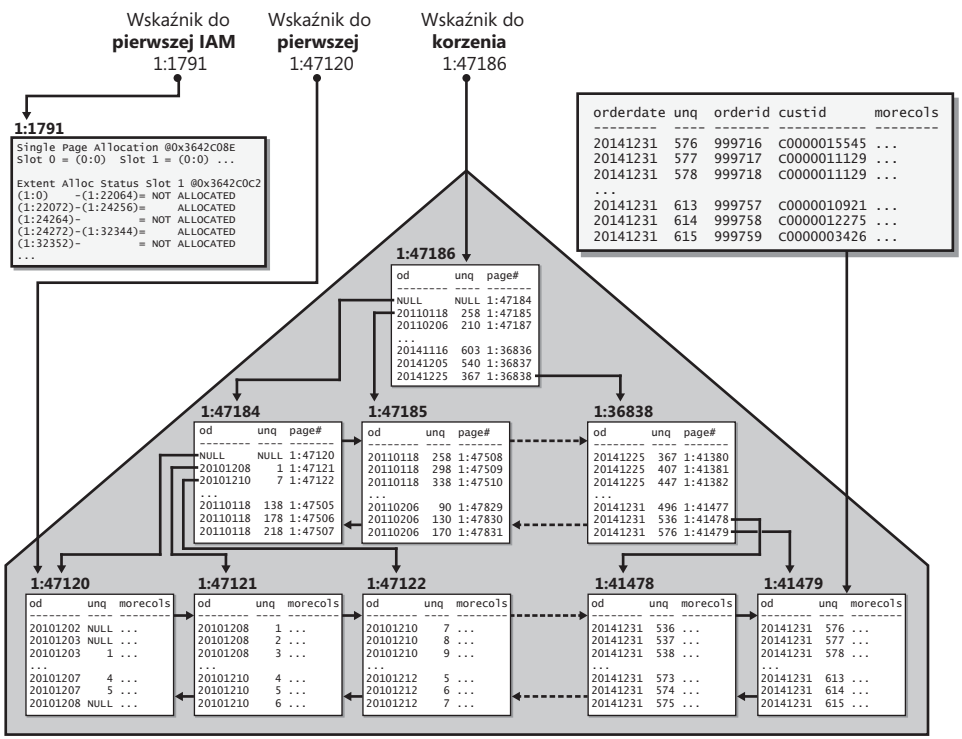
wolnego miejsca, pozostawiając tzw. wskaźnik przenoszący (ang. *forwarding pointer*), który wskazuje nową lokalizację wiersza. Wskaźniki przenoszące eliminują konieczność aktualizowania wskaźników do wierszy w indeksach nieklastrowych, gdy nastąpiło przeniesienie wiersza danych.

Nie objaśniłem jeszcze koncepcji nazywanej *podziałem strony* (ponieważ dotyczy ona jedynie B-drzew); na razie wystarczy mieć świadomość, że podziały stron nie zachodzą na stercie.

B-drzewo (indeks klastrowy)

W systemie SQL Server wszystkie indeksy na tabelach przechowywanych na dysku mają strukturę *B-drzew*, które stanowią specjalny typ *drzew zrównoważonych*. Drzewo zrównoważone to takie drzewo, w którym żaden liść nie znajduje się dużo dalej od korzenia niż inny liść.

Indeks klastrowy ma strukturę B-drzewa, w którym wszystkie dane tabeli są przechowywane na poziomie liści. Indeks klastrowy *stanowi* dane, nie kopię danych. Rysunek 2-4 ilustruje, jak mogłaby wyglądać tabela Orders o strukturze B-drzewa z kolumną *orderdate* pełniącą rolę klucza.



Rysunek 2-4 Ilustracja B-drzewa

Jak widać na rysunku, pełne wiersze danych tabeli Orders są przechowywane na *poziomie liści* indeksu. Wiersze danych na poziomie liści są uporządkowane według klucza (w tym przypadku daty zamówienia – *orderdate*). Lista dwukierunkowa utrzymuje kolejność klucza między stronami, a tablica przesunięć wierszy znajdująca się na końcu strony utrzymuje tę kolejność w ramach strony. Aby przeprowadzić operację uporządkowanego skanowania (lub skanowania zakresu) na poziomie liści indeksu, SQL Server realizuje ją, podążając zgodnie z kierunkiem listy. Tego typu operacje skanowania nazywane są *uporządkowanym skanowaniem indeksu* (ang. *index order scan*).

Jak widać na rysunku 2-4, dla każdego wiersza na poziomie liści indeks utrzymuje specjalną kolumnę *uniquifier* (na rysunku oznaczoną skrótem *unq*). Wartość w tej kolumnie wskazuje wiersze o tej samej wartości klucza i wraz z wartością klucza służy do unikatowego identyfikowania wierszy, gdy kolumny klucza indeksu nie zapewniają unikatowości. Do roli kolumny *uniquifier* powrócę niedługo podczas omawiania indeksów nieklastrowych.

Przetwarzając zapytanie, optymalizator musi wybrać algorytmy służące do realizowania takich operacji, jak złączanie, grupowanie i agregowanie, usuwanie duplikatów, funkcje okna czy określanie porządku prezentacji. Niektóre operacje, np. ustalanie porządku prezentacji (klauzula ORDER BY w zapytaniu), wymagają zastosowania algorytmu opartego na kolejności. Inne operacje dopuszczają zastosowanie więcej niż jednego algorytmu, w tym także algorytmu bazującego na kolejności. Na przykład operacje grupowania/agregowania oraz usuwania duplikatów mogą być realizowane przy użyciu algorytmu wykorzystującego kolejność lub na funkcję skrótu (ang. *hash*). Do przeprowadzenia złączeń można użyć algorytmu bazującego na kolejności, na wartościach skrótu lub na pętli. Aby możliwe było zastosowanie algorytmu opartego na kolejności, dane muszą być uporządkowane w odpowiedniej kolejności. By osiągnąć ten cel, optymalizator dodaje czasami do planu jawną operację sortowania. Jednak tego typu operacja wymaga przydziału pamięci i charakteryzuje się gorszą skalowalnością (proporcjonalnie do N log N, gdzie N jest liczbą przetwarzanych jednostek, czyli wierszy).

Inny sposób uzyskiwania uporządkowanych danych polega na przeprowadzeniu uporządkowanego skanowania indeksu, o ile dostępny jest odpowiedni indeks. Jest to zwykle najbardziej efektywna opcja, która w odróżnieniu od jawnych operatorów sortowania oraz skrótu nie wymaga przydziału pamięci. Jednak gdy indeks nie istnieje, optymalizator jest skazany na pozostałe opcje. Jeśli operacja może zostać zrealizowana wyłącznie przy użyciu algorytmu bazującego na kolejności (jak w przypadku porządku prezentacji), optymalizator musi dodać jawną operację sortowania. Natomiast gdy możliwe jest zastosowanie innego typu algorytmu (jak w przypadku grupowania i agregacji), optymalizator dokonuje wyboru na podstawie szacunkowych kosztów. Ponieważ operacja sortowania charakteryzuje się słabą skalowalnością, optymalizator wybiera zwykle strategię wymagającą sortowania, gdy liczba wierszy jest niewielka. W przypadku wysokiej liczby wierszy optymalizator preferuje algorytmy skrótu, ponieważ zazwyczaj charakteryzują się one lepszą skalowalnością niż sortowanie.

Odnośnie danych na poziomie liści indeksu, kolejność stron w pliku nie musi odpowiadać kolejności klucza indeksu. Jeśli strona *x* wskazuje następną stronę *y*, a strona *y* poprzedza stronę *x* w pliku, strona *y* jest nazywana *stroną poza kolejnością*. Logiczna fragmentacja skanowania (zwana również *średnią fragmentacją w procentach*) wyraża procentowy udział stron poza kolejnością. W związku z tym, jeśli na poziomie liści indeksu znajduje się 25 000 stron i 5000 z nich jest umieszczonych poza kolejnością, logiczna fragmentacja skanowania wynosi 20 procent. Główną przyczyną logicznej fragmentacji skanowania są podziały stron, do których za chwilę powrócę. Im wyższa logiczna fragmentacja skanowania indeksu, tym więcej czasu zajmuje operacja uporządkowanego skanowania indeksu, jeśli dane nie są przechowywane w pamięci podręcznej. Gdy dane znajdują się w pamięci podręcznej, fragmentacja ma znikomy wpływ na tę operację.

Co ciekawe, poza listą SQL Server utrzymuje również strony IAM służące do mapowania danych przechowywanych w indeksie w kolejności pliku, tak jak w przypadku struktury sterty. Jak wspomniałem we wcześniejszej części rozdziału, operacja skanowania danych w oparciu o strony IAM nazywana jest *skanowaniem w kolejności alokacji*. SQL Server może użyć tego typu skanowania do przeprowadzenia nieuporządkowanego skanowania poziomu liści indeksu. Ponieważ operacja skanowania w kolejności alokacji wiąże się z odczytywaniem danych w takiej kolejności, w jakiej są one rozmieszczone w pliku, logiczna fragmentacja skanowania nie obniża jej wydajności, jak w przypadku operacji uporządkowanego skanowania indeksu. W związku z tym operacje skanowania w kolejności alokacji są zwykle bardziej efektywne, niż uporządkowane skanowanie indeksu, w szczególności gdy dane nie znajdują się w pamięci podręcznej i występuje duża logiczna fragmentacja skanowania.

Jak wspomniałem wcześniej, logiczna fragmentacja skanowania powstaje głównie na skutek podziałów stron. Natomiast podziały stron w liściach indeksu następują, gdy pojawia się potrzeba umieszczenia wiersza na stronie, która nie zawiera wystarczającej ilości wolnego miejsca. Należy pamiętać, że dane w indeksie są rozmieszczone w kolejności klucza. Wiersz musi zostać wstawiony na stronę odpowiadającą wartości jego klucza. Gdy docelowa strona jest pełna, SQL Server musi ją podzielić. SQL Server wspiera dwa rodzaje podziałów stron: dla wzorca kluczy rosnących oraz wzorca kluczy nierosnących.

Jeśli wartość nowego klucza jest większa lub równa maksymalnej wartości klucza, SQL Server przyjmuje założenie, że wzorcem wstawiania jest wzorzec kluczy rosnących. Alokuje nową stronę i umieszcza na niej nowy wiersz. Wzorzec kluczy rosnących jest zwykle bardziej efektywny, gdy nowe wiersze są wstawiane przez pojedynczą sesję i system dyskowy dysponuje niewielką liczbą napędów. Nowa strona zostaje alokowana w pamięci podręcznej i wypełniona nowymi wierszami. Jednak w sytuacji, gdy nowe wiersze są wstawiane przez wiele sesji, a system dyskowy dysponuje wysoką liczbą napędów, wzorzec klucza rosnącego nie jest wydajny. Ten wzorzec wiąże się z częstymi konfliktami zatrzaśnięć skrajnej prawej strony indeksu, co obniża wydajność operacji

wstawiania. W tego typu sytuacjach zazwyczaj dużo lepiej sprawdza się wzorzec wstawiania z losową dystrybucją nowych kluczy.

Jeśli nowy klucz ma wartość niższą niż maksymalna wartość klucza, SQL Server przyjmuje założenie, że wzorcem wstawiania jest wzorzec klucza nierosnącego. Alokuje nową stronę, przenosi połowę wierszy z oryginalnej strony na nową i dostosowuje listę tak, aby odzwierciedlała ona odpowiednią logiczną kolejność stron. O tym, czy nowy wiersz zostaje wstawiony na oryginalną czy na nową stronę, decyduje jego wartość klucza. Nowa strona nie musi znajdować się tuż za podzieloną stroną – może zostać umieszczona we wcześniejszej lub dalszej części pliku.

Podziały stron są kosztowne i często skutkują logiczną fragmentacją skanowania. Aby zbadać poziom fragmentacji wybranego indeksu, wystarczy wykonać zapytanie na kolumnie *avg_fragmentation_in_percent* funkcji sys.dm_db_index_physical_stats. Można wyeliminować fragmentację, odbudowując indeks przy użyciu polecenia ALTER INDEX REBUILD. Ponadto można zastosować opcję o nazwie FILLFACTOR, aby tylko częściowo wypełniać strony na poziomie liści odbudowywanego indeksu. Na przykład opcja *FILLFACTOR = 70* służy do żądania, aby strony na poziomie liści były wypełniane w maksymalnie 70 procentach. Jeśli indeks charakteryzuje się wzorcem kluczy nierosnących, pozostawienie pewnej ilości wolnego miejsca na stronach w liściach indeksu pozwala na ograniczenie liczby podziałów i obniżenie logicznej fragmentacji skanowania. Konfigurując współczynnik wypełnienia, warto mieć na uwadze liczbę operacji wstawiania oraz częstotliwość operacji odbudowywania indeksu. Gdy indeks charakteryzuje się wzorcem kluczy rosnących, pozostawianie wolnego miejsca na stronach w liściach indeksu nie ma sensu, ponieważ wszystkie nowe wiersze i tak będą umieszczane na skrajnej prawej stronie indeksu.

Indeks składa się nie tylko z poziomu liści, ale także dodatkowych poziomów, które służą do opisywania niższych poziomów. Każdy wiersz na stronie indeksu niebędącej stroną liścia wskazuje całą stronę na niższym poziomie i wraz z kolejnym wierszem dostarcza informację o zakresie kluczy, za które odpowiada docelowa strona. Wiersz zawiera dwa elementy: minimalną wartość kolumny klucza na wskazywanej stronie indeksu (w przypadku kolejności rosnącej) oraz 6-bajtowy wskaźnik do tej strony. Dzięki temu wiadomo, że strona docelowa odpowiada za zakres kluczy o wartości większej lub równej wartości klucza przechowywanej w bieżącym wierszu oraz mniejszej niż wartość klucza w następnym wierszu. Pierwszy element wiersza, który wskazuje pierwszą stronę na niższym poziomie, ma wartość NULL, co sygnalizuje brak minimum. Natomiast wskaźnik strony składa się z numeru pliku w bazie danych oraz numeru strony w pliku. Gdy SQL Server buduje indeks, rozpoczyna od poziomu liścia i dodaje kolejne, wyższe poziomy. Zatrzymuje się, gdy poziom zawiera tylko jedną stronę, zwaną stroną *korzenia*.

Gdy SQL Server musi znaleźć określony klucz lub zakres kluczy na poziomie liści indeksu, korzysta z metody dostępu nazywanej *przeszukiwaniem indeksu* (ang. *index seek*). Ta metoda dostępu rozpoczyna od odczytania strony korzenia i zidentyfikowania

wiersza, który reprezentuje zakres zawierający wyszukiwany klucz. Następnie przechodzi w dół przez kolejne poziomy, odczytując po jednej stronie na każdym z poziomów, aż dotrze do strony znajdującej się na poziomie liści, która zawiera pierwszy pasujący klucz. Następnie przeprowadza skanowanie zakresu na poziomie liści, aby odnaleźć wszystkie pasujące klucze. Jak wspomniałem, wszystkie strony na poziomie liści indeksu znajdują się w takiej samej odległości od korzenia. To oznacza, że aby dotrzeć do pierwszego pasującego klucza, ta metoda dostępu wymaga odczytania tylu stron, ile poziomów zawiera indeks. Operacje odczytu będą miały charakter *losowych operacji we/wy*, a nie *sekwencyjnych operacji we/wy*, ponieważ strony odczytywane podczas przeszukiwania indeksu aż do momentu dotarcia do poziomu liści będą rzadko znajdowały się obok siebie.

Aby móc szacować wydajność, warto znać liczbę poziomów indeksu, ponieważ to od niej zależy koszt operacji przeszukiwania (wyrażony w liczbie odczytów stron), a niektóre plany wykonania wielokrotnie inicjują operację przeszukiwania (na przykład w operatorze Nested Loops służącym do złączania tabel). Do sprawdzania liczby poziomów istniejącego indeksu można wykorzystać funkcję INDEXPROPERTY wywoływaną z właściwością IndexDepth. Jednak z myślą o indeksach, które nie zostały jeszcze utworzone, warto wiedzieć, jak własnoręcznie oszacować liczbę poziomów.

Poniżej zaprezentowana została procedura i współczynniki potrzebne do wyznaczenia liczby *L* wyrażającej liczbę poziomów w indeksie (należy pamiętać, że o ile nie zaznaczono inaczej, przedstawione obliczenia odnoszą się zarówno do indeksów klastrowych, jak i nieklastrowych):

- **Liczba wierszy w tabeli (oznaczona jako *liczba_wierszy*)** W naszym przypadku wynosi ona 1000 000.
- **Średni całkowity rozmiar wiersza w liściu indeksu (oznaczony jako *rozmiar_wiersza_w_liściu*)** W indeksie klastrowym jest on równy rozmiarowi wiersza danych. Przymiotnik „całkowity” ma przypominać o konieczności uwzględnienia wewnętrznego narzutu wiersza oraz 2-bajtowego wskaźnika w tablicy przesunięcia wierszy. Narzut wiersza wynosi zazwyczaj kilka bajtów. W przypadku tabeli Orders całkowity średni rozmiar wiersza danych wynosi około 200 bajtów.
- **Średnia gęstość strony na poziomie liści (oznaczona jako *zagęszczenie_strony*)** Ta wartość wyraża średnie procentowe zapełnienie stron na poziomie liści. Na gęstość strony mają wpływ takie operacje, jak usuwanie danych, podziały stron oraz odbudowywanie indeksu ze współczynnikiem wypełnienia niższym niż 100. Indeks klastrowy w naszej tabeli bazuje na wzorcu klucza rosnącego, w związku z tym *zagęszczenie_strony* będzie prawdopodobnie zbliżone do 100 procent.
- **Liczba wierszy mieszczących się na stronie na poziomie liści (oznaczona jako *wiersze_per_strona_w_liściu*)** Do wyznaczania tego współczynnika służy następujący wzór *FLOOR((rozmiar_strony – rozmiar_nagłówka) * zagęszczenie_strony / rozmiar_wiersza_w_liściu)*. W naszym przypadku *wiersze_per_strona_w_liściu* wynosi *FLOOR((8192 – 96) * 1 / 200)* = 40.

- **Liczba stron na poziomie liści (oznaczona jako *liczba_stron_w_liściach*)** To prosty wzór: *liczba_wierszy / wiersze_per_strona_w_liściu*. W naszym przypadku liczba ta wynosi *1000000 / 40 = 25000*.
- **Średni całkowity rozmiar wiersza nieznajdującego się w liściu indeksu (oznaczona jako *rozmiar_wiersza_nie_w_liściu*)** Wiersz znajdujący się nad poziomem liści zawiera: kolumny klucza indeksu (w naszym przypadku tylko *orderdate* o rozmiarze 3 bajty), 4-bajtową kolumnę *uniquifier* (tylko w indeksie klastrowym dla klucza, który nie jest unikatowy), wskaźnik do strony (o rozmiarze 6 bajtów), kilka dodatkowych bajtów wewnętrznego narzutu (w naszym przypadku 5 bajtów) oraz znajdujący się na końcu strony wskaźnik przesunięcia wiersza (2 bajty). W naszym przykładzie średni całkowity rozmiar wiersza przechowywanego nad poziomem liści indeksu wynosi 20 bajtów.
- **Liczba wierszy mieszczących się na stronie nie na poziomie liści (oznaczona jako *wiersze_per_strona_nie_w_liściu*)** Wzór służący do obliczania tej wartości przypomina wzór wyznaczania współczynnika *wiersze_per_strona_w_liściu* i wygląda następująco: *FLOOR((rozmiar_strony – rozmiar_nagłówka) / rozmiar_wiersza_nie_w_liściu)*. W naszym przypadku jego wartość wynosi: *FLOOR((8192 – 96) / 20) = 404*.
- **Liczba poziomów znajdujących się nad poziomem liści (oznaczona jako *L–1*)** Do wyznaczania tej wartości służy następujący wzór: *CEILING(LOG(liczba_stron_w_liściach, wiersze_per_strona_nie_w_liściu))*. W naszym przypadku *L–1* wynosi: *CEILING(LOG(25000, 404)) = 2*.
- **Głębokość indeksu (oznaczona jako *L*)** Aby uzyskać liczbę *L*, wystarczy dodać 1 do wyznaczonej w poprzednim kroku liczby *L–1*. Pełny wzór wyznaczania liczby *L* wygląda następująco: *CEILING(LOG(liczba_stron_w_liściach, wiersze_per_strona_nie_w_liściu)) + 1*. W naszym przypadku L wynosi 3.
- **Liczba wierszy, które mogą być reprezentowane przy użyciu indeksu z *L* poziomami (oznaczona jako *N*)** Załóżmy, że chcemy przeprowadzić odwrotne obliczenia, bazując na liczbie *L*. Innymi słowy, chcemy wyznaczyć liczbę wierszy, które można reprezentować przy użyciu indeksu o *L* poziomach. Podobnie jak w poprzednim przykładzie, wyznaczamy wartości *wiersze_per_strona_nie_w_liściu* oraz *wiersze_per_strona_w_liściu*. Następnie do wyznaczenia liczby *N* wykorzystujemy następujący wzór: *POWER(wiersze_per_strona_nie_w_liściu, L–1) * wiersze_per_strona_w_liściu*. Dla *L = 3 wiersze_per_strona_nie_w_liściu = 404*, a *wiersze_per_strona_w_liściu = 40*, w związku z tym otrzymujemy wynik 6 528 540 wierszy.

Manewrując wzorami do wyznaczania współczynników *L* oraz *N* można stwierdzić, że dopóki liczba wierszy w naszej tabeli nie przekroczy 16 tysięcy, indeks będzie miał dwa poziomy. Trzypoziomowy indeks może reprezentować około 6,5 miliona, a czteropoziomowy nawet około 2,6 miliarda wierszy. W przypadku indeksu nieklastrowego wzory pozostają takie same, aczkolwiek na każdej stronie w liściu indeksu można

pomieścić więcej wierszy, o czym będzie się można za chwilę przekonać. W związku z tym w indeksach nieklastrowych poszczególnym poziomom odpowiada jeszcze wyższy limit reprezentowanych wierszy tabeli. Nasza tabela ma milion wierszy i wszystkie zdefiniowane na niej indeksy mają trzy poziomy. W związku z tym koszt operacji przeszukania indeksu na naszej tabeli wynosi trzy odczyty. Warto zapamiętać tę liczbę, ponieważ będę się do niej odwoływał podczas omawiania w dalszej części rozdziału dodatkowych aspektów związanych z wydajnością. Jak można stwierdzić na podstawie przedstawionych informacji, liczba poziomów indeksu zależy od rozmiaru wiersza oraz klucza. O ile mieszczą się one w granicach normy, indeksy dla małych tabel (do kilkudziesięciu tysięcy wierszy) mają zwykle dwa poziomy, dla średnich tabel (do kilku milionów wierszy) trzy poziomy, a dla dużych tabel (do kilku miliardów wierszy) cztery poziomy.

Indeks nieklastrowy dla sterty

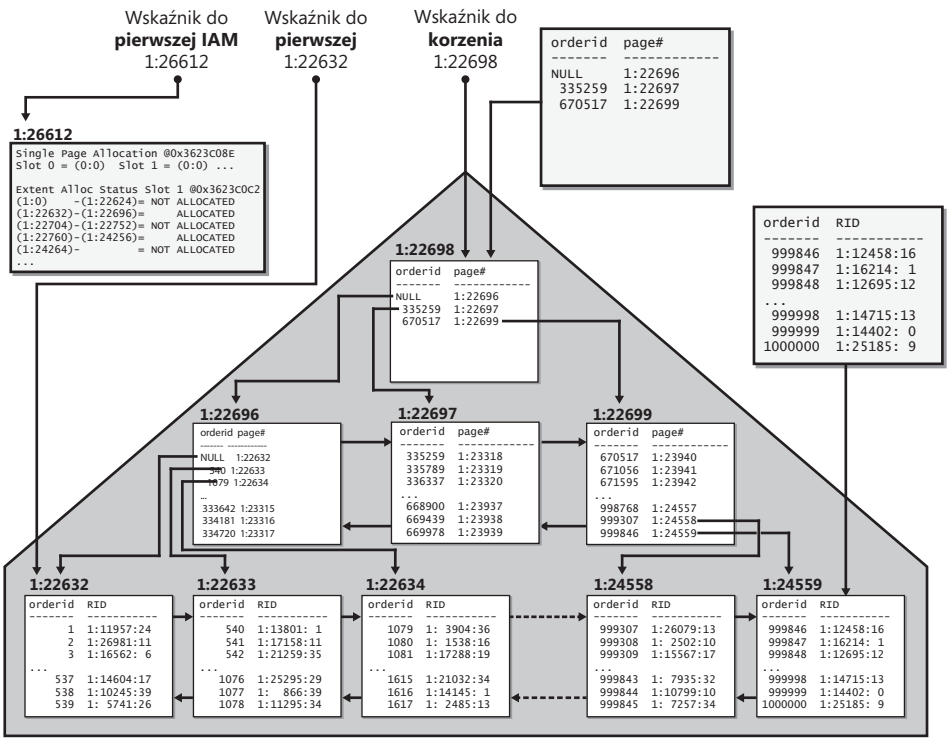
Indeks nieklastrowy ma również strukturę B-drzewa i pod wieloma względami przypomina indeks klastrowy. Główna różnica polega na tym, że wiersz w liściu indeksu nieklastrowego zawiera jedynie kolumny klucza indeksu oraz *lokalizator wiersza*, reprezentujący odpowiedni wiersz. Zawartość lokalizatora wiersza zależy od tego, czy dana tabela ma strukturę stery, czy B-drzewa. W tej części rozdziału omówię indeksy nieklastrowe zdefiniowane na stertach, natomiast w kolejnej części rozdziału zajmę się indeksami nieklastrowymi zdefiniowanymi na B-drzewach (tabelach klastrowych).

Na rysunku 2-5 zilustrowany został indeks nieklastrowy, jaki SQL Server utworzył w celu wymuszania ograniczenia klucza głównego (*PK_Orders*) na kolumnie *orderid*. Indeks, przypisywany przez SQL Server, nosi tę samą nazwę co ograniczenie, czyli *PK_Orders*.

Lokalizator wiersza umieszczany w wierszu liścia indeksu nieklastrowego w celu wskazywania wiersza danych stanowi 8-bajtowy wskaźnik fizyczny nazywany identyfikatorem wiersza (ang. Row Identifier – *RID*). Składa się on z numeru pliku w bazie danych, numeru strony docelowej w pliku oraz pozycji wiersza w tablicy przesunięcia wierszy na stronie docelowej (numeracja rozpoczyna się od zera). Wyszukując wiersz danych na podstawie wybranego klucza indeksu nieklastrowego, SQL Server rozpoczyna od operacji przeszukania indeksu w celu zlokalizowania na poziomie liści indeksu wiersza z odpowiednią wartością klucza. Następnie SQL Server wykonuje operację *wyszukania identyfikatora wiersza* (ang. RID lookup), czyli odczytuje odnalezioną stronę i pobiera z niej odpowiedni wiersz. W związku z tym operacja wyszukania identyfikatora wiersza wymaga wykonania jednego odczytu strony.

Gdy wyszukiwany jest zakres kluczy, SQL Server przeprowadza skanowanie zakresu na poziomie liści indeksu, a następnie wyszukiwanie identyfikatorów wierszy dla pasujących kluczy. Gdy liczba operacji wyszukiwania jest niewielka, ich koszt jest akceptowalny, ale większa liczba operacji może stanowić poważne obciążenie, ponieważ SQL Server dokonuje odczytu całej strony dla każdego wyszukiwanego wiersza. Gdy

zapytanie dotyczące zakresu danych bazuje na indeksie nieklastrowym oraz serii operacji wyszukania (po jednej dla każdego pasującego klucza), całkowity koszt operacji wyszukiwania stanowi zwykle sporą część kosztu wykonania zapytania. Do problemu tego powrócimy w podrozdziale zatytułowanym „Metody dostępu". Szacując koszt operacji przeszukania, warto pamiętać, że zaprezentowane wcześniej wzory odnoszą się nie tylko do indeksów klastrowych, ale także nieklastrowych. Jedyna różnica polega na tym, że *rozmiar_wiersza_w_liściu* jest mniejszy, a w związku z tym wskaźnik *wiersze_per_strona_w_liściu* będzie wyższy, ale wzory są takie same.



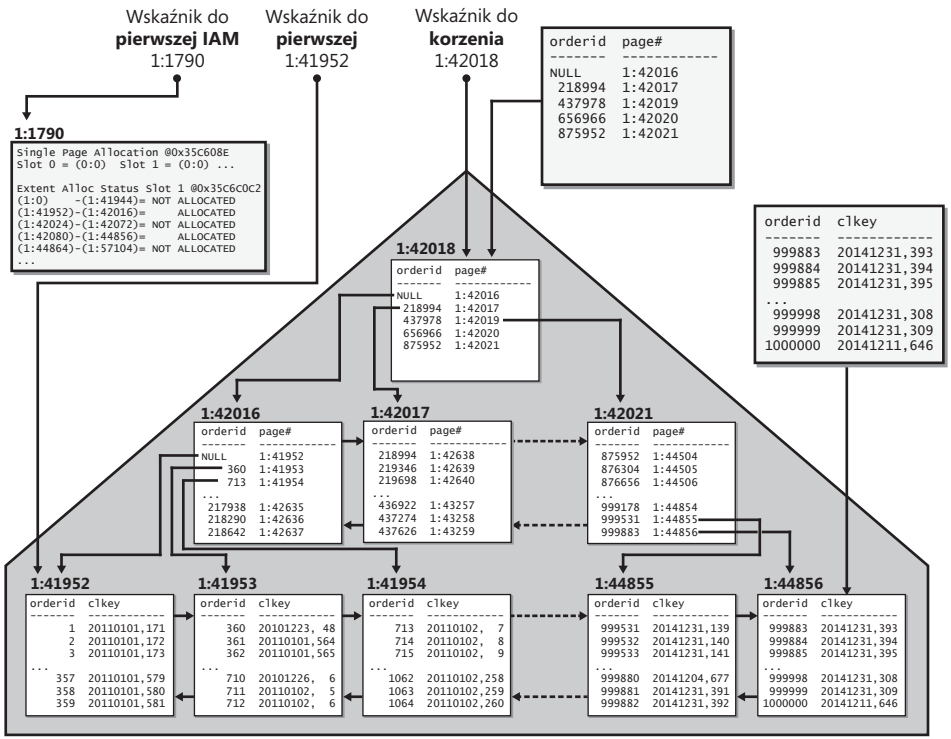
Rysunek 2-5 Indeks nieklastrowy dla sterty

Indeks nieklastrowy dla B-drzewa

Indeksy nieklastrowe tworzone na strukturze B-drzewa (indeksie klastrowym) mają inną budowę, niż te tworzone na stercie. Różnica polega na tym, że lokalizator wiersza w indeksie nieklastrowym utworzonym na strukturze B-drzewa zawiera wartość *klucza klastrowania* zamiast identyfikatora wiersza. Klucz klastrowania składa się z wartości kluczy indeksu klastrowego dla wskazywanego wiersza oraz wartości *uniquifier* (jeśli takowa istnieje). Założenie jest takie, że wiersz ma być wskazywany w sposób *logiczny*, a nie *fizyczny*. Takie rozwiązanie zostało zaprojektowane głównie z myślą o systemach

OLTP, w których indeksy klastrowe prowokują częste podziały stron związane z operacjami wstawiania i modyfikowania danych. Gdyby indeksy nieklastrowe wykorzystywały identyfikatory wierszy w roli lokalizatorów wierszy, wszystkie wskaźniki do przeniesionych wierszy danych musiałyby być aktualizowane przy użyciu nowych identyfikatorów wierszy. Łatwo sobie wyobrazić, jaki wpływ na wydajność miałoby zdefiniowanie dla tabeli wysokiej liczby tego typu indeksów nieklastrowych. Zamiast tego SQL Server utrzymuje wskaźniki logiczne, które nie ulegają zmianie, gdy wiersze danych są przenoszone między stronami na poziomie liści indeksu klastrowego.

Rysunek 2-6 ilustruje, jak mógłby wyglądać indeks nieklastrowy *PK_Orders*, gdyby kolumna *orderid* stanowiła kolumnę klucza indeksu, a w tabeli Orders był zdefiniowany indeks klastrowy z kolumną klucza *orderdate*.



Rysunek 2-6 Indeks nieklastrowy dla B-drzewa

Operacja przeszukania przeprowadzana w celu odnalezienia konkretnego klucza w indeksie nieklastrowym (pewnej wartości *orderid*) zakończy się dotarciem do odpowiedniego wiersza w liściu indeksu i umożliwieniem dostępu do lokalizatora wiersza. W tym przypadku rolę lokalizatora wiersza pełni klucz klastrowania wskazywanego wiersza. Aby dostać się do wskazywanego wiersza, operacja wyszukiwania będzie musiała wywołać operację przeszukiwania indeksu klastrowego, korzystając

z pozyskanego klucza klastrowania. Tego typu operacja wyszukiwania jest nazywana *wyszukaniem klucza* (ang. *key lookup*), w odróżnieniu od wyszukania identyfikatora wiersza. W dalszej części rozdziału zademonstrujemy zastosowanie tej metody dostępu. Koszt każdej operacji wyszukania (mierzony w liczbie odczytów stron) jest równy liczbie poziomów indeksu klastrowego (w naszym przypadku 3). Dla porównania koszt wyszukania identyfikatora wiersza, gdy tabela ma strukturę sterty, to jeden odczyt strony. Oczywiście, jeśli zapytania dotyczą zakresu i wykorzystują indeks nieklastrowy oraz serię operacji wyszukiwania, współczynnik liczby odczytów logicznych dla sterty w stosunku do liczby odczytów dla tabeli klastrowej zbliża się do *1:L*, gdzie *L* to liczba poziomów indeksu klastrowego.

Narzędzia do mierzenia wydajności zapytań

SQL Server oferuje szereg narzędzi służących do mierzenia wydajności zapytań, jednak warto mieć na uwadze, że największe znaczenie ma opinia użytkowników. Użytkownicy zazwyczaj zwracają uwagę na dwa aspekty: czas odpowiedzi (czyli czas oczekiwania na zwrócenie pierwszego wiersza) oraz przepływność (czas oczekiwania na zakończenie pracy zapytania). Oczywiście specjaliści IT dążą do zidentyfikowania różnych współczynników wydajności, które ich zdaniem mają największy wpływ na finalną opinię użytkowników. Najczęściej obserwowane czynniki to liczba odczytów (istotna szczególnie w przypadku zadań wymagających wielu operacji we/wy), jak również czas procesora lub całkowity czas wykonania.

Prezentowane w tym rozdziale przykłady będą bazowały na bazie danych o nazwie PerformanceV3 (szczegółową instrukcję instalacji znaleźć można we Wstępie do książki). Aby nawiązać połączenie z przykładową bazą danych, wystarczy uruchomić następujący kod:

```
SET NOCOUNT ON;
USE PerformanceV3;
```

Do demonstrowania narzędzi do pomiaru wydajności będę wykorzystywał poniższe zapytanie:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid <= 10000;
```

Mierząc wydajność zapytania w środowisku testowym, warto rozważyć, czy pamięć podręczna w środowisku produkcyjnym może już zawierać odpowiednie dane. Jeśli pamięć podręczna będzie najprawdopodobniej wypełniona danymi, należy wykonać zapytanie dwukrotnie i dokonać pomiaru wydajności podczas drugiego wykonania. Pierwsze wykonanie spowoduje, że wszystkie strony zostaną umieszczone w pamięci podręcznej danych. W związku z tym drugie wykonanie zapytania będzie bazowało na przygotowanej pamięci podręcznej. Natomiast gdy pamięć podręczna w środowisku

produkcyjnym najprawdopodobniej nie będzie zawierała relewantnych danych, uruchomienie zapytania warto poprzedzić punktem kontrolnym w celu własnoręcznego zapisania zanieczyszczonych stron na dysku i usunięcia czystych stron z pamięci podręcznej przy pomocy następującego polecenia:

```
CHECKPOINT;
DBCC DROPCLEANBUFFERS;
```

Jednak należy pamiętać, że operacja własnoręcznego opróżniania pamięci podręcznej danych ma oczywiście negatywny wpływ na wydajność zapytań, w związku z tym można przeprowadzać ją jedynie w izolowanym środowisku testowym. Oba polecenia wymagają rozszerzonych uprawnień.

Do analizowania i mierzenia wydajności zapytań wykorzystujemy przede wszystkim trzy standardowe narzędzia: graficzny plan wykonania, opcje sesji STATISTICS IO i STATISTICS TIME, a także sesję Extended Events ze zdarzeniami *sql_statement_completed*. Graficznego planu wykonania używamy do analizowania planu utworzonego przez optymalizator dla danego zapytania. Opcje sesji wykorzystujemy do mierzenia wydajności pojedynczego zapytania lub niewielkiej liczby zapytań. Natomiast sesji Extended Events używamy, gdy musimy zmierzyć wydajność większej liczby zapytań.

Zacznijmy od planu wykonania. Możemy zażądać wyświetlenia oszacowanego planu w programie SQL Server Management Studio (SSMS), zaznaczając wybrane zapytanie i klikając przycisk Display Estimated Execution Plan (Ctrl+L) na pasku narzędzi SQL Editor. Aby zobaczyć rzeczywisty plan, wystarczy kliknąć przycisk Include Actual Execution Plan (Ctrl+M) i uruchomić zapytanie. Z reguły wolę analizować rzeczywisty plan, ponieważ zawiera on informacje z czasu wykonania, takie jak rzeczywista liczba wierszy zwróconych przez każdy z operatorów bądź rzeczywista liczba wykonań. Gdy optymalizator dokonuje mało optymalnych wyborów, zazwyczaj winne są niedokładne oszacowania liczebności, co można wykryć, porównując oszacowania z faktycznymi pomiarami przedstawionymi w rzeczywistym planie wykonania.

Poniższy kod umożliwia mierzenie wydajności zapytania przy użyciu opcji sesji STATISTICS IO (informacje dotyczące operacji we/wy) oraz STATISTICS TIME (informacje dotyczące czasu):

```
SET STATISTICS IO, TIME ON;
```

Po uruchomieniu zapytania program SSMS przedstawi informacje o wydajności w panelu Messages.

Aby zmierzyć wydajność zapytań wykonywanych w trybie ad hoc i przesyłanych z programu SSMS z wykorzystaniem sesji Extended Events (Zdarzenia rozszerzone), należy przechwycić zdarzenie *sql_statement_completed*, dostarczając w roli filtra identyfikator sesji SSMS, z której przesyłane są zapytania. Poniższy kod służy do utworzenia i rozpoczęcia tego typu sesji Extended Events. Aby móc go użyć we własnym środowisku, czytelnik musi zastąpić przykładowy identyfikator sesji identyfikatorem własnej sesji SSMS:

```
CREATE EVENT SESSION query_performance ON SERVER
ADD EVENT sqlserver.sql_statement_completed(
    WHERE (sqlserver.session_id=(53))); -- zastąp własnym ID sesji;

ALTER EVENT SESSION query_performance ON SERVER STATE = START;
```

Aby zobaczyć informacje o zdarzeniu, należy otworzyć panel Object Explorer w programie SSMS, przejść do hierarchii folderów Management\Extended Events\Sessions, rozwinąć folder Sessions, prawym przyciskiem myszy kliknąć sesję *query_performance* i wybrać opcję Watch Live Data.

Aby wyświetlić dane wyjściowe różnych narzędzi do mierzenia wydajności, uruchomimy przykładowe zapytanie po opróżnieniu pamięci podręcznej danych. W moim środowisku wygenerowany został rzeczywisty plan wykonania zapytania przedstawiony na rysunku 2-7.

Rysunek 2-7 Plan wykonania przykładowego zapytania.

Plan jest równoległy. O wykorzystaniu równoległości w operatorze świadczy obecność ikony żółtego kółka z dwoma strzałkami. Plan przeprowadza równoległe skanowanie indeksu klastrowego wraz z zastosowaniem filtra, a następnie zbiera strumienie wierszy zwrócone przez poszczególne wątki. Warto zauważyć, że dane w planie są przesyłane z prawej strony do lewej (jak wskazuje kierunek strzałek), jednak wewnętrzne wykonanie planu rozpoczyna się od lewego węzła (zwanego również *węzłem głównym*). Węzeł główny przesyła żądanie wierszy do węzła znajdującego się po jego prawej stronie, który z kolei przesyła żądanie wierszy do następnego węzła znajdującego się po jego prawej stronie.

Poniżej zaprezentowany został wynik wygenerowany przez opcję STATISTICS IO dla tego zapytania:

```
Table 'Orders'. Scan count 9, logical reads 25339, physical reads 1, read-ahead reads
25138, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Miara *Scan count* (Liczba skanowań) wskazuje, ile razy podczas przetwarzania zapytania trzeba było uzyskać dostęp do obiektu. Moim zdaniem nazwa tej miary jest dość myląca, ponieważ liczba ta uwzględnia wszystkie operacje uzyskiwania dostępu do obiektu, nie tylko operacje skanowania. W związku z tym bardziej adekwatna byłaby prawdopodobnie nazwa *Access count* (Liczba dostępów). W tym przypadku liczba skanowań wynosi 9, ponieważ 8 wątków uzyskiwało dostęp do obiektu w operacji skanowania równoległego i dodatkowo jeden osobny wątek uzyskał dostęp do obiektu, realizując szeregową część planu.

Miara *logical reads* (odczyty logiczne) wskazuje, ile razy odczytane zostały strony znajdujące się w pamięci podręcznej danych (w tym przykładzie 25 339 razy). Jeśli strona została dwukrotnie odczytana z pamięci podręcznej danych, jest liczona dwukrotnie. Niestety narzędzie STATISTICS IO nie pokazuje, ile unikatowych stron zostało odczytanych. Miary *physical reads* (odczyty fizyczne, w przykładzie przeprowadzone 1 raz) oraz *read-ahead reads* (odczyty z wyprzedzeniem, w przykładzie przeprowadzone 25 138 razy) wskazują, ile razy strony zostały fizycznie odczytane z dysku do pamięci podręcznej danych. Jak łatwo się domyślić, odczyt fizyczny jest dużo bardziej kosztowny niż odczyt logiczny. Miara *odczyty fizyczne* reprezentuje standardowy mechanizm odczytu, który realizuje synchroniczne odczyty jednostek o rozmiarze jednej strony. Epitet *synchroniczne* sygnalizuje, że przetwarzanie zapytania nie może być kontynuowane, dopóki operacja odczytu się nie zakończy. Miara *odczyty z wyprzedzeniem* reprezentuje specjalny mechanizm odczytu, który prognozuje, jakie dane będą potrzebne do wykonania zapytania. Ten mechanizm jest nazywany *odczytem z wyprzedzeniem* lub *pobraniem z wyprzedzeniem*. SQL Server decyduje się na odczyty z wyprzedzeniem zazwyczaj wtedy, gdy skanowana jest wystarczająca ilość danych. Ten mechanizm bazuje na odczytach asynchronicznych i może odczytywać fragmenty większe niż pojedyncze strony, aż do 512 KB w pojedynczym odczycie. Aby usprawnić diagnozowanie problemów, można wyłączyć mechanizm odczytu z wyprzedzeniem przy pomocy flagi śledzenia 652 (do włączania służy polecenie DBCC TRACEON, a do wyłączania polecenie DBCC TRACEOFF). Aby obliczyć całkowitą liczbę odczytów fizycznych wykonanych w ramach realizacji danego zapytania, należy zsumować miary *odczyty fizyczne* oraz *odczyty z wyprzedzeniem*. W tym przypadku suma wyniosła 25 139.

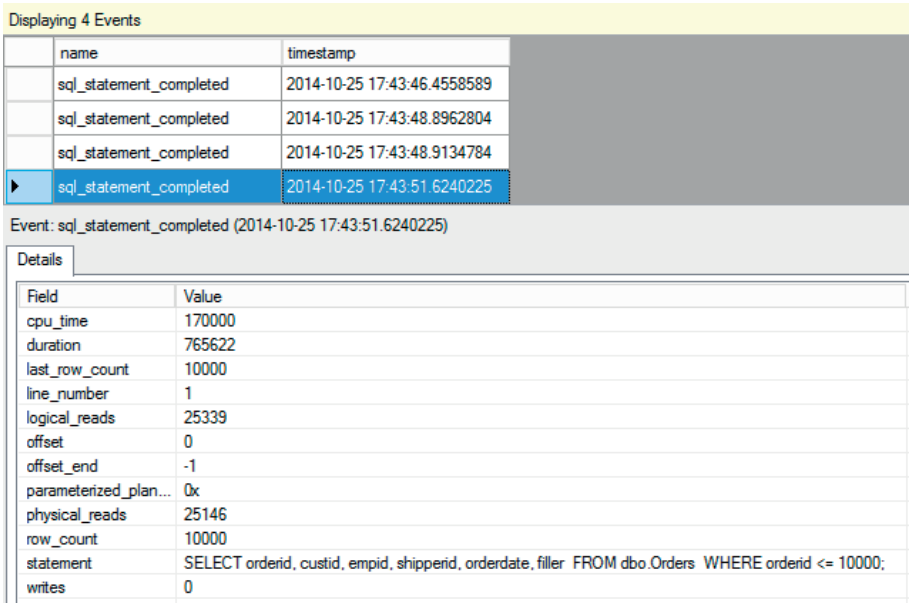
Wynik opcji STATISTICS IO prezentuje osobno odczyty logiczne, odczyty fizyczne oraz odczyty z wyprzedzeniem wykonywane na typie LOB (Large Object Types). Ten przykład nie wymagał przetwarzania danych typu LOB, więc wszystkie trzy miary wynoszą zero.

Opcja STATISTICS TIME prezentuje następujące statystyki czasowe dla przykładowego zapytania:

```
SQL Server Execution Times:
   CPU time = 170 ms,  elapsed time = 765 ms.
```

Opcja ta oferuje kilka zestawów miar. Zwraca nie tylko wskaźniki dotyczące czasu wykonania, ale także fazy analizowania i kompilacji zapytania. Ponadto przedstawia miary dla każdego elementu włączonego w programie SSMS, który wymaga od programu SQL Server wykonania dodatkowej pracy (jak np. graficzny plan wykonania). Wiersz wynikowy STATISTICS TIME, wyświetlony tuż pod wierszem wynikowym STATISTICS IO reprezentuje wykonanie zapytania. Aby przetworzyć zapytanie w kontekście nieprzygotowanej pamięci podręcznej, SQL Server potrzebował 170 ms czasu procesora, a wykonanie zapytania zajęło 765 ms zegarowych.

Na rysunku 2-8 przedstawione zostało okno Watch Live Data dla sesji Extended Events o nazwie *query_performance* (z zaznaczonym wykonaniem przykładowego zapytania przy użyciu nieprzygotowanej pamięci podręcznej).



Displaying 4 Events

name	timestamp
sql_statement_completed	2014-10-25 17:43:46.4558589
sql_statement_completed	2014-10-25 17:43:48.8962804
sql_statement_completed	2014-10-25 17:43:48.9134784
sql_statement_completed	2014-10-25 17:43:51.6240225

Event: sql_statement_completed (2014-10-25 17:43:51.6240225)

Details

Field	Value
cpu_time	170000
duration	765622
last_row_count	10000
line_number	1
logical_reads	25339
offset	0
offset_end	-1
parameterized_plan...	0x
physical_reads	25146
row_count	10000
statement	SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderid <= 10000;
writes	0

Rysunek 2-8 Informacje z sesji Extended Events.

Pamięć podręczna jest już wypełniona danymi, więc gdy ponownie uruchomimy zapytanie, możemy uzyskać następujące dane wyjściowe opcji STATISTICS IO:

```
Table 'Orders'. Scan count 9, logical reads 25339, physical reads 0, read-ahead reads
0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Jak widać, miary reprezentujące odczyty fizyczne są równe zeru. Opcja STATISTICS TIME zwróciła następujący wynik:

```
CPU time = 265 ms,  elapsed time = 299 ms.
```

Gdy pamięć podręczna jest wypełniona odpowiednimi danymi, wykonanie zapytania zajmuje oczywiście mniej czasu.

Metody dostępu

W tym podrozdziale przedstawię różne metody wykorzystywane przez SQL Server do uzyskiwania dostępu do danych. Do tego omówienia będę się odwoływać, analizując plany wykonania prezentowane w dalszej części książki.

W przedstawianych w tym podrozdziale przykładach będziemy wykonywać zapytania na tabeli Orders z bazy danych PerfromanceV3. Ta tabela ma strukturę B-drzewa i został w niej zdefiniowany indeks klastrowy z kolumną *orderdate* w roli klucza. Przy pomocy poniższego fragmentu kodu utworzymy kopię tabeli Orders o nazwie *Orders2* i strukturze sterty:

```
IF OBJECT_ID(N'dbo.Orders2', N'U') IS NOT NULL DROP TABLE dbo.Orders2;
SELECT * INTO dbo.Orders2 FROM dbo.Orders;
ALTER TABLE dbo.Orders2 ADD CONSTRAINT PK_Orders2 PRIMARY KEY NONCLUSTERED (orderid);
```

Demonstrując metody dostępu do tabeli o strukturze B-drzewa, będziemy wykonywać zapytania na tabeli Orders. Natomiast do zilustrowania metod dostępu do tabeli o strukturze sterty posłuży tabela Orders2.

Ważne Do wypełnienia bazy danych PerformanceV3 przykładowymi danymi użyłem generatora danych losowych. W związku z tym czytelnicy uzyskają prawdopodobnie nieco inne wyniki, wykonując zapytania na obiektach tej bazy danych.

Największy koszt realizowania metod dostępu związany jest z operacjami we/wy. W związku z tym liczba odczytów fizycznych będzie traktowana jako podstawowa miara wydajności metody dostępu.

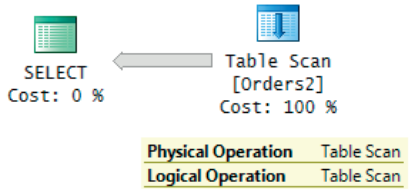
Skanowanie tabeli/nieuporządkowane skanowanie indeksu klastrowego

Pierwszą omawianą metodą dostępu będzie pełne skanowanie tabeli, gdy nie ma konieczności zwracania danych w żadnej konkretnej kolejności. Tego typu pełne skanowanie tabeli występuje zazwyczaj w dwóch sytuacjach: gdy potrzebne są wszystkie wiersze tabeli lub gdy potrzebny jest jedynie podzbiór wierszy tabeli, ale nie istnieje żaden indeks, który mógłby ułatwić filtrowanie. Problem polega na tym, że plan zapytania dla tego typu pełnego skanowania wygląda inaczej w zależności od struktury tabeli (sterta lub B-drzewo). Gdy tabela ma strukturę sterty, plan zawiera operator o nazwie Table Scan (Skanowanie Tabeli). Natomiast gdy tabela ma strukturę B-drzewa, plan zawiera operator o nazwie Clustered Index Scan (Skanowanie indeksu klastrowego) z właściwością Ordered: False (Uporządkowane: Fałsz). Pierwszą operację będziemy nazywać *skanowaniem tabeli* (ang. *table scan*), natomiast drugą *nieuporządkowanym skanowaniem indeksu klastrowego* (ang. *unordered clustered index scan*).

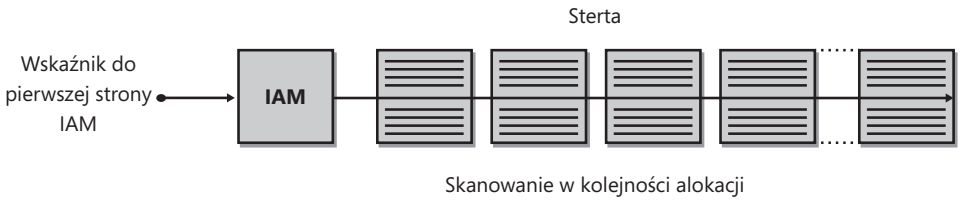
Operacje *skanowania tabeli* oraz *nieuporządkowanego skanowania indeksu klastrowego* wiążą się ze skanowaniem wszystkich stron danych należących do tabeli. Poniższe zapytanie na tabeli Orders2, która ma strukturę sterty, wymagałoby przeprowadzenia skanowania tabeli:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders2;
```

Rysunek 2-9 przedstawia graficzny plan wykonania wygenerowany dla tego zapytania przez optymalizator aparatu relacyjnego, natomiast rysunek 2-10 ilustruje, w jaki sposób ta metoda dostępu jest realizowana przez aparat magazynu.



Rysunek 2-9 Skanowanie sterty (plan wykonania)



Rysunek 2-10 Skanowanie sterty (aparat magazynu)

W moim systemie SQL Server zgłosił dla tego zapytania 24 396 odczytów logicznych.

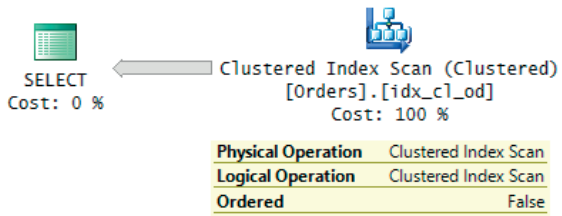
Fakt, iż zarówno właściwości Physical Operation (Operacja fizyczna), jak i Logical Operation (Operacja logiczna) przypisana została wartość Table Scan (Skanowanie tabeli), może wydawać się nieco mylący. Moim zdaniem, operacja logiczna to skanowanie tabeli, natomiast operacja fizyczna to skanowanie sterty.

Jedyny możliwy sposób przetwarzania operatora Table Scan (Skanowanie tabeli) przez aparat magazynu stanowi skanowanie w kolejności alokacji. Jak wspomniałem w poprzedniej części rozdziału zatytułowanej „Struktura tabel", skanowanie w kolejności alokacji jest realizowane w oparciu o strony IAM w kolejności pliku.

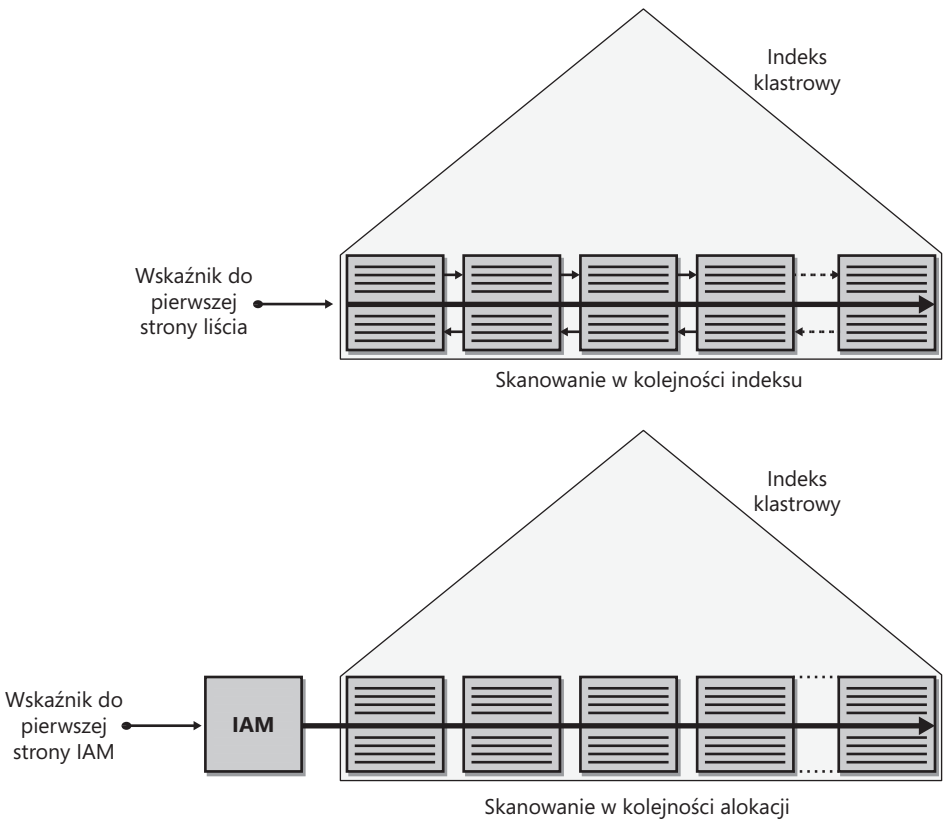
Poniższe zapytanie na tabeli Orders, która ma strukturę B-drzewa, wymagałoby przeprowadzenia nieuporządkowanego skanowania indeksu klastrowego:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders;
```

Rysunek 2-11 przedstawia plan wykonania wygenerowany przez optymalizator dla tego zapytania. Jak widać, właściwość Ordered (Uporządkowane) operatora Clustered Index Scan (Skanowanie indeksu klastrowego) ma wartość False (Fałsz). Rysunek 2-12 prezentuje dwa możliwe sposoby realizowania tej metody dostępu przez aparat magazynu.



Rysunek 2-11 Skanowanie B-drzewa (plan wykonania)



Rysunek 2-12 Skanowanie B-drzewa (aparat magazynu)

W moim systemie SQL Server zgłosił dla tego zapytania 25 073 odczyty logiczne.

Tu również nieco mylący wydaje się fakt, iż plan zawiera tę samą wartość Clustered Index Scan (Skanowanie indeksu klastrowego) we właściwościach Physical Operation (Operacja fizyczna) oraz Logical Operation (Operacja logiczna). Moim zdaniem operacja logiczna to skanowanie tabeli, natomiast operacja fizyczna to skanowanie indeksu klastrowego.

Fakt, iż właściwość Ordered operatora Clustered Index Scan ma wartość False, oznacza, że aparat relacyjny nie wymaga, aby operator zwracał dane w kolejności klucza. To nie oznacza, że zwrócenie posortowanych danych stanowiłoby problem; akceptowana jest dowolna kolejność. Dzięki temu aparat magazynu może wybrać jeden z dwóch rodzajów operacji skanowania: uporządkowane skanowanie indeksu (skanowanie liści indeksu zgodnie z kierunkiem odpowiedniej listy) lub skanowanie w kolejności alokacji (skanowanie w oparciu o strony mapy IAM). Aparat magazynu wybiera odpowiedni rodzaj skanowania, uwzględniając takie czynniki jak wydajność oraz spójność danych. Omówieniem procesu podejmowania decyzji przez aparat magazynu zajmę się po opisaniu operatorów Clustered Index Scan (Uporządkowane skanowanie indeksu klastrowego) oraz Index Scan (Skanowanie indeksu) z właściwością Ordered: True (Uporządkowane: Prawda).

Nieuporządkowane skanowanie pokrywającego indeksu nieklastrowego

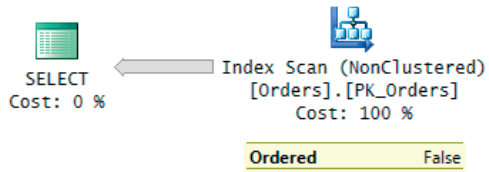
Nieuporządkowane skanowanie pokrywającego indeksu nieklastrowego (ang. *unordered covering nonclustered index scan*) przypomina nieuporządkowane skanowanie indeksu klastrowego. Termin *indeks pokrywający* oznacza, że indeks nieklastrowy zawiera wszystkie kolumny określone w zapytaniu. Indeks pokrywający nie ma żadnych szczególnych cech; właściwość pokrywania dotyczy tylko zakresu konkretnego zapytania. SQL Server może znaleźć wszystkie dane potrzebne do zrealizowania zapytania w danych indeksu, bez konieczności uzyskiwania dostępu do pełnych wierszy danych. Pod pozostałymi względami ta metoda dostępu nie różni się niczym od nieuporządkowanego skanowania indeksu klastrowego. Oczywiście poziom liści pokrywającego indeksu nieklastrowego zawiera mniej stron niż poziom liści indeksu klastrowego, ponieważ jego wiersze są mniejsze i można pomieścić ich więcej na stronie. We wcześniejszej części rozdziału zaprezentowałem metody wyznaczania liczby stron na poziomie liści indeksu (klastrowego i nieklastrowego).

Przykład omawianej metody dostępu może stanowić poniższe zapytanie, które zwraca wszystkie wartości *orderid* z tabeli Orders:

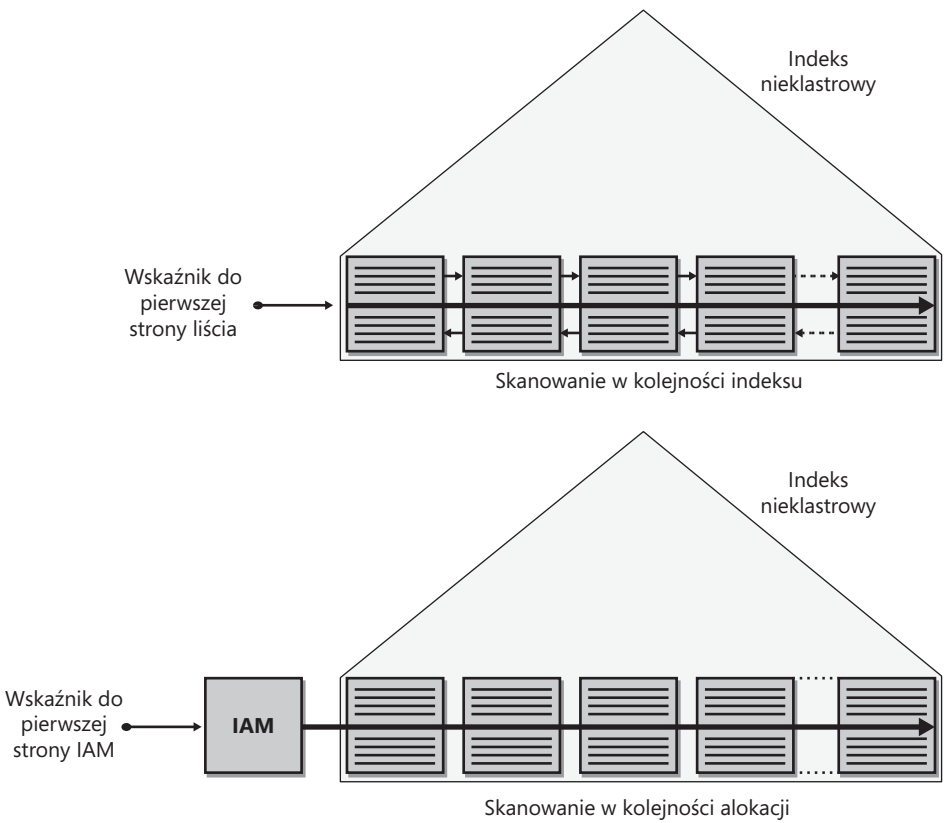
```
SELECT orderid
FROM dbo.Orders;
```

Tabela Orders ma indeks nieklastrowy zdefiniowany na kolumnie *orderid* (*PK_Orders*), co oznacza, że wszystkie identyfikatory *orderid* tabeli znajdują się na poziomie liści

indeksu. Indeks pokrywa przykładowe zapytanie. Na rysunku 2-13 przedstawiony został graficzny plan wykonania tego zapytania, natomiast na rysunku 2-14 dwa potencjalne sposoby przetworzenia go przez aparat magazynu.



Rysunek 2-13 Nieuporządkowane skanowanie pokrywającego indeksu nieklastrowego (plan wykonania)



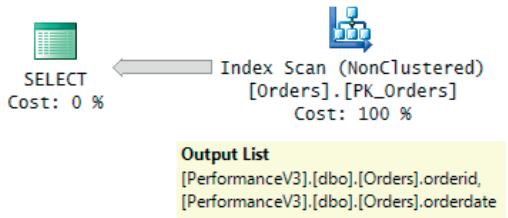
Rysunek 2-14 Nieuporządkowane skanowanie pokrywającego indeksu nieklastrowego (aparat magazynu)

W moim systemie SQL Server zgłosił dla tego zapytania 2611 odczytów logicznych, ponieważ tyle stron znajduje się na poziomie liści indeksu *PK_Orders*.

A oto mała łamigłówka. Dodajmy do zapytania kolumnę *orderdate*:

```
SELECT orderid, orderdate
FROM dbo.Orders;
```

Przeanalizujmy plan wykonania powyższego zapytania, przedstawiony na rysunku 2-15.



Rysunek 2-15 Indeks zawierający klucz klastrowania.

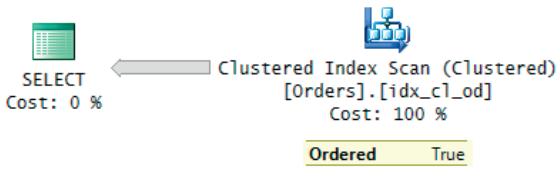
Jak widać, indeks *PK_Orders* nadal traktowany jest jako indeks pokrywający zapytanie. Pytanie brzmi, jak to możliwe, skoro indeks został zdefiniowany tylko na kolumnie *orderid* w roli klucza? Rozwiązanie zagadki stanowi fakt, iż tabela zawiera indeks klastrowy o kluczu *orderdate*. Jak wspomniałem wcześniej, SQL Server wykorzystuje klucze indeksu klastrowego w roli lokalizatorów wierszy w indeksach nieklastrowych. W związku z tym, mimo iż indeks *PK_Orders* został bezpośrednio zdefiniowany tylko na kolumnie *orderid*, w rzeczywistości SQL Server objął nim kolumny *orderid* oraz *orderdate*. Nieważne, że SQL Server dodał kolumnę *orderdate* w roli lokalizatora wiersza. Skoro została ona umieszczona w indeksie, może posłużyć także do celów związanych z przetwarzaniem zapytań.

Uporządkowane skanowanie indeksu klastrowego

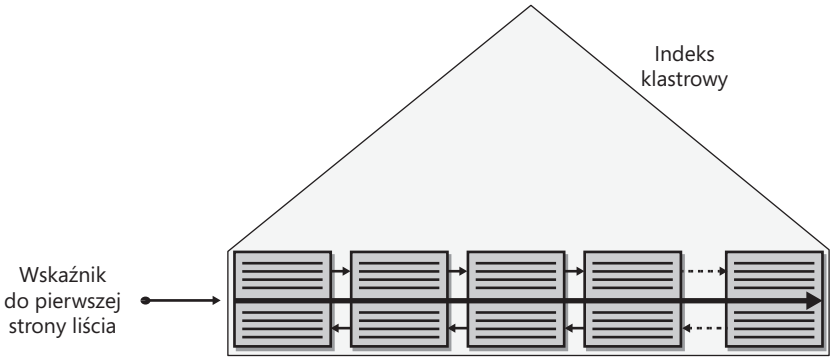
Uporządkowane skanowanie indeksu klastrowego (ang. *ordered clustered index scan*) to operacja pełnego skanowania poziomu liści indeksu klastrowego, która gwarantuje, że dane przekazane do kolejnego operatora będą uporządkowane w kolejności indeksu. Ta metoda dostępu zostanie na przykład umieszczona w planie wykonania poniższego zapytania, w którym wszystkie zamówienia są posortowane według atrybutu *orderdate*:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
ORDER BY orderdate;
```

Plan wykonania tego zapytania został zaprezentowany na rysunku 2-16, a sposób realizowania tej metody dostępu przez aparat magazynu na rysunku 2-17.



Rysunek 2-16 Uporządkowane skanowanie indeksu klastrowego (plan wykonania)



Rysunek 2-17 Uporządkowane skanowanie indeksu klastrowego (aparat magazynu)

W moim systemie SQL Server zgłosił dla tego zapytania 25 073 odczyty logiczne.

Jak można zauważyć, właściwość Ordered ma wartość True. To oznacza, że dane zwracane przez operator muszą być uporządkowane. Gdy operator zawiera właściwość Ordered: True, aparat magazynu może przeprowadzić operację skanowania tylko w jeden sposób, a mianowicie przeprowadzając uporządkowane skanowanie indeksu (skanowanie w oparciu o listę powiązaną z indeksem), jak pokazano na rysunku 2-17. Wydajność uporządkowanego skanowania indeksu, w odróżnieniu od skanowania w kolejności alokacji, zależy od poziomu fragmentacji indeksu. Gdyby fragmentacja w ogóle nie występowała, wydajność uporządkowanego skanowania indeksu powinna być bardzo zbliżona do wydajności skanowania w kolejności alokacji, ponieważ obie operacje wiązałyby się z sekwencyjnym odczytywaniem danych zgodnie z kolejnością w pliku. Jednak gdy pamięć podręczna nie jest wypełniona danymi, a poziom fragmentacji rośnie, różnica wydajności jest dużo większa (oczywiście na korzyść skanowania w kolejności alokacji). W związku z tym, jeśli nie jest to konieczne, nie należy żądać posortowanych danych, aby umożliwić wybór skanowania w kolejności alokacji. Ponadto należy rozwiązać problem fragmentacji indeksów, które służą do uporządkowanego skanowania dużej ilości danych bez przygotowanej pamięci podręcznej. Do kwestii fragmentacji i jej eliminowania powrócę w dalszej części rozdziału.

Warto mieć świadomość, że optymalizator niekoniecznie musi przeprowadzać skanowanie w kierunku zgodnym z porządkiem indeksu. Indeksy wykorzystują listy dwukierunkowe, w których każda strona zawiera zarówno wskaźnik *następny*, jak w wskaźnik *poprzedni*. Gdybyśmy zażądali posortowania danych w kolejności

malejącej, przeprowadzone zostałoby uporządkowane skanowanie indeksu, ale indeks byłby skanowany od końca (od ogona do głowy), zamiast od początku (od głowy do ogona). Warto jednak mieć świadomość, że w wersji SQL Server 2014 aparat magazynu stosuje równoległość tylko w przypadku uporządkowanego skanowania w przód. Operacja uporządkowanego skanowania w tył jest zawsze przeprowadzana w sposób szeregowy.

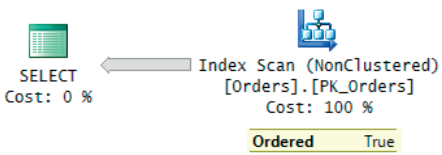
SQL Server wspiera również malejące indeksy. Warto je zastosować, gdy równoległość ma duży wpływ na wydajność zapytania. Inny argument przemawiający za zastosowaniem tego typu indeksów stanowi zapewnienie wsparcia dla sortowania kolumn klucza w różnych kierunkach, na przykład według *col1*, *col2* DESC. Zastosowania indeksów malejących zademonstruję w dalszej części rozdziału, zatytułowanej „Funkcje indeksowania".

Uporządkowane skanowanie pokrywającego indeksu nieklastrowego

Uporządkowane skanowanie pokrywającego indeksu nieklastrowego (ang. *ordered covering nonclustered index scan*) przypomina uporządkowane skanowanie indeksu klastrowego – z tą różnicą, że metoda dostępu operuje na indeksie nieklastrowym (zazwyczaj gdy indeks pokrywa zapytanie). Koszt tej operacji jest rzecz jasna niższy, niż koszt skanowania indeksu klastrowego, ponieważ wymaga odczytania mniejszej liczby stron. Na przykład indeks *PK_Orders* na tabeli klastrowej Orders pokrywa następujące zapytanie i zawiera dane uporządkowane w odpowiedniej kolejności:

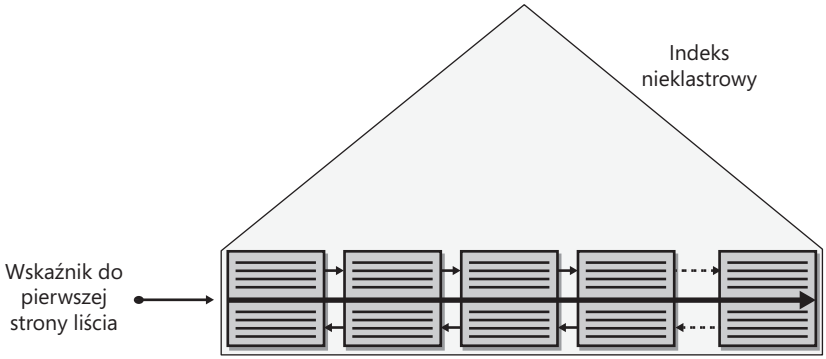
```
SELECT orderid, orderdate
FROM dbo.Orders
ORDER BY orderid;
```

Rysunek 2-18 prezentuje plan wykonania zapytania, natomiast rysunek 2-19 ilustruje sposób realizowania metody dostępu przez aparat magazynu.



Rysunek 2-18 Uporządkowane skanowanie pokrywającego indeksu nieklastrowego (plan wykonania)

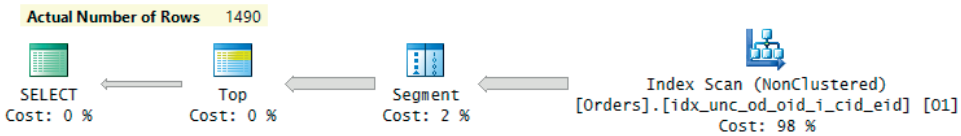
Jak widać, właściwość Ordered (Uporządkowany) operatora Index Scan (Skanowanie indeksu) w planie ma wartość True (Prawda).



Rysunek 2-19 Uporządkowane skanowanie pokrywającego indeksu nieklastrowego (aparat magazynu)

Uporządkowane skanowanie indeksu jest przeprowadzane nie tylko wtedy, gdy zapytanie zawiera jawne żądanie sortowania danych, ale też gdy plan zawiera operator, którego działanie zostaje usprawnione w wyniku posortowania danych wejściowych. Taka sytuacja może nastąpić w związku z przetwarzaniem klauzul GROUP BY, DISTINCT, złączaniem tabel lub w innych, mniej oczywistych sytuacjach. Przyjrzyjmy się na przykład planowi wykonania wygenerowanemu dla następującego zapytania (przedstawionemu na rysunku 2-20):

```
SELECT orderid, custid, empid, orderdate
FROM dbo.Orders AS O1
WHERE orderid =
  (SELECT MAX(orderid)
   FROM dbo.Orders AS O2
   WHERE O2.orderdate = O1.orderdate);
```



Rysunek 2-20 Uporządkowane skanowanie pokrywającego indeksu nieklastrowego z segmentacją

Operator Segment (Segmentacja) organizuje dane w grupy i przesyła kolejne grupy do następnego operatora (w tym przypadku operatora Top). Nasze zapytanie zwraca zamówienia z maksymalną wartością *orderid* dla każdej wartości *orderdate*. Na szczęście posiadamy odpowiedni indeks pokrywający (*idx_unc_od_oid_i_cid_eid*) z kolumnami klucza (*orderdate, orderid*) oraz dołączonymi kolumnami niebędącymi kolumnami klucza (*custid, empid*). Do koncepcji dołączania kolumn, które nie są kolumnami klucza, powrócę w dalszej części rozdziału. Na razie wystarczy mieć świadomość,

że operator Segment organizuje dane w grupy na podstawie wartości *orderdate* i przesyła je w postaci kolejnych pojedynczych grup, w których ostatni wiersz zawiera maksymalną wartość *orderid* w danej grupie, ponieważ *orderid* stanowi drugą kolumnę klucza po kolumnie *orderdate*. W związku z tym plan nie musi obejmować osobnej operacji sortowania danych, ponieważ zostają one zebrane w odpowiedniej kolejności w wyniku uporządkowanego skanowania pokrywającego indeksu posortowanego według *orderdate* oraz *orderid*. Operator Top ma za zadanie jedynie pobrać ostatni wiersz (TOP 1 w kolejności malejącej), który stanowi odpowiedni wiersz dla danej grupy. Operator Top zwróci 1490 wierszy, ponieważ tyle istnieje unikatowych grup (wartości *orderdate*) i dla każdej z nich operator zwraca jeden wiersz. Indeks nieklastrowy pokrywa zapytanie, ponieważ zawiera na poziomie liści wszystkie pozostałe kolumny wspomniane w zapytaniu (*custid, empid*). Dlatego nie ma potrzeby wyszukiwania wierszy danych i można zrealizować zapytanie przy pomocy samego indeksu.

Skanowanie w wykonaniu aparatu magazynu

Zanim przejdę do omówienia kolejnych metod dostępu, wyjaśnię, jak aparat magazynu obsługuje otrzymane od aparatu relacyjnego instrukcje skanowania. *Aparat relacyjny* można porównać do mózgu programu SQL Server. W jego skład wchodzi optymalizator, który zarządza generowaniem planów wykonania zapytań. Natomiast *aparat magazynu* to w pewnym sensie mięśnie programu SQL Server – odpowiada za realizowanie instrukcji otrzymywanych od aparatu relacyjnego w planie wykonania i przeprowadzanie fizycznych operacji na wierszach. Czasami instrukcje optymalizatora pozostawiają aparatowi magazynu pewną swobodę i w takich sytuacjach to aparat magazynu wybiera najlepszą z dostępnych opcji, uwzględniając takie czynniki, jak wydajność czy spójność danych.

Skanowanie w kolejności alokacji a uporządkowane skanowanie indeksu

Gdy plan zawiera operator Table Scan (Skanowanie tabeli), aparat magazynu ma tylko jedną możliwość: skanowanie w kolejności alokacji. A gdy plan zawiera operator Index Scan (Skanowanie indeksu klastrowego lub nieklastrowego) z właściwością Ordered: True (Uporządkowane: Prawda), aparat magazynu musi przeprowadzić uporządkowane skanowanie indeksu.

Natomiast plan z operatorem Index Scan (Skanowanie indeksu) z właściwością Ordered: False (Uporządkowane: Fałsz) sygnalizuje, że aparat relacyjny nie przykłada wagi do kolejności zwracanych wierszy. W takiej sytuacji istnieją dwie możliwości: skanowanie w kolejności alokacji oraz uporządkowane skanowanie indeksu. Aparat magazynu może wybrać dowolną z nich. Niestety finalna decyzja podjęta przez aparat magazynu nie zostaje udokumentowana w planie wykonania ani w innych raportach. Wyjaśnię, jak aparat magazynu podejmuje decyzję, choć trzeba pamiętać, że plan

wykonania przedstawia jedynie instrukcje aparatu relacyjnego, a nie akcje podjęte przez aparat magazynu.

Wydajność operacji skanowania w kolejności alokacji nie zależy od fragmentacji logicznej indeksu, ponieważ i tak jest wykonywana zgodnie z kolejnością pliku. Natomiast gdy uporządkowane skanowanie indeksu wiąże się z dokonywaniem fizycznych odczytów, im wyższa fragmentacja, tym wolniejsze skanowanie. W związku z tym aparat magazynu ocenia skanowanie w kolejności alokacji jako bardziej wydajną opcję. Wyjątek stanowią bardzo małe indeksy (do 64 stron). W takiej sytuacji koszt interpretowania stron IAM stanowi spory koszt na tle pozostałych zadań i aparat magazynu preferuje uporządkowane skanowanie indeksu. Jednak w pozostałych sytuacjach skanowanie w kolejności alokacji stanowi preferowaną opcję w kontekście wydajności.

Jednakże wydajność to nie jedyny aspekt brany pod uwagę przez aparat magazynu, uwzględniana jest również oczekiwana spójność danych wynikająca z obowiązującego poziomu izolacji. Gdy istnieje więcej niż jeden sposób spełniania żądania, aparat magazynu wybiera najszybszą opcję, która spełnia wymóg spójności.

W niektórych sytuacjach operacja skanowania zwraca wiele wystąpień wiersza lub nawet pomija niektóre wiersze. Operacje skanowania w kolejności alokacji są bardziej podatne na tego typu problemy niż operacje uporządkowanego skanowania indeksu. Zacznę od wyjaśnienia, kiedy i w jakich okolicznościach taki fenomen może zaistnieć w wyniku skanowania w kolejności alokacji. Później wyjaśnię, kiedy może on zaistnieć w wyniku uporządkowanego skanowania indeksu.

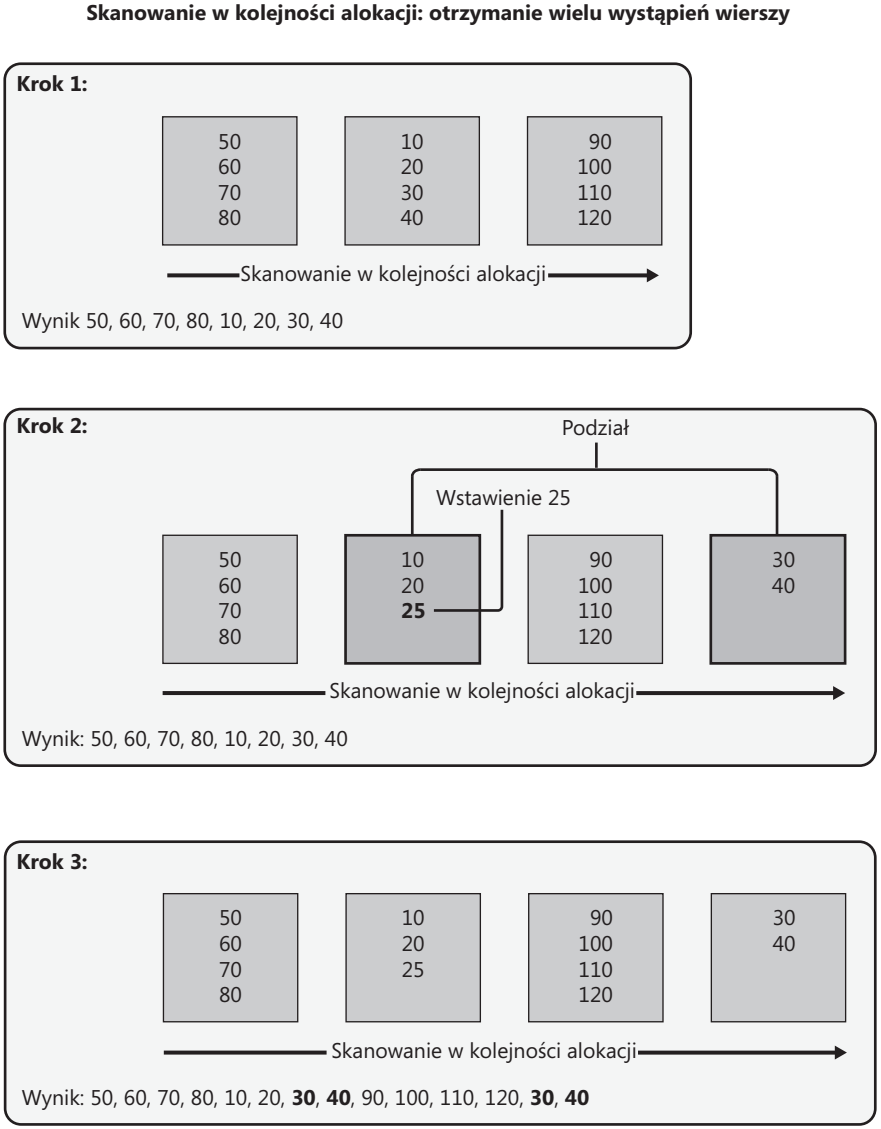
Skanowanie w kolejności alokacji

Rysunek 2-21 ilustruje trzy kroki, które mogą doprowadzić do tego, że skanowanie w kolejności alokacji zwróci wiele wystąpień wierszy.

Krok 1 ilustruje trwający proces skanowania w kolejności alokacji, w którym strony liści pewnego indeksu są odczytywane zgodnie z porządkiem pliku (nie indeksu). Dwie strony zostały już odczytane (klucze 50, 60, 70, 80, 10, 20, 30, 40). W tym momencie, przed odczytaniem trzeciej strony indeksu, ktoś wstawia do tabeli wiersz z kluczem 25.

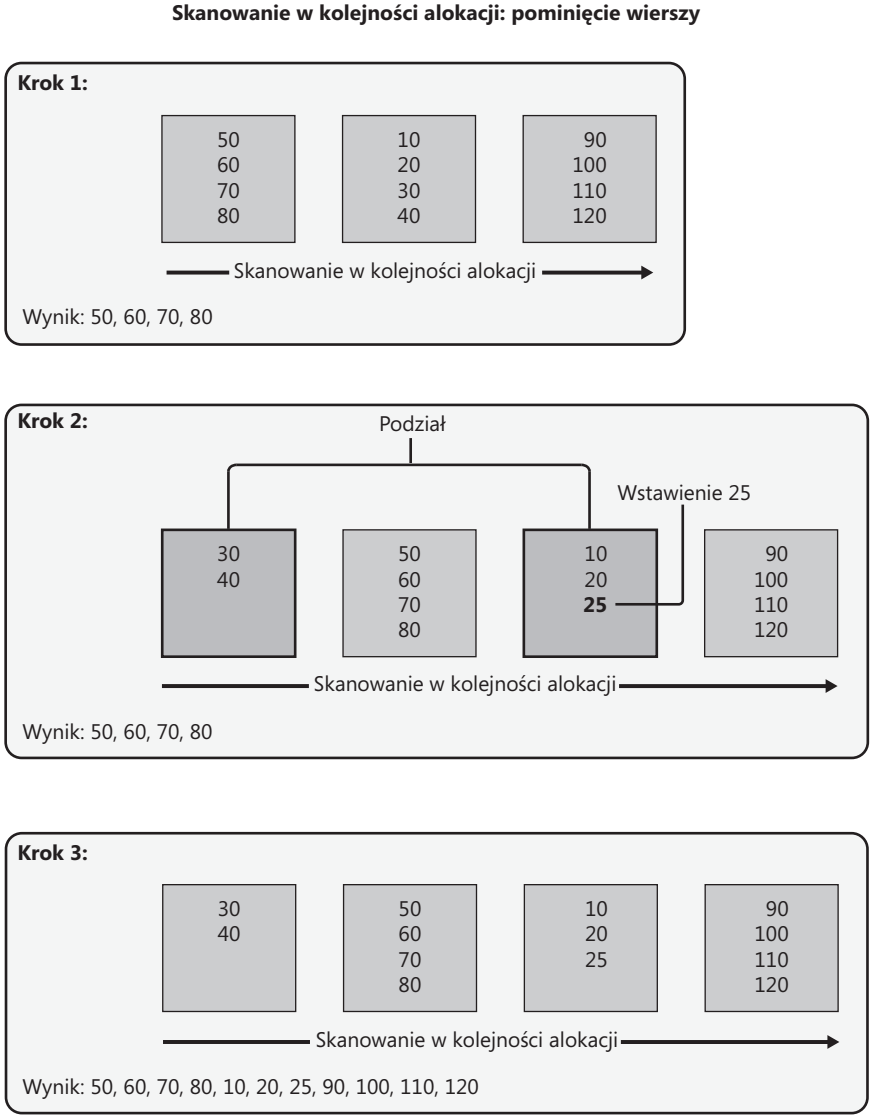
Krok 2 ilustruje podział strony docelowej operacji wstawiania, który nastąpił, ponieważ strona ta była pełna. W związku z podziałem alokowana została nowa strona (w tym przypadku w dalszej części pliku – w miejscu, do którego operacja skanowania jeszcze nie dotarła). Połowa wierszy ze starej strony zostaje przeniesiona na nową stronę (klucze 30, 40), natomiast nowy wiersz zostaje dodany do starej strony ze względu na wartość klucza (klucz 25).

Krok 3 ilustruje kontynuację procesu skanowania: odczyt dwóch pozostałych stron (klucze 90, 100, 110, 120, 30, 40), łącznie ze stroną dodaną w wyniku podziału. Jak widać, wiersze o kluczach 30 oraz 40 zostały odczytane po raz drugi.



Rysunek 2-21 Skanowanie w kolejności alokacji: otrzymanie wielu wystąpień wierszy.

Podobny mechanizm może doprowadzić do pominięcia wierszy – w zależności od tego, jak daleko posunął się proces skanowania do momentu przeprowadzenia podziału oraz gdzie alokowana została nowa strona. Rysunek 2-22 ilustruje, jak może dojść do tego typu sytuacji (w trzech krokach).



Rysunek 2-22 Skanowanie w kolejności alokacji: pominięcie wierszy.

Krok 1 ilustruje trwający proces skanowania w kolejności alokacji, który odczytał już jedną stronę (klucze 50, 60, 70, 80), zanim nastąpiła operacja wstawiania.

Krok 2 ilustruje podział strony docelowej – tym razem nowa strona została umieszczona we wcześniejszej części pliku, przez którą operacja skanowania już przeszła. Podobnie jak w przypadku przedstawionej wcześniej operacji podziału, wiersze o kluczach 30 i 40 zostają przeniesione na nową stronę, natomiast nowy wiersz z kluczem 25 zostaje umieszczony na starej stronie.

Krok 3 ilustruje kontynuację procesu skanowania: odczyt dwóch pozostałych stron (klucze 10, 20, 25, 90, 100, 110, 120). Jak widać, wiersze z kluczami 30 oraz 40 zostały całkowicie pominięte.

Podsumowując, operacja skanowania w kolejności alokacji może zwrócić wiele wystąpień wierszy lub pomijać wiersze w związku z podziałami, które następują w trakcie przeprowadzania operacji. Podział może być konsekwencją wstawienia nowego wiersza, aktualizacji klucza indeksu powodującej konieczność przeniesienia wiersza lub aktualizacji kolumny o zmiennej długości skutkującej wzrostem rozmiaru wiersza. Należy pamiętać, że podziały nie występują na stertach, a jedynie w indeksach. W związku z tym tego typu fenomen nie może mieć miejsca, gdy tabela ma strukturę sterty.

Operacja uporządkowanego skanowania indeksu jest bezpieczniejsza pod tym względem, że nie odczytuje wielu wystąpień tego samego wiersza ani nie pomija wierszy w wyniku podziałów. Jak pamiętamy, uporządkowane skanowanie indeksu jest przeprowadzane zgodnie z kolejnością listy indeksu. Jeśli strona, do której nie dotarła jeszcze operacja skanowania, zostanie podzielona, obie strony zostaną odczytane w ramach skanowania, a zatem wiersze nie zostaną pominięte. Natomiast jeśli podzielona zostanie strona, która została już odczytana przez operację skanowania, odczyt nowej strony nie nastąpi, a zatem nie zostaną zwrócone powtórzone wystąpienia wiersza.

Aparat magazynu zdaje sobie sprawę z tego, że operacje skanowania w kolejności alokacji, w odróżnieniu od operacji uporządkowanego skanowania indeksu, są podatne na tego typu niezgodne odczyty wynikające z podziałów stron. W związku z tym sytuacje, w których operacje Index Scan *Ordered: False* realizowane są poprzez skanowanie w kolejności alokacji, można podzielić na dwie kategorie: *bezpieczną* i *niebezpieczną*.

Sytuacje należące do kategorii niebezpiecznych to takie, w których skanowanie może skutkować zwróceniem wielu wystąpień wierszy lub pominięciem wierszy w wyniku podziałów. Aparat magazynu wykorzystuje tę opcję, gdy rozmiar indeksu przekracza 64 strony i żądanie zostało uruchomione na poziomie izolacji READ UNCOMMITTED (Odczyt niezatwierdzonych danych) – na przykład w wyniku zastosowania wskazówki NOLOCK. Powszechnie uważa się, że poziom izolacji READ UNCOMMITTED oznacza po prostu, że zapytanie nie wymaga założenia blokady współużytkowania i w związku z tym może odczytywać niezatwierdzone zmiany (brudne odczyty). Niestety większość osób nie zdaje sobie sprawy z tego, że poziom izolacji READ UNCOMMITTED sygnalizuje również aparatowi magazynu, że spójność nie stanowi kwestii priorytetowej. Innymi słowy, wybiera on szybszą opcję, nawet jeśli wiąże się ona ze zwracaniem wielu wystąpień wierszy bądź pomijaniem wierszy. Gdy zapytanie jest uruchomione na domyślnym poziomie izolacji READ COMMITTED (Odczytywanie zatwierdzonych) lub wyższym, aparat magazynu wybiera uporządkowane skanowanie indeksu, aby zapobiec przedstawionym fenomenom będącym skutkiem podziału