

DANUTA MENDRALA
MARCIN SZELIGA

WYDANIE III

SQL



PRAKTYCZNY
KURS

Helion

Danuta Mendrala, Marcin Szeliga

Praktyczny kurs SQL

Wydanie III

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

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

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

Redaktor prowadzący: Michał Mrowiec

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/pksql3_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przykady/pksql3.zip>

ISBN: ePub: 978-83-283-0944-9, Mobi: 978-83-283-0945-6

Copyright © Helion 2015

- [Poleć książkę](#)
- [Kup w wersji papierowej](#)

- [Oceń książkę](#)

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

Wstęp

Relacyjne bazy danych pozwalają łatwo, wydajnie i tanim kosztem przechowywać duże ilości najróżniejszych informacji. Nic więc dziwnego, że od 20 lat są ważną częścią większości programów, w tym stron WWW, aplikacji księgowych i magazynowych, narzędzi programistycznych i tak dalej.

Pracując z zapisanymi w bazie informacjami, używamy **serwera bazodanowego — to on odpowiada za bezpieczeństwo i spójność przechowywanych danych oraz pozwala je wydajnie modyfikować i odczytywać**. Komunikujemy się z nim z reguły za pośrednictwem programu klienckiego, np. chcąc wyświetlić informacje o towarach, klikamy odpowiedni przycisk, a następnie w celu ich posortowania klikamy nagłówek odpowiedniej kolumny. Możemy jednak łączyć się bezpośrednio z serwerem i samodzielnie odczytywać przechowywane w nim informacje — ale musimy w tym celu znać język SQL.

 Wskazówka	SQL ^[1] , czyli strukturalny język zapytań (ang. <i>Structured Query Language</i>), umożliwia odczytywanie i modyfikowanie przechowywanych w bazie danych informacji. Jest on też podstawowym językiem programowania baz danych, czyli pozwala na tworzenie i modyfikowanie obiektów bazodanowych, takich jak tabele czy procedury.
--	---

Znajomość języka SQL jest obowiązkowa dla programistów^[2] i niezbędna dla administratorów (nie tylko baz danych, ale również systemów operacyjnych — bez niej nie będą potrafiли np. przeanalizować danych zapisywanych w dziennikach zabezpieczeń). **Książka ta ma pomóc użytkownikom, programistom i administratorom w opanowaniu języka SQL** (a dokładniej — standardu ANSI SQL99 i jego późniejszych rozszerzeń), **głównie w zakresie odczytywania i modyfikowania danych**. Osobom, które już znają ten język, pomoże ona pogłębić i usystematyzować wiedzę — z myślą o nich umieściliśmy w książce rozwiązania zaawansowanych problemów (takich jak wyznaczanie trendów, np. odczytywanie informacji o okresach, w których rosła wartość sprzedaży danego towaru), a także wskazówki dotyczące poprawy wydajności zapytań.

Na końcu każdego rozdziału umieściliśmy kilka zadań do samodzielnego wykonania. Stopień ich trudności jest zróżnicowany. Rozwiązywanie wszystkich zadań będzie wymagać trochę czasu, ale jeżeli uda Ci się znaleźć ponad 70% prawidłowych odpowiedzi, to znaczy, że naprawdę dobrze poznajeś język SQL.

Serwery bazodanowe

Na rynku dostępnych jest co najmniej kilkadziesiąt różnych serwerów baz danych. Do najpopularniejszych należą:

1. **SQL Server** firmy Microsoft;
2. **MySQL** — powstał w ramach projektu Open Source, następnie został wykupiony przez szwedzką firmę MySQL AB, a w lutym 2008 roku przejęty przez Sun Microsystems;
3. **Oracle Database** firmy Oracle;
4. **PostgreSQL** — początkowo dzieło naukowców z uniwersytetu w Berkeley, teraz rozwijany w ramach otwartego projektu przez społeczność PostgreSQL Global Development Group;
5. Wchodzący w skład pakietu Microsoft Office **Access**;
6. Bazujący na udostępnionym w 2000 roku kodzie serwera InterBase **Firebird**;
7. **DB2** firmy IBM;
8. **InterBase** firmy Borland.

Te serwery różnią się prawie wszystkim: niektóre są niesamowicie drogie, inne — dostępne za darmo; serwery firmy Microsoft działają tylko w środowisku systemu Windows, pozostałe — w różnych systemach operacyjnych; niektóre wydajnie zarządzają bazami danych o wielkości tysiący gigabajtów i realizują tysiące transakcji na sekundę, inne są przeznaczone do domowego użytku. **Jedynym wspólnym punktem wszystkich tych serwerów jest język SQL.**

O książce

Pierwsze wydanie tej książki trafiło do księgarni w 2008 roku. W tym czasie SQL Server nie obsługiwał niektórych standardowych rozszerzeń języka SQL (takich jak funkcje okienkowe i analityczne klauzuli OVER czy sekwencje), a więc nie zostały one opisane w tym

wydaniu. Trzy lata później na rynek trafiło drugie, rozszerzone wydanie książki. Jako przykładowego serwera użyliśmy wtedy przedpremierowej wersji SQL Server 2012, niestety — jak to bywa z wersjami przedpremierowymi — w wersji finalnej niektóre funkcjonalności (np. składnia instrukcji `MERGE`) zostały zmienione.

Pracując nad bieżącym wydaniem, postanowiliśmy wprowadzić kilka istotnych zmian i raz jeszcze rozszerzyć książkę o nowe tematy:

1. Żeby ułatwić czytelnikowi samodzielne wykonanie opisanych zadań, zdecydowaliśmy się opracować wszystkie przykłady jeszcze raz w taki sposób, aby korzystały one z jednej, niewielkiej bazy danych *AdventureWorks 2012 LT*.
2. Jako że partycjonowanie dość istotnie różni się od grupowania wierszy, wydzieliśmy osobny rozdział o partycjonowaniu, w którym szczegółowo opisaliśmy klauzulę `OVER`, funkcje analityczne oraz mechanizmy okienkowego przetwarzania danych.
3. Dodaliśmy rozdział poświęcony analizie planów wykonania zapytań. W ten sposób mogliśmy przedstawić najważniejsze zalecenia dotyczące pisania wydajnych (szybko wykonywanych) zapytań.

 Wskazówka	Chociaż pierwsze wydanie książki miało miejsce w 2008 roku, jej historia zaczyna się sześć lat wcześniej, w roku 2002. Wtedy to nakładem wydawnictwa Helion ukazała się książka autorstwa Marcina Szeligi zatytułowana <i>ABC języka SQL</i> . Jako że zyskała ona sporą popularność, na podstawie opisanych w niej tematów powstała pierwsza wersja kursu tego języka.
---	---

SQL Server firmy Microsoft

Niestety, **zaimplementowane w różnych serwerach bazodanowych (a nawet w kolejnych wersjach tego samego serwera) wersje języka SQL dość znacznie się od siebie różnią**. Najmniejszym problemem są różnice leksykalne (słownikowe) — np. w większości serwerów procedury składowane wywołuje się instrukcją `CALL`, ale w niektórych do tego samego celu służy instrukcja `EXEC`. Poważniejszym problemem są różnice wynikające z funkcjonalności danego serwera — np. Access nie obsługuje procedur składowanych, a więc w jego wersji języka SQL

nie występuje ani instrukcja CALL, ani EXEC.

 Wskazówka	Producenci serwerów bazodanowych powoli wdrażają w swoich produktach standard ANSI SQL. Do dziś żaden serwer nie implementuje wszystkich instrukcji standardu z roku 1999, za to każdy zawiera niestandardowe rozszerzenia języka SQL.
---	--

Mimo że żaden serwer nie jest w 100% zgodny ze standardem języka SQL, na potrzeby książki musieliszy wybrać jeden. Zdecydowaliśmy się na SQL Server firmy Microsoft w wersji 2012 lub nowszej, ponieważ:

1. W pełni funkcjonalna, wyposażona w graficzne narzędzia administracyjne edycja Express tego serwera jest dostępna za darmo.
2. Można go łatwo zainstalować w systemie Windows 7 i Windows 8.
3. Jest w dużym stopniu zgodny ze standardem ANSI SQL3 (jest to związane ze składnią używanego w tym serwerze języka T-SQL^[3] oraz faktem, że SQL Serwer firmy Microsoft implementuje większość zdefiniowanych w tym standardzie funkcjonalności).
4. W internecie ogólnie dostępne są różne materiały dotyczące tego serwera, w tym:
 - a. kompletna dokumentacja języka T-SQL nazwana BOL (ang. *Book On-Line*);
 - b. liczne portale i fora dyskusyjne (np. <http://wss.pl/SQLServer/Default.aspx>), na których można znaleźć odpowiedzi na związane z tym serwerem pytania.
3. **Serwer SQL firmy Microsoft ciągle umacnia swoją pozycję na rynku serwerów bazodanowych i nic nie wskazuje na to, żeby w ciągu kilku lat miało się to zmienić.** Coraz więcej popularnych programów (np. e-Audytör, Płatnik, GenRap WF-MAG, a także pakiety CDN OPT!MA i Optivum) oraz narzędzia firmy Microsoft (np. WSUS, MOSS, System Center Essentials czy Microsoft Dynamics NAV) używa różnych wersji tego serwera.

Instalacja

Przed przystąpieniem do wykonywania opisanych w książce ćwiczeń

należy zainstalować na komputerze darmową wersję Microsoft SQL Server 2012 lub 2014 Express with Tools. Aby było to możliwe, muszą być spełnione następujące wymagania sprzętowe:

1. procesor Pentium III lub kompatybilny o prędkości co najmniej 1 GHz^[4];
2. minimum 512 MB pamięci RAM;
3. co najmniej 2 GB wolnego miejsca na dysku systemowym.

Dodatkowo na komputerze muszą być wcześniej zainstalowane:

1. W przypadku serwera SQL Express 2012:
 - a. system Windows Vista z dodatkiem SP2, Windows Server 2008 SP2, Windows 7 lub Windows Server 2008 R2^[5].
2. W przypadku serwera SQL Express 2014:
 - a. Windows 7 z dodatkiem SP1, Windows 8 lub nowszy, a w przypadku systemów serwerowych Windows Server 2008 SP2 lub nowszy.
Ponadto, niezależnie od wersji, serwer ten wymaga:
 - b. Programu Microsoft Windows Installer 4.5 lub nowszego — jest on dostępny w witrynie Microsoft Download (<http://www.microsoft.com/download>). Jeżeli nie wyłączyles aktualizacji automatycznych, powinien już być zainstalowany na Twoim komputerze.
 - c. Środowiska Microsoft .NET Framework 3.5 oraz 4.0 — plik instalatora najnowszej wersji platformy .NET można pobrać z witryny Microsoft Download.

 Wskazówka	Przed instalacją serwera SQL należy uaktualnić system operacyjny — można to zrobić, wybierając dostępny w menu <i>Start</i> skrót do usługi <i>Windows Update</i> .
---	---

Po przygotowaniu komputera możemy przystąpić do instalacji serwera SQL. **W pierwszej kolejności należy sprawdzić, czy na komputerze nie jest już zainstalowana jakaś wersja serwera**

SQL (wspominaliśmy, że wchodzi on w skład wielu różnych programów). W tym celu:

1. Uruchom konsolę MMC *Zarządzanie komputerem* (np. klikając prawym przyciskiem myszki ikonę *Komputer*).
2. Rozwiń sekcję *Usługi i aplikacje*, następnie kliknij *Usługi*.
3. Sprawdź, czy na liście nie znajdują się usługi o nazwie *SQL Server (nazwa instancji)*^[6].
4. Jeżeli tak, sprawdź, czy serwer jest uruchomiony i czy usługa jest automatycznie uruchamiana. Jeśli nazwa instancji jest inna niż *MSSQLServer*, zapamiętaj ją — będzie potrzebna do połączenia się z serwerem.
5. Ponieważ wszystkie ćwiczenia będą wykonywane na przykładowej bazie *AdventureWorks 2012 LT* albo na stworzonej w ramach rozdziału 10. testowej bazie danych, **możesz bez obaw o dane innych programów używać do ich wykonywania istniejącego serwera SQL**.

Jeżeli na Twoim komputerze nie ma zainstalowanego serwera SQL 2012/2014 lub jeżeli chcesz zainstalować dodatkową instancję Express tego serwera:

6. Z witryny Microsoft Download pobierz wersję instalacyjną serwera SQL 2012 bądź 2014 Express with Tools.

Uruchom program instalacyjny. Krok po kroku kreator **przeprowadzi Cię przez proces instalacji wybranego serwera — wystarczy, że zastosujesz się do jego wskazówek**.

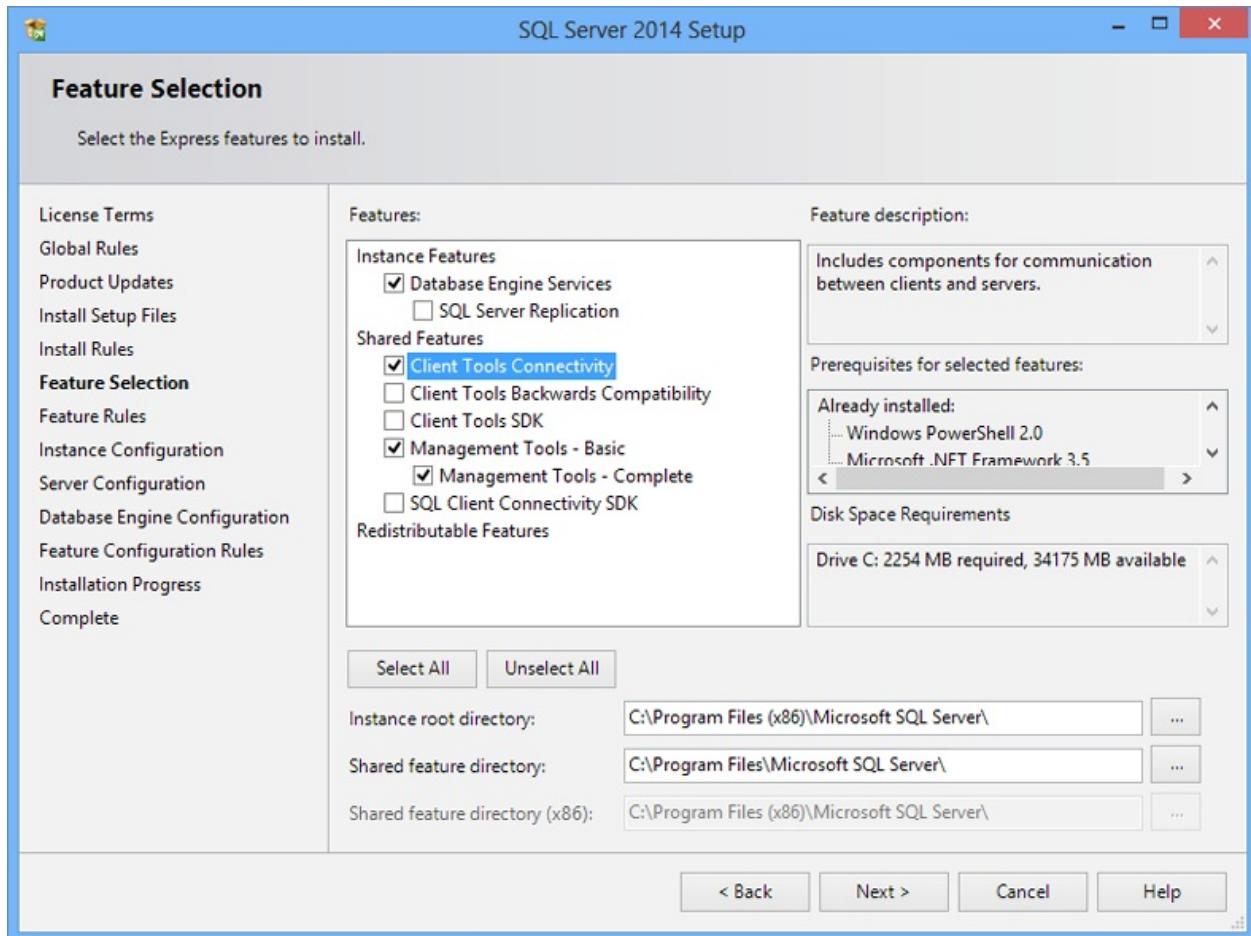
Poniżej przedstawiony został proces instalacji wersji 2014 serwera SQL Server Express w systemie Windows 8. Zaczniemy od pobrania właściwej wersji pliku instalacyjnego:

1. Wpisz w wyszukiwarce frazę Microsoft SQL Server 2014 Express. Jedną ze znalezionych stron będzie strona opisana jako *Download Microsoft SQL Server 2014 Express and install it*. Otwórz ją, a trafisz na portal msdn.microsoft.com.
2. Kliknij przycisk *GET STARTED NOW*.
3. Pobranie serwera w tej wersji wymaga zalogowania się na konto Microsoft. Jeżeli jeszcze nie masz takiego konta (użytkownicy systemów Windows 8 i Windows 8.1 domyślnie korzystają z takich kont), będziesz mógł je założyć za darmo.

4. Po zalogowaniu się na swoje konto Microsoft wypełnij krótki formularz rejestracyjny. Pamiętaj, żeby odpowiadając na pytanie *Please select which version of SQL Server Express you would like to download*, zaznaczyć opcję *Express with Tools*.
5. Kliknij przycisk *Continue*.
6. Odpowiadając na kolejne pytania, wybierz architekturę (32- lub 64-bitową) serwera. W systemach 64-bitowych możliwa jest instalacja obu (32- i 64-bitowych) wersji serwera SQL, w naszym przypadku wybór architektury nie ma znaczenia — wszystkie opisane elementy języka oraz ilustrujące je zadania działają identycznie w obu środowiskach.
7. Kliknij przycisk *Continue*.
8. Wybierz wersję językową serwera SQL (nie ma polskojęzycznej wersji serwera SQL). My wybraliśmy wersję angielską.
9. Kliknij przycisk *Continue*.
10. Zapisz plik instalatora (np. plik *SQLExprWT_x86_ENU*) na dysku.

Instalacja serwera SQL Server Express przebiega automatycznie, a więc nasza rola sprowadzi się do uruchomienia pobranego pliku instalacyjnego. Następnie:

1. Wskaż folder, do którego zostaną wypakowane pliki. Po ich wypakowaniu instalator zostanie automatycznie uruchomiony.
2. Z listy zadań (ang. *Task*) wybrana zostanie nowa instalacja (ang. *Installation*). Kliknij odnośnik *New SQL Server stand-alone installation or add features to an existing installation*. Instalacja przebiega w kilku etapach.
3. Zaakceptuj licencję końcowego użytkownika^[7] i kliknij *Next*.
4. Instalator sprawdzi zgodność komputera z wymaganiami instalacyjnymi (ang. *Setup Support Rules*). Jeżeli Twój komputer spełnia minimalne wymagania, wyświetcone zostanie okienko pozwalające wybrać instalowane składniki serwera SQL (rysunek W.1).



Rysunek W.1. W książce zostały wykorzystane tylko pokazane na rysunku składniki serwera SQL

5. Kliknij *Next*.
6. Wybierz nazwę instalowanej instancji (kopii) serwera SQL 2014. Odpowiadając na to pytanie, pozostaw domyślną opcję *Named instance* i nazwę instancji *SQLExpress*.
7. Domyślnie użytkownik, który przeprowadza instalację, zostanie administratorem serwera SQL. Jeżeli nie potrzebujesz dodawać kolejnych administratorów tego serwera, kliknij *Next*.
8. Rozpocznie się proces instalacji. Po jego zakończeniu wyłącz program instalacyjny.

Edycja SQL Server Express with Tools instalowana jest razem z graficzną konsolą SSMSE (ang. *SQL Server Management Studio Express*). Konsola ta służy do administracji serwerem SQL, a także do jego programowania — my natomiast będziemy jej używać do wykonywania instrukcji języka SQL.

Po zakończeniu instalacji:

1. Uruchom konsolę SSMSE.
2. Połącz się z serwerem SQL Express.
3. W nowym okienku kodu (można je wyświetlić, klikając przycisk *New Query*) wpisz instrukcję `SELECT @@version`, następnie uruchom ją klawiszem *F5* lub przyciskiem *EXECUTE*. W dolnej części centralnego okna programu zostanie wyświetlona informacja o zainstalowanej przez Ciebie wersji serwera SQL.

Przykładowa baza danych

Ostatnią wymaganą do wykonania znajdującej się w książce ćwiczeń operacją jest zainstalowanie przykładowej bazy danych *AdventureWorks 2012 LT*. Plik skryptu tworzącego tę bazę i wypełniającego ją danymi jest dostępny na serwerze FTP Wydawnictwa Helion (razem z przykładami do książki). Po pobraniu pliku:

1. Wypakuj plik z przykładami do książki do folderu *c:\SQL*.
2. Dwukrotnie kliknij lewym przyciskiem myszki plik *c:\SQL\setup.sql*. Uruchomioną w ten sposób konsolę SSMSE połącz z zainstalowanym serwerem SQL (w większości przypadków wystarczy w tym celu kliknąć przycisk *Connect*).
3. Wykonaj wszystkie znajdujące się w tym skrypcie instrukcje. W tym celu kliknij przycisk *Execute* lub naciśnij klawisz *F5* (rysunek W.2).

The screenshot shows the Microsoft SQL Server Management Studio interface. The title bar reads "Setup.sql - MS\SQLExpress master (MS\szeloz (51)) - Microsoft SQL Server Management Studio". The main window displays a Transact-SQL script named "Setup.sql" which creates the AdventureworksLT2012 database and its schema. The "Messages" pane at the bottom shows the execution progress with messages like "(1 row(s) affected)" repeated several times, followed by "Query executed successfully.". The "Properties" pane on the right shows connection details for the current session.

```

USE [master]
GO

CREATE DATABASE [AdventureworksLT2012]
ON PRIMARY
( NAME = 'AdventureworksLT2012_Data', FILENAME = 'N'C:\SQL\AdventureworksLT2012_Data.ndf' , SIZE = 6464KB , MAXSIZE = UNLIMITED, FILEGROWTH
LOG ON
( NAME = 'AdventureworksLT2012_Log', FILENAME = 'N'C:\SQL\AdventureworksLT2012_Log.ldf' , SIZE = 2048KB , MAXSIZE = 2048GB , FILEGROWTH = 16
GO

ALTER DATABASE [AdventureworksLT2012] SET RECOVERY SIMPLE
GO

USE [AdventureworksLT2012]
GO
/*===== Object: Schema [SalesLT] Script Date: 2014-07-27 07:35:28 =====*/
CREATE SCHEMA [SalesLT]
GO
/*===== Object: XslSchemaCollection [SalesLT].[ProductDescriptionSchemaCollection] Script Date: 2014-07-27 07:35:28 =====*/
CREATE XML SCHEMA COLLECTION [SalesLT].[ProductDescriptionSchemaCollection] AS N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >'
GO
/*===== Object: UserDefinedDataType [dbo].[AccountNumber] Script Date: 2014-07-27 07:35:28 =====*/
CREATE TYPE [dbo].[AccountNumber] FROM [nvarchar](15) NULL
GO
/*===== Object: UserDefinedDataType [dbo].[Flag] Script Date: 2014-07-27 07:35:28 =====*/
CREATE TYPE [dbo].[Flag] FROM [bit] NOT NULL
GO
/*===== Object: UserDefinedDataType [dbo].[Name] Script Date: 2014-07-27 07:35:28 =====*/
CREATE TYPE [dbo].[Name] FROM [nvarchar](50) NULL
GO
/*===== Object: UserDefinedDataType [dbo].[NameStyle] Script Date: 2014-07-27 07:35:28 =====*/
CREATE TYPE [dbo].[NameStyle] FROM [bit] NOT NULL
GO
/*===== Object: UserDefinedDataType [dbo].[OrderNumber] Script Date: 2014-07-27 07:35:28 =====*/
CREATE TYPE [dbo].[OrderNumber] FROM [nvarchar](25) NULL
GO

```

Rysunek W.2. Udane wykonanie skryptu tworzącego przykładową bazę danych powinno zająć mniej niż minutę

Konwencje i oznaczenia

Poniżej zostały przedstawione zastosowane w książce konwencje i symbole.

Czcionka o stałej szerokości

Czcionka ta używana jest do oznaczania wszystkich poleceń wprowadzanych za pomocą klawiatury, słów kluczowych języka SQL, nazw obiektów bazodanowych (np. tabel czy procedur), stałych, a także wyników przykładowych instrukcji. Wszystkie słowa kluczowe języka SQL zapisywane są dużymi literami, a opcjonalne słowa kluczowe umieszczane są ponadto w nawiasach kwadratowych.

Kursywa

Kursywa używana jest do oznaczania nazw plików i katalogów oraz adresów internetowych, jak również do wyróżniania

angielskich nazw i zwrotów oraz wybieranych za pomocą myszki nazw opcji.

Pogrubienie

Najważniejsze fragmenty tekstu zostały wyróżnione za pomocą pogrubienia.

 Wskazówka	W ten sposób są oznaczone wskazówki i uwagi do tekstu.
--	--

[1] Skrót SQL najczęściej wymawia się *sequel* — tak nazywała się pierwsza wersja języka SQL opracowana w latach siedemdziesiątych XX wieku przez firmę IBM.

[2] W trakcie rekrutacji programistów prawie zawsze sprawdzana jest ich znajomość języka SQL — wynika to z założenia, że przedżej czy później każdy programista będzie pracował nad programem, który korzysta z baz danych.

[3] Transact-SQL, w skrócie T-SQL, jest nazwą zaimplementowanego w serwerze firmy Microsoft języka SQL.

[4] Wymagania dla 64-bitowej wersji serwera SQL 2011 są nieco wyższe.

[5] Jeżeli nadal używasz systemu Windows XP, zainstaluj Microsoft SQL Server 2008 Express Edition with Advanced Services — prawie wszystkie opisane w książce ćwiczenia będą działały z tą wersją serwera.

[6] Domyślną nazwą instancji wersji Express jest *SQLEXPRESS*, a instancji płatnych wersji serwera — *MSSQLServer*.

[7] Serwer SQL Express jest darmowy również w komercyjnych zastosowaniach, a więc można go zgodnie z prawem instalować nie tylko na domowych, ale także firmowych komputerach.

Część I

Trochę teorii, czyli modele i standardy

Zanim poznamy i zaczniemy stosować język SQL, powinniśmy znać odpowiedzi na poniższe pytania:

1. Czym są relacyjne bazy danych?
2. Co odróżnia język SQL od innych języków programowania?

Wyjaśnienie tych kwestii znajdziesz w dwóch pierwszych rozdziałach książki — opisaliśmy w nich uniwersalny model relacyjnych baz danych i ogólnie przyjęty standard języka SQL. Jeżeli zupełnie nie znasz języka SQL, a materiał z pierwszej części książki wyda Ci się niejasny, pomiń dwa pierwsze rozdziały i zacznij kurs od przeczytania rozdziałów 3 – 6, a następnie wróć do tej części.

Rozdział 1. Relacyjny model baz danych

- Czym jest tabela?
- Czym jest baza danych?
- Dlaczego model relacyjny jest najpopularniejszym modelem baz danych?
- Jakie warunki musi spełniać serwer bazodanowy, żeby mógł być nazywany serwerem relacyjnych baz danych?
- Na czym polega normalizacja baz danych?

Tabele jako zbiory danych

Podstawowym obiektem bazy danych jest tabela – dwuwymiarowa konstrukcja zbudowana z pionowych kolumn i poziomych wierszy, na przecięciu których znajdują się komórki. Tabelę można sobie wyobrazić jako arkusz Excela lub książkę adresową (tabela 1.1).

Tabela 1.1. Tabele przechowują interesujące nas informacje, w tym przypadku dane naszych znajomych

Nazwisko	Imię	Nr telefonu	Adres	Miasto	E-mail
Mendrala	Danuta	0 999 234567	ul. Bajkowa 12	Katowice	dm@foo.pl
Szeliga	Marcin	0 888 765432	ul. Krótka 991	Katowice	ms@foo.com

Zauważ, że **liczba kolumn tabeli jest stała, natomiast liczba wierszy zmienia się** odpowiednio do liczby przechowywanych w niej obiektów.

 Wskazówka	Obiektem jest każda rzecz, która ma swoją tożsamość (możemy ją odróżnić od innych rzeczy tego samego typu) i dowolną liczbę opisujących ją cech (atrributów). Tak więc obiektem jest konkretna osoba (ale nie ludzie), książka, którą kupiłeś, czy wystawiona za nią faktura.
--	---

Zwrót również uwagę, że **w poszczególnych kolumnach znajdują się zawsze dane tego samego typu** (np. w pierwszej kolumnie są tylko nazwiska, a w ostatniej jedynie adresy e-mail). Z kolei **w poszczególnych wierszach znajduje się komplet informacji o**

konkretnych obiektach — w pierwszym wierszu znajdują się dane dotyczące wyłącznie Danuty, w drugim — tylko Marcina.

 Wskazówka	Wiersz jest nazywany również rekordem, a pojedyncza komórka tabeli — polem.
--	---

Kolejną ważną cechą tabeli jest to, że **każdy jej rekord składa się z takiej samej liczby pól (kolumn)**. Żaden wiersz tabeli nie może być krótszy lub dłuższy od innych, niedopuszczalne są również przerwy pomiędzy komórkami.

Niezbędna dla opanowania języka SQL jest umiejętność wyobrażenia sobie tabel jako zbiorów rekordów — tak jak zbiór matematyczny składa się z dowolnej liczby elementów, tak tabela może zawierać dowolną liczbę wierszy. I tak samo jak w zbiorze kolejność elementów jest nieistotna, tak kolejność rekordów tabeli (wierszy), a nawet kolejność jej atrybutów (kolumn) nie ma żadnego znaczenia dla serwerów bazodanowych (rysunek 1.1).



Rysunek 1.1. Zbiór danych o naszych znajomych

Kolumny mają niepowtarzalne nazwy i zawierają określone typy danych

Przyjęło się, że nazwy kolumn odpowiadają nazwom atrybutów,

których wartości są w nich przechowywane. W naszym przykładzie w kolumnie **Imię** przechowywane są imiona naszych znajomych, a w kolumnie **Nr telefonu** — numery ich telefonów. **Nazwy kolumn tej samej tabeli muszą być niepowtarzalne**, więc gdybyśmy chcieli w przykładowej tabeli zapisać numery telefonów domowych, musielibyśmy dodać do niej jedną kolumnę i nazwać ją np. **Nr telefonu domowego**.

Ponieważ w kolumnach są przechowywane wartości tego samego atrybutu, to **wszystkie pola kolumny zawierają dane tego samego typu**. Mogą to być same liczby (np. w kolumnie **Cena**), ciągi znaków (w kolumnie **Imię**) lub daty (w kolumnie **Wiek**).

Wiersze powinny być unikatowe

Tabele reprezentują zbiory, ale mają też charakterystyczne dla siebie cechy (jak choćby niepowtarzalność nazw kolumn). **Główna różnica między tabelą a zbiorem polega na tym, że zbiory nie zawierają kilku kopii tego samego elementu.** Przykładowo zbiór liczb całkowitych zawiera tylko jedną liczbę 1 i jedną liczbę 2, a zbiór kolorów — tylko jeden element o nazwie „niebieski”^[1]. Natomiast w tabeli może się wielokrotnie powtórzyć ten sam wiersz, np. w wyniku przypadkowego ponownego wpisania danych tej samej osoby.

 Wskazówka	Ponieważ powtórzenie tego samego rekordu prowadzi do trudno wykrywalnych błędów logicznych i niespójności danych, przyjęto się dodawać do tabeli specjalną kolumnę, w której zapisuje się identyfikatory poszczególnych wierszy. Taka kolumna nazywa się kluczem podstawowym tabeli .
---	--

Tabela 1.2 pokazuje przykładową tabelę po dodaniu kolumny klucza podstawowego.

Tabela 1.2. Dzięki dodaniu identyfikatorów osób, nawet jeżeli pomyłkowo dwukrotnie wstawimy do tabeli dane tej samej osoby, to będą to dwa różne, rozpoznawalne przez identyfikatory rekordy

Osoba ID	Nazwisko	Imię	Nr telefonu	Adres	Miasto	E-mail
1	Mendrala	Danuta	0 999 234567	ul. Bajkowa 12	Katowice	dm@foo.pl
2	Szeliga	Marcin	0 888 765432	ul. Krótka 991	Katowice	ms@foo.com

Kolejność kolumn jest bez znaczenia

Kolejność elementów w zbiorach jest bez znaczenia. Na przykład oba poniższe zbiory są identyczne:

Znajomi1 {Danuta, Marcin}

Znajomi2 {Marcin, Danuta}

Skoro tabele są reprezentacjami zbiorów, kolejność ich kolumn też nie powinna mieć żadnego znaczenia^[2] — obie poniższe tabele są takie same:

Znajomi1 {OsobaID, Nazwisko, Imię, NrTelefonu, Adres, Miasto, E-mail}

Znajomi2 {NrTelefonu, OsobaID, Imię, Nazwisko, Adres, E-mail, Miasto}

Kolejność wierszy jest bez znaczenia

O ile kolejność kolumn może mieć wpływ na pracę z danymi (taka sytuacja ma miejsce, gdy odczytując dane z tabeli, używamy niezalecanej składni `SELECT * FROM Tabela`), o tyle kolejność wierszy jest całkowicie nieistotna. Serwery bazodanowe i tak odczytują i modyfikują poszczególne wiersze w taki sposób, żeby wykonanie danej operacji (np. znalezienie nazw towarów, których sprzedaż w tym miesiącu jest o ponad 10% niższa niż w zeszłym) było jak najszysze, a nie w kolejności, w jakiej rekordy zostały zapisane w tabelach.

 Wskazówka	Ponieważ użytkownicy z reguły spodziewają się zobaczyć listę rekordów uporządkowaną w pewien sposób (np. listę towarów ułożoną alfabetycznie), język SQL pozwala sortować odczytywane z tabel dane. Sortowane są jednak wyłącznie wyniki zapytań, a nie przechowywane w tabelach wiersze. Żeby posortować przechowywane na dysku dane, trzeba założyć indeksy.
---	--

Bazy danych

Bazą danych w ścisłym tego słowa znaczeniu jest zbiór przechowywanych w tabelach danych uzupełniony o informacje o samych tabelach (takie jak ich nazwy, typy i nazwy kolumn oraz wiele innych)^[3]. W tym sensie **baza danych jest zbiorem informacji o ścisłe określonej strukturze**.

Potocznie termin „baza danych” funkcjonuje jako określenie

programu (serwera bazodanowego), którego zadaniem jest przechowywanie, przetwarzanie i udostępnianie danych.

 Wskazówka	Standard ANSI SQL99 nie ustala znaczenia terminu „baza danych”, definiuje natomiast następującą hierarchię obiektów: powiązane ze sobą tabele tworzą schemat (ang. <i>Schema</i>), jeden lub więcej schematów składa się na katalog (najbliższy odpowiednik bazy danych), zbiór katalogów wchodzi w skład klastra (ang. <i>Cluster</i>), którego najbliższym odpowiednikiem jest serwer bazodanowy.
--	---

Trzy modele baz danych: relacyjny, obiektowy i jednorodny

Wszystkie wymienione we wstępie serwery bazodanowe są serwerami relacyjnych baz danych. Model relacyjny, choć zdecydowanie najpopularniejszy, nie jest jednak jedyny — oprócz niego są używane dwa inne modele: jednorodny i obiektowy.

Model jednorodny

W tym modelu wszystkie dane są przechowywane w jednym arkuszu, tabeli, kostce analitycznej lub pliku (stąd nazwa modelu). Przykład tego typu danych został pokazany na tabeli 1.3.

Tabela 1.3. Arkusze kalkulacyjne to najpopularniejsze programy przechowujące i przetwarzające dane jednorodne

Nazwa	Cena	Ilość	Data zakupu	Dostawca	Adres dostawcy	Tel. kontaktowy
Sandały	99 zł	2	12-02-2011	Dobry But SA	ul. Handlowa 12a	99 01234565
Sandały	75 zł	2	01-03-2011	Dobry But SA	ul. Handlowa 12a	99 01234565
Jacht VI-1	9 999 999 zł	1	15-01-2011	Jachting SA	ul. Morska 1	88 233454546
Sznurowadła	1 zł	10	22-12-2010	Dobry But SA	ul. Handlowa 12a	99 01234565

Zaletami tego modelu są łatwość i szybkość odczytywania interesujących nas danych — w tym przypadku wystarczy tylko znaleźć rekord opisujący szukany zakup, żeby poznać wszystkie szczegóły operacji.

Wadą modelu jednorodnego jest duża liczba duplikatów (powtarzających się danych) — w naszym przykładzie nazwa dostawcy, jego adres i telefon kontaktowy wpisane są tyle razy, ile razy kupiliśmy u niego towar. Nazwy produktów również powtarzają się kilka razy.

Wielokrotne zapisywanie tych samych danych nie tylko zajmuje więcej miejsca na dysku i w pamięci, lecz także:

1. Utrudnia modyfikowanie danych. Gdyby firma Dobry But SA zmieniła adres lub numer telefonu, musielibyśmy je zmienić w wielu różnych rekordach. Gdybyśmy przeoczyli jeden z nich, dane byłyby niespójne, tj. odczytując dwa razy tę samą informację (adres lub telefon tej samej firmy), moglibyśmy otrzymać różne wyniki.
2. Zwiększa ryzyko wpisania błędnych danych. Podając po raz kolejny nazwę tej samej firmy, możemy przypadkowo dodać spację, zmienić wielkość liter czy w inny sposób pomylić się przy jej wpisywaniu. W rezultacie w bazie zostaną zapisane różne nazwy firmy i gdybyśmy chcieli w przyszłości np. podsumować zakupy u poszczególnych dostawców, otrzymane wyniki byłyby nieprawdziwe.

Model relacyjny

W modelu relacyjnym dane są przechowywane w wielu odrębnych, ale powiązanych ze sobą tabelach. Prawidłowe rozmieszczenie informacji w osobnych tabelach zostało dokładniej opisane w punkcie „Normalizacja”; na potrzeby porównania modeli baz danych możemy przyjąć, że w **jednej tabeli powinno się zapisywać dane o obiektach jednego typu**, np. wyłącznie informacje o znajomych czy firmach.

Przekształcone do modelu relacyjnego informacje o zakupach zostały pokazane w tabeli 1.4.

Tabela 1.4. W modelu relacyjnym dane o towarach, dostawcach i sprzedaży są zapisywane w osobnych tabelach

Nazwa	Cena	Ilość	Data zakupu	Dostawca	Adres dostawcy	Tel. kontaktowy
Sandały	99 zł	2	12-02-2011	Dobry But SA	ul. Handlowa 12a	99 01234565

Jacht VI-1	75 zł	2	01-03-2011	Jachting SA	ul. Morska 1	88 233454546
Sznurowadła	9 999 999 zł	1	15-01-2011			
	1 zł	10	22-12-2010			

Zaletą modelu relacyjnego jest zapobieganie tworzeniu duplikatów danych — dane o poszczególnych sprzedawcach i nazwy towarów są zapisane tylko raz. Nie tylko zmniejsza to ilość przechowywanych w bazie informacji, lecz także ułatwia ich modyfikowanie i wstawianie^[4].

Wadą modelu relacyjnego jest skomplikowane i wolne odczytywanie danych. Chcąc poznać komplet informacji na temat sprzedaży określonego produktu, musimy odczytać aż trzy tabele. W dodatku dane odczytane z jednej tabeli należy właściwie połączyć z danymi odczytanymi z pozostałych tabel — inaczej nie dowiemy się, u kogo i za ile kupiliśmy dany towar.

W ten sposób doszliśmy do drugiej charakterystycznej cechy modelu relacyjnego: dane są przechowywane w osobnych, ale powiązanych ze sobą tabelach. **Te łączące tabele powiązania nazywa się relacjami** (stąd nazwa modelu).

Żeby relacje były jednoznaczne, połączone nimi tabele muszą mieć klucze podstawowe, a zapisane w nich identyfikatory muszą być powtórzone w każdej z połączonych tabel. W innym przypadku nie bylibyśmy w stanie stwierdzić, u jakiego dostawcy, kiedy i za ile kupiliśmy dany towar. **Kolumna, która zawiera identyfikator rekordu innej tabeli, nazywa się kluczem obcym.** Tabele uzupełnione o kolumny klucza podstawowego i kluczy obcych pokazano w tabeli 1.5.

Tabela 1.5. Liczba duplikatów została ograniczona do powtarzających się (w kolumnach kluczy obcych) identyfikatorów rekordów

ID towaru	Nazwa	ID kupna	Cena	Ilość	Data zakupu	ID towaru	ID firmy
1	Sandały	1	99 zł	2	12-02-2011	1	1
2	Jacht VI-1	2	75 zł	2	01-03-2011	1	1

3	Sznurowadła	3	9 999 999 zł	1	15-01-2011	2	2
		4	1 zł	10	22-12-2010	3	1

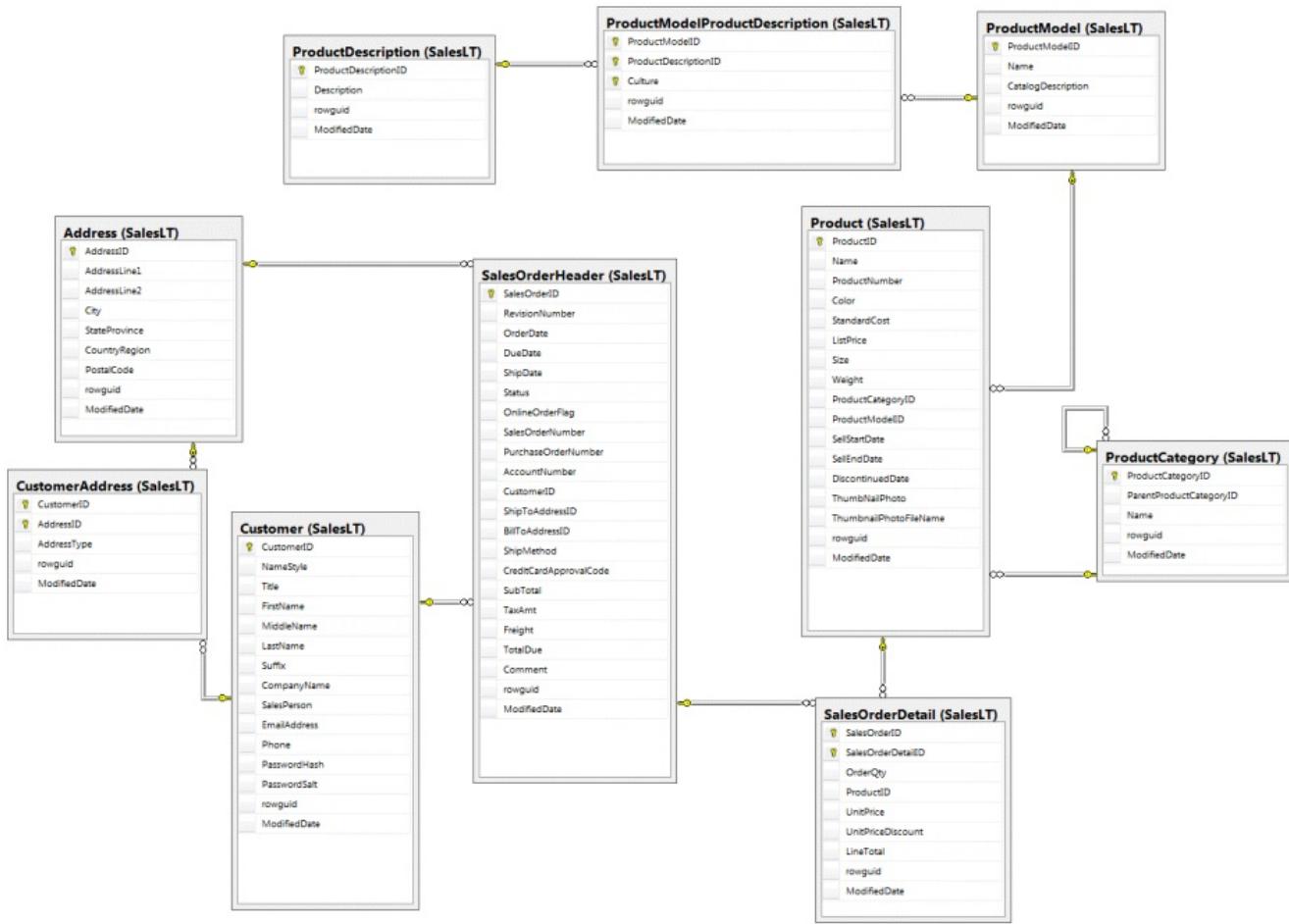
ID firmy	Dostawca	Adres dostawcy	Tel. kontaktowy
1	Dobry But SA	ul. Handlowa 12a	99 01234565
2	Jachting SA	ul. Morska 1	88 233454546

W praktyce relacyjne bazy danych składają się z wielu powiązanych ze sobą relacjami tabel. Informacje na temat tego, która tabela jest powiązana z innymi tabelami, są prezentowane w postaci diagramów E/R (diagramów Encja/Relacja).

Diagram E/R przykładowej bazy danych

Aby utworzyć diagram bazy AdventureWorksLT2012:

1. Uruchom konsolę SQL Server Management Studio.
2. Połącz się z serwerem SQL Server (upewnij się, że w polu *Server name*: widoczna jest nazwa zainstalowanego na potrzeby ćwiczeń serwera, i kliknij przycisk *Connect*).
3. Z lewej strony ekranu pojawi się okienko *Object Explorer*. Zawiera ono hierarchicznie ułożone wszystkie obiekty serwera. Dwukrotnie kliknij lewym przyciskiem myszy folder *Databases*, a następnie nazwę przykładowej bazy danych *AdventureWorksLT2012*.
4. Kliknij prawym przyciskiem myszy wyświetlony w ten sposób folder *Database Diagrams* i wybierz z menu kontekstowego opcję *New Database Diagram*.
5. Odpowiedz Yes na pytanie, czy chcesz dodać do tej bazy potrzebne do utworzenia diagramu obiekty.
6. Wyświetlone zostanie okienko z listą tabel bazy danych. Zaznacz (aby zaznaczyć kilka tabel naraz, należy przytrzymać wciśnięty klawisz *Ctrl*) wszystkie te tabele, których nazwy po prawej stronie zawierają frazę (*SalesLT*)^[5], i kliknij przycisk *Add*.
7. Zamknij okno dodawania tabel przyciskiem *Close*.
8. Kliknij widoczny na pasku menu przycisk *Arrange Tables* — tabele zostaną uporządkowane na diagramie (rysunek 1.2).



Rysunek 1.2. Utworzony za pomocą konsoli SSMSE diagram E/R

Analizując utworzony w ten sposób diagram E/R, można m.in. zauważyc, że:

1. Dane o produktach zapisane są w tabeli **Product** — jeden wiersz tej tabeli zawiera takie dane o konkretnym produkcie, jak:
 - jego identyfikator (**ProductID**),
 - nazwę (**Name**),
 - kolor (**Color**),
 - czy cenę (kolumna **ListPrice**).
2. Produkty są przypisane do konkretnych kategorii (tabela **Product** jest powiązana relacją z tabelą **ProductCategory**) w taki sposób, że klucz podstawowy tabeli nadzędnej (kolumna **ProductCategoryID** tabeli **ProductCategory**) jest skopiowany do tabeli podrzędnej (tabeli **Product**). Innymi słowy — kolumna

- ProductCategoryID w tabeli Product jest kolumną klucza obcego.
3. Każdy produkt może być sprzedany dowolną liczbę razy. Dane o sprzedanych w ramach poszczególnych zamówień produktach znajdują się w tabeli SalesOrderDetail, a jedną z kolumn tej tabeli jest kolumna ProductID. Czyli w tym przypadku to tabela Product jest tabelą nadzędną, a podzędna tabela SalesOrderDetail zawiera (w kolumnie klucza obcego) kopie identyfikatorów produktów (dane z kolumny klucza podstawowego tabeli nadzędnej).

Poświęć trochę czasu na poznanie wszystkich tabel przykładowej bazy danych i łączących je relacji — wiedza ta znacznie ułatwi Ci rozwiązywanie zadań z kolejnych rozdziałów.

Model obiektowy

Relacyjną bazę danych można porównać do magazynu z meblami, w którym każda część mebla jest umieszczana w osobnym, przeznaczonym wyłącznie dla niej kontenerze (odpowiednikami kontenerów są tabele). Umieszczając meble w takim magazynie, rozbieramy je na części i osobno układamy drzwi, półki i tak dalej. Taki sposób przechowywania jest bardzo wydajny, ale wyjmując mebel z magazynu, każdorazowo musimy złożyć go w jedną całość.

 Wskazówka	Charakterystyczną cechą modelu relacyjnego jest prostota struktur danych (poszczególne tabele przechowują informacje o bardzo uproszczonych obiektach). Ta prostota początkowo była uważana za główną zaletę modelu relacyjnego, teraz natomiast takie niezgodne z intuicyjnym obrazem świata rozproszenie informacji pomiędzy różnymi tabelami jest uznawane za jego główną wadę.
---	--

Tymczasem model obiektowy przypomina magazyn, w którym meble są ustawiane bez rozbierania na części. W rezultacie korzystanie z magazynu jest prostsze (ale niekoniecznie szybsze), za to liczba mebli, jaką można zmieścić na tej samej przestrzeni, jest znacznie mniejsza.

 Wskazówka	Model obiektowy, bardzo popularny wśród programistów takich języków programowania, jak C#, Java czy VB.NET, zakłada, że wszystko jest obiektem o nieznanej nam wewnętrznej budowie. Owa wewnętrzna budowa z reguły nas jednak nie interesuje, za to musimy wiedzieć, jak używać tych obiektów do własnych celów [6] .
---	---

W opracowanym w latach 90. XX wieku modelu obiektowym informacje są przechowywane w bazie nie w postaci rekordów, ale całych obiektów. Tak zapisane dane są dostępne za pośrednictwem metod tych obiektów, np. obiekt `Towar` może mieć metodę pozwalającą odczytać nazwę towaru i inną metodę zwracającą dane producenta tego towaru.

Zaletą modelu obiektowego jest zgodność z obowiązującym paradygmatem programowania. W tym przypadku programiści nie natrafiają na problemy związane z „niezgodnością impedancji”, takie jak:

1. Niezgodność składni — składnia języka SQL jest zupełnie inna niż składnia takich języków, jak C, Java czy Visual Basic.
2. Niezgodność typów — większość języków programowania, w przeciwieństwie do języka SQL, ma wbudowaną statyczną kontrolę typów, a w prawie żadnym z nich nie występuje podstawowy dla języka SQL typ „relacja”, wiele języków programowania nie obsługuje też wartości NULL.
3. Niezgodność użycia — w języku SQL programista określa wynik, jaki chcetrzymać, a nie sposób, w jaki ma on być osiągnięty. Ponadto SQL jest językiem interpretowanym, a nie komplelowanym.

Główną wadą modelu obiektowego jest to, że nie został jeszcze sprawdzony. Do dziś nie opracowano standardu tego modelu, a nieliczne obiektowe bazy danych są używane głównie w celach badawczych, ponadto nie umożliwiają one na razie przechowywania dużych (liczonych w setkach gigabajtów czy terabajtów) ilości danych, nie mówiąc już o wydajnym zarządzaniu nimi.

 Wskazówka	Relacyjne bazy danych zdobyły tak silną pozycję, że ich zastąpienie w najbliższej przyszłości bazami obiektowymi wydaje się mało prawdopodobne. Za to ogólną popularność zdobywają technologie ORM (ang. <i>Object-Relational Mapping</i>), które pozwalają programistom traktować relacyjne bazy danych tak, jakby były bazami obiektowymi.
---	---

Założenia relacyjnego modelu baz danych



Wskazówka

Relacyjny model baz danych został opracowany w latach 70. XX wieku przez pracującego dla firmy IBM naukowca Edgara Franka Coda. Chociaż matematyczne podstawy tego modelu (teoria zbiorów matematyka Georga Cantora) były znane już od ponad 100 lat, to zbudowanie serwera relacyjnych baz danych okazało się dość trudne. Doktor Codd został uhonorowany prestiżową nagrodą Turinga wcześniej (otrzymał ją w 1981 roku), niż na rynku pojawił się pierwszy, oparty na stworzonym przez niego modelu, serwer bazodanowy.

W modelu relacyjnym dane są reprezentowane przez zbiory krotek (czyli pół tabel), do których dostęp zapewniają operatory algebry relacji, takie jak selekcja, projekcja czy suma. Prawie cała praca dra Coda (model relacyjny został opisany w wydanej w 1970 roku publikacji *A Relational Model of Data for Large Shared Data Banks*) składa się z definicji i lematów, my jednak skoncentrujemy się na zamieszczonych w niej dwunastu postulatach, które stały się de facto kryteriami oceny serwerów bazodanowych. **Tylko serwery zgodne z wszystkimi postulatami E.F. Coda uznawane są za serwery relacyjnych baz danych.**

Postulaty Coda dotyczące struktury danych

Spełnienie postulatów dotyczących struktur danych pozwala w ten sam sposób, niezależnie od wykorzystywanego serwera bazodanowego, zarządzać przechowywanymi w bazach informacjami. Do postulatów należą:

1. **Postulat informacyjny.** Informacje są reprezentowane w postaci logicznych tabel. Oznacza to, że fizyczny sposób organizacji i przechowywania danych przez serwer bazodanowy nie może mieć wpływu na działanie programów klienckich.
2. **Postulat dostępu.** Każda informacja musi być dostępna za pomocą nazwy tabeli, kolumny i wartości klucza podstawowego. Innymi słowy, znajomość struktury tabeli i wartości identyfikatorów rekordów musi wystarczyć do odczytania dowolnej informacji z bazy.
3. **Postulat fizycznej niezależności danych.** Sposób przechowywania danych i wewnętrzne mechanizmy dostępu do nich przez serwer nie mogą mieć wpływu na aplikacje klienckie. Na przykład to, czy dane są przechowywane w jednym pliku, czy w wielu plikach, dla programów klienckich musi być całkowicie niewidoczne.

4. **Postulat logicznej niezależności danych.** Zmiany w strukturze bazy danych, np. zmiana definicji tabeli, o ile tylko nie powodują utraty informacji i są poprawne semantycznie, nie mogą mieć wpływu na aplikację kliencką. Przykładowo, zgodnie z tym postulatem, dodanie kolumny do tabeli nie może zakłócać działania programów klienckich.
5. **Postulat niezależności dystrybucyjnej.** Odwołania do danych za pomocą języka SQL muszą być niezależne od fizycznej lokalizacji danych. Innymi słowy, aplikacje klienckie powinny mieć taki sam dostęp do danych znajdujących się na lokalnym dysku twardym jak do danych rozproszonych pomiędzy różne lokalizacje.
6. **Postulat zabezpieczenia przed modyfikacjami przeprowadzanymi za pomocą języków proceduralnych.** Jeśli serwer bazodanowy umożliwia bezpośrednie modyfikowanie poszczególnych rekordów za pomocą języków niższego poziomu, zmiany te nie mogą naruszać spójności danych, w szczególności nie mogą być sprzeczne z nałożonymi na tabele ograniczeniami.

Postulaty Codda dotyczące przetwarzania danych

Relacyjny model baz danych został stworzony z myślą o wydajnym i łatwym modyfikowaniu zapisanych w bazach informacji. Te zmiany muszą być przeprowadzane z uwzględnieniem poniższych postulatów:

1. **Postulat pełnego języka danych.** Serwer baz danych musi implementować jeden język pozwalający definiować tabele, widoki i ograniczenia (więzy spójności), zarządzać dostępem użytkowników i transakcjami oraz odczytywać i modyfikować dane^[7].
2. **Postulat modyfikowania bazy danych przez widoki.** Zmiany danych przeprowadzane poprzez widoki muszą być odzwierciedlane w odpowiednich tabelach, a bezpośrednie zmiany danych w tabelach muszą być automatycznie widoczne poprzez widoki.
3. **Postulat modyfikowania danych na wysokim poziomie abstrakcji.** Odczytanie, zmodyfikowanie, wstawienie lub usunięcie danych musi być możliwe za pomocą pojedynczej operacji.

Postulaty Coda dotyczące integralności danych

Przestrzeganie postulatów dotyczących integralności danych gwarantuje zachowanie logicznej spójności przechowywanych w bazie informacji:

1. **Postulat wartości NULL.** Serwer bazodanowy w spójny sposób przetwarza specjalną wartość NULL jak brakującą informację, a nie jak zero (0) czy pusty串 znaków („ ”).
2. **Postulat słownika danych.** Metadane (informacje o strukturze bazy danych) są przechowywane i udostępniane tak samo (czyli w postaci tabel) jak zapisane w bazie informacje.
3. **Postulat niezależności ograniczeń.** Ograniczenia (więzy spójności) muszą być definiowane w tym samym języku SQL i przechowywane po stronie bazy danych, a więc nie jest obowiązkowe implementowanie ich po stronie aplikacji klienckiej. Serwer baz danych musi umożliwiać zdefiniowanie przynajmniej dwóch typów ograniczeń:
 - a. Ograniczenia klucza podstawowego, które gwarantują spójność danych w ramach tabel.
 - b. Ograniczenia klucza obcego, które gwarantują spójność danych zapisanych w powiązanych tabelach.

Normalizacja

Proces dostosowywania schematu bazy danych do wymogów modelu relacyjnego nazywa się normalizacją. Jego głównym celem jest wyeliminowanie wynikających z nadmiarowości danych anomali, które mogłyby doprowadzić do utraty spójności danych. Podczas normalizacji zmienia się strukturę tabel, tworzy nowe tabele i określa łączące je relacje, ale nie usuwa się ani nie modyfikuje przechowywanych w bazie informacji.

 Wskazówka	Edgar Frank Codd zdefiniował trzy postacie normalne i chociaż później zostały dodane kolejne trzy, to prawie wszystkie bazy danych doprowadza się do trzeciej postaci normalnej (3PN), a ponad 75% baz danych znormalizowanych jest do czwartej postaci normalnej (4NF). Baza danych znajdująca się w wyższej postaci normalnej musi spełniać wymogi wszystkich niższych postaci normalnych.
---	--

Pierwsza postać normalna

Głównym celem doprowadzania do pierwszej postaci normalnej jest wyeliminowanie nieatomowych atrybutów (tabela jest zgodna z 1PN, jeśli wszystkie jej kolumny przechowują atomowe, niepodzielne wartości). Na przykład kolumnę Adres należy rozbić na kilka kolumn przechowujących kod, nazwę miasta i ulicę, a kolumnę Osoba — na dwie: Imię i Nazwisko.

Za atomowe wartości należy przyjąć takie, które mogą być użyte w przyszłości do:

1. wyszukiwania, np. znalezienia osoby o podanym nazwisku;
2. sortowania, np. przygotowania listy osób ułożonej alfabetycznie według imion;
3. grupowania, np. policzenia osób mieszkających w poszczególnych miastach.

Ponadto aby spełnić wymogi pierwszej postaci normalnej, tabela musi posiadać kolumnę klucza podstawowego.

Druga postać normalna

Doprowadzenie tabeli do drugiej postaci normalnej polega na usunięciu z niej atrybutów (kolumn), które zależą funkcyjnie od części klucza podstawowego (tabela jest zgodna z 2PN, jeżeli znajduje się w pierwszej postaci normalnej i wartości jej wszystkich niekluczowych kolumn zależą od całego klucza podstawowego). **W praktyce oznacza to, że jeśli klucz podstawowy tabeli jest prosty** (założony na pojedynczej kolumnie), a nie złożony (założony na kilku kolumnach), **i tabela jest w 1PN, to spełnia ona też automatycznie wymogi drugiej postaci normalnej.**

Sytuacja taka miałaby miejsce, gdyby w tabeli SalesOrderDetail, której kluczem podstawowym są kolumny SalesOrderID oraz SalesOrderDetailID, znajdowała się kolumna z datą zamówienia. Wtedy kolumna OrderDate zależałaby wyłącznie od jednej części klucza podstawowego (kolumny SalesOrderID), innymi słowy — znając identyfikator zamówienia, moglibyśmy ze 100-procentową dokładnością odczytać jego datę i znajomość identyfikatora SalesOrderDetailID byłaby do tego niepotrzebna.

Trzecia postać normalna

Doprowadzenie tabeli do trzeciej postaci normalnej polega na znalezieniu i usunięciu przechodnich zależności pomiędzy atrybutami (tabela jest zgodna z 3PN, jeżeli jest już w drugiej postaci normalnej i wartości jej kolumn nie są zależne od niekluczowych atrybutów).

Na przykład jeśli w tabeli `Product` znajdowałaby się kolumna z nazwami kategorii, wartości w tej kolumnie powtarzałyby się tyle razy, ile produktów należały do danej kategorii. Nadmiarowość ta wynika z faktu, że nazwa kategorii zależy funkcyjnie nie tylko od identyfikatora produktu (znając go, możemy jednoznacznie odczytać nazwę kategorii), ale również od innych atrybutów produktu, takich jak jego nazwa czy numer.

Doprowadzanie tabel do trzeciej postaci normalnej polega na:

1. Utworzeniu tabel słownikowych, np. tabeli z nazwami miast.
Takie tabele zawierają listy (słowniki) używanych w bazie terminów, dzięki czemu zamiast każdorazowo posługiwać się danym terminem, wystarczy użyć jego identyfikatora.
2. Utworzeniu tabel łącznikowych, czyli takich, które umożliwiają budowanie relacji typu „wiele do wielu”. Na przykład jeżeli przyjmiemy, że ta sama osoba może zapisać się jednocześnie na kilka kursów, a na ten sam kurs z reguły może zapisać się wiele osób, powinniśmy utworzyć tabelę `KursOsoba` i umieścić w niej klucze obce tabel `Osoby` i `Kursy` oraz atrybuty konieczne dla połączenia kursanta z jego zajęciami, takie jak numer sali czy godzina rozpoczęcia zajęć.

Postać Boyce'a-Codda

Kolejną, czasami nazwaną postacią trzecią i pół, postacią normalną jest postać Boyce'a-Codda (BCNF). Jej formalna definicja brzmi następująco: tabela jest zgodna z BCNF, jeżeli jest już w trzeciej postaci normalnej i dla każdej nietrywialnej zależności między podzbiorami jej atrybutów zbiór będący wyznacznikiem jest jej zbiorem identyfikującym. Ponieważ łatwiej jest wyjaśnić postać Boyce'a-Codda na przykładzie niż na podstawie jej definicji, przypuśćmy, że w bazie danych znajduje się tabela z danymi na temat sklepów, sprzedawanych w nich towarów oraz ich kategorii. Założymy też, że ten sam sklep może sprzedawać dowolne produkty i ten sam produkt może być sprzedawany w różnych sklepach (taką relację

nazywa się relacją „wiele do wielu”) i że jeden produkt nie może należeć do różnych kategorii, ale do tej samej kategorii może należeć wiele produktów (jest to przykład znanej nam już relacji „jeden do wielu”) (tabela 1.6).

Tabela 1.6. Przykład tabeli niezgodnej z wymaganiami postaci Boyce'a-Codda

Sklep	Towar	Kategoria
Sklep A	Towar 1	Kategoria X
Sklep A	Towar 2	Kategoria X
Sklep A	Towar 3	Kategoria Y
Sklep B	Towar 1	Kategoria X
Sklep B	Towar 3	Kategoria Y

Kluczem podstawowym takiej tabeli mogłyby być kolumny Sklep i Towar, bo na podstawie obu tych wartości można jednoznacznie zidentyfikować pozostałe atrybuty, w tym kategorię. W takim wypadku tabela byłaby znormalizowana do trzeciej postaci normalnej. Gdybyśmy jednak chcieli dopisać do tej tabeli nowy sklep, mielibyśmy problem — jej kluczem jest kombinacja identyfikatora sklepu i sprzedawanych w nim produktów, a więc sklepu, w którym nic jeszcze nie sprzedaliśmy, nie dałoby się dopisać.

Aby rozwiązać ten problem, należało by podzielić tabelę na cztery osobne, spełniające wymogi postaci BCNF tabele, w których byłyby zapisane dane:

1. w pierwszej o sklepach;
2. w drugiej o produktach;
3. w trzeciej o kategoriach produktów;
4. w czwartej o sprzedawanych w danym sklepie produktach (to byłaby tak zwana tabela łącznikowa).

Czwarta postać normalna

Omówione do tej pory postacie normalne definiowane były za pomocą pojęcia zależności funkcyjnej, czyli zależności, w której na podstawie wartości jednej kolumny (lub kolumn) można wywnioskować wartości innych kolumn. Normalizując tabele zgodnie z tymi postaciami, minimalizowaliśmy liczbę powtarzających się wartości.

W definicji czwartej postaci normalnej termin „zależność funkcyjna” jest zastąpiony terminem „zależność wielowartościowa” (tabela jest zgodna z 4PN, jeżeli jest już w postaci Boyce'a-Codda i nie występują w niej zależności wielowartościowe). Ponownie najłatwiej postać tę wytlumaczyć na przykładzie.

Wyobraźmy sobie tabelę `Produkty`, w której w kolumnie `Nazwa` zapisane są nazwy różnych przedmiotów (np. krzesło, lornetka i tak dalej). W kolejnych kolumnach znajdują się wartości różnych atrybutów tych przedmiotów, np. w kolumnie `Obicie` kolor obicia, a w kolumnie `Ogniskowa` dane o ogniskowej obiektywu. Jako że różne przedmioty mają różne cechy (np. krzesła nie mają ogniskowej), wiele komórek takiej tabeli będzie pustych.

Powodem tej anomalii jest występowanie zależności wielowartościowej, czyli takiej, w której na podstawie jednej kolumny (nazwy przedmiotu) można wnioskować o wielu kolumnach (takich jak `Ogniskowa` czy `Obwód w pasie`). Aby doprowadzić tabelę do postaci zgodnej z czwartą postacią normalną, należy ją rozbić na osobne tabele, których kolumny będą zawierały wyłącznie nazwy cech obiektów danego typu.

Podsumowanie

- Tabele, tak jak zbiory, są pojemnikami na dane.
- Tabele są zbudowane ze stałej liczby kolumn określonego typu i o niepowtarzalnych nazwach oraz ze zmiennej liczby wierszy.
- Tabele powinny mieć klucz podstawowy, który gwarantuje niepowtarzalność ich wierszy i umożliwia jednoznaczne wskazanie szukanego rekordu.
- Kolejność wierszy i kolumn jest bez znaczenia.
- Baza danych to zbiór powiązanych ze sobą informacji o ścisłe określonej strukturze.
- W modelu jednorodnym wszystkie informacje są przechowywane w jednej tabeli.
- W modelu relacyjnym informacje są przechowywane w wielu powiązanych ze sobą tabelach.
- W modelu obiektowym informacje są przechowywane w postaci obiektów różnych typów.

- Teoretyczne podstawy modelu relacyjnego, w tym trzy pierwsze postacie normalne, zostały zdefiniowane w latach 70. XX wieku przez dra E.F. Codda.

Zadania

1. Wyodrębnij kolumny, które zgodnie z definicją 3PN powinny być przeniesione do tabel słownikowych.

Płyty {ID płyty, Nazwa wykonawcy, Nazwa gatunku, Czas trwania, Opinia, Narodowość wykonawcy}

2. Jak należy przekształcić poniższe tabele, aby każda książka mogła mieć dowolnie wielu autorów, a każdy autor mógł napisać dowolną liczbę książek?

Autorzy {ID autora, Imię, Nazwisko}

Książki {ID książki, Tytuł, ID autora1, ID autora2, ID autora3}

3. Doprowadź poniższą tabelę kolejno do pierwszej, drugiej i trzeciej postaci normalnej.

Uczniowie {Imię, Nazwisko, Adres, Ocena, Data wystawienia oceny, Uwagi}

[1] Zbiór z powtórzeniami nazywa się wielozbiorem.

[2] W praktyce pierwsza kolumna tabeli prawie zawsze jest kolumną klucza podstawowego, a druga zawiera najbardziej charakterystyczny dla danego typu obiektów atrybut, np. nazwisko w przypadku osób lub nazwę w przypadku towarów. Taką kolejność kolumn zakładają różne narzędzia automatyzujące pisanie programów klienckich baz danych. Dobrą praktyką jest też umieszczanie kolumn, w których mogą wystąpić wartości `NULL`, jako ostatnich kolumn tabeli.

[3] Dane opisujące inne dane, czyli np. informacje o typach czy nazwach kolumn, nazywa się metadanymi.

[4] Dopuszając informacje o kolejnym zakupie u tego samego sprzedawcy, będziemy mogli wybrać jego dane z listy.

[5] Po prawej stronie nazw tabel znajduje się nazwa schematu, w którym zostały one stworzone. W przykładowej bazie danych

schemat SalesLT zawiera wszystkie tabele związane ze sprzedażą produktów.

[6] Programiści wspomnianych wcześniej języków powiedzieliby, że musimy znać interfejs (zbiór metod, atrybutów i zdarzeń) danego obiektu.

[7] Tym językiem jest oczywiście SQL.

Rozdział 2. Standardy języka SQL

- Jak za pomocą pojedynczej instrukcji odczytać lub zmodyfikować wiele danych?
- Co to znaczy, że SQL jest językiem strukturalnym?
- Dlaczego nie ma jednego, uniwersalnego języka SQL?
- Po co uczyć się standardu ANSI języka SQL?

Strukturalny język zapytań

Pierwszą wersję strukturalnego języka zapytań opracowała w latach 70. XX wieku firma IBM. Był to język SEQUEL (ang. *Structured English Query Language*), który z czasem przekształcił się w SQL (ang. *Structured Query Language*).

Zaletą języka SQL jest przede wszystkim upraszczanie pracy z relacyjnymi bazami danych — zamiast krok po kroku określić, jak serwer bazodanowy ma wykonać dane polecenie, użytkownik deklaruje, w języku przypominającym potoczny angielski, spodziewany wynik (strukturę zwracanych danych). Zadaniem serwera bazodanowego jest zinterpretowanie i wykonanie takiego polecenia.

Język SQL umożliwia również wydajne przetwarzanie dużych ilości danych — tak dużych, że w większości przypadków niemieszczących się w pamięci operacyjnej komputera i z tego powodu w razie potrzeby odczytywanych i zapisywanych przez serwer na dyskach twardych.

Przetwarzanie zbiorów a przetwarzanie pojedynczych danych

Większość języków, np. C, Pascal czy Visual Basic, umożliwia przetwarzanie danych prostych typów, takich jak liczby czy ciągi znaków. Z kolei języki obiektowe, przykładowo C++, Java czy Visual Basic .NET, zostały stworzone z myślą o przetwarzaniu danych złożonych typów, takich jak instancje klas Osoba czy System.Web.UI.Control.

To, co łączy oba rodzaje języków, to konieczność pojedynczego przetwarzania danych wartość po wartości. Przedstawimy to na przykładzie algorytmu przeszukiwania binarnego (połówkowego)[\[1\]](#). Żeby znaleźć dany element według tego algorytmu, należy:

1. Podzielić dane wejściowe na dwie równe części.
2. Sprawdzić, czy szukany element jest równy elementowi wybranemu w poprzednim kroku do podzielenia danych. Jeżeli tak, szukany element został znaleziony.
3. W przeciwnym razie należy sprawdzić, czy:
 - a. szukany element jest mniejszy od elementu wybranego do podziału w pierwszym kroku — jeżeli tak, trzeba powtórzyć dwie pierwsze operacje dla pierwszej połowy danych wejściowych;
 - b. szukany element jest większy od elementu wybranego do podziału w pierwszym kroku — jeżeli tak, trzeba powtórzyć dwie pierwsze operacje dla drugiej połowy danych wejściowych.

Przykładowa implementacja tego algorytmu w Pascalu wygląda następująco:

```
function przeszukiwanieBinarne(var A : Array of Integer; p,n,m : Integer)
: Integer;
var
  i : Integer;
begin
  i:= n+ ((m-n) div 2); // Podziel pozostałą część tablicy na pół.
  if A[i]=p then przeszukiwanieBinarne:=i // Sprawdź, czy element tablicy
                                              // jest równy poszukiwanemu.
  else
    if A[i]<p then przeszukiwanieBinarne:=przeszukiwanieBinarne(A,p,i+1,m)
    else przeszukiwanieBinarne:=przeszukiwanieBinarne(A,p,n,i-1);
end;
```

W tym przypadku zmienna *A* zawiera przeszukiwaną tablicę, *p* — poszukiwany element, *n* — dolny indeks tablicy, a *m* — górny indeks tablicy[\[2\]](#).

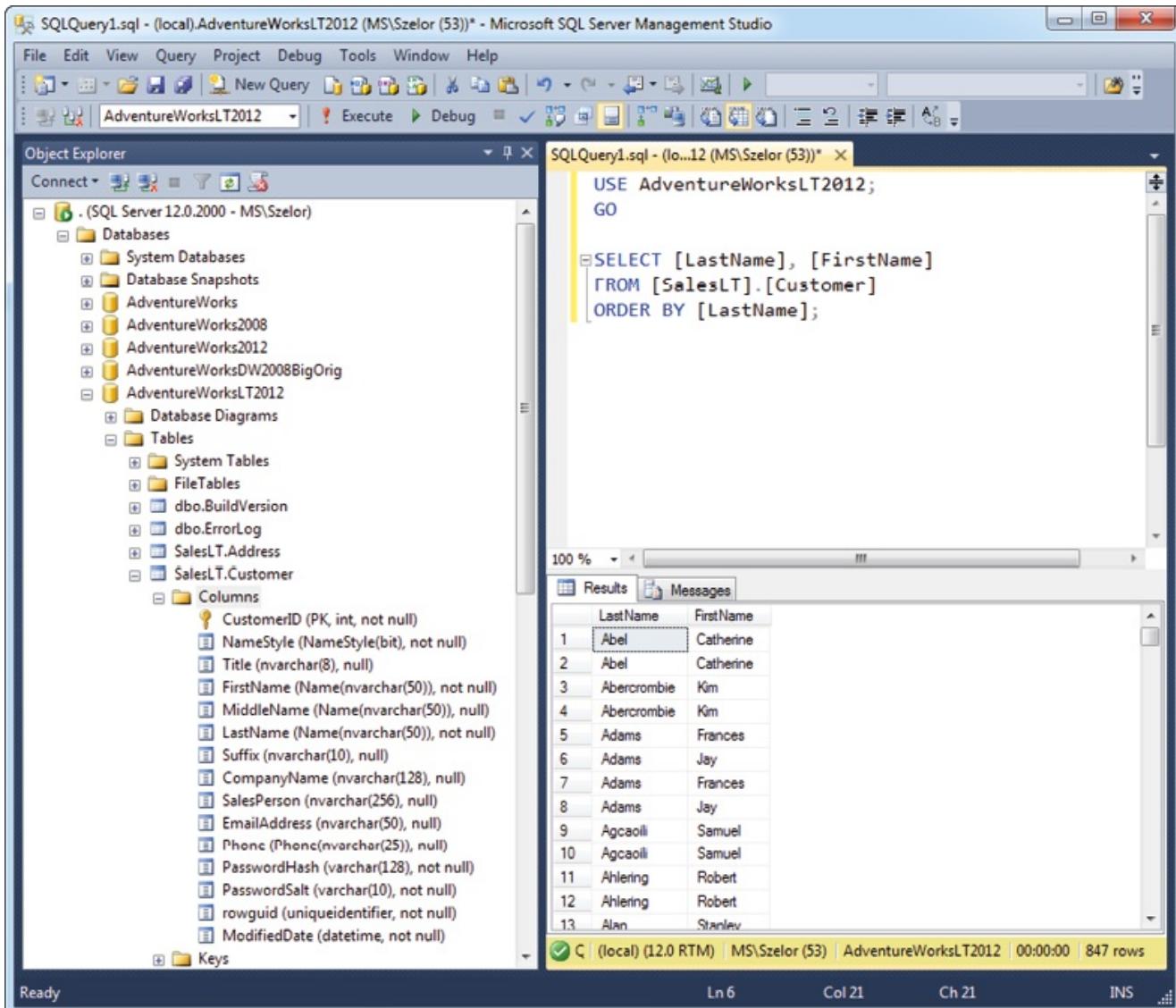
W odróżnieniu od innych języków SQL umożliwia przetwarzanie wielu danych za pomocą pojedynczej instrukcji. Wejściowy zbiór danych określa się w klauzuli FROM instrukcji SELECT, a pozostałe klauzule tej instrukcji określają operacje, które serwer bazodanowy wykona na wszystkich wierszach wskazanych tabel. W rezultacie wyszukująca żądany element funkcja może być zastąpiona poniższą instrukcją:

```
SELECT i AS ZwracanaKolumna  
FROM TabelaA  
WHERE p=i;
```

Język strukturalny a język proceduralny

W przeciwieństwie do języków proceduralnych SQL pozwala określić wynik, nie sposób jego osiągnięcia. Instrukcje języka SQL nie zawierają żadnych wskazówek dotyczących metody ich wykonywania przez serwer bazodanowy. W poprzednim przykładzie instrukcja SELECT nie określała sposobu (algorytmu), według którego serwer miał znaleźć szukany element w tabeli. Ponieważ w języku SQL deklaruje się spodziewany rezultat wykonania instrukcji, nazywa się go językiem strukturalnym.

Na przykład deklaracja *Odczytaj nazwiska i imiona klientów posortowane rosnąco według nazwisk* w języku SQL będzie wyglądać w sposób pokazany na rysunku 2.1.



Rysunek 2.1. Z lewej strony konsoli SSMSE znajduje się okienko eksploratora obiektów zawierające informacje m.in. o tabelach i ich kolumnach. Z prawej strony znajduje się okienko edytora SQL z przykładowym poleceniem, a poniżej — okienko z wynikiem wykonania tego polecenia przez serwer bazodanowy

Sposób wykonania instrukcji języka SQL zależy od serwera bazodanowego. To jego zadaniem jest znalezienie najlepszej (z reguły oznacza to: najszybszej) metody. W przeciwnieństwie do języków proceduralnych, których instrukcje są wykonywane w tej samej kolejności, w jakiej zostały zapisane, **w języku SQL kolejność wykonywania poleceń zależy od serwera i niekoniecznie odpowiada kolejności, w jakiej zapisano poszczególne instrukcje i ich klauzule^[3].**

Język interpretowany a język komplgowany

Wspomniane wcześniej języki: C, C++, Pascal, Visual Basic i Java są językami kompilowanymi — ich instrukcje są przekształcane do postaci kodu wykonywalnego przez kompilatory. W wyniku tego jednorazowego procesu powstają skompilowane wersje programów i to one są instalowane na komputerach użytkowników i mogą być przez nich wielokrotnie uruchamiane.

Instrukcje języka SQL, niezależnie od tego, czy są zakodowane w programie klienckim, wpisane bezpośrednio przez użytkowników (takie instrukcje nazywa się zapytaniami Ad-hoc), czy też zapisane na stałe w bazie danych (jako procedury składowane, funkcje użytkownika lub widoki), **przed wykonaniem są interpretowane przez serwer bazodanowy**. Wynikiem tej interpretacji jest plan wykonania instrukcji, który następnie jest realizowany przez serwer.

 Wskazówka	Podczas interpretacji przeprowadzana jest optymalizacja polegająca na znalezieniu jak najlepszego (w przypadku serwera SQL Server jak najtańszego) planu wykonania. Odpowiedzialne za to optymalizatory są z reguły najbardziej skomplikowanymi elementami serwerów bazodanowych.
--	---

Ponieważ instrukcje języka SQL są interpretowane, a nie kompilowane, niektóre błędy (takie jak próba odwołania się do nieistniejącej tabeli czy próba wstawienia danych tekstowych do kolumny typu liczbowego) zostaną wykryte dopiero podczas ich wykonywania. Problem ten nie występuje w językach kompilowanych.

Kolejną konsekwencją każdorazowego interpretowania instrukcji SQL jest dłuższy czas (a więc wyższy koszt) ich wykonania. Dotyczy to jednak wyłącznie sytuacji, w których język SQL jest używany niezgodnie z jego przeznaczeniem, czyli do przetwarzania pojedynczych, a nie całych zbiorów danych. Przy przetwarzaniu zbiorów optymalizatory współczesnych serwerów bazodanowych są w stanie znaleźć kilka, a nawet kilkudziesiąt tysięcy razy bardziej efektywne plany wykonania instrukcji SQL niż najlepszy algorytm zaimplementowany w językach kompilowanych. Wynika to:

1. ze sposobu odczytywania i zapisywania danych — serwery bazodanowe jednorazowo zapisują i odczytują całe bloki danych, a nie pojedyncze rekordy;
2. z dostosowywania typu i kolejności przeprowadzanych operacji

do odczytywanych w celu zrealizowania żądania użytkownika danych.

Składnia języka SQL

W języku SQL występuje pięć głównych kategorii syntaktycznych:

1. **identyfikatory**, czyli nazwy obiektów;
2. **literały**, czyli stałe;
3. **operatorы**, czyli spójniki;
4. **słowa kluczowe**, czyli wyrazy interpretowane przez serwer bazodanowy w określony sposób;
5. ignorowane przez serwery bazodanowe **komentarze**.

Instrukcja języka SQL zaczyna się poleciem, czyli słowem kluczowym określającym operację, która ma być wykonana, następnie z reguły występują dookreślające tę operację klauzule. W książce, dla lepszej czytelności, poszczególne klauzule są zapisywane w nowych wierszach. W rzeczywistości znak końca wiersza jest ignorowany przez interpretatory języka SQL, a znakiem końca całej instrukcji jest średnik.

Identyfikatory

Obiekty baz danych tworzą hierarchię — serwer zawiera wiele baz danych, baza danych może zawierać wiele schematów, w każdym schemacie może znajdować się wiele tabel, a każda tabela może się składać z wielu kolumn. **Każdy z tych obiektów** (baza, tabela czy kolumna) **musi mieć niepowtarzalną nazwę, czyli swój identyfikator.**

Identyfikatory muszą być zgodne ze zdefiniowanymi w standardzie języka SQL regułami:

1. Nie mogą składać się z więcej niż 128 znaków.
2. Mogą zawierać litery, cyfry oraz symbole: @, \$, #. Pozostałe symbole, w tym znak spacji, są niedozwolone.
3. Mogą zaczynać się literą, ale nie cyfrą. Identyfikatory zaczynające się jednym z dwóch dozwolonych symboli mają specjalne znaczenie:
 - a. Identyfikator rozpoczynający się symbolem @ oznacza zmienną.

- b. Identyfikator rozpoczynający się symbolem # oznacza obiekt tymczasowy.
4. Nie mogą być słowami kluczowymi języka SQL.

Dodatkowo identyfikatory powinny być zgodne z poniższymi konwencjami nazewniczymi:

1. Powinny być krótkie, ale jednoznacznie opisywać dany obiekt. Na przykład tabela zawierająca zamówienia z roku 2008 powinna nazywać się nie Z08, ale raczej Zamówienia2008.
2. Wielkość liter powinna być zgodna z przyjętymi w ramach projektu regułami. Nam najbardziej podoba się zasada mówiąca, że każdy wyraz (z wyjątkiem pierwszego) powinien zaczynać się od wielkiej litery, np. udfNajdroższeTowary.
3. Przedrostek nazw widoków, funkcji użytkownika, procedur składowanych czy wyzwalaczy powinien wskazywać na typ obiektu, np. udf — funkcja użytkownika (ang. *User Define Function*), usp — procedura użytkownika (ang. *User Stored Procedure*), v — widok (ang. *View*), tr — wyzwalacz (ang. *Trigger*).

Literały

Wszystkie cyfry, ciągi znaków i daty, jeżeli nie są identyfikatorami, są traktowane jako stałe, czyli literały. **W języku SQL ciągi znaków umieszcza się w apostrofach:**

```
SELECT 'Przykładowy ciąg znaków';
```

Operatory

Operatory odgrywają rolę spójników. Niektóre z nich mogą (ale nie powinny, bo instrukcje z nimi są czytelniejsze) być zastąpione odpowiednimi funkcjami. Operatory dzielą się na:

1. Arytmetyczne, do których należą: iloczyn *, iloraz /, modulo %, suma + i różnica -.
2. Znakowe, do których należą: konkatenacja (złączenie ciągów znaków) +, symbol wieloznaczny (zastępujący dowolny ciąg znaków) % i symbol wieloznaczny (zastępujący jeden znak) _.
3. Logiczne, do których należą: koniunkcja AND, alternatywa OR i negacja NOT.

4. Porównania, do których należą: równy =, mniejszy niż <, większy niż >, mniejszy lub równy <=, większy lub równy >= i różny != lub <>.
5. Charakterystyczne dla języka SQL. Należą do nich m.in. [\[4\]](#): przynależność do zbioru IN, przynależność do domkniętego przedziału BETWEEN ... AND, zgodność ze wzorem LIKE, kaskadowe wykonanie operacji CASCADE oraz wywołanie funkcji tabelarycznej APPLY.

 Wskazówka	To, czy operator + oznacza sumę, czy konkatenację, zależy od kontekstu jego użycia. Dotyczy to również operatora %.
---	---

Słowa kluczowe

Słowa kluczowe to zastrzeżone, mające ściśle określone znaczenie ciągi znaków. Należą do nich:

1. instrukcje języka SQL, takie jak SELECT czy CREATE;
2. klauzule języka SQL, np. WHERE lub JOIN;
3. nazwy typów danych, np. INT lub CHAR;
4. nazwy funkcji systemowych, takie jak ISNULL() lub ABS();
5. terminy zarezerwowane dla przyszłego użycia w danym serwerze bazodanowym.

Komentarze

W języku SQL występują dwa rodzaje komentarzy:

1. Podwójny znak myślnika oznacza komentarz w wierszu. Część wiersza, która znajduje się za znakami --, jest traktowana jako komentarz:

```
SELECT 'test'; -- przykładowy komentarz
```

2. Znaki /* oznaczają początek bloku komentarza, a znaki */ — jego koniec. Wiersze znajdujące się pomiędzy tymi znakami są traktowane jako komentarz:

```
/*
  Funkcja zwraca ostatnie zamówienie
  złożone przez klienta w danym sklepie
```

*/

Dialekty języka SQL

Każdy producent serwerów bazodanowych stara się przekonać użytkowników do wyboru właśnie jego produktu. Jednym ze sposobów na uczynienie danego serwera atrakcyjniejszym od innych jest zaimplementowanie w nim specyficznych i niedostępnych w produktach konkurencji funkcji. W efekcie w różnych serwerach zastosowane są różne dialekty języka SQL.

Jeszcze kilkanaście lat temu główną przyczyną powstawania odmiennych dialektów języka SQL było to, że użytkownicy oczekiwali od serwerów bazodanowych funkcji nieujętych w standardzie tego języka. Sytuacja zmieniła się w roku 1999, kiedy został przyjęty standard ANSI SQL99. Dziś większa część instrukcji języka SQL zaimplementowanych w produktach największych producentów serwerów bazodanowych jest zgodna z tym standardem.

Jednak standard SQL99 nie definiuje wielu rozszerzeń języka SQL, przede wszystkim instrukcji sterujących wykonaniem programu oraz metod obsługi błędów. Producenci serwerów bazodanowych mogli więc implementować te rozszerzenia na swój sposób.

Standard SQL99 nie obejmuje także wewnętrznych mechanizmów wykonywania i optymalizacji instrukcji języka SQL przez serwery bazodanowe. Tu różnice pomiędzy poszczególnymi serwerami są największe — istnienie rozmaitych mechanizmów składowania i odczytywania danych powoduje np. różnice w strukturze i przydatności indeksów. Ponadto prawie każdy dialect języka SQL umożliwia sterowanie pracą optymalizatora za pomocą dyrektyw, np. wymuszanie łączenia tabel w określony sposób lub poprzez wskazany algorytm. Lista i działanie takich dyrektyw są zupełnie różne dla poszczególnych serwerów.

Do najpopularniejszych dialektów języka SQL należą:

1. T-SQL — chociaż stosowane w serwerach Microsoft SQL Server i Sybase Adaptive Server odmiany tego języka różnią się, to wciąż klasyfikuje się je jako jeden dialect.
2. PL/SQL (ang. *Procedural Language/SQL*) — przypominający język ADA, bardzo rozbudowany dialect języka SQL stosowany w

- serwerach firmy Oracle.
3. PL/pgSQL (ang. *Procedural Language/PostgreSQL Structured Query Language*) — przypominająca dialekt PL/SQL wersja języka SQL zaimplementowana w serwerze PostgreSQL.
 4. SQL PL (ang. *SQL Procedural Language*) — dialekt używany w serwerach bazodanowych firmy IBM.

Standardy ANSI

Żeby zachęcić producentów serwerów bazodanowych do ujednolicenia dialektów języka SQL i jednocześnie ułatwić im tę czynność, Amerykański Narodowy Instytut Standardów ANSI (ang. *American National Standards Institute*) oraz Międzynarodowa Organizacja Normalizacyjna ISO (ang. *International Standard Organisation*) opracowują i publikują standardy języka SQL.

Historia

Pierwszą próbą uporządkowania i ujednolicenia dialektów języka SQL był standard ANSI z roku 1986. Rok później ISO zaakceptowała ten standard, wydając własny dokument normalizacyjny. W 1989 roku został opublikowany zweryfikowany standard ANSI, znany powszechnie jako SQL1. Niestety, głównie ze względu na sprzeczne interesy producentów systemów bazodanowych, standard ten nie określał wielu podstawowych cech języka, a liczne właściwości zdefiniowano jako zależne od implementacji.

Kolejną próbą ujednolicenia języka było przyjęcie w 1992 roku standardu SQL2. Niezbędnym kompromisem okazało się wprowadzenie trzech poziomów zgodności z nowym standardem:

1. Podstawowy poziom zgodności (ang. *Entry-level conformance*) był właściwie powtórzeniem standardu SQL1. Ten poziom zgodności wdrożyli w swoich produktach prawie wszyscy producenci serwerów bazodanowych.
2. Pośredni poziom zgodności (ang. *Intermediate-level conformance*) stanowił zbiór zasadniczych ujednolicień języka; poziom ten był ogólnie osiągalny.
3. Pełny poziom zgodności (ang. *Full conformance*) oznaczał spełnienie wszystkich wymogów standardu SQL2.

Najlepiej znanym i najpowszechniejszym stosowanym elementem standardu SQL2 był podział instrukcji na trzy kategorie:

1. Instrukcje DDL (ang. *Data Definition Language* — język definiowania danych) służące do tworzenia, modyfikowania i usuwania obiektów. Do tej kategorii należały instrukcje CREATE, ALTER i DROP.
2. Instrukcje DML (ang. *Data Manipulation Language* — język modyfikowania danych) pozwalające odczytywać i modyfikować dane. Do tej kategorii zostały zaklasyfikowane instrukcje SELECT, INSERT, UPDATE i DELETE.
3. Instrukcje DCL (ang. *Data Control Language* — język kontroli dostępu do danych) umożliwiające nadawanie i odbieranie uprawnień użytkownikom. Do tej kategorii należały instrukcje GRANT i REVOKE.

W 1999 roku organizacje ANSI i ISO opracowały SQL3. Był to pierwszy standard obejmujący zaawansowane funkcje i obszary zastosowań języka SQL, takie jak modele obiektowo-relacyjnych baz danych, mechanizmy wywoływania instrukcji języka SQL czy techniki zarządzania spójnością danych.

Cztery lata później, w roku 2003, został przyjęty czwarty standard języka SQL. Był on rozszerzeniem SQL3 o takie elementy, jak:

1. obsługa danych typu XML;
2. funkcje rankingu;
3. instrukcja MERGE;
4. jednolity mechanizm generowania wartości, np. identyfikatorów wierszy.

W 2006 roku został przyjęty kolejny standard języka SQL, w którym prawie wszystkie nowości dedykowane zostały obsłudze dokumentów XML przez serwery bazodanowe. Definiuje on m.in. sposoby przechowywania danych XML w tabelach oraz przeszukiwania i modyfikowania zawartości dokumentów XML za pomocą języka XQuery.

Rok 2008 przyniósł kolejny standard języka SQL, w którym zdefiniowano zgodną z teorią zbiorów (operującą na wielu, a nie na pojedynczych rekordach) klauzulę ORDER BY, wyzwalacze typu INSTEAD

`OF` (uruchamiane zamiast oryginalnej instrukcji użytkownika) oraz instrukcję obcięcia tabeli `TRUNCATE`.

Siódma wersja standardu ANSI SQL została opracowana w roku 2011. Wzbogaciła ona język SQL o obsługę temporalnych danych, m.in. określony został sposób definiowania kluczy czasowych (kluczy definiujących początek i koniec danego okresu), tabel przechowujących wersje rekordów z różnego czasu i składania zapytań wybierających dane na podstawie kluczy czasowych.

SQL3

W książce przedstawiśmy standard SQL3 z uwzględnieniem późniejszych rozszerzeń, w wersji zaimplementowanej przez firmę Microsoft w serwerze SQL 2012 i późniejszych. Opisane w niej przykłady i ćwiczenia można, po nielicznych modyfikacjach, wykonać na każdym serwerze bazodanowym zgodnym z tym standardem.

Klasy instrukcji

Wspomniany wcześniej podział instrukcji języka SQL na trzy kategorie (DDL, DML i DCL) okazał się mało precyzyjny i musiał być uzupełniony o nowe funkcje serwerów bazodanowych. Dlatego w standardzie SQL3 zostało zdefiniowanych siedem klas instrukcji:

1. *Connection Statements* — instrukcje umożliwiające nawiązanie i zakończenie połączenia z serwerem, np. `CONNECT` i `DISCONNECT`;
2. *Control Statements* — instrukcje sterujące wykonaniem programu, np. `CALL` i `RETURN`;
3. *Data Statements* — instrukcje mające trwały wpływ na dane, np. `SELECT`, `INSERT`, `UPDATE` i `DELETE`;
4. *Diagnostics Statements* — instrukcje diagnostyczne umożliwiające obsługę błędów, np. `GET DIAGNOSTICS`, `TRY ... CATCH` i `RAISERROR`;
5. *Schema Statements* — instrukcje mające trwały wpływ na obiekty baz danych, np. `CREATE`, `ALTER` i `DROP`;
6. *Session Statements* — instrukcje kontrolujące opcje sesji użytkowników, np. `SET`;
7. *Transaction Statements* — instrukcje umożliwiające rozpoczęcie i zakończenie transakcji, np. `START`, `COMMIT` i `ROLLBACK`.

Typy danych

Typ danych określa, jakiego rodzaju informacje mogą być przechowywane w poszczególnych kolumnach tabel lub w zmiennych oraz jakiego typu dane mogą być przekazywane jako parametry wywołania procedury lub funkcji. Listę typów danych zdefiniowanych w standardzie SQL3 zawiera tabela 2.1.

Tabela 2.1. Typy danych zaimplementowane w poszczególnych serwerach bazodanowych mogą się różnić od standardowych typów danych, nawet jeżeli mają te same nazwy

Kategoria	Przykładowe typy danych	Opis
Typy liczbowe	INTEGER (INT), SMALLINT NUMERIC (DECIMAL) REAL	Reprezentują liczby całkowite. Reprezentuje liczby o określonej skali i precyzji. Reprezentuje liczby o zmiennej precyzji.
Typy daty i czasu	DATE TIME	Reprezentuje datę. Reprezentuje czas.
Typy znakowe	CHAR VARCHAR NCHAR, NVARCHAR	Reprezentuje ciąg znaków o określonej długości. Reprezentuje ciąg znaków o zmiennej długości. Reprezentują ciąg znaków o stałej lub zmiennej długości zakodowanych w UNICODE.
Typy binarne	BINARY VARBINARY BLOB	Reprezentuje ciąg bitów o określonej długości. Reprezentuje ciąg bitów o zmiennej długości. Reprezentuje duże obiekty binarne.
Dokumenty XML	XML	Reprezentuje całe dokumenty XML.

Wartość NULL

Zgodnie z jednym z postulatów dra Codda serwery bazodanowe powinny spójnie przetwarzać wartość specjalną NULL. **Wartość NULL** reprezentuje brakujące, nieznane lub nieistotne dane i jest różna od 0 oraz od pustego ciągu znaków. Na przykład brak ceny produktu nie oznacza, że jest on darmowy, tylko że jego cena nie została jeszcze ustalona.

Z powodu występowania wartości NULL w serwerach bazodanowych obowiązuje logika trójwartościowa, a nie dwuwartosciowa. Porównanie wartości NULL z dowolną inną wartością daje więc w wyniku wartość nieznaną (ang. Unknown), a nie prawdę lub fałsz. Możemy się o tym

przekonać, wykonując poniższe zapytania — żadne z nich nie zwróci ani 1, ani 0:

```
SELECT 1  
WHERE NULL = NULL  
UNION  
SELECT 0  
WHERE NULL <> NULL;
```

```
SELECT 1  
WHERE NULL>1  
UNION  
SELECT 0  
WHERE NULL<1;
```



Wskazówka Wartość specjalna NULL nie jest równa jakiekolwiek innej wartości ani od niej różna.

Z tego powodu suma liczby produktów o określonym rozmiarze (np. w rozmiarze XL) i produktów o innym rozmiarze nie musi się równać liczbie wszystkich produktów — jeżeli rozmiar pewnych produktów jest nieznany lub nieokreślony (czyli w kolumnie `Size` jest wartość `NULL`), oba warunki logiczne `[Size] = 'XL'` oraz `[Size] <> 'XL'` dadzą w wyniku wartość nieznaną, a jak dowiemy się w następnych rozdziałach, tylko te wiersze, dla których klauzula `WHERE` zwraca prawdę, trafiają do wyniku zapytania.

Sytuację tę ilustrują poniższe zapytania (nie martw się, jeżeli ich składnia nie jest oczywista — wszystkie elementy tych zapytań zostały dokładnie opisane w dalszej części książki):

```
SELECT COUNT([Size])  
FROM [SalesLT].[Product];
```

```
SELECT COUNT([Size])  
FROM [SalesLT].[Product]  
WHERE [Size] = 'XL';
```

```
SELECT COUNT([Size])
FROM [SalesLT].[Product]
WHERE [Size] <> 'XL';
```

```
-----  
211  
3  
208
```

Reguły przetwarzania wartości `NULL` są następujące:

- Dwie wartości `NULL` nie są ani sobie równe, ani różne od siebie. Wartość `NULL` nie jest też równa jakiekolwiek innej wartości ani mniejsza czy większa od niej. Sensowne wyniki daje jedynie sprawdzenie (za pomocą operatora `IS`), czy dana wartość jest [nieznana](#)^[5].
- Wynikiem wszystkich operacji zawierających `NULL` jest wartość `NULL`, co pokazuje poniższy przykład:

```
SELECT NULL/0, 'Ala' + NULL, 5 + NULL, 10 * NULL, NULL + NULL;
```

```
-----  
NULL           NULL           NULL           NULL           NULL
```

- Wartość `NULL` jest ignorowana przez wszystkie funkcje grupujące z wyjątkiem funkcji `COUNT(*)`.

Poziomy zgodności

Poziomy zgodności pojawiły się po raz pierwszy w standardzie SQL2. Miały one ułatwić producentom serwerów bazodanowych stopniowe uzgadnianie swoich produktów z wymogami tego standardu. Okazało się jednak, że osiągnięcie pierwszego, podstawowego poziomu zgodności wystarczyło producentom do ogłoszenia, iż ich produkty są zgodne ze standardem SQL2, choć w praktyce dialekty języka SQL były zupełnie różne.

W standardzie SQL3 wprowadzono nowy model zgodności — serwer bazodanowy musi implementować wszystkie najważniejsze funkcje, żeby producent mógł ogłosić jego podstawową (ang. *Core*) zgodność z tym standardem.

Wyższe poziomy zgodności zostały zastąpione pakietami

dodatkowymi — ich lista znajduje się w tabeli 2.2. Serwer bazodanowy musi implementować funkcje przynajmniej jednego pakietu dodatkowego, żeby producent mógł ogłosić jego rozszerzoną (ang. *Enhanced*) zgodność ze standardem SQL3.

Tabela 2.2. Lista dodatkowych pakietów założeń standardu SQL3

Nazwa	Najważniejsze funkcje
PKG01 Rozszerzona obsługa typów danych	Dodatkowe typy danych: <ul style="list-style-type: none">• dokładny typ daty i czasu;• typ daty i czasu uwzględniający strefy czasowe;• typy danych użytkownika.
PKG02 Dodatkowe mechanizmy spójności danych	Dodatkowe mechanizmy zapewniania spójności danych: <ul style="list-style-type: none">• asercje;• podzapytania w ograniczeniu CHECK;• wyzwalacze uruchamiane dla każdej instrukcji;• kaskadowe usuwanie i aktualizowanie danych.
PKG03 Funkcje OLAP	Dodatkowe funkcje analityczne: <ul style="list-style-type: none">• operatory CUBE i ROLLUP;• konstruktory wierszy i tabel;• operator INTERSEC;• obustronnełączenie zewnętrzne.
PKG04 Dodatkowe moduły programistyczne	Proceduralne rozszerzenia języka SQL: <ul style="list-style-type: none">• instrukcje CASE, IF, WHILE, LOOP, FOR, REPEAT;• widoki systemowe INFORMATION_SCHEMA;• możliwość dołączania modułów kodu.
PKG05 Interfejs kliencki	Zgodność z dodatkowym API (ang. <i>Application Programming Interface</i>), pozwalającym na osadzanie instrukcji języka SQL i wywoływanie modułów SQL z poziomu programu klienckiego.
PKG06 Podstawowe techniki obiektowe	Obiektowe rozszerzenia języka SQL: <ul style="list-style-type: none">• przeciążanie funkcji i procedur;• złożone typy użytkownika;• dziedziczenie;• wskaźniki;• tablice.
PKG07 Zaawansowane techniki obiektowe	Obiektowe rozszerzenia języka SQL: <ul style="list-style-type: none">• konstruktory i destruktory;• zagnieżdżanie tabel;• operator ONLY.
PKG08	Możliwość automatycznego wywoływania procedur składowanych.

Wyzwalcze	
PKG09 Multimedia	Możliwość przechowywania i przetwarzania strumieni danych, takich jak pliki audio lub wideo.

Podsumowanie

- Pojedyncza instrukcja języka SQL pozwala odczytać lub zmodyfikować wiele danych.
- W języku SQL deklaruje się wynik, a nie sposób jego osiągnięcia.
- Instrukcje języka SQL są przed wykonaniem interpretowane przez serwer bazodanowy.
- W języku SQL instrukcja rozpoczyna się polecением (czasownikiem), po którym umieszcza się dookreślające to polecenie klauzule.
- Każdy obiekt w bazie danych musi mieć jednoznacznie identyfikującą go nazwę.
- Do obiektów baz danych odwołujemy się za pomocą ich nazw.
- Wartość NULL jest różna od jakiekolwiek innej wartości, w tym od 0 i pustego ciągu znaków.
- Standardy języka SQL miały na celu ujednolicenie różnych dialektów tego języka.
- Dziś prawie każdy serwer bazodanowy jest w podstawowym stopniu zgodny ze standardem SQL3.

Zadania

1. Zaimplementuj w języku SQL poniższy, zapisany w pseudokodzie, algorytm przeszukiwania tabeli:
 - a. Odczytaj wiersz tabeli Customer.
 - b. Sprawdź, czy w kolumnie LastName znajduje się wartość Kumar.
 - c. Jeżeli tak, skopiuj wiersz do tabeli tymczasowej.
 - d. Sprawdź, czy tabela zawiera dalsze wiersze:
 - Jeżeli tak, powtórz poprzednie kroki.
 - Jeżeli nie, zwróć użytkownikowi zawartość tabeli tymczasowej.

2. Czy potrafisz bez wykonywania poniższych instrukcji powiedzieć, jakie będą ich wyniki?

```
SELECT 1  
WHERE NOT NULL = - NULL;  
SELECT 1+'1'+NULL;
```

3. Czy to, że SQL jest językiem interpretowanym, a serwer bazodanowy wykona każdą poprawną instrukcję przysłaną przez użytkownika, może mieć wpływ na bezpieczeństwo? Uzasadnij swoją odpowiedź, podając przykład.
-

[1] Algorytm ten wykorzystuje fakt, że większość danych przechowywanych w pamięciach komputerów jest uporządkowana w pewien sposób — alfabetycznie według nazwisk, rosnąco według pensji itp. Znając ten porządek, możemy wyeliminować z danych wejściowych te, które na pewno nie zawierają szukanego elementu.

[2] Program posiada pewną charakterystyczną dla algorytmów rekurencyjnych wadę — jeśli poszukiwany element nie zostanie znaleziony (bo nie ma go w przeszukiwanej tabeli), program wpada w nieskończoną pętlę. Żeby tego uniknąć, należy sprawdzić poprawność danych wejściowych.

[3] Więcej informacji na temat sposobu wykonywania zapytań przez serwery bazodanowe, w tym na temat kolejności wykonywania poszczególnych klauzul instrukcji SELECT, znajduje się w rozdziale 9.

[4] Lista dostępnych operatorów języka SQL zależy od konkretnego serwera bazodanowego.

[5] Od tej reguły są wyjątki. Zostały one omówione w punktach poświęconych klauzulom ORDER BY i GROUP BY.

Część II

Pobieranie danych, czyli instrukcja SELECT

Najczęściej używaną i jednocześnie najbardziej rozbudowaną instrukcją języka SQL jest instrukcja SELECT. Nazywa się ją **zapytaniem**, ponieważ pozwala pytać serwer bazodanowy o przechowywane przez niego dane.

W kolejnych rozdziałach drugiej części książki, poznając składnię instrukcji SELECT oraz ucząc się budować coraz bardziej skomplikowane zapytania, dowiesz się:

1. W jakiej kolejności serwery bazodanowe wykonują poszczególne klauzule zapytań.
2. Jak wykorzystywać serwer bazodanowy także do obliczania wyników zapytań, a nie tylko do odczytywania zawartości tabel.
3. Jak wybierać interesujące Cię w danej chwili dane.
4. Jak można odczytywać zapisane w wielu tabelach dane.
5. Na czym polega grupowanie danych i jak używać funkcji grupujących.
6. Czym partycjonowanie wierszy różni się od ich grupowania.
7. Jak używać funkcji rankingu oraz funkcji analitycznych.
8. Na czym polega przetwarzanie okienek danych.
9. Czym są podzapytania oraz jak używać ich do wyszukiwania danych i poprawy czytelności zapytań.
10. W jaki sposób serwery bazodanowe wykonują zapytania, czym są plany wykonania zapytań i jak je analizować w celu rozwiązania problemów z wydajnością zapytań.

Rozdział 3. Odczytywanie danych z wybranej tabeli

- Jak wskazać odczytywaną tabelę?
- Czym jest projekcja tabeli?
- Jak z wyniku zapytań wyeliminować powtarzające się wiersze?
- Jak za pomocą wyrażeń i funkcji systemowych obliczać zwracane dane?
- W jaki sposób poprawić czytelność wyników zapytań?
- Dlaczego wyniki zapytań się sortuje?

Klauzula FROM

Prawie każde zapytanie zawiera klaузę FROM. Wyjątkiem od tej reguły są instrukcje SELECT, które nie odczytują żadnych danych, a jedynie zwracają sumę dwóch liczb lub w inny sposób przekształcają przesłane do serwera w klauzuli SELECT dane^[1]:

```
SELECT 123 + 654, UPPER('sql');
```

```
-----  
777      SQL
```

Ponieważ rzadko używa się serwera bazodanowego jako kalkulatora, typowe zapytanie zawiera przynajmniej klauzulę FROM. Na przykład żeby odczytać nazwy firm klienckich i nazwiska ich przedstawicieli, należy wykonać poniższą instrukcję:

```
USE AdventureWorksLT2012  
SELECT [CompanyName], [LastName]  
FROM [SalesLT].[Customer];
```

```
-----  
CompanyName          LastName  
A Bike Store        Gee  
Progressive Sports  Harris  
Advanced Bike Components Carreras  
Modular Cycle Systems Gates
```

Metropolitan Sports Supply	Harrington
Aerobic Exercise Company	Carroll
Associated Bikes	Gash
...	

Przyjrzyjmy się poszczególnym klauzulom tego zapytania:

1. Instrukcja SELECT oznacza, że chcemy odczytać jakieś dane.
2. Po prawej stronie instrukcji SELECT znajdują się oddzielone przecinkami nazwy kolumn tabeli, które chcemy odczytać.
3. **Klauzula FROM umożliwia wskazanie tabeli zawierającej te kolumny i przechowującej interesujące nas dane.**

Serwery bazodanowe w pierwszej kolejności wykonują klauzulę FROM — dopiero po sprawdzeniu, czy tabela o takiej nazwie istnieje i czy zawiera wymienione w klauzuli SELECT kolumny, odczytują z tej tabeli żądane dane. Natomiast klauza SELECT jest wykonywana jako jedna z ostatnich. Z tego powodu w większości klauzul nie można odwołać się do wyrażeń zdefiniowanych w klauzuli SELECT — poniższa instrukcja jest błędna i nie zostanie wykonana:

```
SELECT [CompanyName], [LastName] as Nazwisko
FROM [SalesLT].[Customer]
WHERE Nazwisko ='Gee' ;
```

 Wskazówka	Zwróć uwagę, że wynik zapytania nie jest w żaden sposób posortowany. Wynik tego samego zapytania wykonanego na innym serwerze lub w innym czasie może zwrócić te same, ale inaczej uporządkowane dane.
---	--

W pełni kwalifikowane nazwy obiektów

Kompletne nazwy obiektów (czyli ich identyfikatory) odzwierciedlają opisaną w poprzednim rozdziale hierarchię: serwer może zarządzać wieloma bazami danych, w każdej bazie danych może istnieć wiele schematów, a każdy schemat może zawierać wiele obiektów, takich jak tabele czy procedury. Dlatego **w pełni kwalifikowana nazwa obiektu ma postać *nazwa serwera.nazwa bazy danych.nazwa schematu.nazwa obiektu*, przy czym:**

1. Opcjonalna *nazwa serwera* wskazuje serwer bazodanowy. Jeżeli jej nie podamy, instrukcja zostanie wykonana przez serwer, z

którym jesteśmy połączeni.

2. Opcjonalna *nazwa bazy danych* wskazuje nazwę bazy danych, w której znajduje się żądany obiekt. Jeżeli nie zostanie ona podana, serwer założy, że obiekt znajduje się w bieżącej bazie danych. W poprzednim przykładzie przed wykonaniem zapytania połączymy się z bazą *AdventureWorksLT2012*^[2], ale gdybyśmy tego nie zrobili, musielibyśmy wskazać nazwę tej bazy w klauzuli *FROM*:

```
SELECT [CompanyName], [LastName]  
FROM [AdventureWorksLT2012].[SalesLT].[Customer];
```

3. Opcjonalna *nazwa schematu* wskazuje schemat (przestrzeń nazwy grupującą powiązane ze sobą obiekty), w którym znajduje się żądana tabela. Gdybyśmy ją pominęli, serwer bazodanowy założyłby, że obiekt o podanej nazwie znajduje się w domyślnym schemacie użytkownika wykonującego daną instrukcję. Ponieważ tabela *Customer* znajduje się w schemacie *SalesLT*, a domyślnym schematem użytkownika jest schemat *dbo*, poniższa instrukcja nie zostałaby wykonana:

```
SELECT [CompanyName], [LastName]  
FROM [Customer];  
  
Msg 208, Level 16, State 1, Line 18  
  
Invalid object name 'Customer'.
```

Inaczej wyglądałaby sytuacja w przypadku tabeli *BuildVersion*. Ta tabela została zdefiniowana w domyślnym schemacie *dbo*, a więc wykonanie poniższego zapytania zakończy się odczytaniem interesujących nas danych:

```
SELECT SystemInformationID, [Database Version]  
FROM [BuildVersion];
```

```
-----  
SystemInformationID    Database Version  
1                      10.50.91013.00
```



Wskazówka

Podając nazwę schematu, unikniemy błędów wynikających z tego, że poszczególni użytkownicy mogą mieć przypisane różne schematy domyślne. Ponadto w przypadku większości serwerów bazodanowych posługiwanie się nazwami schematów skraca czas wykonywania instrukcji.

4. Obowiązkowa *nazwa obiektu* wskazuje obiekt, do którego chcemy się odwołać, w naszym przypadku tabelę **Klienci**.

Wybieranie kolumn

Wynikiem zapytania jest tabela zbudowana z kolumn i wierszy odczytanych z tabeli wskazanej w klauzuli `FROM` — pierwsza kolumna wyniku poprzedniego zapytania zawierała nazwy firm, a druga nazwiska ich przedstawicieli.

Kolejność kolumn wyniku zapytania definiuje się w klauzuli SELECT. Pierwsza wymieniona w niej nazwa będzie pierwszą kolumną wyniku, druga — drugą i tak dalej. W ten sposób wynik zapytania nie zależy od kolejności kolumn odczytywanej tabeli, co jest zgodne z opisany w rozdziale 1. relacyjnym modelem baz danych.

Oba poniższe zapytania zwracają te same wyniki, ale kolejność kolumn każdego z nich jest inna:

```
SELECT [LastName], [FirstName]
```

```
FROM [SalesLT].[Customer];
```

```
-----  
LastName           FirstName  
Gee                Orlando  
Harris              Keith  
Carreras            Donna  
Gates               Janet  
Harrington          Lucy
```

```
...
```

```
SELECT [FirstName], [LastName]
```

```
FROM [SalesLT].[Customer];
```

```
-----  
FirstName          LastName  
Orlando            Gee  
Keith               Harris  
Donna              Carreras  
Janet               Gates
```

Lucy
Rosmarie

Harrington
Carroll



Wybranie niektórych kolumn tabeli nazywa się selekcją pionową, rzutem lub projekcją tabeli. Wynikiem projekcji jest tabela o liczbie kolumn mniejszej niż liczba kolumn odczytywanej tabeli.

Jeżeli chcemy odczytać zawartość wszystkich kolumn tabeli i nie chcemy zmieniać ich kolejności^[3], możemy w klauzuli SELECT użyć symbolu *:

```
SELECT *
FROM [SalesLT].[ProductCategory];
```

```
-----  
ProductCategoryID ParentProductCategoryID Name rowguid  
ModifiedDate  
1 NULL Bikes CFBDA25C-DF71-47A7-B81B-64EE161AA37C  
2002-06-01 00:00:00.000  
2 NULL Components C657828D-D808-4ABA-91A3-  
AF2CE02300E9 2002-06-01 00:00:00.000
```

Odradzamy jednak używanie symbolu * — stosując go, narażamy się na otrzymanie błędnych, innych, niż się spodziewaliśmy, wyników. Sytuacja taka będzie miała miejsce, gdy ktoś zmieni kolejność lub liczbę kolumn tabeli (np. poprzez dodanie bądź usunięcie kolumny), do której odwołujemy się w klauzuli FROM. Co więcej, stosowanie symbolu * może znacznie wydłużyć czas wykonywania zapytania.



Używając symbolu *, zmuszamy serwer bazodanowy do odczytania wszystkich kolumn tabeli, co może wielokrotnie wydłużyć czas wykonywania zapytania, ponieważ serwer bazodanowy nie używa istniejącego indeksu. Jeżeli interesują nas dane tylko z niektórych kolumn, nie powinniśmy odczytywać całej tabeli, a następnie usuwać niepotrzebnych kolumn po stronie programu klienckiego.

Eliminowanie duplikatów

Tabele w relacyjnych bazach danych nie powinny zawierać powtarzających się wierszy, ale mogą zawierać powtórzenia niektórych danych (np. kluczy obcych). I tak, skoro kilka różnych produktów zostało sprzedanych w ramach tego samego zamówienia,

poniższe zapytanie zwróci wartość SalesOrderID powtózoną tyle razy, ile było pozycji danego zamówienia:

```
SELECT [SalesOrderID], [LineTotal]
FROM [SalesLT].[SalesOrderDetail];
```

```
-----
SalesOrderID    LineTotal
71774          356.898000
71774          356.898000
71776          63.900000
71780          873.816000
71780          923.388000
71780          406.792800
...
```

Domyślnie wyniki zapytań zawierają powtórzone dane, co odpowiada użyciu w klauzuli SELECT słowa kluczowego ALL. Możemy to sprawdzić, wykonując poniższe zapytania — oba zwracają te same 542 wiersze:

```
SELECT [ProductID]
FROM [SalesLT].[SalesOrderDetail];
SELECT ALL [ProductID]
FROM [SalesLT].[SalesOrderDetail];
```

Powtarzające się dane możemy wyeliminować za pomocą słowa kluczowego DISTINCT — poniższe zapytanie zwraca tylko 142 unikatowe wiersze reprezentujące 142 różne sprzedane towary:

```
SELECT DISTINCT [ProductID]
FROM [SalesLT].[SalesOrderDetail];
```

```
-----
ProductID
707
708
711
712
714
715
...
```

 Wskazówka	<p>Jeżeli odczytywane dane powinny być niepowtarzalne, użycie słowa kluczowego DISTINCT tylko ukryje błędy w strukturze bazy lub niespójność przechowywanych w niej informacji. Gdyby ktoś pomyłkowo wpisał dane tego samego produktu do tabeli [SalesLT].[Product], to odczytując nazwy produktów za pomocą słowa kluczowego DISTINCT, nie dowiedzielibyśmy się o tym błędzie. Innymi słowy, nie powinniśmy nadużywać słowa DISTINCT z dwóch powodów: po pierwsze dane w dobrze zaprojektowanej bazie powinny być spójne, po drugie wyeliminowanie duplikatów jest kosztowną operacją, która może znaczco wydłużyć czas wykonywania zapytania.</p>
---	---

Słowo kluczowe DISTINCT eliminuje z wyniku zapytania powtórzone wiersze, a nie powtórzone wartości wybranych kolumn. Z tego powodu może ono w klauzuli SELECT wystąpić tylko raz i zawsze dotyczyć wszystkich wymienionych w tej klauzuli kolumn.

Ponieważ kombinacje identyfikatorów produktów i identyfikatorów zamówień są unikatowe, poniższe zapytanie zwraca wszystkie 542 wiersze tabeli[\[4\]](#):

```
SELECT DISTINCT [ProductID],[SalesOrderID]
FROM [SalesLT].[SalesOrderDetail];
-----
```

ProductID	SalesOrderID
707	71782
707	71783
707	71784
707	71797
707	71902
707	71936
707	71938
708	71782
708	71783
708	71784
708	71797
708	71902
...	

Wyrażenia

Instrukcje SELECT mogą zawierać wyrażenia zbudowane z nazw kolumn i stałych połączonych standardowymi operatorami, np. operatorem + (dodawanie). W skład wyrażeń mogą również wchodzić funkcje systemowe, takie jak UPPER() czy LEFT().

 Wskazówka	Umieszczone w klauzuli SELECT wyrażenia muszą zwracać skalarne (pojedyncze) wartości.
--	---

Operatory arytmetyczne

Wszystkie standardowe operatory arytmetyczne, czyli + (dodawanie), - (odejmowanie), * (mnożenie), / (dzielenie bez reszty) i % (modulo^[5]), mogą być używane w języku SQL. Ich argumentami mogą być liczby lub dane typów, które serwer bazodanowy może automatycznie konwertować na liczby.

Poniższe zapytanie zwraca nazwy produktów, ich ceny i wynik pomnożenia tych cen przez 5:

```
SELECT [Name], [ListPrice], [ListPrice]*5  
FROM [SalesLT].[Product];
```

```
-----  
Name          ListPrice  (No column name)  
HL Road Frame - Black, 58    1431,50      7157,50  
HL Road Frame - Red, 58     1431,50      7157,50  
Sport-100 Helmet, Red      34,99       174,95  
Sport-100 Helmet, Black    34,99       174,95  
Mountain Bike Socks, M     9,50        47,50  
...
```

Kolejny przykład pokazuje, że oba argumenty wyrażenia mogą być nazwami kolumn — w takim przypadku serwer bazodanowy obliczy wynik wyrażenia dla wszystkich wierszy tabeli:

```
SELECT [Name], [ListPrice], [Weight], [ListPrice]*[Weight]  
FROM [SalesLT].[Product];
```

```
-----  
Name          ListPrice  Weight      (No column name)
```

```
HL Road Frame - Black, 58    1431,50      1016.04      1454461.260000
HL Road Frame - Red, 58     1431,50      1016.04      1454461.260000
Sport-100 Helmet, Red     34,99       NULL        NULL
```

...

Domyślną kolejność wykonywania operacji (najpierw mnożenie, dzielenie i modulo, następnie dodawanie i odejmowanie) możemy zmienić za pomocą nawiasów — w takim wypadku serwer bazodanowy najpierw wykona operacje zapisane w nawiasie:

```
SELECT [ListPrice] - [StandardCost] * [Weight]
```

```
FROM [SalesLT].[Product];
```

```
-----
```

```
-1074869.832400
```

```
-1074869.832400
```

...

```
SELECT ([ListPrice] - [StandardCost]) * [Weight]
```

```
FROM [SalesLT].[Product];
```

```
-----
```

```
378159.927600
```

```
378159.927600
```

...

Łączenie danych tekstowych

Normalizując projekt bazy danych, w poszczególnych kolumnach umieściliśmy niepodzielne dane. Na przykład imiona i nazwiska zostały zapisane w odrębnych kolumnach, a adres został rozbity na kilka części, z których każda (miasto, kod i ulica) trafiła do odrębnej kolumny.

Taki podział umożliwia nam wydajne wyszukiwanie i grupowanie danych, ale czasami chcemy połączyć dane z kilku kolumn (z reguły kolumn przechowujących dane tekstowe) w jedną całość. Umożliwia nam to operator konkatenacji, czyli łączenia ciągów znaków:

```
SELECT [ProductNumber], [Color], [ProductNumber] + ' ' + [Color]
FROM [SalesLT].[Product];
```

```
-----
```

ProductName	Color	(No column name)
HL Road Frame - Black, 58	1431,50	
HL Road Frame - Red, 58	1431,50	
Sport-100 Helmet, Red	34,99	

FR-R92B-58	Black	FR-R92B-58 Black
FR-R92R-58	Red	FR-R92R-58 Red
HL-U509-R	Red	HL-U509-R Red
HL-U509	Black	HL-U509 Black
FK-1639	NULL	NULL
FK-5136	NULL	NULL

...

Warto zwrócić uwagę na wynik wyrażenia w dwóch ostatnich z pokazanych wierszy — ponieważ w kolumnie `Color` występowała w nich wartość `NULL`, wynikiem również jest wartość nieznana. Dlatego jeżeli tylko dany serwer bazodanowy zawiera specjalną funkcję do łączenia ciągów znaków `CONCAT`, lepiej jest użyć tej funkcji niż operatora `+` (jeśli jednym z argumentów jej wywołania jest `NULL`, funkcja `CONCAT` zwróci wartość drugiego argumentu, a nie `NULL`):

```
SELECT [ProductNumber], [Color], CONCAT([ProductNumber],[Color])
FROM [SalesLT].[Product];
-----
```

ProductNumber	Color	(No column name)
FR-R92B-58	Black	FR-R92B-58Black
FR-R92R-58	Red	FR-R92R-58Red
HL-U509-R	Red	HL-U509-RRed
HL-U509	Black	HL-U509Black
FK-1639	NULL	FK-1639
FK-5136	NULL	FK-5136

...

Funkcje systemowe

Tak jak inne języki programowania, SQL udostępnia nam całą gamę funkcji systemowych. Ponieważ listy zaimplementowanych w poszczególnych serwerach bazodanowych funkcji systemowych są różne, poniżej przedstawiliśmy tylko kilka najpopularniejszych funkcji i proste przykłady ich użycia (dla lepszej czytelności argumentami wywołania funkcji są stałe, ale wszystkie funkcje mogą być wywoływane z nazwami kolumn).

Funkcje arytmetyczne

Do funkcji arytmetycznych zaklasyfikowaliśmy te, których argumentami i wynikiem są dane liczbowe (tabela 3.1).

Tabela 3.1. Typowe funkcje arytmetyczne

Funkcja	Opis	Przykład
ABS()	Zwraca wartość bezwzględną liczby.	SELECT ABS(1), ABS(-1); 1 1
CEILING()	Zwraca najmniejszą liczbę całkowitą równą argumentowi wywołania lub większą od niego.	SELECT CEILING (123.23); 124
FLOOR()	Zwraca największą liczbę całkowitą równą argumentowi wywołania lub mniejszą od niego.	SELECT FLOOR (123.93); 123
POWER()	Podnosi liczbę do potęgi określonej przez drugi argument wywołania.	SELECT POWER (5,3); 125
RAND()	Zwraca pseudolosową liczbę z zakresu od 0 do 1.	SELECT RAND(); 0,715245333903122
ROUND()	Zaokrąglą liczbę do określonej przez drugi argument wywołania liczby miejsc po przecinku. Jeżeli drugi argument będzie ujemny, zaokrąglona zostanie część całkowita.	SELECT ROUND (81.46,1), ROUND (9,-1); 81,50 10
SQRT()	Zwraca pierwiastek kwadratowy liczby.	SELECT SQRT (81); 9

Funkcje znakowe

Do funkcji znakowych zaklasyfikowaliśmy te, których argumentami wywołania są ciągi znaków. Niektóre z wymienionych w tabeli 3.2 funkcji zwracają dane znakowe, inne — liczby.

Tabela 3.2. Typowe funkcje znakowe

Funkcja	Opis	Przykład
LEN()	Zwraca długość przekazanego ciągu znaków.	SELECT LEN ('Kurs SQL'); 8
LOWER()	Zwraca przekonwertowany na małe litery ciąg znaków.	SELECT LOWER('udfDaneTowaru'); udfdanetowar
	Usuwa z ciągu znaków wiodące spacje.	SELECT LTRIM (' Kurs SQL');

LTRIM()		Kurs SQL
REPLACE()	Zamienia w podanym ciągu znaków wskazaną frazę na inną.	SELECT REPLACE ('Stop','St','H'); Hop
REPLICATE()	Powtarza ciąg znaków określoną liczbę razy.	SELECT REPLICATE ('A1',3); A1A1A1
RTRIM()	Usuwa z ciągu znaków wolne spacje.	SELECT LEN (RTRIM ('Kurs SQL ')); 8
SUBSTRING()	Zwraca część ciągu znaków o określonej długości, zaczynając od znaku o podanej pozycji.	SELECT SUBSTRING ('Kurs SQL',6,3); SQL
UPPER()	Zwraca przekonwertowany na duże litery ciąg znaków.	SELECT UPPER('udfDaneTowaru'); UDFDANE TOWARU

Funkcje daty i czasu

Do funkcji daty i czasu zaliczyliśmy te, których wynikami lub argumentami wywołania są zapisy daty oraz czasu (tabela 3.3).

Tabela 3.3. Typowe funkcje daty i czasu

Funkcja	Opis	Przykład
DATEADD()	Zwraca datę i czas zwiększone o podaną liczbę wskazanych jednostek.	SELECT DATEADD(DAY,20,'20071228'); 2008-01-17 00:00:00.000
DATEDIFF()	Zwraca liczbę podanych jednostek dzielących przekazane jako argumenty wywołania daty.	SELECT DATEDIFF (HOUR,'20070101','20071231'); 8736
DAY()	Zwraca numer dnia przekazanej jako argument daty.	SELECT DAY(GETDATE()); 21
GETDATE()	Zwraca bieżącą datę i czas.	SELECT GETDATE(); 2007-12-21 11:48:41.947
MONTH()	Zwraca numer miesiąca przekazanej jako argument daty.	SELECT MONTH(GETDATE()); 12
YEAR()	Zwraca rok przekazanej jako argument daty.	SELECT YEAR('20071231'); 2007

Konwersja typów

Chociaż większość serwerów bazodanowych przeprowadza automatyczną konwersję (rzutowanie) typów danych, radzimy zawsze jawnie określać typ przetwarzanych danych. W przeciwnym razie serwer posłuży się wbudowaną tablicą konwersji typów i na jej podstawie ustali typ wynikowy wyrażenia oraz znaczenie użytych w nim operatorów.

Na przykład w serwerze SQL 2012 wynikiem pierwszego zapytania będzie 3 (typy liczbowe mają wyższy priorytet niż znakowe, a więc oba operatory + zostaną zinterpretowane jako operatory dodawania), a drugiego — 12 (operacje są wykonywane od lewej do prawej, a pierwszy operator + zostanie zinterpretowany jako operator konkatenacji):

```
SELECT '1'+1+'1', '1'+'1'+1;
```

```
-----  
3      12
```

Jawną konwersję typów umożliwiają funkcje `CAST(wyrażenie AS typ)` oraz `CONVERT (typ, wyrażenie)`.

Pierwszym argumentem funkcji `CAST` jest konwertowana wartość, po której występuje słowo kluczowe `AS` i nazwa docelowego typu danych. Poniższe zapytanie, w którym próbujemy połączyć ze sobą kody produktów (dane tekstowe) z ich cenami (danymi walutowymi), zgłosi błąd:

```
SELECT [ProductNumber] + [ListPrice]  
FROM [SalesLT].[Product];
```

```
-----  
Msg 235, Level 16, State 0, Line 1
```

```
Cannot convert a char value to money. The char value has incorrect  
syntax.
```

Przeprowadzanie jawniej konwersji typu walutowego na tekstowy, czy to za pomocą funkcji `CAST`, czy `COVERT`, rozwiązuje problem:

```
SELECT [ProductNumber] + CAST([ListPrice] AS VARCHAR(15))  
FROM [SalesLT].[Product];
```

```
(No column name)
FR-R92B-581431.50
FR-R92R-581431.50
HL-U509-R34.99
HL-U50934.99
SO-B909-M9.50
SO-B909-L9.50
```

...

Specjalna funkcja CASE

Funkcja CASE jest SQL-owym odpowiednikiem instrukcji warunkowej IF ... THEN ... ELSE^[6]. Pozwala ona dla każdego wiersza wyniku zapytania sprawdzić wartość podanego wyrażenia i w zależności od wyniku testu zwrócić określoną po słowie kluczowym THEN wartość.

Funkcja CASE pozwoli np. na podstawie odczytanej ceny towarów zwrócić jej słowny opis:

```
SELECT [ListPrice],
CASE
    WHEN [ListPrice] <10 THEN 'Tani'
    WHEN [ListPrice] <50 THEN 'Średnia półka'
    ELSE 'Drogi'
END
FROM [SalesLT].[Product];
```

```
-----
```

ListPrice	(No column name)
1431,50	Drogi
1431,50	Drogi
34,99	Średnia półka
34,99	Średnia półka
9,50	Tani

```
...
```

	Kolejne warunki logiczne (klauzule WHEN) są sprawdzane w takiej kolejności, w jakiej zostały zapisane. Ta kolejność jest bardzo ważna — po znalezieniu pierwszego prawdziwego warunku funkcja CASE natychmiast zwraca
--	---



Wskazówka

odpowiedającą mu wartość, nie sprawdzając dla tego wiersza następnych warunków (kolejnych klauzul WHEN)[\[7\]](#).

Formatowanie wyników

Wyniki zapytań nie muszą wyglądać dokładnie tak jak odczytywana tabela — język SQL umożliwia nadawanie aliasów (alternatywnych, z reguły skróconych nazw) kolumnom i tabelom oraz dodawanie do wyniku kolumn zawierających stałe.

Aliases

Aliases kolumn definiuje się w klauzuli SELECT, albo podając je bezpośrednio po oryginalnej nazwie kolumny, albo poprzedzając je słowem kluczowym AS. **Radzimy zawsze używać słowa kluczowego AS** — poprawia ono czytelność zapytań, dzięki czemu łatwiej zauważyc brak przecinka pomiędzy nazwami dwóch odczytywanych kolumn.



Wskazówka

Alias to nie to samo co synonim. Aliasy są definiowane w ramach zapytania i można się do nich odwołać tylko w zapytaniu, w którym zostały zdefiniowane. Natomiast synonim to obiekt bazodanowy wskazujący na jakiś inny obiekt, tak jak umieszczony na pulpicie skrót do pliku lub folderu.

W poniższym zapytaniu nazwa kolumny Name została zastąpiona aliasem Nazwa produktu, a kolumna wyliczeniowa (czyli taka, która nie została bezpośrednio odczytana z tabeli, tylko wyliczona w zapytaniu) otrzymała nazwę Zysk:

```
SELECT [Name] AS [Nazwa Produktu], [ListPrice] - [StandardCost] AS Zysk  
FROM [SalesLT].[Product];
```

Nazwa Produktu	Zysk
HL Road Frame - Black, 58	372,19
HL Road Frame - Red, 58	372,19
Sport-100 Helmet, Red	21,9037
Sport-100 Helmet, Black	21,9037
Mountain Bike Socks, M.	6,1037

...

W kolejnym zapytaniu zdefiniowany został alias dla kolumny zwracającej zaokrąglone ceny produktów:

```
SELECT [ListPrice], ROUND([ListPrice],0) AS [W Zaokrągleniu]  
FROM [SalesLT].[Product];
```

ListPrice	W Zaokrągleniu
1431,50	1432,00
1431,50	1432,00
34,99	35,00
34,99	35,00

...

 Wskazówka	Ponieważ klauzula SELECT, choć w zapytaniach występuje jako pierwsza, jest wykonywana jako jedna z ostatnich, zdefiniowanych w niej aliasów nie można użyć w innych klauzulach zapytania. Wyjątkiem od tej reguły jest przedstawiona w podrozdziale „Sortowanie wyników” klauzula ORDER BY.
---	---

Aliases tabel definiuje się w ten sam sposób, ale w klauzuli FROM. Do zdefiniowanych w klauzuli FROM aliasów można się odwołać w innych klauzulach zapytania:

```
SELECT P.[ListPrice], ROUND([ListPrice],0) AS [W Zaokrągleniu]  
FROM [SalesLT].[Product] AS P;
```

ListPrice	W Zaokrągleniu
1431,50	1432,00
1431,50	1432,00
34,99	35,00
34,99	35,00

...

Jednak ponieważ klauzula FROM jest wykonywana w pierwszej kolejności, po zdefiniowaniu w niej aliasu dla tabeli w pozostałych klauzulach zapytania nie możemy już używać oryginalnej nazwy tabeli:

```
SELECT [SalesLT].[Product].[ListPrice], ROUND([ListPrice],0) AS [W  
Zaokrągleniu]
```

```
FROM [SalesLT].[Product] AS P;
```

```
-----  
Msg 4104, Level 16, State 1, Line 1
```

```
The multi-part identifier "SalesLT.Product.ListPrice" could not be bound.
```

Jak już powiedzieliśmy, w takim przypadku zamiast nazwą tabeli musimy się posłużyć zdefiniowanym dla niej aliasem.

Stałe (literały)

W klauzuli `SELECT` można również umieszczać literały, czyli dowolne stałe wartości, takie jak liczby, daty lub ciągi znaków. Dodane w ten sposób do wyniku zapytania kolumny będą zawierały te same, powtórzone w każdym wierszu, wartości.

Poniższe zapytanie zawiera dwie poprawiające czytelność wyniku stałe i funkcję `DATEPART()`, która zwraca określoną część znacznika daty oraz czasu:

```
SELECT 'Zamówienie ' + CAST([SalesOrderID] AS CHAR(5)) + ' zostało  
złożone w roku ' + CONVERT(CHAR(4),DATEPART(YEAR,[OrderDate]))
```

```
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
(No column name)
```

```
Zamówienie 71774 zostało złożone w roku 2008
```

```
Zamówienie 71776 zostało złożone w roku 2008
```

```
Zamówienie 71780 zostało złożone w roku 2008
```

```
Zamówienie 71782 zostało złożone w roku 2008
```

```
...
```

Jako specjalny rodzaj stałych należy też traktować większość wywoływanych bez parametrów i umieszczanych w klauzuli `SELECT` funkcji. **Takie funkcje są wywoływane raz dla całego zapytania, a nie dla każdego wiersza jego wyniku**^[8], a więc zwrócone przez nie wartości zostaną powtórzone:

```
SELECT [ProductNumber], GETDATE(), RAND()  
FROM [SalesLT].[Product];
```

```
-----  
ProductNumber (No column name) (No column name)
```

```
BB-7421 2014-07-29 09:43:11.830 0,761818926125455
```

BB-8107	2014-07-29 09:43:11.830	0,761818926125455
BB-9108	2014-07-29 09:43:11.830	0,761818926125455
BC-M005	2014-07-29 09:43:11.830	0,761818926125455
BC-R205	2014-07-29 09:43:11.830	0,761818926125455
BK-M18B-40	2014-07-29 09:43:11.830	0,761818926125455

...

W serwerze SQL Server wyjątkiem od tej reguły jest funkcja NEWID — ta, i tylko ta, funkcja systemowa zostanie wywołana osobno dla każdego wiersza, co pokazuje kolejne zapytanie:

```
SELECT [ProductNumber], NEWID()
FROM [SalesLT].[Product];
```

```
-----  
ProductNumber          (No column name)  
BB-7421                64C44871-5C6D-4719-8C59-6E22D8387668  
BB-8107                F1CDBCD8-0B30-407F-AA74-5B09BC8FE610  
BB-9108                9212DAD6-0243-4163-A38B-272962586714  
BC-M005                820BE6C1-F154-4ADE-BCCE-3C6C79858F2C
```

Sortowanie wyników

Kolejność wierszy wyniku zapytania jest niedeterministyczna i zależy od sposobu, w jaki serwer bazodanowy w danej chwili wykona to zapytanie. Jeżeli chcemy posortować wynik, do zapytania musimy dodać klauzulę ORDER BY.

 Wskazówka	Kolejność klauzul instrukcji SELECT jest ściśle określona, a opcjonalna klauzula ORDER BY musi być ostatnią klauzulą zapytania.
---	---

W klauzuli ORDER BY umieszcza się nazwy lub numery kolumn, według których chcemy posortować wynik zapytania. Ponieważ drugą kolumną zapytania jest ListPrice, wyniki obu zapytań zostaną posortowane w ten sam sposób:

```
SELECT [Name],[ListPrice]
FROM [SalesLT].[Product]
```

```
ORDER BY [ListPrice];
```

```
-----  
Name          ListPrice  
Patch Kit/8 Patches    2,29  
Road Tire Tube      3,99  
Touring Tire Tube    4,99  
Mountain Tire Tube   4,99  
Water Bottle - 30 oz. 4,99  
Bike Wash - Dissolver 7,95
```

```
...  
SELECT [Name],[ListPrice]  
FROM [SalesLT].[Product]  
ORDER BY 2;
```

```
-----  
-----  
Name          ListPrice  
Patch Kit/8 Patches    2,29  
Road Tire Tube      3,99  
Touring Tire Tube    4,99  
Mountain Tire Tube   4,99  
Water Bottle - 30 oz. 4,99  
Bike Wash - Dissolver 7,95
```

```
...
```

Domyślnie dane są szeregowane w porządku rosnącym, czyli od wartości najmniejszych do największych w przypadku danych liczbowych i od najwcześniejszych do najpóźniejszych w przypadku dat. Aby odwrócić kolejność sortowania, należy bezpośrednio po nazwie kolumny użyć słowa kluczowego **DESC** (ang. *Descending*):

```
SELECT [Name],[ListPrice]  
FROM [SalesLT].[Product]  
ORDER BY [ListPrice] DESC;
```

```
-----  
Name          ListPrice  
Road-150 Red, 62      3578,27
```

Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27
Mountain-100 Silver, 38	3399,99

...

Wyniki zapytań mogą być sortowane według wartości wielu kolumn. W takim przypadku kolejność, w jakiej te kolumny będą wymienione w klauzuli, wyznacza kolejność sortowania. W poniższym przykładzie wynik jest najpierw sortowany rosnąco według kolorów, a następnie malejąco według cen produktów o tym samym kolorze:

```
SELECT [ProductNumber], [Color], [ListPrice]
FROM [SalesLT].[Product]
ORDER BY [Color] DESC, [ListPrice];
```

...

ProductNumber	Color	ListPrice
S0-R809-M.	White	8,99
S0-R809-L	White	8,99
S0-B909-M.	White	9,50
S0-B909-L	White	9,50
PD-R347	Silver/Black	40,49
PD-M282	Silver/Black	40,49

...

Poszczególne klauzule zapytania są od siebie niezależne, co oznacza, że w każdej z nich możemy odwołać się do innych kolumn. W szczególności możliwe jest posortowanie wyniku zapytania według wartości kolumn niewymienionych w klauzuli SELECT[\[9\]](#):

```
SELECT [Color]
FROM [SalesLT].[Product]
ORDER BY [ListPrice];
```

Color

...

NULL

White

White

Multi

White

White

...

Na tym przykładzie możemy zauważyc, że:

1. Sortowanie według kolumny zawierającej duplikaty jest niedeterministyczne — wiersze zawierające powtórzone wartości użytej do sortowania kolumny nadal nie będą posortowane.



Powtórzone wiersze mogą być usunięte z wyniku za pomocą słowa kluczowego DISTINCT, a więc wynik zapytania może liczyć mniej wierszy niż oryginalna tabela. W takim przypadku sortowanie go według wartości nieistniejącej w nim kolumny jest niemożliwe. Innymi słowy, **jeżeli w zapytaniu zostało użyte słowo kluczowe DISTINCT, wszystkie wymienione w klauzuli ORDER BY kolumny i wyrażenia muszą występować w klauzuli SELECT.**

2. Sortowanie według kolumny niedołączonej do wyniku zapytania jest dla użytkowników mało przydatne.

Sortowanie danych tekstowych

Dane tekstowe są przechowywane i przetwarzane w postaci kodów — każdej literze alfabetu, cyfrze i znakowi specjalnemu odpowiada inna cyfra kodu. Serwery bazodanowe używają dwóch rodzajów kodowania danych tekstowych:

1. W kodowaniu ANSI jeden znak jest zakodowany za pomocą jednego bajta. Ogranicza to liczbę możliwych kodów do 256 (jeden bajt składa się z ośmiu bitów, z których każdy może przyjąć wartość 0 lub 1, a $2^8 = 256$). Ponieważ liczba stosowanych w różnych językach liter alfabetu jest znacznie większa^[10], w kodowaniu ANSI stosuje się strony kodowe.
2. W kodowaniu UNICODE jeden znak jest zakodowany za pomocą dwóch bajtów. Daje to 65 536 ($2^{16} = 65\,536$) różnych kodów, co wystarcza do zapisania wszystkich znaków diakrytycznych, a wtedy stosowanie stron kodowych jest niepotrzebne.

Oprócz typu kodowania (i strony kodowej w przypadku kodowania ANSI) serwery bazodanowe pozwalają też określić sposób sortowania danych tekstowych. Z reguły możemy zadecydować:

1. Czy ciągi znaków mają być sortowane według kodów (takie sortowanie nazywa się sortowaniem binarnym), czy w porządku alfabetycznym.
2. Czy mają być uwzględniane znaki diakrytyczne (w przypadku języka polskiego: czy litera *q* ma być umieszczona pomiędzy literami *a* i *b*, czy po *z*).

Wszystkie trzy czynniki (sposób kodowania, wybrana strona kodowa i sposób sortowania) mają wpływ na to, jak serwer bazodanowy będzie sortował ciągi znaków. Żeby się o tym przekonać, posortujmy malejąco imiona wszystkich klientów, przy czym za pierwszym razem posortujmy je alfabetycznie, używając domyślnej strony kodowej i standardowego porządku sortowania, a za drugim razem wymuśmy sortowanie binarne według znaków polskiej strony kodowej:

```
SELECT imie  
FROM imiona  
ORDER BY imie;
```

```
-----  
imie  
Łucja  
Rafał  
Stanisław  
Tadeusz  
Tomasz
```

```
SELECT imie  
FROM imiona  
ORDER BY imie COLLATE Polish_BIN;
```

```
-----  
Rafał  
Stanisław  
Tadeusz  
Tomasz
```

Łucja

Jak widać, w drugim przypadku imię zaczynające się od litery Ł trafiło na koniec listy.

Podsumowanie

- Nazwa odczytywanej tabeli jest podawana w klauzuli FROM.
- To klauzula FROM jest wykonywana jako pierwsza klauzula zapytania.
- Nazwy odczytywanych kolumn wymienia się oddzielone przecinkami w klauzuli SELECT.
- Nie powinno się odczytywać wszystkich kolumn za pomocą operatora *.
- Nazwy tabel (tak samo jak nazwy obiektów innych typów) powinno się poprzedzać nazwami schematów.
- Wynikiem zapytania jest tabela.
- Wynik zapytania zawiera powtórzone wiersze, chyba że zostaną one usunięte za pomocą słowa kluczowego DISTINCT.
- Wynik zapytania jest nieuporządkowany, chyba że zostanie posortowany za pomocą klauzuli ORDER BY.
- W klauzulach SELECT i ORDER BY można umieszczać wyrażenia i wywoływać funkcje systemowe, o ile tylko ich wynikami są pojedyncze wartości.
- Bezparametrowe funkcje są wywoływane raz dla całego zapytania, a nie dla poszczególnych zwracanych przez niego wierszy.
- Aliasy nazw tabel definiuje się w klauzuli FROM i obowiązują one w całym zapytaniu. Aliasy nazw kolumn definiuje się w klauzuli SELECT i można ich użyć tylko w klauzuli ORDER BY.

Zadania

1. Odczytaj z tabeli [SalesLT].[Product] nazwy towarów (kolumna [Name]) oraz powiększone o 20% ich ceny katalogowe (oblicz je na podstawie danych z kolumny [ListPrice]).
2. Oblicz, ile dni upłynęło pomiędzy złożeniem a wysłaniem zrealizowanych zamówień (różnice między wartościami kolumn [ShipDate] i [OrderDate] tabeli [SalesLT].

- [SalesOrderHeader]).
3. Wykorzystaj wiadomości z tego rozdziału, w tym opisane funkcje systemowe, do napisania zapytania, którego skrócony wynik został zamieszczony poniżej:

(No column name) name)	(No column name)	(No column name)
Produkt Touring-1000 Blue, 54	kosztuje	2384,10
Produkt Touring-1000 Blue, 60	kosztuje	2384,10
Produkt Mountain-200 Silver, 38	kosztuje	2320,00
Produkt Mountain-200 Silver, 42	kosztuje	2320,00
Produkt Mountain-200 Silver, 46	kosztuje	2320,00
Produkt Mountain-200 Black, 38	kosztuje	2295,00
Produkt Mountain-200 Black, 42	kosztuje	2295,00

Podpowiedź: dane o produktach znajdują się w tabeli [SalesLT].[Product].

4. Odczytaj z tabeli [SalesLT].[SalesOrderHeader] ułożoną od najnowszych do najstarszych listę dat wszystkich zamówień. Wynik zapytania powinien mieć formę rok-miesiąc-dzień.

(No column name)
2014-4-1
2008-6-1

W rozwiązaniu użyj funkcji YEAR(), MONTH() i DAY().

5. Odczytaj z tabeli [SalesLT].[Product] kolumny [ProductNumber] oraz [Size] i posortuj wynik rosnąco według wartości kolumny [Size], ale w taki sposób, żeby wartości NULL znalazły się na końcu, a nie na początku wyniku.

Podpowiedź: użyj funkcji CASE.

[1] Serwer SQL Server umożliwia wykonanie instrukcji SELECT niezawierającej klauzuli FROM. Nie wszystkie serwery bazodanowe to umożliwiają, np. w serwerze Oracle konieczne jest dopisanie klauzuli FROM i odwołanie się w niej do pseudotabeli o nazwie dual.

[2] W serwerze SQL Server do połączenia się z wybraną bazą danych służy instrukcja USE.

[3] Domyślnie kolejność kolumn wyniku odpowiada kolejności kolumn odczytywanej tabeli.

[4] Każdy wiersz wyniku zapytania musi się składać z takiej samej liczby kolumn. Użycie w klauzuli SELECT słowa kluczowego DISTINCT w celu usunięcia powtórzenia tylko z wybranych kolumn danego wyniku jest więc niemożliwe.

[5] Operator modulo zwraca resztę z dzielenia, np. 777 modulo 10 = 7.

[6] Instrukcja IF ... THEN ... ELSE (jeżeli..., to..., w przeciwnym razie) pozwala sterować wykonaniem programu. Jeśli określony w klauzuli IF warunek jest prawdziwy, wykonywany jest blok instrukcji THEN, w przeciwnym razie wykonywane są instrukcje z bloku ELSE.

[7] Gdyby zawsze były sprawdzane wszystkie warunki, przykładowe zapytanie zwróciłoby niepoprawny wynik — niezależnie od liczby towarów ich słownym opisem zawsze byłoby Drogi.

[8] Wyjątkiem od tej reguły są funkcje zwracające unikatowe identyfikatory. W serwerze SQL 2011 taką funkcją jest NEWID().

[9] Z tego samego powodu możemy również sortować wyniki zapytań na podstawie wyrażeń, o ile tylko zwracają one wartości skalarne. Wymóg umieszczania jedynie wyrażeń zwracających wartości skalarne jest kolejną cechą wspólną klauzul ORDER BY i SELECT.

[10] Większość alfabetów zawiera specyficzne dla siebie znaki diakrytyczne, np. w języku polskim występują m.in. litery *ą*, *ć* oraz *ż*.

Rozdział 4. Wybieranie wierszy

- Dlaczego obowiązująca w języku SQL logika trójwartościowa różni się od klasycznej logiki dwuwartościowej?
- Czym jest selekcja tabeli?
- Za pomocą jakich operatorów można wybierać wiersze?
- Jak tworzyć złożone warunki logiczne?
- Dlaczego klauzule TOP i ORDER BY występują razem?
- Na czym polega stronicowanie wierszy?

Logika trójwartościowa

Zanim przejdziemy do omówienia klauzuli WHERE, przedstawimy obowiązującą w języku SQL logikę trójwartościową i trzy podstawowe operatory (spójniki) logiczne: NOT, AND i OR.

Podstawą używanej na co dzień klasycznej^[1] logiki są trzy intuicyjnie uznawane za prawdziwe zasady:

1. Zasada tożsamości, na mocy której $a=a$, czyli każda rzecz jest równa samej sobie.
2. Zasada sprzeczności, na mocy której $\sim(a \wedge \sim a)$, czyli z dwóch sprzecznych zdań (predykatów) jedno jest prawdziwe^[2].
3. Zasada wyłączonego środka, na mocy której $a \vee \sim a$, czyli zdanie (predykat) albo jest prawdziwe, albo fałszywe^[3].

W języku SQL żadna z tych zasad nie stosuje się do wartości NULL. Pozostałe wartości są przetwarzane i porównywane zgodnie z trzema zasadami logiki klasycznej.

Wartość NULL

Wartość NULL reprezentuje brakującą lub nieznaną wartość, a więc w rzeczywistości w ogóle nie jest wartością żadnego typu. Jednak żeby uniknąć nieporozumieńst terminologicznych, powszechnie stosowany jest zwrot *wartość NULL*.

Z rozdziału 2. książki wiesz, że wartość `NULL` jest w specyficzny sposób przetwarzana przez serwery bazodanowe:

1. Porównywanie wartości `NULL` z innymi wartościami, w tym z nią samą, daje w wyniku wartość nieznaną `UNKNOWN`, a nie prawdę lub fałsz. Jest to sprzeczne z wszystkimi trzema zasadami logiki klasycznej:
 - a. Wartość `NULL` nie jest równa samej sobie (wyjątkiem od tej reguły jest opisana w rozdziale 6. klazula `GROUP BY`).
 - b. Ani wartość `NULL`, ani jej negacja nie są prawdziwe.
 - c. Wartość `NULL` nie jest ani prawdziwa, ani fałszywa.
4. Wynikiem wyrażeń zawierających wartość `NULL` jest zawsze wartość `NULL`.

Operatory logiczne

Operatory logiczne można porównać do spójników zdań — możemy dzięki nim połączyć kilka prostych warunków logicznych w jeden złożony.

 Wskazówka	Klasycznym warunkiem logicznym jest wyrażenie prawdziwe lub fałszywe. W języku SQL warunek logiczny może mieć trzecią wartość — wartość <code>UNKNOWN</code> .
---	--

Rolę operatorów logicznych przedstawimy na kilku przykładach:

1. Operator `AND` (logiczne I, czyli koniunkcja) może być użyty do połączenia następujących warunków logicznych: `cena < 500 AND kolor = niebieski`. Otrzymany w ten sposób złożony warunek logiczny będzie prawdziwy tylko wtedy, gdy cena jest niższa niż 500, a kolor to niebieski. Oznacza to, że operator `AND` eliminuje kolejne wiersze z wyniku.
2. Operator `OR` (logiczne LUB, czyli alternatywa) może być użyty do połączenia następujących warunków logicznych: `kolor = niebieski LUB kolor = czerwony LUB kolor = żółty`. Otrzymany w ten sposób złożony warunek logiczny będzie prawdziwy, jeżeli kolor to niebieski, czerwony lub żółty.

Oznacza to, że operator OR dodaje wiersze do wyniku zapytania.

3. Operatory AND, OR i NOT (logiczne NIE, czyli negacja) mogą być użyte do połączenia kilku prostych warunków logicznych: `cena < 500 AND (kolor NOT czarny OR marka NOT Ford)`. Otrzymany w ten sposób złożony warunek logiczny będzie prawdziwy tylko wtedy, gdy cena jest niższa niż 500 i albo kolor nie jest czarny, albo marką nie jest Ford.

Możliwe wartości (TRUE, FALSE lub UNKNOWN) operatorów logicznych przedstawia się najczęściej w postaci tabel prawdziwości zawierających wszystkie możliwe kombinacje parametrów danego operatora.

Operator NOT

Operator NOT jest operatorem jednoargumentowym. W klasycznej logice jego wynikiem jest zaprzeczenie (negacja) argumentu. W języku SQL może on również zwrócić wartość UNKNOWN (tabela 4.1).

Tabela 4.1. Tabela prawdziwości operatora NOT

A	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	UNKNOWN

Operator OR

Operator OR jest operatorem dwuargumentowym. W klasycznej logice alternatywa jest nieprawdziwa tylko wtedy, gdy oba jej argumenty są nieprawdziwe, w pozostałych przypadkach wynikiem alternatywy logicznej jest prawda. Wszystkie możliwe w języku SQL kombinacje parametrów tego operatora zostały przedstawione w tabeli 4.2.

Tabela 4.2. Tabela prawdziwości operatora OR

A	b	a OR b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE

TRUE	NULL	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
FALSE	NULL	UNKNOWN
NULL	TRUE	TRUE
NULL	FALSE	UNKNOWN
NULL	NULL	UNKNOWN

Operator AND

Operator AND również jest operatorem dwuargumentowym. W klasycznej logice koniunkcja jest prawdziwa tylko wtedy, gdy oba argumenty są prawdziwe, w pozostałych przypadkach jej wynikiem jest fałsz. Wszystkie możliwe w języku SQL kombinacje parametrów tego operatora zostały przedstawione w tabeli 4.3.

Tabela 4.3. Tabela prawdziwości operatora AND

a	b	a AND b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	NULL	UNKNOWN
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
FALSE	NULL	FALSE
NULL	TRUE	UNKNOWN
NULL	FALSE	FALSE
NULL	NULL	UNKNOWN

Klauzula WHERE

Poznane w poprzednim rozdziale zapytania zwracały wszystkie wiersze tabeli. Jednak w praktyce takie zapytania są używane dość rzadko — najczęściej chcemy ograniczyć wynik zapytania do interesujących nas w danej chwili danych.

Wybieranie zwracanych przez zapytanie wierszy umożliwia klauzula WHERE. Umieszczone w niej warunki logiczne są sprawdzane dla każdego wiersza tabeli:

Jeżeli wynikiem testu logicznego jest prawda, wiersz trafia do wyniku zapytania.

W przeciwnym razie, czyli **jeżeli wynikiem testu logicznego jest fałsz lub wartość nieznana, wiersz jest usuwany z wyniku zapytania.**

 Wskazówka	Wybieranie niektórych wierszy tabeli nazywa się selekcją rekordów tabeli. Wynikiem selekcji jest tabela o mniejszej liczbie wierszy niż odczytywana tabela. Typowe zapytania łączą poznaną w poprzednim rozdziale projekcję (wybieranie kolumn) z selekcją (wybieraniem wierszy).
---	---

Opcjonalna klauzula WHERE musi wystąpić bezpośrednio po klauzuli FROM. Ponieważ poszczególne klauzule zapytania są niezależne, **wymienione w klauzuli WHERE kolumny nie muszą występować w klauzuli SELECT.** Tak jest w przypadku poniższego zapytania zwracającego nazwy produktów, których ceny nie przekraczają dziesięciu:

```
SELECT ProductNumber, ListPrice  
FROM [SalesLT].[Product]  
WHERE [ListPrice]<10;
```

ProductNumber	ListPrice
S0-B909-M.	9,50
S0-B909-L	9,50
CA-1098	8,99
WB-H098	4,99
BC-M005	9,99
BC-R205	8,99
PK-7098	2,29
S0-R809-M	8,99
S0-R809-L	8,99
CL-9009	7,95
TT-M928	4,99

TT-R982	3,99
TT-T092	4,99

Standardowe operatory porównania

Warunki logiczne buduje się za pomocą standardowych operatorów porównania, takich jak:

1. = (równy) — operator zwraca prawdę, jeżeli porównywane wartości są takie same.
2. <>, != (różny) — operator zwraca prawdę, jeżeli porównywane wartości są różne.
3. < (mniejszy niż) — operator zwraca prawdę, jeżeli pierwsza (umieszczona po jego lewej stronie) wartość jest mniejsza od drugiej (umieszczonej po prawej stronie operatora).
4. <= (mniejszy lub równy) — operator zwraca prawdę, jeżeli pierwsza wartość nie jest większa od drugiej.
5. > (większy niż) — operator zwraca prawdę, jeżeli pierwsza wartość jest większa od drugiej.
6. >= (większy lub równy) — operator zwraca prawdę, jeżeli pierwsza wartość nie jest mniejsza od drugiej.

Na przykład żeby odczytać nazwy i ceny produktów droższych niż 3500, należy w klauzuli WHERE umieścić warunek [ListPrice]>3500:

```
SELECT ProductNumber, ListPrice
FROM [SalesLT].[Product]
WHERE [ListPrice]>3500;
```

ProductNumber	ListPrice
BK-R93R-62	3578,27
BK-R93R-44	3578,27
BK-R93R-48	3578,27
BK-R93R-52	3578,27
BK-R93R-56	3578,27

Kolejny przykład pokazuje, jak odczytać posortowaną od najnowszych do najstarszych listę produktów, które trafiły do

sprzedaży po 1 czerwca 2007 roku:

```
SELECT [Name], [SellStartDate]
FROM [SalesLT].[Product]
WHERE [SellStartDate]>'20070601'
ORDER BY [SellStartDate] DESC;
```

```
-----
Name           SellStartDate
Classic Vest, S   2007-07-01 00:00:00.000
Classic Vest, M   2007-07-01 00:00:00.000
Classic Vest, L   2007-07-01 00:00:00.000
Women's Mountain Shorts, S 2007-07-01 00:00:00.000
Women's Mountain Shorts, M 2007-07-01 00:00:00.000
```

...

Język SQL pozwala też na użycie w klauzuli WHERE wyrażeń i funkcji systemowych, o ile tylko zwracają one wartości skalarne. Możemy więc w prosty sposób sprawdzić np., które produkty trafiły do sprzedaży w wybranym roku:

```
SELECT [Name], [SellStartDate]
FROM [SalesLT].[Product]
WHERE YEAR([SellStartDate]) = 2007
ORDER BY [SellStartDate] DESC;
```

```
-----
Name           SellStartDate
Classic Vest, S   2007-07-01 00:00:00.000
Classic Vest, M   2007-07-01 00:00:00.000
Classic Vest, L   2007-07-01 00:00:00.000
Women's Mountain Shorts, S 2007-07-01 00:00:00.000
Women's Mountain Shorts, M 2007-07-01 00:00:00.000
Women's Mountain Shorts, L 2007-07-01 00:00:00.000
```

...

Nazwy kolumn mogą być użyte w wyrażeniach znajdujących się po obu stronach operatorów porównania. W takim przypadku porównywane ze sobą będą wartości różnych pól tego samego wiersza. Kolejny przykład pokazuje, jak znaleźć towary,

których cena jest co najmniej dwukrotnie wyższa niż koszt ich uzyskania:

```
SELECT [ProductNumber], [StandardCost], [ListPrice]
FROM [SalesLT].[Product]
WHERE [StandardCost]*2<[ListPrice];
```

```
-----
ProductNumber      StandardCost      ListPrice
S0-B909-L          3,3963           9,50
HL-U509-B          13,0863          34,99
FK-1639             65,8097          148,22
FK-5136             77,9176          175,49
FK-9939             101,8936         229,49
```

...

Operatory SQL

Oprócz standardowych operatorów porównania w klauzuli WHERE możemy użyć specyficznych dla języka SQL operatorów IN, BETWEEN ... AND, LIKE oraz IS NULL.

Operator IN

Operator IN pozwala sprawdzić, czy szukana wartość należy do podanego zbioru, np. czy wielkość produktu wynosi 44, 48 lub 50:

```
SELECT [ProductNumber], [Size]
FROM [SalesLT].[Product]
WHERE [Size] IN ('44','48','50');
```

```
-----
ProductNumber      Size
FR-M21S-44        44
FR-M21S-48        48
FR-M21B-44        44
FR-M21B-48        48
BK-T79Y-50        50
```

...

Operator IN może być używany z każdym typem danych, nie tylko z

danymi liczbowymi (w sumie w poprzednim zapytaniu argumentami operatora IN również były dane tekstowe, a nie liczby). Kolejne zapytanie zwraca numery czarnych lub czerwonych produktów:

```
SELECT [ProductNumber], [Color]
FROM [SalesLT].[Product]
WHERE [Color] IN ('Black', 'Red');
```

```
-----
ProductNumber          Color
FR-R92B-58            Black
FR-R92R-58            Red
HL-U509-R             Red
HL-U509               Black
```

...

 Wskazówka	Niektóre serwery bazodanowe podczas tworzenia planu wykonania zapytania automatycznie zastępują wieloargumentowy operator IN równoważną alternatywą, czyli zamiast operatora IN w planie wykonania zapytania znajdziemy listę operatorów OR.
--	--

Operator BETWEEN ... AND

Operator BETWEEN ... AND pozwala sprawdzić, czy szukana wartość należy do określonego przedziału domkniętego. Dzięki niemu możemy np. poznać nazwy towarów o cenach nie niższych niż 10 i nie wyższych niż 20:

```
SELECT [ProductNumber], [ListPrice]
FROM [SalesLT].[Product]
WHERE [ListPrice] BETWEEN 10 AND 20;
```

```
-----
ProductNumber          ListPrice
PU-0452                19,99
LT-T990                 13,99
```

Z powodu okresowej natury znaczników daty i czasu (jeden dzień to 24 godziny, jedna godzina to 60 minut i tak dalej) operator BETWEEN ... AND jest powszechnie stosowany do sprawdzania danych typu data i czas. Następny przykład pokazuje, jak odczytać identyfikatory

zamówień złożonych w ciągu pierwszego półrocza roku 2014:

```
SELECT [SalesOrderID], [OrderDate]
FROM [SalesLT].[SalesOrderHeader]
WHERE [OrderDate] BETWEEN '20140101' AND '20140630';
```

```
-----  
SalesOrderID          OrderDate  
77774                2014-04-01 00:00:00.000
```

 Wskazówka	W książce używamy formatu znaczników daty, w którym rok, numer miesiąca i numer dnia nie są oddzielone myślnikami. Serwer SQL zawsze interpretuje tak zapisane daty w ten sam sposób, niezależnie od ustawień regionalnych serwera i sesji użytkownika.
---	---

Operator BETWEEN ... AND może być też użyty do sprawdzenia, czy dane tekstowe należą do podanego zakresu. W takim przypadku stosowany jest domyślny, określony na poziomie kolumny, tabeli, bazy lub całego serwera, porządek sortowania. Poniższe zapytanie zwraca nazwiska klientów zaczynające się na litery od B do D:

```
SELECT [LastName]
FROM [SalesLT].[Customer]
WHERE LEFT([LastName],1) BETWEEN 'B' AND 'D';
```

```
-----  
LastName  
Caprio  
Beck  
Beaver  
Burnett  
Blanton  
Blackwell  
Deborde  
...  
...
```

 Wskazówka	Niektóre serwery bazodanowe podczas tworzenia planu wykonania zapytania automatycznie zastępują wieloargumentowy operator BETWEEN ... AND równoważną koniunkcją operatorów => i <= , czyli zamiast operatora BETWEEN ... AND w planie wykonania zapytania znajdziemy listę operatorów AND.
---	--

Operator LIKE

Z pomocą operatora `LIKE` możemy przeszukiwać dane tekstowe pod kątem ich zgodności z podanym wzorcem. Do tworzenia wzorca można użyć dwóch symboli o specjalnym znaczeniu:

1. Symbol `%` (procent) zastępuje dowolny ciąg znaków.
2. Symbol `_` (podkreślenie) zastępuje jeden dowolny znak.

Dzięki nim możemy np. odczytać dane produktów, których nazwy zaczynają się na literę `S`:

```
SELECT [ProductID], [Name]  
FROM [SalesLT].[Product]  
WHERE [Name] LIKE 'S%';  
-----
```

ProductID	Name
883	Short-Sleeve Classic Jersey, L
882	Short-Sleeve Classic Jersey, M
881	Short-Sleeve Classic Jersey, S
884	Short-Sleeve Classic Jersey, XL
708	Sport-100 Helmet, Black
711	Sport-100 Helmet, Blue
707	Sport-100 Helmet, Red

Możliwe jest też ograniczenie wyniku do tych produktów, których nazwy zaczynają się na literę `S`, a kończą literą `L`:

```
SELECT [ProductID], [Name]  
FROM [SalesLT].[Product]  
WHERE [Name] LIKE 'S%L';  
-----
```

ProductID	Name
883	Short-Sleeve Classic Jersey, L
884	Short-Sleeve Classic Jersey, XL

Operator `LIKE` pozwoli nam też odczytać dane produktów, których numer zaczyna się literą `S`, druga litera jest dowolna, a następnie występuje w nich myślnik i litera `M`:

```
SELECT [ProductID], [ProductNumber]
FROM [SalesLT].[Product]
WHERE [ProductNumber] LIKE 'S_-M%';
```

```
-----  
ProductID          ProductNumber  
857                SB-M891-L  
856                SB-M891-M  
855                SB-M891-S  
908                SE-M236  
909                SE-M798  
910                SE-M940  
850                SH-M897-L  
849                SH-M897-M  
841                SH-M897-S  
851                SH-M897-X
```



Wskazówka

Apostrof oznacza ciąg znaków, więc jeżeli chcemy wyszukać dane tekstowe zawierające apostrof, musimy poprzedzić go dodatkowym apostrofem.

Operator IS NULL

Chociaż nie możemy porównywać wartości `NULL` z innymi wartościami, to możemy sprawdzić, czy dana wartość jest nieokreślona. Służy do tego operator `IS NULL`. Poniższe zapytanie zwraca kody produktów, które nie mają określonej daty wycofania ze sprzedaży:

```
SELECT [ProductNumber]
FROM [SalesLT].[Product]
WHERE [SellEndDate] IS NULL;
```

```
-----  
ProductNumber  
BB-7421  
BB-8107  
BB-9108  
BC-M005
```

BC-R205

BK-M18B-40

...

Złożone warunki logiczne

Wszystkie przedstawione operatory porównań możemy łączyć ze sobą za pomocą trzech operatorów logicznych: NOT, AND i OR. Utworzone w ten sposób złożone warunki logiczne pozwalają na zaawansowane wyszukiwanie danych.

Na przykład żeby odczytać nazwy będących ciągle w sprzedaży czarnych produktów, które należą do kategorii o identyfikatorze 5, 16 lub 19, oraz produktów kosztujących nie więcej niż 50, należy wykonać poniższe zapytanie:

```
SELECT [Name]
FROM [SalesLT].[Product]
WHERE [ProductCategoryID] IN (5,16,19)
AND [SellEndDate] IS NOT NULL
AND [Color] = 'Black'
OR [ListPrice]<50;
```

Name

Sport-100 Helmet, Red
Sport-100 Helmet, Black
Mountain Bike Socks, M
Mountain Bike Socks, L
Sport-100 Helmet, Blue

W przypadku użycia operatora AND każdy kolejny warunek logiczny zmniejsza liczbę wierszy wyniku zapytania. Możemy się o tym przekonać, komentując poszczególne wiersze^[4] klauzuli WHERE — jeżeli będziemy je komentować od ostatniego, zapytanie będzie zwracać kolejno 69, 20 i wreszcie 14 wierszy.

Natomiast operator OR dodaje wiersze do wyniku zapytania. Jeśli początkowo nasze zapytanie zwróciło nazwy 14 produktów, to po dodaniu warunku OR ich liczba zwiększyła się do 67.

Hierarchia operatorów

Końcowy wynik złożonych warunków logicznych zależy od kolejności, w jakiej serwery bazodanowe sprawdzają prawdziwość składających się na nie prostych warunków. Ta kolejność wynika natomiast z przyjętej w standardzie języka SQL hierarchii operatorów^[5]:

1. Jako pierwsze wykonywane są operacje mnożenia i dzielenia.
2. Następnie — dodawania i odejmowania.
3. W trzeciej kolejności wykonywane są standardowe operacje porównania, takie jak równy lub mniejszy niż.
4. Pierwszym wykonywanym operatorem logicznym jest operator NOT.
5. Następnie wykonywany jest operator AND.
6. Jako ostatnie — operatory SQL (IN, BETWEEN ... AND, LIKE) oraz operator OR.

Jeżeli wyrażenie zawiera kilka operatorów o tym samym priorytecie, są one wykonywane od lewej do prawej.

Sprawdźmy, czy na podstawie tych reguł potrafisz powiedzieć, jaki będzie wynik poniższego zapytania.

```
SELECT *
FROM [SalesLT].[Customer]
WHERE NOT [CompanyName] ='A Bike Store'
AND [CustomerID]>100
ORDER BY [CompanyName], [CustomerID];;
```

Ponieważ operator NOT ma wyższy priorytet niż AND, w wyniku znajdziemy tylko informacje o klientach, którzy nie pracują dla firmy A Bike Store, **o identyfikatorach wyższych od 100**.

Kolejność wykonywania poszczególnych operacji możemy zmienić za pomocą nawiasów. Umieszczone w nawiasach operatory są wykonywane jako pierwsze i tylko wynik ich wykonania jest przekazywany do znajdujących się poza nawiasami operatorów. Tak więc poniższe zapytanie:

```
SELECT *
```

```

FROM [SalesLT].[Customer]
WHERE NOT ([CompanyName] ='A Bike Store'
AND [CustomerID]>100)
ORDER BY [CompanyName],[CustomerID];

```

zwróci dane **wszystkich klientów z wyjątkiem pracowników firmy A Bike Store**, których identyfikatory mają wartość powyżej 100.

Przeanalizujmy jeszcze jeden, bardziej skomplikowany przykład z zagnieżdzonymi nawiasami^[6]:

```

SELECT [ProductCategoryID], [ListPrice], [StandardCost]
FROM [SalesLT].[Product]
WHERE NOT (([StandardCost] > 20 AND [ProductCategoryID] =1)
OR [ListPrice] BETWEEN 10 AND 20);
-----
```

ProductCategoryID	ListPrice	StandardCost
18	1431,50	1059,31
18	1431,50	1059,31
35	34,99	13,0863
35	34,99	13,0863
27	9,50	3,3963
27	9,50	3,3963
35	34,99	13,0863
23	8,99	6,9223
25	49,99	38,4923
25	49,99	38,4923
25	49,99	38,4923
25	49,99	38,4923

...

1. W pierwszej kolejności sprawdzany jest najbardziej zagnieżdzony warunek W1: `[StandardCost] > 20 AND [ProductCategoryID] =1`. Jest on prawdziwy dla produktów z pierwszej kategorii, których koszt standardowy przekracza 20.
2. Następnie sprawdzany jest warunek W2: `[ListPrice]`

BETWEEN 10 AND 200. Ten warunek spełniają produkty o cenie nie mniejszej niż 10 i nie większej niż 20.

3. Gdybyśmy w tym momencie mogli przerwać wykonywanie zapytania, jego wynik zawierałby wiersze spełniające pierwszy lub drugi warunek (W3 jest prawdziwy, jeżeli W1 lub W2 był prawdziwy).
4. Ponieważ cały warunek logiczny jest poprzedzony operatorem NOT, do wyniku zapytania trafią wiersze niespełniające poprzedniego warunku (W4 jest prawdziwy, gdy W3 był fałszywy).
5. Doszliśmy do momentu, w którym musimy sprawdzić, kiedy W3 będzie fałszywy lub nieznany. Skorzystajmy w tym celu z tabeli 4.2 (tabeli prawdziwości operatora OR). Okazuje się, że W3 będzie fałszywy, gdy oba warunki W1 i W2 będą fałszywe, a nieznany, gdy oba warunki będą nieznane oraz gdy jeden z nich będzie nieznany, a drugi fałszywy.
6. Na podstawie tabeli prawdziwości operatora AND (tabela 4.3) możemy stwierdzić, że W1 będzie fałszywy, jeżeli cena będzie równa bądź mniejsza niż 20 lub jeśli produkt nie będzie należał do pierwszej kategorii. Warunek będzie nieznany, gdy oba proste warunki będą nieznane oraz gdy jeden z nich będzie nieznany, a drugi prawdziwy.
7. W rezultacie warunek W2 będzie fałszywy dla produktów o cenie mniejszej niż 10 lub większej niż 20.
8. **Wynik zapytania będzie więc zawierał dane produktów o cenie mniejszej niż 10 lub większej niż 20 oraz produktów nienależących do pierwszej kategorii, których koszt standardowy jest równy lub mniejszy niż 20.**

Klauzula TOP

Opcjonalna klauzula TOP pozwala ograniczyć wynik zapytania do podanej w niej liczby wierszy. Docelowa liczba wierszy może być określona bezwzględnie lub procentowo. Jeżeli występuje w

zapytaniu, klauzula `TOP` musi się znaleźć bezpośrednio po instrukcji `SELECT`, a przed nazwami zwracanych przez zapytanie kolumn.

 Wskazówka	Klauzula <code>TOP</code> ma inne znaczenie niż <code>WHERE</code> — nie formułuje się w niej warunku logicznego, który może być spełniony przez nieznaną z góry liczbę wierszy, tylko sztucznie zmniejsza się liczbę wierszy wyniku do z góry znanej liczby. Obie klauzule łączy tylko to, że pozwalają na ograniczenie liczby wierszy wyniku zapytania.
--	---

Przyjrzyjmy się kolejnemu przykładowi: potrafisz powiedzieć, jaki będzie wynik tego zapytania?

```
SELECT TOP 1 [Name], [ListPrice]  
FROM [SalesLT].[Product];
```

Na pewno zwróci ono jeden wiersz z nazwą i ceną produktu. Ale który to będzie wiersz odczytywanej tabeli? My też tego nie wiemy — zależy to od kolejności, w jakiej serwer bazodanowy, wykonując zapytanie, odczyta wiersze tabeli `[SalesLT].[Product]`.

Kolejność wierszy wyniku zapytania możemy określić tylko za pomocą klauzuli `ORDER BY`; bez niej klauzula `TOP` jest niedeterministyczna i praktycznie bezużyteczna. Jeżeli poprzednie zapytanie miało zwrócić nazwę i cenę najdroższego towaru, powinniśmy posortować rosnąco jego wynik (w ten sposób pierwszy wiersz będzie zawierał dane najdroższego produktu):

```
SELECT TOP 1 [Name], [ListPrice]  
FROM [SalesLT].[Product]  
ORDER BY [ListPrice] DESC;
```

Name	ListPrice
Road-150 Red, 62	3578,27

Ale czy ten wynik na pewno jest poprawny? Tak, jeżeli tylko jeden towar kosztuje 3578,27. Jeśli towarów w tej cenie jest więcej, wynik nadal jest niedeterministyczny. Rozwiązać ten problem możemy na dwa sposoby:

1. Dodając do sortowania kolumnę, w której nie występują duplikaty. Taką kolumnę dodaje się jako ostatnią w klauzuli `ORDER BY`.

2. Dodając do wyniku wszystkie powtórzone wiersze, w tym przypadku dane wszystkich towarów o cenie 3578,27.

Pierwsze rozwiązanie powinno być stosowane, jeżeli liczba zwracanych wierszy musi być z góry znana. Na przykład jeśli klienta interesują dane dokładnie trzech najdroższych towarów, powinniśmy z nim ustalić, co oprócz ceny będzie kryterium wyboru wierszy, i zmodyfikować odpowiednio klauzulę ORDER BY:

```
SELECT TOP 3 [Name], [ListPrice]
FROM [SalesLT].[Product]
ORDER BY [ListPrice] DESC, [ProductID];
```

```
-----  
Name          ListPrice  
Road-150 Red, 62    3578,27  
Road-150 Red, 44    3578,27  
Road-150 Red, 48    3578,27
```

Natomiast aby dodać do wyniku dodatkowe, zawierające powtórzenia wiersze, należy użyć rozszerzonej składni TOP ... WITH TIES (rozszerzona składnia klauzuli TOP wymaga użycia klauzuli ORDER BY, bez niej serwer bazodanowy zgłosi błąd składni):

```
SELECT TOP 3 WITH TIES [Name], [ListPrice]
FROM [SalesLT].[Product]
ORDER BY [ListPrice] DESC;
```

```
-----  
NWTCFV-92    1,20  
NWTCFV-93    1,20
```

Możemy również określić, jaki procent wierszy odczytywanej tabeli ma zwrócić zapytanie. Kolejny przykład pokazuje, jak odczytać dane o 5% najdroższych produktów:

```
SELECT TOP 5 PERCENT WITH TIES [Name], [ListPrice]
FROM [SalesLT].[Product]
ORDER BY [ListPrice] DESC;
```

```
-----  
Name          ListPrice
```

Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99
Mountain-100 Silver, 44	3399,99
Mountain-100 Silver, 48	3399,99
Mountain-100 Black, 38	3374,99
Mountain-100 Black, 42	3374,99
Mountain-100 Black, 44	3374,99
Mountain-100 Black, 48	3374,99
Road-250 Red, 44	2443,35

...



Argumentem klauzuli TOP mogą być nie tylko stałe, lecz także zmienne i wyrażenia. Wyrażeniem może być dowolne, zwracające wartość skalarną podzapytanie — podzapytania zostały opisane w rozdziale 8.

Stronicowanie wierszy

Możliwość wybierania wierszy na podstawie ich kolejności pozwala również je stronicować, czyli ograniczać wynik zapytania do wierszy o określonych numerach. W serwerze SQL klauzule OFFSET (w której określa się liczbę wcześniejszych, przeznaczonych do pominięcia wierszy) oraz FETCH NEXT (w której określa się liczbę zwracanych wierszy) umieszcza się (co jest zgodne ze standardem ANSI języka SQL) za klauzulą ORDER BY.

Przypuśćmy, że chcemy podzielić na strony listę produktów, używając do stronicowania identyfikatora modelu (w ten sposób te same modele będą wyświetlane na tych samych bądź sąsiednich stronach). Punktem wyjścia jest poniższe zapytanie:

```
SELECT [Name], [ProductModelID]
FROM [SalesLT].[Product]
```

```
ORDER BY [ProductModelID];
```

```
-----  
Name          ProductModelID  
Classic Vest, S      1  
Classic Vest, M      1  
Classic Vest, L      1  
AWC Logo Cap        2  
Full-Finger Gloves, S 3  
Full-Finger Gloves, M 3  
Full-Finger Gloves, L 3  
Half-Finger Gloves, S 4  
Half-Finger Gloves, M 4  
Half-Finger Gloves, L 4  
HL Mountain Frame - Silver, 42 5
```

```
...
```

Pominąć określoną liczbę wierszy możemy, dodając na końcu zapytania klauzulę `OFFSET` i podając w niej liczbę początkowych wierszy, które mają być usunięte z wyniku zapytania:

```
SELECT [Name], [ProductModelID]  
FROM [SalesLT].[Product]  
ORDER BY [ProductModelID]  
OFFSET 5 ROWS;
```

```
-----  
Name          ProductModelID  
Full-Finger Gloves, M      3  
Full-Finger Gloves, L      3  
Half-Finger Gloves, S      4  
Half-Finger Gloves, M      4  
Half-Finger Gloves, L      4  
HL Mountain Frame - Silver, 42 5
```

```
...
```

Natomiast ograniczyć liczbę zwracanych wierszy możemy za pomocą klauzuli `FETCH NEXT`:

```

SELECT [Name], [ProductModelID]
FROM [SalesLT].[Product]
ORDER BY [ProductModelID]
OFFSET 3 ROWS
FETCH NEXT 5 ROWS ONLY;
-----
```

Name	ProductModelID
AWC Logo Cap	2
Full-Finger Gloves, S	3
Full-Finger Gloves, M	3
Full-Finger Gloves, L	3
Half-Finger Gloves, S	4

Technika ta nazywa się stronicowaniem, bo pozwala zwracać jedynie podzbiór wierszy wyniku, który ma być wyświetlony w danym momencie przez program kliencki. W kolejnym przykładzie do zilustrowania tej techniki zostały użyte dwie zmienne — @Strona reprezentująca wielkość (liczbę wierszy) kolejnych stron oraz @Bieżąca zawierająca numer aktualnie zwracanej przez serwer bazodanowy strony:

```

DECLARE @Strona tinyint = 5
      , @Bieżąca tinyint = 2;
SELECT [Name], [ProductModelID]
FROM [SalesLT].[Product]
ORDER BY [ProductModelID]
OFFSET (@Strona * (@Bieżąca - 1)) ROWS
FETCH NEXT @Strona ROWS ONLY;
-----
```

Name	ProductModelID
Full-Finger Gloves, M	3
Full-Finger Gloves, L	3
Half-Finger Gloves, S	4
Half-Finger Gloves, M	4
Half-Finger Gloves, L	4

Podsumowanie

- W języku SQL obowiązuje logika trójwartościowa.
- Porównywanie wartości NULL z innymi wartościami daje w wyniku wartość nieznaną. Wyjątkiem od tej reguły jest sprawdzanie, czy jakaś wartość jest nieznana, za pomocą operatora IS NULL.
- W klauzuli WHERE umieszcza się sprawdzany dla każdego wiersza odczytywanej tabeli warunek logiczny.
- Na podstawie tego testu wiersz tabeli trafia (lub nie) do wyniku zapytania:
 - Jeżeli wynikiem testu jest prawda, wiersz jest dodawany do wyniku.
 - Jeżeli wynikiem testu jest fałsz lub wartość nieznana, wiersz jest z niego usuwany.
- Operatory SQL IN oraz BETWEEN ... AND pozwalają uprościć warunki logiczne i poprawiają czytelność zapytań, nie wpływając w żaden sposób na ich wydajność.
- Chociaż zapytanie może zawierać tylko jedną klauzulę WHERE, w której wolno umieścić tylko jeden warunek logiczny, to może on być złożeniem wielu warunków.
- Klauzula TOP pozwala ograniczyć liczbę zwracanych przez zapytania wierszy.
- Do stronicowania wierszy służą klauzule OFFSET oraz FETCH NEXT.

Zadania

1. Odczytaj z tabeli [SalesLT].[Product] nazwy produktów, których koszt jest ponad dwukrotnie niższy od ceny, a ostatnim znakiem kodu towaru jest 4 lub 8.
2. Z tabeli [SalesLT].[SalesOrderHeader] odczytaj numery (kolumna [SalesOrderID]) i wysokości opłat (kolumna [Freight]) 5% zamówień o najniższym koszcie wysyłki złożonych (kolumna OrderDate) w drugim półroczu 2008 roku.
3. Napisz zapytanie zwracające za każdym razem nazwę innego, losowo wybranego produktu.

-
- [1] Klasycznej, bo opracowanej przez starożytnych Greków, głównie przez Arystotelesa.
 - [2] W klasycznej logice symbol \sim oznacza negację (logiczne NIE), a symbol \wedge koniunkcję (logiczne I).
 - [3] W klasycznej logice symbol \vee oznacza alternatywę (logiczne LUB).
 - [4] Znakiem komentarza są dwa myślniki. Poprzedzenie wiersza zaczynającego się operatorem AND dwoma myślnikami spowoduje wyłączenie danego warunku.
 - [5] Ta sama hierarchia obowiązuje w matematyce i większości innych niż SQL języków programowania.
 - [6] Ten przykład ma zaprezentować różne zastosowania operatorów. W praktyce nie używa się operatora NOT do zmiany działania operatorów OR na AND i odwrotnie.

Rozdział 5. Łączenie tabel i wyników zapytań

- Jak za pomocą jednego zapytania odczytać dane z wielu tabel?
- Czym się różni złączenie naturalne od nienaturalnego?
- Dlaczego łącząc tabele, powinniśmy używać aliasów ich nazw?
- Czym jest złączenie zewnętrzne?
- Kiedy kolejność łączenia tabel ma wpływ na wynik zapytania?
- Jak można złączyć tabelę z nią samą?
- Czym się różni łączenie wyników zapytań od łączenia odczytywanych tabel?
- Jak dla każdego zwróconego przez zapytanie wiersza wywołać funkcję tabelaryczną?

Złączenia naturalne i nienaturalne

Opisane w poprzednich rozdziałach zapytania odczytywały dane tylko z jednej tabeli. Tymczasem w relacyjnych bazach danych, takich jak baza *AdventureWorksLT2012*, większość zapytań odwołuje się do wielu powiązanych ze sobą tabel.

 Wskazówka	Możliwość łączenia odczytywanych tabel, wyników zapytań i wyników wyrażeń tabelarycznych (takich jak podzapytania, widoki oraz funkcje) jest jednym z największych atutów języka SQL. Pozwala ona nie tylko odtwarzać powiązane ze sobą dane, które w trakcie normalizacji zostały podzielone między różne tabele, lecz także w dowolny sposób je łączyć.
---	---

Na przykład odczytując tabelę [SalesLT].[Product], jesteśmy w stanie poznać identyfikatory kategorii, do których należą poszczególne towary. Nie możemy jednak poznać nazw tych kategorii, ponieważ są zapisane w tabeli [SalesLT].[ProductCategory]:

```
SELECT [Name], [ProductCategoryID]
FROM [SalesLT].[Product];
```

Name	ProductCategoryID
HL Road Frame - Black, 58	18
HL Road Frame - Red, 58	18

Sport-100 Helmet, Red	35
Sport-100 Helmet, Black	35
Mountain Bike Socks, M	27
Mountain Bike Socks, L	27
Sport-100 Helmet, Blue	35
AWC Logo Cap	23

...

```
SELECT [ProductCategoryID], [Name]
FROM [SalesLT].[ProductCategory];
```

ProductCategoryID	Name
4	Accessories
22	Bib-Shorts
30	Bike Racks
31	Bike Stands
1	Bikes
32	Bottles and Cages
9	Bottom Brackets

...

Klucze obce

W rozdziale 1. wyjaśniliśmy, że kolumna kluczy obcych przechowuje duplikaty identyfikatorów wierszy (kluczy podstawowych) innej tabeli, dzięki czemu możemy powiązać ze sobą przechowywane w tych tabelach dane. Ponieważ wartości klucza podstawowego muszą być niepowtarzalne, ale w kolumnie klucza obcego każda z nich może być powtórzona dowolną liczbę razy, klucze obce umożliwiają powiązanie tabel związkiem typu „jeden do wielu”.

W ten sposób powiązane są ze sobą m.in. tabele [SalesLT].[ProductCategory] i [SalesLT].[Product] — do jednej kategorii może należeć wiele produktów, ale ten sam produkt nie może należeć do różnych kategorii. Ponieważ identyfikatory kategorii są zapisane w obu tych tabelach (w tabeli [SalesLT].[ProductCategory] w kolumnie klucza podstawowego, a w tabeli [SalesLT].[Product] w kolumnie klucza obcego), możemy ich użyć do złączenia obu tabel i odczytania

numerów zamówień zrealizowanych przez firmę wysyłkową:

```
SELECT C.Name, P.Name  
FROM [SalesLT].[ProductCategory] AS C  
JOIN [SalesLT].[Product] AS P  
ON P.ProductCategoryID=C.ProductCategoryID;
```

Name	Name
Mountain Bikes	Mountain-500 Black, 44
Mountain Bikes	Mountain-500 Black, 48
Mountain Bikes	Mountain-500 Black, 52
Road Bikes	Road-350-W Yellow, 40
Road Bikes	Road-350-W Yellow, 42

...

Przeanalizujmy to zapytanie:

1. W klauzuli **SELECT** nazwy kolumn zostały poprzedzone aliasami nazw tabel. W innym wypadku próba wykonania zapytania mogłaby się skończyć błędem — wyjaśnienie tego problemu znajduje się w następnym punkcie, poświęconym aliasom.
2. W klauzuli **FROM** pojawił się nowy operator **JOIN ... ON**. Pozwala on na:
 - a. wymienienie nazw wszystkich odczytywanych tabel;
 - b. określenie warunków ich łączenia. W tym przypadku do wyniku zapytania trafią tylko te wiersze z obu tabel, w których wartości kolumny **ProductCategoryID** są takie same.



Wskazówka

Działanie operatora **JOIN ... ON** polega na wygenerowaniu wszystkich możliwych kombinacji wierszy łączonych tabel, a następnie usunięciu z tak otrzymanego zbioru pośredniego wierszy niespełniających warunku łączenia.

Chociaż można operator **JOIN ... ON** zastąpić warunkiem **WHERE**:

```
SELECT C.Name, P.Name
```

```
FROM [SalesLT].[ProductCategory] AS C, [SalesLT].[Product] AS P  
WHERE P.ProductCategoryID=C.ProductCategoryID;
```

Name	Name
Mountain Bikes	Mountain-500 Black, 44
Mountain Bikes	Mountain-500 Black, 48
Mountain Bikes	Mountain-500 Black, 52
Road Bikes	Road-350-W Yellow, 40
Road Bikes	Road-350-W Yellow, 42

...

to rozwiązanie takie ma jednak kilka wad:

1. Jest niezgodne ze standardem języka SQL.
2. Pogarsza czytelność zapytań, szczególnie tych z rozbudowaną klauzulą WHERE.
3. W złączeniach zewnętrznych może być przyczyną trudnych do wykrycia błędów logicznych.

 Wskazówka	Klauzula WHERE jest wykonywana po klauzuli FROM, a więc możemy w niej wyeliminować dowolne, zwracane przez klauzulę FROM wiersze, ale operacja odwrotna jest niemożliwa. Z tego powodu warunki złączenia i wyboru nie powinny być stosowane zamiennie.
---	--

Złączenie tabel na podstawie wartości kolumn klucza podstawowego i klucza obcego nazywa się złączeniem naturalnym, ponieważ jego wynik odzwierciedla naturalny związek pomiędzy danymi zapisanymi w różnych tabelach.

Z reguły chcemy, żeby wynik zapytania reprezentował naturalny związek między danymi, dlatego złączenie naturalne jest najczęściej stosowanym typem złączenia. I tak aby poznać dane klienta (zapisane w tabeli [SalesLT].[Customer]), który złożył konkretne zamówienie, wraz z danymi na temat tego zamówienia (zapisanymi w tabeli [SalesLT].[SalesOrderHeader]), musimy złączyć obie tabele na podstawie wartości par kluczy podstawowy - obcy, czyli kolumn CustomerID:

```
SELECT *
```

```
FROM [SalesLT].[Customer] AS C
JOIN [SalesLT].[SalesOrderHeader] AS OH
ON c.CustomerID=oh.CustomerID
WHERE [SalesOrderID]=71796;
```

 Wskazówka	Serwer SQL nie pozwala naturalnie łączyć tabel za pomocą standardowego operatora SQL NATURAL JOIN. Jego odpowiednikiem jest operator JOIN ... ON, o ile w klauzuli ON zostaną wskazane kolumny klucza podstawowego i obcego ^[1] . Serwer SQL nie obsługuje też skróconego zapisu JOIN ... USING (kolumna). Zamiast raz podać nazwę kolumny występującej w obu łączonych tabelach, musimy w klauzuli ON dwukrotnie ją powtórzyć.
---	--

Język SQL pozwala też na złączenie tabel na podstawie wartości niekluczowych kolumn. **Ponieważ wynik takiego złączenia nie odzwierciedla naturalnego połączenia danych, nazywa się je złączeniem nienaturalnym.**

Im bardziej znormalizowana baza danych, tym mniej zawiera duplikatów danych i tym rzadziej używane są w niej złączenia nienaturalne. Na przykład w znormalizowanej bazie danych *AdventureWorks* adresy są zapisane w tabeli [SalesLT].[CustomerAddress]. Jako że każdy klient może mieć kilka adresów, a pod tym samym adresem mogą mieszkać różni klienci, tabele te są powiązane relacją typu „wiele do wielu” i nie mają wspólnej kolumny. W obu jednak znajduje się kolumna z datą ostatniej modyfikacji rekordu — chociaż poniższa instrukcja nie ma sensu w znaczeniu biznesowym, jest jednak poprawna i zostanie wykonana przez serwer bazodanowy:

```
SELECT *
FROM [SalesLT].[Customer] AS C
JOIN [SalesLT].[Address] AS A
ON A.ModifiedDate=C.ModifiedDate;
```

Analizując wynik tego zapytania, możemy się przekonać, dlaczego w relacyjnych bazach danych do złączenia tabel używa się kluczy. Przyczyną jest to, że złączenia nienaturalne, jako odwołujące się do kolumn, które mogą zawierać powtarzające się wartości, są niejednoznaczne. W skrajnym przypadku taki warunek złączenia nie wyeliminuje z wyniku zapytania żadnych wierszy. I tak powyższe złączenie liczącej 847 wierszy tabeli [SalesLT].[Customer] z

zawierającą 450 wierszy tabelą [SalesLT].[Address] zwróciło aż 36 864 rekordy — bez wątpienia wynik zapytania zawiera powtarzające się dane.

Aliases

Jeżeli zapytanie odwołuje się tylko do jednej tabeli, poprzedzanie nazw kolumn nazwą tej tabeli jest niepotrzebne. Wynika to z tego, że tabela nie może mieć kilku kolumn o tej samej nazwie, a więc umieszczane w klauzulach instrukcji SELECT nazwy są jednoznaczne.

W przypadku zapytań odwołujących się do wielu tabel sytuacja wygląda zupełnie inaczej. **Skoro kolumny o tej samej nazwie mogą występować w różnych tabelach, serwer bazodanowy nie jest w stanie tylko na podstawie ich nazw określić, do której z nich chcemy się odwołać:**

```
SELECT [ProductID]
FROM [SalesLT].[Product]
JOIN [SalesLT].[SalesOrderDetail]
    ON [SalesLT].[Product].ProductID = [SalesLT].[SalesOrderDetail].ProductID;
-----
Ambiguous column name 'ProductID'.
```

Nazwy tabel są jednak dość długie. Ich ciągłe powtarzanie we wszystkich klauzulach zapytań byłoby kłopotliwe i pogorszyłoby czytelność zapytania. Dlatego **odwołując się do więcej niż jednej tabeli, powinniśmy nadawać aliasy wszystkim tabelom i konsekwentnie używać ich w całym zapytaniu:**

```
SELECT P.[ProductID]
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
    ON [SalesLT].[Product].ProductID = [SalesLT].[SalesOrderDetail].ProductID;
-----
Msg 4104, Level 16, State 1, Line 45
```

The multi-part identifier "SalesLT.Product.ProductID" could not be bound.

```
Msg 4104, Level 16, State 1, Line 45
```

The multi-part identifier "SalesLT.SalesOrderDetail.ProductID" could not

be bound.

Jak widać, w klauzuli `ON` również nie możemy posłużyć się oryginalnymi nazwami tabel, dla których zdefiniowane zostały aliasy:

```
SELECT P.[ProductID]
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
    ON P.ProductID = OD.ProductID;
```

ProductID

707

707

707

708

708



Używając aliasów nazw tabel, unikniemy trudnych do wykrycia błędów związanych ze sposobem, w jaki dany serwer bazodanowy sprawdza nazwy obiektów. Gdybyśmy się pomyliły, wpisując nazwę kolumny, a w którejś ze złączonych tabel istniałaby kolumna o wprowadzonej przez nas nazwie, serwer nie zgłosiłby błędu, tylko odczytał dane z innej kolumny, niż chcieliśmy.

Złączenia równościowe i nierównościowe

Przekonaliśmy się już, że wyniki zapytań nie muszą dokładnie odpowiadać odczytywanym z tabel danym. Tak jak za pomocą umieszczonych w klauzuli `SELECT` wyrażeń możemy zmienić wartości zwracanych przez zapytania danych, tak za pomocą złączenia nienaturalnego możemy wygenerować zbiór „przypadkowo” powiązanych ze sobą wierszy.

Wyjaśnimy to przy okazji omawiania złączeń równościowych, które w warunku złączenia zawierają operator `=`. **Wyniki zapytań ze złączeniem równościowym zawierają te wiersze złączonych tabel, w których wartości użytych do złączenia kolumn są takie same.** Ponieważ wartości różnych kluczy podstawowych są z reguły takie same^[2], poniższe zapytanie zwróci wynik, ale będzie to wynik niepoprawny:

```
SELECT P.[ProductID], M.ProductModelID, P.ProductCategoryID
```

```

FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductModel] AS M
ON P.ProductCategoryID = M.ProductModelID;
-----
```

ProductID	ProductModelID	ProductCategoryID
680	18	18
706	18	18
707	35	35
708	35	35
709	27	27
710	27	27
711	35	35

...

Zapytanie zostało przygotowane tak, żebyśmy mogli zobaczyć, gdzie popełniliśmy błąd. Tabele powinny być złączone na podstawie pary kolumn klucz podstawowy – klucz obcy, ponieważ do wyniku trafiły wiersze, w których numer kategorii przez przypadek był równy numerowi modelu.

Tabele mogą być też łączone na podstawie warunków złączeń zawierających inne operatory porównania niż =, np. operator > – takie złączenia nazywane są nierównościowymi. Chociaż wydaje się, że poniższe zapytanie zwraca informacje o produktach sprzedanych za cenę niższą niż wartość zamówienia, to w rzeczywistości zwraca ono połączone ze sobą w trybie każdy z każdym, spełniające ten warunek wiersze obu tabel:

```

SELECT H.SalesOrderID, H.SalesOrderNumber, D.LineTotal, H.SubTotal
FROM [SalesLT].[SalesOrderHeader] AS H
JOIN [SalesLT].[SalesOrderDetail] AS D
ON D.LineTotal<H.SubTotal;
-----
```

SalesOrderID	SalesOrderNumber	LineTotal	SubTotal
71774	S071774	2.994000	880,3484
71774	S071774	4.770000	880,3484
71774	S071774	5.394000	880,3484

71774	S071774	5.394000	880,3484
71774	S071774	8.244000	880,3484
71774	S071774	8.244000	880,3484
71774	S071774	10.788000	880,3484
71774	S071774	10.788000	880,3484
71774	S071774	10.788000	880,3484
71774	S071774	10.992000	880,3484
...			

Zapytanie to zwraca znacznie więcej wierszy, niż się spodziewaliśmy — 14 067 wierszy, podczas gdy cała tabela [SalesLT].[SalesOrderDetail] (składająca się z dwóch łączonych tabel) liczy zaledwie 542 wiersze.

Złączenia nierównościowe są bardzo rzadko używane — jednym z ich nielicznych zastosowań jest analiza danych polegająca na wyszukiwaniu istniejących między tymi danymi zależności. Ponieważ wyniki takich złączeń zawierają mnóstwo powtórzonych wierszy, złączenia nierównościowe z reguły występują razem ze złączeniami naturalnymi:

```
SELECT H.SalesOrderID, H.SalesOrderNumber, D.LineTotal, H.SubTotal
FROM [SalesLT].[SalesOrderHeader] AS H
JOIN [SalesLT].[SalesOrderDetail] AS D
    ON D.SalesOrderID=H.SalesOrderID
    AND D.LineTotal<H.SubTotal;
```

SalesOrderID	SalesOrderNumber	LineTotal	SubTotal
71774	S071774	356.898000	880,3484
71774	S071774	356.898000	880,3484
71776	S071776	63.900000	78,81
71780	S071780	873.816000	38418,6895
71780	S071780	923.388000	38418,6895
...			

Złączenia zewnętrzne

Wszystkie omówione do tej pory złączenia były złączeniami wewnętrznymi — ich wyniki zawierały tylko wiersze

spełniające podany w nich warunek złączenia.

Złączenie wewnętrzne jest domyślnym typem złączenia, dlatego do tej pory pomijaliśmy opcjonalne słowo kluczowe INNER operatora JOIN. Brak tego słowa oznacza właśnie złączenie wewnętrzne, czyli oba poniższe zapytania są identyczne i zwracają takie same wyniki:

```
SELECT ProductNumber, C.Name  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON C.ProductCategoryID=P.ProductCategoryID  
WHERE P.Color = 'WHITE';
```

ProductNumber	Name
S0-B909-M	Socks
S0-B909-L	Socks
S0-R809-M	Socks
S0-R809-L	Socks

```
SELECT ProductNumber, C.Name  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON C.ProductCategoryID=P.ProductCategoryID  
WHERE P.Color = 'WHITE';
```

ProductNumber	Name
S0-B909-M	Socks
S0-B909-L	Socks
S0-R809-M	Socks
S0-R809-L	Socks

Ponieważ klauzula ON również filtryje wiersze (żeby produkt trafił do wyniku zapytania, w tabeli [SalesLT].[ProductCategory] musi istnieć powiązany z nim rekord), zapytanie to nie zwraca wszystkich białych produktów. W przykładowej bazie danych jest jeden produkt, który nie należy do jakiejkolwiek kategorii, co oznacza, że zostanie on usunięty w klauzuli ON.

Jeżeli wynik zapytania miał zawierać tylko numery produktów, które

należą do jakiejkolwiek kategorii, to zapytanie jest poprawne. Jeśli jednak chcieliśmy odczytać numery wszystkich produktów, to musielibyśmy zastosować złączenie zewnętrzne, czyli takie, które zwróci również niepasujące (tj. niemające odpowiednika w powiązanej tabeli) wiersze.

Złączenia zewnętrzne dzielą się na lewostronne, prawostronne i obustronne.

Złączenie lewostronne

Lewostronne złączenie zewnętrzne (`LEFT OUTER JOIN`) powoduje pozostawienie w wyniku niepasujących wierszy z pierwszej (lewej) tabeli. Ponieważ te wiersze nie mają swoich odpowiedników w złączonej tabeli, w kolumnach wyniku zwracającego dane z drugiej (prawej) tabeli zostanie wstawiona wartość `NULL`:

```
SELECT ProductNumber, C.Name  
FROM [SalesLT].[Product] AS P  
LEFT OUTER JOIN [SalesLT].[ProductCategory] AS C  
    ON C.ProductCategoryID=P.ProductCategoryID  
WHERE P.Color = 'WHITE';  
-----
```

ProductNumber	Name
S0-B909-M	Socks
S0-B909-L	Socks
S0-R809-M	Socks
S0-R809-L	Socks
NW-1S	NULL

Tym razem wynik zapytania zawiera numery wszystkich produktów, w tym nienależącego do jakiejkolwiek kategorii.

Złączenie prawostronne

Tak jak lewostronne złączenie zewnętrzne dodaje do wyniku zapytania niepasujące wiersze z tabeli, której nazwa jest podana jako pierwsza, tak prawostronne złączenie zewnętrzne (`RIGHT OUTER JOIN`) dodaje niepasujące wiersze z prawej tabeli. Czyli zmieniając kolejność tabel w klauzuli `FROM`, możemy zastąpić złączenie lewostronne równoważnym złączeniem prawostronnym. Poniższe

zapytanie zwraca nazwiska pracowników, którzy nie zrealizowali żadnego zamówienia:

```
SELECT ProductNumber, C.Name  
FROM [SalesLT].[ProductCategory] AS C  
RIGHT OUTER JOIN [SalesLT].[Product] AS P  
    ON C.ProductCategoryID=P.ProductCategoryID  
WHERE P.Color = 'WHITE';
```

```
-----  
ProductNumber          Name  
S0-B909-M              Socks  
S0-B909-L              Socks  
S0-R809-M              Socks  
S0-R809-L              Socks  
NW-1S                  NULL
```

Złączenie obustronne

Obustronne złączenia zewnętrzne (`FULL OUTER JOIN`) zwracają wszystkie wiersze obu złączonych tabel, w tym te, które nie spełniają warunku złączenia. Kolejne zapytanie zwraca niepasujące do siebie wiersze obu powiązanych tabel:

```
SELECT ProductNumber, C.Name  
FROM [SalesLT].[Product] AS P  
FULL OUTER JOIN [SalesLT].[ProductCategory] AS C  
    ON C.ProductCategoryID=P.ProductCategoryID  
WHERE P.ProductCategoryID IS NULL;
```

```
-----  
ProductNumber          Name  
NW-1S                  NULL  
NULL                  Bikes  
NULL                  Components  
NULL                  Clothing  
NULL                  Accessories  
NULL                  Fancy Stuff
```

Złączenie krzyżowe (iloczyn kartezjański)

Iloczynem kartezjańskim dwóch zbiorów jest zbiór zawierający wszystkie kombinacje (pary) ich elementów. Tabele reprezentują zbiory, dlatego **iloczynem kartezjańskim dwóch tabel jest tabela zawierająca wszystkie możliwe kombinacje wierszy złączonych tabel**.

Ponieważ przy złączeniu krzyżowym nie określa się warunku złączenia (bo wynik ma zawierać wszystkie możliwe połączenia wierszy), możemy w ten sposób złączyć dowolne dwie tabele:

```
SELECT P.Name, C.FirstName  
FROM [SalesLT].[Product] AS P  
CROSS JOIN [SalesLT].[Customer] AS C;
```

Name	FirstName
All-Purpose Bike Stand	Orlando
All-Purpose Bike Stand	Keith
All-Purpose Bike Stand	Donna
All-Purpose Bike Stand	Janet
All-Purpose Bike Stand	Lucy
All-Purpose Bike Stand	Rosmarie
All-Purpose Bike Stand	Dominic
All-Purpose Bike Stand	Kathleen
All-Purpose Bike Stand	Katherine
All-Purpose Bike Stand	Johnny
All-Purpose Bike Stand	Christopher
All-Purpose Bike Stand	David
All-Purpose Bike Stand	John
...	

Wynik złączenia krzyżowego zawiera mnóstwo powtórzeń — imię każdego klienta zostało powtórzone 296 razy (tyle wierszy liczyła tabela [SalesLT].[Customer]), a nazwa każdego towaru 847 razy (tylko jest klientów). W rezultacie zapytanie to zwróciło 250 712 ($296 * 847 = 250\ 712$) wierszy.

Ponieważ uzyskane w wyniku złączenia wiersze są niepowtarzalne (z wyjątkiem powtarzających się wierszy, w których imię klienta jest przez przypadek takie samo jak nazwa towaru), klauzula DISTINCT ich nie wyeliminuje:

```
SELECT DISTINCT P.Name, C.FirstName  
FROM [SalesLT].[Product] AS P  
CROSS JOIN [SalesLT].[Customer] AS C;
```

```
-----  
Name           FirstName  
All-Purpose Bike Stand    Orlando  
All-Purpose Bike Stand    Keith  
All-Purpose Bike Stand    Donna  
All-Purpose Bike Stand    Janet  
All-Purpose Bike Stand    Lucy  
All-Purpose Bike Stand    Rosmarie  
All-Purpose Bike Stand    Dominic  
All-Purpose Bike Stand    Kathleen  
All-Purpose Bike Stand    Katherine  
All-Purpose Bike Stand    Johnnny  
All-Purpose Bike Stand    Christopher  
All-Purpose Bike Stand    David  
All-Purpose Bike Stand    John  
...  
...
```

Wynik krzyżowego złączenia tabel może być ogromny. Na przykład złączenie w ten sposób tabeli [SalesLT].[SalesOrderDetail](542 wiersze) z liczącą 847 wierszy tabelą [SalesLT].[Customer] dałoby 459 074 wiersze ($542 * 847 = 459\ 074$). Ponieważ tak otrzymane dane są w większości wynikiem przypadkowego połączenia wierszy, złączenia krzyżowe są używane przede wszystkim do:

1. **Generowania danych testowych.** Wynik poniższego zapytania zawiera 250 tysięcy kombinacji kodów (zwróć uwagę, że słowo kluczowe CROSS JOIN może być pominięte):

```
SELECT [ProductNumber], C.[rowguid]  
FROM [SalesLT].[Product], [SalesLT].[Customer] AS;
```

ProductNumber	rowguid
BB-7421	947BCAF1-1F32-44F3-B9C3-0011F95FBE54
BB-7421	45F0A45D-6EC0-4F4C-A5AB-00B128AF62FD
BB-7421	D5AE552E-FAFD-49AB-AC05-014484CA1139
BB-7421	3CEF570C-26DD-478E-9F28-015037841EE2
BB-7421	7C96C878-A038-4B2F-825F-016C922D6407
BB-7421	C8099813-1F08-41C6-8567-025CAB01F0FA
BB-7421	592B130D-BFF6-4744-B0B6-02F706504993
...	

2. Poprawy wydajności podzapytań zawierających kilka funkcji grupujących. Grupowanie danych zostało omówione w kolejnym rozdziale, a podzapytania w rozdziale 8. Tam też zamieściliśmy przykład optymalizacji podzapytań z użyciemłączenia krzyżowego.

Złączenia wielokrotne

W jednym zapytaniu możemy się odwołać do więcej niż dwóch tabel. Choć górnny limit liczby złączonych tabel zależy od serwera bazodanowego, to z reguły jest on na tyle wysoki, że nie musimy się nim przejmować^[3]. Powinniśmy jednak pamiętać, że z każdą następną złączoną tabelą spada wydajność zapytania, w związku z czym musimy podzielić zapytanie na kilka mniejszych.

Niezależnie od tego, do ilu tabel się odwołujemy, serwery bazodanowe zawsze łączą ze sobą dwie tabele. Otrzymany po połączeniu dwóch pierwszych tabel zbiór pośredni jestłączony z kolejną tabelą, dając w wyniku następny zbiór pośredni. Ta operacja powtarza się aż do złączenia wszystkich tabel.

Proces wielokrotnego złączenia tabel przeanalizujemy na przykładzie zapytania zwracającego nazwiska klientów i nazwy kupionych przez nich produktów. W bazie *AdventureWorksLT2012* te informacje znajdują się w różnych, ale powiązanych ze sobą tabelach. Żeby móc skonstruować właściwe zapytanie, musimy wiedzieć, jakie tabele odczytać i jak są one nawzajem powiązane. Wiemy już, że tego typu

informacje o strukturze bazy danych najczęściej przedstawia się w postaci diagramów E/R (rysunek 5.1).



Rysunek 5.1. Na diagramie E/R (Encja/Relacja) znajdziemy nazwy tabel i ich kolumn, nazwę kolumny klucza podstawowego i informacje o łączących poszczególne tabele (encje) relacjach

Choć zadanie odczytania nazwisk oraz nazw towarów może się wydawać skomplikowane, nie jest wcale trudniejsze od wcześniejszych przykładów. Wymaga jedynie napisania dłuższego zapytania:

1. Zaczniemy od odczytania wszystkich nazw produktów z tabeli [SalesLT].[Product]:

```
SELECT P.Name
FROM [SalesLT].[Product] AS P;
```

Name
All-Purpose Bike Stand
AWC Logo Cap
Bike Wash - Dissolver
Cable Lock

```
Chain  
Classic Vest, L  
Classic Vest, M  
...  
...
```

2. W wyniku otrzymaliśmy 296 wierszy, czyli nazwy wszystkich sprzedawanych przez firmę AdventureWorks produktów. Następnie złączymy tę listę nazw z powiązaną tabelą zawierającą informacje o szczegółach zamówień, czyli z tabelą [SalesLT].[SalesOrderDetail]:

```
SELECT P.Name  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[SalesOrderDetail] AS D  
ON P.ProductID = D.ProductID;  
-----
```

```
Name  
Sport-100 Helmet, Red  
Sport-100 Helmet, Black  
Sport-100 Helmet, Black  
...  
...
```

3. Tym razem wynik zawiera już 542 wiersze — ponieważ wykonaliśmy naturalnełączenie wewnętrzne, oznacza to, że niektóre produkty były zamawiane częściej niż raz. Pozostało nam tylko dołączyć do wyniku kolumnę z nazwiskami klientów.
4. Jeżeli spróbujemy dodać kolumnę LastName tabeli [SalesLT].[Customer], okaże się, że nie możemy wykonaćnaturalnegołączenia — kluczem tej tabeli jest zawierająca identyfikatory klientów kolumna CustomerID, która nie występuje w żadnej z dwóch złączonych do tej pory tabel:

```
SELECT P.Name  
FROM [SalesLT].[Product] AS P
```

```

JOIN [SalesLT].[SalesOrderDetail] AS D
    ON P.ProductID = D.ProductID
JOIN [SalesLT].[Customer] AS C
    ON C.CustomerID = ???

```

5. Z pokazanego na rysunku 5.1 diagramu E/R wynika, że dołączenia z tabelą [SalesLT].[Customer] musimy użyć tabeli pośredniej [SalesLT].[SalesOrderHeader]. Dodajmy tę tabelę i na koniec posortujmy jeszcze wynik według nazwisk klientów:

```

SELECT C.LastName, P.Name
FROM [SalesLT].[Customer] AS C
JOIN [SalesLT].[SalesOrderHeader] AS H
    ON C.CustomerID=H.CustomerID
JOIN [SalesLT].[SalesOrderDetail] AS D
    ON D.SalesOrderID=H.SalesOrderID
JOIN [SalesLT].[Product] AS P
    ON D.ProductID=P.ProductID
ORDER BY C.LastName;

```

LastName	Name
Abel	Hydration Pack - 70 oz.
Abel	Sport-100 Helmet, Blue
Abel	Sport-100 Helmet, Red
Beck	Road-350-W Yellow, 40
Beck	Hitch Rack - 4-Bike
Beck	ML Road Frame-W - Yellow, 44
Beck	Road-350-W Yellow, 42
...	



Kolejność poszczególnych sekcji JOIN ... ON odpowiada logicznej kolejności łączenia tabel. Jednak serwery bazodanowe optymalizują zapytania, a więc w rzeczywistości tabele mogły być złączone w innej kolejności.

Określanie kolejności złączeń

W złączeniach równościowych kolejność łączenia tabel nie ma żadnego wpływu na wynik zapytania. Natomiast w przypadku złączeń zewnętrznych klauzula `ON` wykonywana po dodaniu niepasujących wierszy z powrotem usuwa je z wyniku.

Przeanalizujmy to na przykładzie produktów, które nigdy nie zostały sprzedane. Wiemy już, że łącząc lewostronnie tabele `[SalesLT].[Product]` i `[SalesLT].[SalesOrderDetail]`, otrzymamy listę wszystkich produktów:

```
SELECT P.ProductNumber, D.SalesOrderID  
FROM [SalesLT].[Product] AS P  
LEFT OUTER JOIN [SalesLT].[SalesOrderDetail] AS D  
ON D.ProductID=P.ProductID;
```

ProductNumber	SalesOrderID
BB-7421	71782
BB-7421	71898
BB-7421	71936
BB-8107	NULL
BB-9108	71782
BB-9108	71845
BB-9108	71898
BB-9108	71936
BC-M005	NULL

...

To zapytanie zwróciło 696 wierszy. Gdybyśmy chcieli, żeby wynik zapytania zawierał jeszcze jakieś dane z tabeli `[SalesLT].[SalesOrderHeader]`, np. wartości zamówień, musielibyśmy dołączyć do zapytania tę tabelę:

```
SELECT P.ProductNumber, D.SalesOrderID, H.TotalDue  
FROM [SalesLT].[Product] AS P  
LEFT OUTER JOIN [SalesLT].[SalesOrderDetail] AS D  
ON D.ProductID=P.ProductID  
JOIN [SalesLT].[SalesOrderHeader] AS H  
ON D.SalesOrderID=H.SalesOrderID;
```

ProductNumber	SalesOrderID	TotalDue
FR-R72Y-48	71774	972,785
FR-R72Y-38	71774	972,785
RB-9231	71776	87,0851
FR-M63S-42	71780	42452,6519
BK-M38S-46	71780	42452,6519

...

To zapytanie zwróciło tylko 542 wiersze i nie znajdziemy w jego wyniku m.in. produktu o numerze BB-8107:

```
SELECT P.ProductNumber, D.SalesOrderID, H.TotalDue
FROM [SalesLT].[Product] AS P
LEFT OUTER JOIN [SalesLT].[SalesOrderDetail] AS D
    ON D.ProductID=P.ProductID
JOIN [SalesLT].[SalesOrderHeader] AS H
    ON D.SalesOrderID=H.SalesOrderID
WHERE P.ProductNumber = 'BB-8107';
```

(0 row(s) affected)

Kolejność wykonywania operacji łączenia jest ustalana przez optymalizator i nie wynika z kolejności, w jakiej nazwy tabel zostały wymienione w klauzuli `FROM`. **Wynik tego zapytania nie zawiera więc informacji o niesprzedanych produktach, które początkowo dodane zostały z niego usunięte w trakcie wykonywania drugiej klauzuli `ON`.** Problem ten można rozwiązać na dwa sposoby:

- Łącząc zewnętrznie wszystkie tabele. Wadą tego rozwiązania jest to, że gdyby w kolejnych tabelach (w tym przypadku w tabeli `[SalesLT].[SalesOrderHeader]`) również znajdowały się wiersze niepasujące do wierszy z ostatniej łączonej tabeli, one także znalazłyby się w wyniku zapytania:

```
SELECT P.ProductNumber, D.SalesOrderID, H.TotalDue
FROM [SalesLT].[Product] AS P
LEFT OUTER JOIN [SalesLT].[SalesOrderDetail] AS D
```

```

        ON D.ProductID=P.ProductID
    LEFT JOIN [SalesLT].[SalesOrderHeader] AS H
        ON D.SalesOrderID=H.SalesOrderID;

```

ProductNumber	SalesOrderID	TotalDue
BB-7421	71782	43962,7901
BB-7421	71898	70698,9922
BB-7421	71936	108597,9536
BB-8107	NULL	NULL
BB-9108	71782	43962,7901
BB-9108	71845	45992,3665
BB-9108	71898	70698,9922
BB-9108	71936	108597,9536
BC-M005	NULL	NULL

...

2. Określając za pomocą nawiasów kolejność, w jakiej tabele powinny być połączone. Jeżeli złączenie zewnętrzne będzie wykonane jako ostatnie (co wymaga przeniesienia określającej warunek tego złączenia klauzuli ON poza nawias), niepasujący wiersz znajdzie się w wyniku zapytania:

```

SELECT P.ProductNumber, D.SalesOrderID, H.TotalDue
FROM [SalesLT].[Product] AS P
LEFT OUTER JOIN
(
    [SalesLT].[SalesOrderDetail] AS D
    JOIN [SalesLT].[SalesOrderHeader] AS H
        ON D.SalesOrderID=H.SalesOrderID
)
ON D.ProductID=P.ProductID;

```

ProductNumber	SalesOrderID	TotalDue
BK-R64Y-44	71938	98138,2131
SE-T924	71946	43,0437
FR-T67Y-58	NULL	NULL

SO-B909-L	NULL	NULL
ST-1401	NULL	NULL
...		

Złączenie tabeli z nią samą

Złączenie tabeli z nią samą jest wykonywane w taki sam sposób jak omawiane do tej pory złączenia różnych tabel. Chociaż serwery bazodanowe nie tworzą kopii złączonej tabeli, to wszystkie operacje przeprowadzane są tak, jakby dotyczyły dwóch identycznych tabel.

Zaczniemy od prostego przykładu. Spróbujmy złączyć krzyżowo tabelę [SalesLT].[Product] z nią samą i odczytać uzyskane w wyniku nazwy towarów:

```
SELECT ProductNumber  
FROM [SalesLT].[Product]  
CROSS JOIN [SalesLT].[Product];
```

Msg 1013, Level 16, State 1, Line 192

The objects "SalesLT.Product" and "SalesLT.Product" in the FROM clause have the same exposed names. Use correlation names to distinguish them.

Ponieważ obie wymienione w klauzuli FROM nazwy tabeli są takie same, musimy nadać im różne aliasy, żeby odróżnić od siebie wirtualne kopie tej tabeli:

```
SELECT ProductNumber  
FROM [SalesLT].[Product] AS P1  
CROSS JOIN [SalesLT].[Product] AS P2;
```

Msg 209, Level 16, State 1, Line 197

Ambiguous column name 'ProductName'...

Nadal jednak zapytanie nie działa. Kolejny problem jest spowodowany tym, że obie wirtualne kopie tabeli zawierają kolumnę ProductNumber, a więc jej nazwa jest niejednoznaczna i musi być poprzedzona aliasem:

```
SELECT P1.ProductNumber
```

```
FROM [SalesLT].[Product] AS P1  
CROSS JOIN [SalesLT].[Product] AS P2;
```

ProductNumber

BB-7421

BB-7421

BB-7421

BB-7421

BB-7421

BB-7421

...

To, czy odczytamy kolumnę ProductNumber z wirtualnej kopii P1, czy z wirtualnej kopii P2, nie ma znaczenia — za każdym razem w wyniku uzyskamy te same, powtórzone 295 razy (tyle wierszy liczy tabela) nazwy wszystkich produktów. Przekonaliśmy się więc, że złączenie krzyżowe tabeli z nią samą jest wykonywane dokładnie w ten sam sposób co złączenie krzyżowe dwóch różnych tabel.

Złączenia tabel z nimi samymi są często wykonywane w celu analizy danych. Na przykład poniższe zapytanie zwraca dane towarów, które są co najmniej sześciokrotnie tańsze niż jakikolwiek inny towar z tej samej kategorii:

```
SELECT P1.ProductID, P1.ListPrice, P2.ProductID, P2.ListPrice  
FROM [SalesLT].[Product] AS P1  
JOIN [SalesLT].[Product] AS P2  
    ON P1.ProductCategoryID=P2.ProductCategoryID  
WHERE P1.ListPrice*6 < P2.ListPrice  
ORDER BY P1.ProductID;
```

ProductID	ListPrice	ProductID	ListPrice
873	2,29	928	24,99
873	2,29	929	29,99
873	2,29	930	35,00
873	2,29	931	21,49
873	2,29	932	24,99

873	2,29	933	32,60
873	2,29	934	28,99
921	4,99	929	29,99

Eliminacja duplikatów

Łącząc tabelę z nią samą, powinniśmy się spodziewać w wyniku zapytania powtarzających się wierszy. Powtórzenia te mogą wynikać z dwukrotnego umieszczenia w klauzuli SELECT nazwy tej samej kolumny. W tym przypadku **usunięcie dowolnej kopii nazwy kolumny z klauzuli SELECT** rozwiąże problem.

Trudniejsze do wyeliminowania są powtórzenia drugiego rodzaju. Ponieważ złączymy tabelę z nią samą, te same kolumny są odczytywane z obu jej wirtualnych kopii. Jeżeli warunek złączenia jest symetryczny^[4], wiersze odczytane z tabeli P1 zostaną połączone ze swoimi kopiami odczytanymi z tabeli P2.

Tego typu powtórzeń nie wyeliminujemy za pomocą słowa kluczowego DISTINCT. **Najprostszym sposobem na wyeliminowanie takich powtórzeń jest dodanie asymetrycznego warunku złączenia.** Może to być np. warunek sprawdzający, czy cena produktu odczytana z tabeli P1 jest mniejsza od ceny produktu odczytanego z tabeli P2. Dzięki temu wynik zapytania nie będzie zawierał powtórzonych wierszy.

Klucze obce w obrębie jednej tabeli

Skoro najczęstszym typem złączeń są złączenia naturalne, to tabela przeważnie jestłączana z nią samą, jeżeli w jej obrębie występuje relacja klucz podstawowy – klucz obcy.

Taka sytuacja ma miejsce np. w tabeli [SalesLT].[ProductCategory] bazy AdventureWorksLT2012:

1. Kluczem podstawowym tej tabeli jest kolumna [ProductCategoryID] — jego zadaniem jest identyfikowanie kategorii.
2. Kluczem obcym jest kolumna [ParentProductCategoryID] — ten klucz reprezentuje relację pomiędzy podkategorią (identyfikowaną za pomocą wartości ProductCategoryID) a kategorią nadzczną, czyli kategorią, której identyfikator

został zapisany w kolumnie ParentProductCategoryID.

Łącząc tabelę [SalesLT].[ProductCategory] z nią samą, możemy więc odczytać listę podkategorii wybranej kategorii^[5]:

```
SELECT C1.ParentProductCategoryID, C1.ProductCategoryID, C1.Name,
C2.ParentProductCategoryID, C2.ProductCategoryID, C2.Name
FROM [SalesLT].[ProductCategory] AS C1
JOIN [SalesLT].[ProductCategory] AS C2
    ON C1.ParentProductCategoryID=C2.ProductCategoryID
WHERE C1.ParentProductCategoryID=1;
```

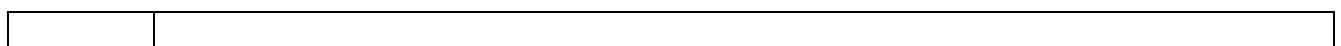
ParentProductCategoryID	ProductCategoryID	Name
ParentProductCategoryID	ProductCategoryID	Name
1 5	Mountain Bikes	NULL 1 Bikes
1 6	Road Bikes	NULL 1 Bikes
1 7	Touring Bikes	NULL 1 Bikes

Serwer bazodanowy wykonał to zapytanie następująco:

1. Tabela [SalesLT].[ProductCategory] została złączona ze swoją wirtualną kopią.
2. Z otrzymanego w ten sposób zbioru pośredniego zostały wyeliminowane te wiersze, które nie spełniały warunku złączenia, czyli na tym etapie wynik zawierał identyfikatory wszystkich kategorii i ich podkategorii.
3. Klauzula WHERE wyeliminowała dane tych kategorii, które nie były bezpośrednimi podkategoriami kategorii o podanym numerze.

Łączenie wyników zapytań

Oprócz tabel język SQL pozwala również łączyć wyniki zapytań. **O ile łączenie tabel polega na dodawaniu** do wyniku zapytania dodatkowych, pochodzących ze złączonych tabel **kolumn**, **o tyle łączenie wyników polega na dodawaniu** (lub usuwaniu) **wierszy** zwróconych przez złączone zapytania.





Wskazówka

Łączone wyniki zapytań muszą się składać z tej samej liczby kolumn, a odpowiadające sobie kolumny muszą być tych samych albo umożliwiających niejawną konwersję typów.

Suma

Dodać do siebie wyniki zapytań możemy za pomocą operatora `UNION`. Odpowiada on teoriomnogościowemu operatorowi sumy zbiorów, czyli w wyniku uzyskamy wiersze zwrócone przez pierwsze lub drugie zapytanie.

Poniższe zapytanie zwraca nazwy dostawców produktów złączone z nazwami kategorii:

```
SELECT [Name]
FROM [SalesLT].[Product]
UNION
SELECT [Name]
FROM [SalesLT].[ProductCategory];
```

```
-----
Name
Accessories
All-Purpose Bike Stand
AWC Logo Cap
Bib-Shorts
Bike Racks
Bike Stands
Bike Wash - Dissolver
Bikes
...
```

Ponieważ złączone ze sobą zapytania są traktowane przez serwer bazodanowy jako jedno zapytanie, średnik należy umieścić tylko na końcu całego zapytania. Z tego samego powodu tylko w ostatnim zapytaniu może wystąpić klauzula `ORDER BY`:

```
SELECT [Name]
FROM [SalesLT].[Product]
ORDER BY 1
```

```
UNION
SELECT [Name]
FROM [SalesLT].[ProductCategory]
ORDER BY 1;
-----
Msg 156, Level 15, State 1, Line 225
Incorrect syntax near the keyword 'UNION'.
```

Jako że nazwy kolumnłączonych wyników zapytań nie muszą być takie same, w klauzuli ORDER BY najczęściej używa się ich numerów, a nie nazw:

```
SELECT [Name]
FROM [SalesLT].[Product]
UNION
SELECT [Name]
FROM [SalesLT].[ProductCategory]
ORDER BY 1;
-----
```

```
Name
Accessories
All-Purpose Bike Stand
AWC Logo Cap
Bib-Shorts
Bike Racks
Bike Stands
Bike Wash - Dissolver
Bikes
...
```

Operator UNION powoduje usunięcie z wyniku powtórzonych wierszy, zatem wynik poniższego zapytania będzie liczył 1 153 wiersze, a nie 1 297 wierszy:

```
SELECT [AddressID]
FROM [SalesLT].[Address]
UNION
SELECT [CustomerID]
```

```
FROM [SalesLT].[Customer];
```

```
AddressID
```

```
451
```

```
466
```

```
467
```

```
475
```

```
487
```

```
502
```

```
504
```

```
...
```

Jeżeli chcemy tylko złączyć ze sobą wyniki zapytań, bez eliminowania ewentualnych duplikatów, albo jeżeli mamy pewność, że łączone wyniki są rozłączne^[6], powinniśmy użyć operatora UNION ALL — **takie łączenie wyników zapytań, jako niewymagające wyszukiwania i usuwania powtarzających się wierszy, jest znacznie szybsze:**

```
SELECT [AddressID]
FROM [SalesLT].[Address]
UNION ALL
SELECT [CustomerID]
FROM [SalesLT].[Customer];
```

```
AddressID
```

```
451
```

```
466
```

```
467
```

```
475
```

```
487
```

```
502
```

```
504
```

```
...
```

Operatory UNION i UNION ALL pozwalają dodać wiersze do wyniku zapytania, podobnie jak operator logiczny OR. Nic więc dziwnego, że

zastąpienie zapytań z operatorem OR zapytaniami złączonymi operatorem UNION ALL jest dość proste:

1. Poniższe zapytanie zwraca dane zamówień złożonych przez dwóch wybranych klientów:

```
SELECT [SalesOrderID], [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
WHERE [CustomerID] IN (29847,30072);
```

```
-----  
SalesOrderID      TotalDue  
71774            972,785  
71776            87,0851  
77774            972,785
```

2. Każdy z tych warunków logicznych możemy umieścić w odrębnym zapytaniu i połączyć ich wyniki operatorem UNION ALL:

```
SELECT [SalesOrderID], [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
WHERE [CustomerID] = 29847
UNION ALL
SELECT [SalesOrderID], [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
WHERE [CustomerID] = 30072;;
```

```
-----  
SalesOrderID      TotalDue  
71774            972,785  
77774            972,785  
71776            87,0851
```

Operatory UNION i UNION ALL są też często używane do łączenia zapytań zwracających różne dane dotyczące tych samych obiektów, np. największych i najmniejszych wartości sprzedaży przeprowadzonych w danym dniu przez poszczególnych sprzedawców. Ponieważ w praktyce tego typu zapytania wymagają grupowania danych i używania podzapytań, zostały omówione w rozdziale 8.

Część wspólna

Operator `INTERSECT` zwraca część wspólną wyników dwóch zapytań, a więc tylko te wiersze, które zostały zwrócone przez oba zapytania. Odpowiada on teoriomnogościowemu operatorowi części wspólnej zbiorów, czyli w wyniku uzyskamy wiersze zwrócone przez pierwsze i drugie zapytanie.

 Wskazówka	Chociaż standard SQL3 uwzględnia operatory <code>INTERSECT ALL</code> i <code>EXCEPT ALL</code> , to jednak żaden serwer bazodanowy ich nie implementuje. Spowodowane jest to tym, że usuwanie wierszy na podstawie nie pojedynczego ich wystąpienia, ale wielokrotnego powtórzenia jest mało intuicyjne i prowadzi do błędów logicznych.
--	---

Wynikiem poniższego zapytania są identyfikatory zarówno klientów, jak i adresów:

```
SELECT [AddressID]
FROM [SalesLT].[Address]
INTERSECT
SELECT [CustomerID]
FROM [SalesLT].[Customer];
```

```
-----  
AddressID
```

```
451
523
613
648
668
560
587
```

Różnica

Operator `EXCEPT` zwraca te wiersze, które znalazły się wyłącznie w wyniku pierwszego zapytania i nie było ich w wyniku drugiego zapytania. Odpowiada on teoriomnogościowemu operatorowi różnicy zbiorów, czyli w wyniku uzyskamy wiersze zwrócone przez pierwsze, ale nie przez drugie zapytanie.

W przeciwieństwie do sumy i opisanej powyżej części wspólnej

operator EXCEPT jest asymetryczny — zmiana kolejności zapytań spowoduje zmianę wyniku. W tym przypadku wynik zawiera te numery, które zostały użyte jako identyfikatory adresów, ale które nie występują w tabeli [SalesLT].[Customer]:

```
SELECT [AddressID]
FROM [SalesLT].[Address]
EXCEPT
SELECT [CustomerID]
FROM [SalesLT].[Customer];
```

```
-----  
AddressID
```

```
593
```

```
902
```

```
1042
```

```
879
```

```
1065
```

```
524
```

```
...
```

Jak już wspomnieliśmy, operator EXCEPT jest asymetryczny (kolejność łączonych zbiorów ma w jego przypadku znaczenie), więc powyższe zapytanie zwróciło 306 wierszy, a po zmianie kolejności zapytań wynikiem ich odjęcia będą 703 wiersze:

```
SELECT [CustomerID]
FROM [SalesLT].[Customer]
EXCEPT
SELECT [AddressID]
FROM [SalesLT].[Address];
```

```
-----  
CustomerID
```

```
12
```

```
29784
```

```
29606
```

```
200
```

```
29603
```

Łączenie wierszy i wyników funkcji tabelarycznych

Zapytania nie muszą odczytywać danych bezpośrednio z tabel — w wielu produkcyjnych bazach danych bezpośredni dostęp do tabel jest wręcz niemożliwy (ze względów bezpieczeństwa i dla poprawy wydajności zapytań), a użytkownicy odczytują i modyfikują dane za pośrednictwem widoków, procedur lub funkcji.

Na przykład w bazie *AdventureWorksLT2012* znajduje się funkcja tabelaryczna [`ufnGet AllCategories`], która zwraca dane wszystkich kategorii, ich podkategorii i należących do nich produktów. Dzięki temu zamiast pisać rekurencyjne zapytania, wystarczy wywołać wspomnianą funkcję (funkcje tabelaryczne zwracają dane w postaci tabel, a więc ich nazwy wywołuje się w klauzuli `FROM`):

```
SELECT *
FROM [dbo].[ufnGetAllCategories];
-----
ParentProductCategoryName    ProductCategoryName    ProductCategoryID
Accessories                  Bike Racks           30
Accessories                  Bike Stands          31
Accessories                  Bottles and Cages   32
Accessories                  Cleaners            33
Accessories                  Fenders             34
Accessories                  Helmets            35
Accessories                  Hydration Packs    36
Accessories                  Lights              37
Accessories                  Locks               38
Accessories                  Panniers            39
Accessories                  Pumps               40
Accessories                  Tires and Tubes   41
Clothing                     Bib-Shorts          22
...

```

Jak widać, nazwy widoków, tak jak nazwy tabel, czy funkcji

tabelarycznych umieszcza się w klauzuli `FROM`. To, czy dane będą odczytane z tabeli, czy z widoku, nie ma żadnego wpływu na budowę zapytania — nadal możemy w nim używać wszystkich poznanych do tej pory klauzul i stosować wszystkie opisane w książce techniki odczytywania danych.

Innym typem obiektów, których nazwy można umieszczać w klauzuli `FROM`, są widoki. W bazie `AdventureWorksLT` jest m.in. widok, który też zwraca dane o kategoriach, ich podkategoriach i należących do nich produktach:

```
SELECT *
FROM [SalesLT].[vGetAllCategories];
-----
ParentProductCategoryName      ProductCategoryName
ProductCategoryID
Accessories                      Racks                  30
Accessories                      Bike Stands            31
Accessories                      Bottles and Cages    32
Accessories                      Cleaners               33
Accessories                      Fenders                34
Accessories                      Helmets                35
Accessories                      Hydration Packs     36
Accessories                      Lights                 37
Accessories                      Locks                  38
Accessories                      Panniers               39
Accessories                      Pumps                 40
Accessories                      Tires and Tubes     41
Clothing                         Bib-Shorts             22
...
...
```

Główna różnica między odczytywaniem danych poprzez widok a odczytywaniem ich poprzez funkcję tabelaryczną polega na tym, że widoku nie można wywołać z parametrami. Drugą, wynikającą zresztą z pierwszej, różnicą jest to, że nazwa funkcji tabelarycznej musi być uzupełniona o nawiasy, nawet jeżeli wywołujemy tę funkcję bez parametrów. W przykładowej bazie danych znajduje się funkcja tabelaryczna, która zwraca dane klienta

o przekazanym jako parametr jej wywołania identyfikatorze. I tak żeby poznać dane klienta o numerze 1, wystarczy wykonać poniższe zapytanie:

```
SELECT *  
FROM [dbo].[ufnGetCustomerInformation](1);
```

```
-----  
CustomerID      FirstName      LastName  
1                Orlando        Gee
```

Język SQL umożliwia nie tylko odczytywanie danych poprzez funkcje i widoki, lecz takżełączenie każdego zwróconego przez zapytanie wiersza z wynikiem funkcji tabelarycznej lub podzapytania — służy do tego specjalny operator **APPLY**.

Operator **APPLY**

Operator **APPLY** umożliwia wywołanie dla każdego wiersza pierwszej (lewej) tabeli dowolnego wyrażenia zwracającego dane w postaci tabelarycznej, np. podzapytania albo funkcji tabelarycznej. Czyli możemy nie tylko złączyć wynik funkcji tabelarycznej z tabelą za pomocą operatora **JOIN**, lecz także wywołać funkcję tabelaryczną w zapytaniu.

Pokażemy to na kilku przykładach. Pierwszy z nich przedstawia wywołanie znanej nam funkcji `dbo.ufnGetContactInformation()`, jednak tym razem będzie ona wywołana tyle razy, ile wierszy zwraca zapytanie. W rezultacie wynik zapytania zawiera dane odczytane z tabeli połączone z danymi zwróconymi przez tę funkcję, w tym przypadku informacje o klientkach:

```
SELECT C.[CustomerID], F.*  
FROM [SalesLT].[Customer] AS C  
CROSS APPLY [dbo].[ufnGetCustomerInformation](C.[CustomerID]) AS F  
WHERE C.Title = 'Ms.';
```

```
-----  
CustomerID      CustomerID      FirstName      LastName  
3                3                Donna         Carreras  
4                4                Janet         Gates  
6                6                Rosmarie    Carroll
```

...

Funkcja `dbo.ufnGetContactInformation()` zwraca tylko jeden wiersz, ale operator `APPLY` umożliwia wywoływanie dowolnych funkcji tabelarycznych. Żeby się o tym przekonać, wywołamy funkcję zwracającą informacje o ostatnich zamówieniach złożonych przez klienta, którego identyfikator będzie przekazany jako pierwszy parametr wywołania. Liczba interesujących nas zamówień będzie drugim parametrem wywołania funkcji^[7]:

```
SELECT F.*  
FROM [SalesLT].[Customer] AS C  
CROSS APPLY [dbo].[udfLastOrders] (C.[CustomerID],2) AS F  
ORDER BY C.CustomerID;
```

SalesOrderID	OrderDate	TotalDue
71946	2008-06-01 00:00:00.000	43,0437
71923	2008-06-01 00:00:00.000	117,7276
71797	2008-06-01 00:00:00.000	86222,8072
77774	2014-04-01 00:00:00.000	972,785
71774	2008-06-01 00:00:00.000	972,785

...

Operator `CROSS APPLY` odpowiada złączeniu wewnętrzemu, a więc eliminuje z wyniku zapytania te wiersze, dla których funkcja zwróciła wartości `NULL` — w tym przypadku byłyby to dane klientów, którzy nigdy nie złożyli u nas zamówienia. Żeby się o tym przekonać, wystarczy umieścić w niej warunek sprawdzający, czy dowolna ze zwróconych przez funkcję kolumn nie zawiera wartości `NULL`:

```
SELECT C.FirstName, F.*  
FROM [SalesLT].[Customer] AS C  
CROSS APPLY [dbo].[udfLastOrders] (C.[CustomerID],2) AS F  
WHERE F.[SalesOrderID] IS NULL;
```

(0 row(s) affected)

Zgodnie z oczekiwaniemi zapytanie nie zwróciło żadnych wierszy.

Język SQL pozwala jednak na dołączenie do wyniku zapytania tych wierszy, dla których funkcja tabelaryczna zwróciła wartość `NULL`, a więc uzyskanie efektu podobnego do złączenia zewnętrznego. W tym celu operator `CROSS APPLY` należy zastąpić operatorem `OUTER APPLY`:

```
SELECT C.FirstName, C.[CustomerID], F.*  
FROM [SalesLT].[Customer] AS C  
OUTER APPLY [dbo].[udfLast0rders](C.[CustomerID],2) AS F;
```

FirstName	CustomerID	SalesOrderID	OrderDate	
Abigail	29792	NULL	NULL	NULL
Michael	29793	NULL	NULL	NULL
Derek	29794	NULL	NULL	NULL
Jon 86222,8072	29796	71797	2008-06-01 00:00:00.000	

Podsumowanie

- Język SQL pozwala na złączenie w jednym zapytaniu danych odczytyanych z wielu tabel.
- Złączenie tabel na podstawie par wartości klucz podstawowy – klucz obcy jest nazywane złączeniem naturalnym.
- Tabele mogą być złączone za pomocą dowolnego operatora, nie tylko za pomocą `=`.
- Wynik złączenia wewnętrznego zawiera tylko pasujące, tj. spełniające warunek złączenia, wiersze z obu tabel.
- Złączenie zewnętrzne pozwala dodać do wyniku dodatkowe, niepasujące wiersze z jednej tabeli lub obu połączonych tabel.
- Wynikiem złączenia krzyżowego są wszystkie możliwe kombinacje wierszy złączonych tabel.
- Chociaż w jednym zapytaniu możemy złączyć wiele tabel, to serwery bazodanowe zawsze łączą ze sobą dwie tabele. Tak otrzymany zbiór pośredni jest łączony z kolejną tabelą i proces ten jest powtarzany aż do złączenia wszystkich tabel.
- Złączenie tabeli z nią samą jest realizowane tak, jakbyśmy łączyli dwie identyczne tabele, a więc według takich samych reguł jak w przypadku złączenia różnych tabel. Duplikaty wierszy otrzymane

w wyniku wewnętrznegołączenia nienaturalnego można usunąć, dodając niesymetryczny warunekłączenia.

- Łączenie wyników zapytań przypominałączenie zbiorów i jest używane do podzielenia skomplikowanego zapytania na kilka prostszych.
- Operator **APPLY** pozwala wywołać dla poszczególnych zwróconych przez zapytanie wierszy funkcje tabelaryczne.

Zadania

1. Odczytaj alfabetycznie uporządkowaną listę nazw produktów sprzedanych kiedykolwiek klientom o imieniu Jeffrey.
2. Odczytaj imiona i nazwiska klientów, którzy nie złożyli ani jednego zamówienia.
3. Bez używania funkcji CASE napisz zapytanie zwracające numer zamówienia (kolumna [**SalesOrderID**]), wysokość opłaty (kolumna [**Freight**]) i wyraz **High** lub **Low**, przy czym za kosztowne uznaj te zamówienia, których wartość opłaty przekracza 100, a pozostałe oceń jako małe. Przykładowy wynik pokazany jest poniżej:

SalesOrderID	Freight	(No column name)
71936	2456,9673	High
71938	2220,3216	High
71774	22,0087	Low
71776	1,9703	Low

[1] Zgodne ze standardem języka SQL działanie operatora **NATURAL JOIN** może być przyczyną błędów. Jeżeli obie łączone tabele nie mają żadnych kolumn o tych samych nazwach, z wyjątkiem kolumn klucza podstawowego i obcego, zapytanie `SELECT * FROM TabelaA NATURAL JOIN TabelaB` będzie wykonane zgodnie z oczekiwaniami — tabele zostaną łączone na podstawie wartości kolumn klucza podstawowego i obcego. Gdyby jednak w obu tych tabelach występowała kolumna **Miasto**, łączenie zostałoby wykonane na podstawie wartości wszystkich kolumn o tych samych nazwach, czyli odpowiadałoby mu zapytanie `SELECT * FROM TabelaA JOIN TabelaB ON TabelaA.TowarID =`

TabelaB.TowarID AND TabelaA.Miasto = TabelaB.Miasto;.

[2] Wynika to z tego, że wartości kluczy podstawowych często są automatycznie generowanymi przez serwery bazodanowe liczbami całkowitymi, prawie zawsze zaczynającymi się od jedynki.

[3] Na przykład serwer SQL 2011 pozwala złączyć ze sobą do 256 tabel.

[4] Operator równości jest symetryczny, czyli oba poniższe warunki: T1.[Cena katalogowa] = T2.[Cena katalogowa] oraz T2.[Cena katalogowa] = T1.[Cena katalogowa] są identyczne.

[5] Hierarchia kategorii jest przechodnia — podkategoria B jest jednocześnie kategorią nadzczną kategorii C, która pośrednio też jest podkategorią kategorii A. Ponieważ odczytanie kompletnej listy kategorii wymaga użycia podzapytania, odpowiedni przykład znajduje się w rozdziale 8.

[6] Dwa zbiory są rozłączne, jeżeli nie mają wspólnych elementów, a dwie tabele lub dwa wyniki zapytań — jeżeli nie mają takich samych wierszy.

[7] Ponieważ składnia instrukcji CREATE FUNCTION zależy od serwera bazodanowego, tworzenie własnych funkcji wykracza poza zakres tej książki.

Rozdział 6. Grupowanie wierszy

- Czym funkcje grupujące różnią się od skalarnych?
- Jak można pogrupować zwracane przez zapytanie dane?
- Jak pogrupować te same dane na wiele różnych sposobów?
- Jak przedstawić wynik zapytania w postaci tabeli przestawnej?
- Jak używać funkcji grupujących do filtrowania zwracanych przez zapytania danych?

Funkcje grupujące

Poznane do tej pory zapytania zwracały szczegółowe, z reguły odczytane bezpośrednio z tabel dane. W tym rozdziale dowiesz się, jak używać funkcji grupujących do analizy danych, dzięki czemu można uzyskać bardziej ogólne wyniki, np. średnią cenę produktów czy zestawienie sprzedaży towarów z różnych kategorii.

Grupowanie danych polega na łączeniu wielu wierszy w jeden. W najprostszym przypadku łączy się wszystkie wiersze tabeli w jedną grupę, ale możliwe jest też ich podzielenie pomiędzy wiele grup. Wtedy podstawą zaklasyfikowania wiersza do danej grupy jest wartość jednej z kolumn lub wynik użytego do grupowania wyrażenia (rysunek 6.1).

Charakterystyczną cechą funkcji grupujących jest operowanie na zbiorach, a nie pojedynczych wartościach. Dzięki temu otrzymane w wyniku grupowania dane mogą być użyte jako argumenty ich wywołania. Jeżeli wszystkie wiersze tabeli zostały połączone w jedną grupę, funkcja grupująca zostanie wywołana tylko raz, w innym przypadku zostanie wywołana dla każdej grupy. Funkcje grupujące zwracają pojedyncze (skalarne) wartości, więc wywołuje się je w klauzuli SELECT, tak jak wcześniej poznane funkcje systemowe.

	Kolumna A	Kolumna B	Wartość
3 wiersze → 1 wiersz	A	B	1
	A	B	2
	A	B	3
2 wiersze → 1 wiersz	C	D	4
	C	D	5

	Kolumna A	Kolumna B	SUM(Wartość)
1 wiersz	A	B	6
1 wiersz	C	D	9

Rysunek 6.1. Grupowanie zmniejsza liczbę wierszy wyniku, upraszczając analizę danych, ale powoduje utratę informacji o szczegółach

Funkcja COUNT()

Funkcja COUNT() zwraca liczbę przekazanych jako argument wywołania wartości. Używa się jej do sprawdzenia liczby grup lub policzenia wierszy tabeli. Na przykład poniższe zapytanie zwraca liczbę wszystkich wierszy tabeli (kolumna SalesOrderID jest kluczem podstawowym, a więc nie mogą wystąpić w niej wartości NULL):

```
SELECT COUNT([SalesOrderID])
FROM [SalesLT].[SalesOrderHeader];
-----
```

33

Domyślnie funkcje grupujące są wywoływane dla wszystkich wartości w grupie z wyjątkiem wartości NULL. Tak więc wynikiem poniższego zapytania jest liczba zamówień złożonych przez wszystkich klientów. Ponieważ każde zamówienie zostało złożone przez jakiegoś klienta, otrzymaliśmy w wyniku liczbę wszystkich zamówień:

```
SELECT COUNT(CustomerID)
FROM [SalesLT].[SalesOrderHeader];
-----
```

33

Funkcje grupujące mogą być też wywoływane tylko dla

wartości niepowtarzających się w grupie. Po poprzedzeniu argumentu funkcji grupującej (nazwy kolumny) słowem kluczowym DISTINCT zapytanie zwróci liczbę klientów, którzy złożyli przynajmniej jedno zamówienie:

```
SELECT COUNT(DISTINCT CustomerID)  
FROM [SalesLT].[SalesOrderHeader];
```

32

Słowo kluczowe DISTINCT jest częścią argumentu funkcji grupującej i z tego powodu zostało umieszczone w nawiasie. Gdybyśmy przenieśli je przed nawias, oznaczałoby to, że z wyniku zapytania mają być usunięte powtarzające się wiersze. Ponieważ zapytanie zwraca tylko jeden wiersz, próba wyeliminowania duplikatów jest nielogiczna i nie ma żadnego wpływu na wynik, chociaż składnia zapytania jest poprawna:

```
SELECT DISTINCT COUNT(CustomerID)  
FROM [SalesLT].[SalesOrderHeader];
```

33

Zliczanie wierszy

Funkcja COUNT() może być wywołana z * jako argumentem — zwraca wtedy liczbę wierszy tabeli lub grupy:

```
SELECT COUNT(*)  
FROM [SalesLT].[SalesOrderHeader];
```

33

Wyjątkowość składni COUNT(*) polega na tym, że **zliczane są nie tylko duplikaty wierszy, lecz także puste wiersze**, czyli funkcja COUNT() wywołana z * jako jedyna funkcja grupująca uwzględnia wartość NULL. Z tego powodu symbolu * nie można poprzedzić słowem kluczowym DISTINCT:

```
SELECT COUNT(DISTINCT *)  
FROM [SalesLT].[SalesOrderHeader];
```

Msg 102, Level 15, State 1, Line 24 Incorrect syntax near '*'.

Funkcje SUM() i AVG()

Argumentami funkcji SUM() i AVG() mogą być tylko liczby. Pierwsza z tych funkcji zwraca sumę wartości w grupie, druga — średnią wartości w grupie.

Poniższe zapytanie zwraca całkowitą wartość i średnią cenę wszystkich produktów:

```
SELECT SUM([ListPrice]), AVG([ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
219778,59      742,4952
```

Aby przekonać się, że funkcje te ignorują wartość NULL, utworzymy tabelę tymczasową i wstawimy do niej dwa wiersze:

```
CREATE TABLE #Tab
(id INT IDENTITY,
val INT);
GO
INSERT INTO #TAB (val)
VALUES
(2),(4);
```

Sprawdźmy zawartość naszej tabeli:

```
SELECT *
FROM #TAB;
```

```
-----  
1      2  
2      4
```

Suma wartości z kolumn val wynosi $2 + 4$, czyli 6, a średnia $(2 + 4) / 2$, czyli 3, co potwierdza poniższy wynik:

```
SELECT SUM(val), AVG(val)
FROM #TAB;
```

```
-----  
6      3
```

Jeżeli teraz wstawimy do tej tabeli kolejny wiersz, w którym wartość

`val` będzie nieokreślona, wynik funkcji `AVG(val)` nie zmieni się i nadal będzie wynosił 3, chociaż teraz tabela zawiera trzy, a nie dwa wiersze:

```
INSERT INTO #TAB(val)
VALUES (NULL);
SELECT AVG(val), COUNT(*), COUNT(val)
FROM #TAB;
```

```
-----  
3      3      2
```

Pomimo że wszystkie funkcje grupujące mogą uwzględniać tylko niepowtarzalne wartości, słowo kluczowe `DISTINCT` jest używane prawie wyłącznie z funkcją `COUNT()` — rzadko kiedy interesuje nas nie suma wartości produktów, a suma ich niepowtarzalnych wartości:

```
SELECT SUM([ListPrice]), SUM(DISTINCT [ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
219778,59      44698,24
```

Funkcje `MIN()` i `MAX()`

Argumentami funkcji `MIN()` i `MAX()` mogą być oprócz danych liczbowych dane daty i czasu oraz tekstowe. Pierwsza z tych funkcji zwraca najmniejszą, druga — największą wartość w grupie, przy czym:

1. W przypadku dat za najmniejszą uznawana jest najwcześniejsza, a za największą najpóźniejsza data w grupie:

```
SELECT MIN([OrderDate]), MAX([OrderDate])
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
2008-06-01 00:00:00.000      2014-04-01 00:00:00.000
```

2. Dla liczb i walut funkcja `MIN` zwraca wartość najmniejszą, a funkcja `MAX` — największą:

```
SELECT MIN([TotalDue]), MAX([TotalDue])
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
43,0437      119960,824
```

3. Przy porównywaniu ciągów znaków obowiązują takie same zasady jak przy ich sortowaniu w porządku rosnącym:

```
SELECT MIN(Name), MAX(Name)  
FROM [SalesLT].[Product];
```

```
-----  
All-Purpose Bike      Stand      Women's Tights, S
```

Używając poznanych funkcji grupujących, możemy sprawdzić, dlaczego rozwiązanie ostatniego zadania z rozdziału 4. wymagało użycia klauzuli ORDER BY, a nie WHERE:

```
SELECT MIN(ID), MAX(ID), COUNT(*), MAX(ID)-MIN(ID)-COUNT(*)  
FROM dbo.Produkty;
```

```
-----  
680      1000      296      24
```

Jak widać, choć tabela liczy 296 wierszy, to produkty mają numery z zakresu od 680 do 1000, a więc 24 numery są niewykorzystane.

Inne funkcje grupujące

Oprócz opisanych powyżej pięciu podstawowych funkcji grupujących niektóre serwery bazodanowe pozwalają używać kilku innych, mniej popularnych. W przypadku serwera SQL Server są to:

1. Funkcja **CHECKSUM_AGG()**, której argumentem muszą być liczby całkowite, zwraca sumę kontrolną wartości w grupie (ta funkcja jest używana głównie do porównywania grup, np. sprawdzenia, czy nie zmieniły się ceny produktów):

```
SELECT CHECKSUM_AGG([SalesOrderID]),  
CHECKSUM_AGG(CAST([UnitPrice] AS INT))  
FROM [SalesLT].[SalesOrderDetail];
```

```
-----  
267      19
```

2. Funkcja **COUNT_BIG()** od funkcji **COUNT()** różni się tylko typem zwracanych wartości — w jej przypadku liczba wierszy w grupie jest zwracana jako dane typu **BIGINT**.
3. Opisana w dalszej części rozdziału funkcja **GROUPING()** zwraca 1 lub 0, w zależności od tego, czy dana grupa została dodana

do wyniku zapytania przez operatory CUBE, czy ROLLUP.

4. Funkcja STDEV() zwraca odchylenie standardowe wartości w grupie:

```
SELECT AVG([ListPrice]), STDEV([ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
742,4952      891,      781241845919
```

5. Funkcja STDEVP() zwraca odchylenie standardowe próbki wartości w grupie:

```
SELECT STDEV([ListPrice]), STDEVP([ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
891,781241845919      890,273580172876
```

6. Funkcja VAR() zwraca wariancję wartości w grupie:

```
SELECT AVG([ListPrice]), VAR([ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
742,4952          795273,78330825
```

7. Funkcja VARP() zwraca wariancję próbki wartości w grupie:

```
SELECT VAR([ListPrice]), VARP([ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
795273,78330825      792587,04755383
```

Wyrażenia

W jednym z wcześniejszych przykładów wyniki funkcji grupujących były od siebie odejmowane. W rzeczywistości funkcje grupujące mogą być częścią dowolnych wyrażeń, o ile tylko te wyrażenia są poprawne składniowo, czyli np. zwracają pojedyncze wartości właściwego typu.

Poniższe zapytanie zwraca różnice pomiędzy minimalną i maksymalną oraz maksymalną i średnią ceną produktów:

```
SELECT MAX([ListPrice]) - MIN([ListPrice]),
```

```
MAX([ListPrice]) - AVG( [ListPrice])
FROM [SalesLT].[Product];
-----
```

```
3575,98      2835,7748
```

Argumentami funkcji grupujących również mogą być wyrażenia. Na przykład możemy policzyć średnią z cen brutto produktów (cen pomnożonych przez 1,23):

```
SELECT AVG([ListPrice]*1.23)
FROM [SalesLT].[Product];
-----
```

```
913.269140
```

Oznacza to jednak, że najpierw pomnożymy ceny wszystkich produktów przez określoną stałą (w tym przypadku 1,23), a następnie na podstawie tak otrzymanych wyników obliczymy średnią. Jeżeli serwer bazodanowy nie zoptymalizuje planu wykonania tego zapytania, wykona 296 mnożeń (jedno dla każdego produktu), a później raz policzy średnią. Tymczasem mnożenie wartości przez stałą można wykonać tylko raz, już po obliczeniu średniej, a otrzymany wynik będzie taki sam:

```
SELECT AVG([ListPrice])*1.23
FROM [SalesLT].[Product];
-----
```

```
913.269140
```

Zagnieżdżanie funkcji grupujących

Standard SQL3 nie zezwala na zagnieżdżanie funkcji grupujących, tak więc argumentem funkcji grupującej nie może być wynik innej funkcji grupującej:

```
SELECT AVG(SUM([ListPrice]))
FROM [SalesLT].[Product];
-----
```

```
Msg 130, Level 15, State 1, Line 102
```

```
Cannot perform an aggregate function on an expression containing an
aggregate or a subquery.
```

Klauzula GROUP BY

Funkcje grupujące mogą być wywoływanne dla określonych grup, a nie tylko dla całych tabel. Podział wierszy na logiczne grupy umożliwia właśnie klauzula GROUP BY, przy czym najczęściej wiersz jest klasyfikowany do danej grupy na podstawie wartości kolumny innej niż ta, dla której wywoływana jest funkcja grupująca.

Na przykład żeby odczytać średnią cenę i liczbę towarów należących do poszczególnych kategorii, należy pogrupować wiersze na podstawie identyfikatora lub nazwy kategorii i wyliczyć średnią dla każdej grupy. Możemy to zrobić, wybierając w każdym zapytaniu dane z jednej kategorii, obliczając średnią oraz zliczając produkty i łącząc tak otrzymane wyniki operatorem UNION ALL. Klauzula GROUP BY pozwala jednak znacznie uprościć takie zapytanie:

```
SELECT AVG([ListPrice]) AS Średnia,  
COUNT([ProductID]) AS Liczba  
FROM [SalesLT].[Product]  
GROUP BY [ProductCategoryID];
```

Średnia	Liczba
123,00	1
1683,365	32
1597,45	43
1425,2481	22
73,89	8
92,24	3
106,50	2

...

Wynik zapytania liczy tyle wierszy, do ilu kategorii mogą należeć produkty, a funkcje grupujące zostały wywołane raz dla każdej grupy. Skoro liczba wierszy wyniku odpowiada liczbie kategorii, możemy dodać do wyniku zapytania użytą do grupowania kolumnę:

```
SELECT [ProductCategoryID],  
AVG([ListPrice]) AS Średnia,  
COUNT([ProductID]) AS Liczba
```

```
FROM [SalesLT].[Product]
GROUP BY [ProductCategoryID];
```

ProductCategoryID	Średnia	Liczba
NULL	123,00	1
5	1683,365	32
6	1597,45	43
7	1425,2481	22
8	73,89	8
9	92,24	3
10	106,50	2

Teraz wyraźnie widać, że serwer bazodanowy zaklasyfikował wiersze do poszczególnych grup na podstawie nazwy kategorii — trafiły do nich produkty należące do tych samych kategorii, a następnie dla każdej grupy została wywołana funkcja grupująca.

Gdybyśmy jednak spróbowali dodać do wyniku zapytania kolumnę z kodami produktów:

```
SELECT [ProductCategoryID], AVG([ListPrice]), [Name]
FROM [SalesLT].[Product]
GROUP BY [ProductCategoryID];;
```

```
-----  
Msg 8120, Level 16, State 1, Line 111
```

```
Column 'SalesLT.Product.Name' is invalid in the select list because it is
not contained in either an aggregate function or the GROUP BY clause.
```

próba wykonania zapytania skończyłaby się błędem, którego opis powinniśmy zapamiętać jako praktyczną wskazówkę: **jeżeli zapytanie zawiera klauzulę GROUP BY, w klauzuli SELECT dopuszczalne są wyłącznie funkcje grupujące oraz nazwy kolumn lub wyrażenia użyte do pogrupowania danych, czyli występujące w klauzuli GROUP BY.**

Z tego powodu poniższe zapytanie również jest nieprawidłowe:

```
SELECT P.[ProductNumber], SUM(OD.[LineTotal])
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
    ON P.ProductID = OD.ProductID;
```

Msg 8120, Level 16, State 1, Line 116

Column 'SalesLT.Product.Name' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

Chociaż w bazie *AdventureWorksLT2012* numer produktu jednoznacznie identyfikuje poszczególne produkty, jeżeli w klauzuli SELECT użyliśmy kolumny ProductNumber, musimy ją też umieścić w klauzuli GROUP BY:

```
SELECT P.[ProductNumber], SUM(OD.[LineTotal])
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
    ON P.ProductID = OD.ProductID
GROUP BY P.[ProductNumber];
```

ProductNumber	(No column name)
BB-7421	226.758000
BB-9108	1093.410000
BK-M18B-40	971.982000
BK-M18B-42	971.982000
BK-M18B-44	1619.970000
BK-M18B-48	3887.928000

...

Kolejny przykład pokazuje, jak do grupowania danych używać wyrażeń. Założymy, że chcemy sztucznie zmniejszyć liczbę grup otrzymanych na podstawie kolorów różnych produktów:

```
SELECT [Color], COUNT(*)
FROM [SalesLT].[Product]
GROUP BY [Color];
```

Color	(No column name)
NULL	50
Black	89
Blue	26
Grey	1

Multi	8
Red	38
Silver	36
Silver/Black	7
White	5
Yellow	36

Początkowo lista otrzymanych w ten sposób grup liczy 10 pozycji. Liczbę grup możemy zmniejszyć następująco:

```
SELECT LEFT([Color],1), COUNT(*)
FROM [SalesLT].[Product]
GROUP BY LEFT([Color],1);
```

(No column name)	(No column name)
NULL	50
B	115
G	1
M	8
R	38
S	43
W	5
Y	36

Klauzula GROUP BY umożliwia również tworzenie podgrup. Przypuśćmy, że chcemy poznać liczbę oraz średnią cenę produktów z poszczególnych kategorii z rozbiciem na kolory tych produktów. W tym celu musimy w klauzuli GROUP BY umieścić nazwę dodatkowej kolumny:

```
SELECT C.Name, P.Color, COUNT(*) AS Liczba, AVG(ListPrice) AS Średnia
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
GROUP BY C.Name, P.Color
ORDER BY C.Name;
```

Name	Color	Liczba	Średnia
------	-------	--------	---------

Gloves	Black	6	31,24
Handlebars	NULL	8	73,89
Headsets	NULL	3	87,0733
Helmets	Black	1	34,99
Helmets	Blue	1	34,99
Helmets	Red	1	34,99

...

W wyniku tego zapytania nazwa kategorii powtarza się tyle razy, ile produktów o różnych kolorach do niej należy, a więc zawiera ona grupy (nazwy kategorii) i podgrupy (nazwy kolorów).

W większości serwerów bazodanowych klauzula `GROUP BY` musi być pominięta, jeżeli chcemy wszystkie wiersze zaklasyfikować do jednej grupy (sytuacja taka miała miejsce w przykładach omawiających poszczególne funkcje grupujące). W rezultacie takie zapytania jak poniższe:

```
SELECT COUNT( CustomerID )
FROM [SalesLT].[SalesOrderHeader];
```

33

choćiąż grupuję dane, to nie zawierają klauzuli `GROUP BY`. Z tego powodu są nazywane zapytaniami niejawnie grupującymi dane. Serwer SQL pozwala na umieszczenie w tego typu zapytaniach pustej klauzuli `GROUP BY`:

```
SELECT COUNT( CustomerID )
FROM [SalesLT].[SalesOrderHeader]
GROUP BY ();-----
```

33

Kolejność wykonywania klauzuli `GROUP BY`

Skoro poznałeś nową klauzulę, powinieneś wiedzieć, kiedy jest ona wykonywana przez serwery bazodanowe. Logiczna kolejność wykonywania zapytania zawierającego wszystkie poznane do tej pory klauzule jest następująca:

1. Jako pierwsza wykonywana jest klauzula `FROM`. Jeżeli zapytanie odwołuje się do wielu tabel, są one kolejno ze sobą

złączane.

2. Otrzymany w ten sposób zbiór pośredni jest filtrowany na podstawie warunku logicznego umieszczonego w klauzuli WHERE. Tylko te wiersze, dla których jest on prawdziwy, trafiają do kolejnego zbioru pośredniego.
3. Następnie wykonywana jest klauzula GROUP BY, czyli grupowane są tylko przefiltrowane wiersze.
4. Po zgrupowaniu wykonywana jest klauzula SELECT.
5. W dalszej kolejności serwer sortuje wiersze, czyli wykonywana jest klauzula ORDER BY.
6. Na końcu, podczas wykonywania klauzuli TOP, liczba wierszy wyniku zapytania zostaje ograniczona.

Fakt, że klauzula może odwołać się wyłącznie do zbioru pośredniego będącego rezultatem wykonania poprzedniej klauzuli, ma trzy interesujące nas w tej chwili konsekwencje:

1. Po pierwsze w klauzuli WHERE nie można umieścić funkcji grupujących (podczas wykonywania tej klauzuli dane nie są jeszcze pogrupowane):

```
SELECT [ProductCategoryID], AVG([ListPrice])
FROM [SalesLT].[Product]
WHERE AVG([ListPrice])>5
GROUP BY [ProductCategoryID];
```

Msg 147, Level 15, State 1, Line 157
An aggregate may not appear in the WHERE clause unless it is in a subquery contained in a HAVING clause or a select list, and the column being aggregated is an outer reference.

2. Po drugie wiersze wyeliminowane w klauzuli WHERE nie zostaną pogrupowane:

```
SELECT Color, AVG([ListPrice])
FROM [SalesLT].[Product]
WHERE Color LIKE 'S%'
GROUP BY Color;
```

Color	(No column name)
Silver	1015,6425
Silver/Black	64,0185

3. Po trzecie dane są najpierw grupowane, a potem sortowane, czyli w klauzuli ORDER BY możemy umieścić tylko kolumny lub wyrażenia użyte do grupowania bądź funkcje grupujące:

```
SELECT Color, AVG([ListPrice])
FROM [SalesLT].[Product]
WHERE Color LIKE 'S%'
GROUP BY Color
ORDER BY [Name];
```

Msg 8127, Level 16, State 1, Line 5

Column "SalesLT.Product.Name" is invalid in the ORDER BY clause because it is not contained in either an aggregate function or the GROUP BY clause.

Operatory CUBE i ROLLUP

Jeżeli dane są grupowane według wartości kilku kolumn, to — jak pokazały wcześniejsze przykłady — kolejność ich występowania w klauzuli GROUP BY wyznacza podział na grupy i podgrupy. Jeśli tych grup i podgrup jest niewiele, użytkownicy będą potrafili samodzielnie wyliczyć brakujące podsumowania, np. sumy cen produktów z poszczególnych kategorii sprzedanych w dwóch kolejnych miesiącach. Możemy jednak dodać do wyniku zapytania takie sumy pośrednie — służy do tego operator ROLLUP.

 Wskazówka	Operatory CUBE i ROLLUP mają ograniczoną funkcjonalność — te same rezultaty, ale szybciej i z pełną możliwością kontrolowania tworzonych podsumowań oferuje operator GROUPING SETS. Z tych powodów operatory CUBE i ROLLUP nie powinny być używane w nowych aplikacjach.
--	--

Poniższe zapytanie zwraca więcej (86) wierszy niż to samo zapytanie bez operatora ROLLUP. Te dodatkowe wiersze zawierają sumy pośrednie, w tym wypadku liczbę wszystkich produktów z danej kategorii oraz sumę ich cen:

```
SELECT C.Name, P.Color, COUNT(*) AS Liczba, SUM(ListPrice) AS Średnia
```

```

FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
GROUP BY ROLLUP (C.Name, P.Color)
ORDER BY C.Name;
-----
```

Name	Color	Liczba	Średnia
Mountain Bikes	Black	16	27404,84
Mountain Bikes	Silver	16	26462,84
Mountain Bikes	NULL	32	53867,68
Mountain Frames	Black	14	9391,99
Mountain Frames	Silver	14	9599,11
Mountain Frames	NULL	28	18991,10
Panniers	Grey	1	125,00
Panniers	NULL	1	125,00

...

Operator `ROLLUP` dodał podsumowania dla grup utworzonych na podstawie wartości pierwszej z kolumn wymienionych w klauzuli `GROUP BY`. Dodane przez niego wiersze zawierają wartości `NULL` w kolumnach użytych do grupowania, natomiast w ostatniej kolumnie tych wierszy znajdziemy sumy pośrednie:

1. Trzeci wiersz zawiera podsumowanie dla kategorii Mountain Bikes.
2. Szósty wiersz zawiera podsumowanie dla kategorii Mountain Frames.
3. Ostatni pokazany wiersz zawiera podsumowanie dla kategorii Panniers. Ponieważ do tej kategorii należą wyłącznie szare produkty, wartości w tym wierszu są takie same jak w poprzedzającym go wierszu.

Jako że liczba możliwych sposobów pogrupowania danych rośnie wykładniczo wraz ze wzrostem liczby kolumn wymienionych w klauzuli `GROUP BY` (dla trzech kolumn jest ich 8, dla czterech już 16), przygotowanie wielu wersji tego samego zapytania jest w praktyce

niewykonalne. Dzięki operatorowi CUBE możemy uniknąć pisania wielu wersji tego samego zapytania — jego działanie polega bowiem na dodaniu do wyniku zapytania podsumowań dla wszystkich możliwych kombinacji grup i podgrup:

```
SELECT C.Name, P.Color, COUNT(*) AS Liczba, SUM(ListPrice) AS Średnia  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON P.ProductCategoryID=C.ProductCategoryID  
GROUP BY CUBE (C.Name, P.Color)  
ORDER BY C.Name;
```

Name	Color	Liczba	Średnia
NULL	NULL	50	2750,12
NULL	Black	89	67436,26
NULL	Grey	1	125,00
NULL	Multi	8	478,92
NULL	Red	38	53274,10
NULL	Yellow	36	34527,29
NULL	NULL	295	219655,59
NULL	Blue	26	24015,66
NULL	Silver	36	36563,13
NULL	White	4	36,98
...			

Tym razem wynik liczy 96, a nie 86 wierszy. Dodatkowe dziesięć wierszy dodanych przez operator CUBE zawiera sumy pośrednie policzone dla kolorów, np. drugi wiersz zawiera liczbę i sumę cen wszystkich czarnych produktów, niezależnie od kategorii, do jakiej one należą. Dodane przez operator CUBE wiersze z podsumowaniami można rozpoznać po wartościach NULL występujących w kolumnach, według których pogrupowane zostały dane:

1. Wartość NULL w kolumnie Color oznacza podsumowanie dla koloru.
2. Wartość NULL w kolumnie Name — podsumowanie dla kategorii.

3. Wartość NULL w obu użytych do grupowania kolumnach oznacza podsumowanie całego zapytania.

Funkcje GROUPING i GROUPING_ID

Oznaczanie dodatkowych grup za pomocą wartości NULL jest niejednoznaczne — gdyby w użytych do grupowania kolumnach występowała wartość nieokreślona, wiersze z wartością NULL byłyby nie do odróżnienia od wierszy zawierających sumy pośrednie.

Serwer SQL rozwiązuje ten problem za pomocą:

1. Funkcji GROUPING(), która zwraca 1, jeżeli dany wiersz zawiera podsumowania dodane do wyniku przez operatory CUBE lub ROLLUP, i 0, jeżeli dany wiersz zawiera wartości wyliczone dla danej grupy:

```
SELECT C.Name, GROUPING(C.Name) AS ByName,
P.Color, GROUPING(P.Color) AS ByColor,
AVG(ListPrice) AS Średnia
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
GROUP BY CUBE (C.Name, P.Color)
ORDER BY C.Name;
```

Name Średnia	ByName	Color	ByColor	
NULL 923,6792	1	Blue	0	
NULL 1015,6425	1	Silver	0	
NULL	1	White	0	9,245
NULL 64,0185	1	Silver/Black	0	
Bib-Shorts	0	NULL	1	89,99
Bib-Shorts	0	Multi	0	89,99
Bike Racks 120,00	0	NULL	0	

- ...
2. Funkcji GROUPING_ID(), która zwraca numer poziomu podsumowania, przy czym 0 oznacza wiersz zawierający wartości wyliczone dla grupy, a największy numer — wiersz zawierający podsumowanie całego zapytania:

```

SELECT C.Name, GROUPING_ID(C.Name,P.Color) AS GroupLevel,
P.Color, AVG(ListPrice) AS Średnia
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
GROUP BY CUBE (C.Name, P.Color)
ORDER BY C.Name;
-----
```

Name	GroupLevel	Color	Średnia
NULL 744,5952	3	NULL	
NULL 923,6792	2	Blue	
NULL 1015,6425	2	Silver	
NULL	2	White	9,245
NULL	2	Silver/Black	64,0185
Bib-Shorts	1	NULL	89,99
Bib-Shorts	0	Multi	89,99

...

Operator GROUPING SETS

Klauzula GROUP BY pozwala zdefiniować hierarchię grup, dla których będą wywoływane funkcje grupujące — w ten sposób wiele złączonych operatorem UNION ALL zapytań możemy zastąpić jednym. Jeżeli jednak serwer bazodanowy nie obsługuje operatorów CUBE, ROLLUP lub GROUPING SETS, to chcąc pogrupować te same dane na różne sposoby, nadal musimy napisać kilka zapytań i złączyć ich wyniki.

Wydajność i możliwości operatora GROUPING SETS przedstawimy przy użyciu składającej się z czterech kolumn (kolumny a, b, c będą użyte

do grupowania, kolumna x zawiera sumowane wartości) tabeli, do której wstawimy cztery wiersze:

```
CREATE TABLE #T
(
    a int,
    b int,
    c int,
    x int
);
INSERT #T VALUES
    (1,2,3,8),
    (1,2,4,1),
    (1,2,5,5),
    (1,3,4,9);
GO
```

```
-----  
(4 row(s) affected)
```

Policzyć sumy wartości x dla grup zdefiniowanych na podstawie wartości kolumny a (ponieważ zawiera ona tylko jedną wartość, będzie to suma wszystkich wartości x), b (w tym wypadku będą to dwie sumy) możemy za pomocą dwóch złączonych operatorem UNION ALL zapytań. Dodatkowo do wyniku całego zapytania dodamy wiersze ze szczególnowymi danymi (grupując dane według wartości kolumn a , b i c , uzyskamy jedną grupę dla jednego wiersza):

```
SELECT a, NULL as b, NULL as c, sum(x) as sumx
FROM #T
GROUP BY a
UNION ALL
SELECT NULL as a, b, NULL as c, sum(x) as sumx
FROM #T
GROUP BY b
UNION ALL
SELECT a, b, c, sum(x) as sumx
FROM #T
```

```
GROUP BY a, b, c;
```

```
-----  
a      b      c      sumx  
1      NULL    NULL    23  
NULL   2       NULL    14  
NULL   3       NULL    9  
1      2       3       8  
1      2       4       1  
1      2       5       5  
1      3       4       9
```

Takie rozwiązanie wymaga pisania długich zapytań, lecz przede wszystkim jest mało wydajne. Serwer bazodanowy wielokrotnie (w tym przypadku trzykrotnie) odczytuje te same dane, co przy dużych tabelach wiąże się z długim czasem wykonania zapytania. Gdybyśmy mogli określić kilka sposobów grupowania danych w jednym zapytaniu, znalezlibyśmy jedno rozwiązanie dla obu tych problemów.

Możliwość zdefiniowania wielu metod grupowania tych samych danych daje nam operator GROUPING SETS. Poniższe zapytanie zwraca te same, pogrupowane na trzy sposoby dane — tym razem tabele źródłowe zostały odczytane tylko raz, dzięki czemu czas wykonania zapytania skrócił się trzykrotnie:

```
SELECT a, b, c, SUM(x) AS sumx  
FROM #T  
GROUP BY  
      GROUPING SETS (a,  
                     (b),  
                     (a, b, c));
```

```
-----  
a      b      c      sumx  
1      2       3       8  
1      2       4       1  
1      2       5       5  
NULL   2       NULL    14  
1      3       4       9
```

NULL	3	NULL	9
1	NULL	NULL	23



Każdy porządek grupowania wyznaczony przez dowolną liczbę wyrażeń lub nazw tabel jest umieszczany w nawiasie, a przecinki między nawiasami służą do oddzielania od siebie poszczególnych porządków grupowania.

Żeby w pełni docenić możliwości i wydajność operatora GROUPING SETS, wystarczy porównać ze sobą dwa poniższe, zwracające te same dane, zapytania:

```
SELECT [CountryRegion], [StateProvince],[City],COUNT(*)  
FROM [SalesLT].[Address]  
GROUP BY [CountryRegion],[StateProvince],[City]  
UNION ALL  
SELECT [CountryRegion], [StateProvince],NULL,COUNT(*)  
FROM [SalesLT].[Address]  
GROUP BY [CountryRegion],[StateProvince]  
UNION ALL  
SELECT [CountryRegion], NULL,[City],COUNT(*)  
FROM [SalesLT].[Address]  
GROUP BY [CountryRegion],[City];  
  
SELECT [CountryRegion], [StateProvince],[City],COUNT(*)  
FROM [SalesLT].[Address]  
GROUP BY GROUPING SETS (  
    ([CountryRegion],[StateProvince],[City]),  
    ([CountryRegion],[StateProvince]),  
    ([CountryRegion],[City] )  
);
```

Podsumowując — klauzula GROUP BY umożliwia wskazanie jednego porządku grupowania, a operator GROUPING SETS rozszerza jej funkcjonalność, umożliwiając zdefiniowanie w jednym zapytaniu wielu różnych porządków grupowania.

Operatory PIVOT i UNPIVOT

Serwer SQL pozwala na przekształcanie wierszy w kolumny i odwrotnie, czyli na pracę z tabelami przestawnymi. Pierwsza operacja pozwala poprawić czytelność wyników zapytań, grupujących dane (zmniejszając liczbę wierszy, poprawiamy czytelność raportów), druga — dostosować dane pochodzące z arkuszy kalkulacyjnych do struktury tabel bazodanowych.

PIVOT

Operator PIVOT przekształca dane z wierszy w kolumny; jednocześnie grupuje je i wywołuje dla każdej grupy wskazaną funkcję grupującą. Otrzymany w ten sposób zbiór pośredni przypomina tabelę przestawną dostępną m.in. w programie Excel.

Dzięki zastąpieniu wskazanych wierszy kolumnami tabela przestawna jest znacznie mniejsza i czytelniej prezentuje dane pogrupowane według dwóch kolumn — zamiast wyświetlać wartości podgrup w kolejnych wierszach, tabela umieszcza wynik funkcji grupującej na przecięciu kolumny wyznaczającej jeden porządek grupowania z wierszem wyznaczającym drugi porządek grupowania.

Pokazuje to poniższy przykład:

1. Oryginalne zapytanie odczytujące nazwy wybranych kategorii, nazwiska osób, które kupiły produkt z danej kategorii, i wartość sprzedanych tej osobie produktów zwraca 20 wierszy:

```
SELECT TOP 20 CAT.Name, C.LastName, SUM(OD.LineTotal) AS Total  
FROM [SalesLT].[SalesOrderDetail] AS OD  
JOIN [SalesLT].[SalesOrderHeader] AS H  
    ON OD.SalesOrderID = H.SalesOrderID  
JOIN [SalesLT].[Product] AS P  
    ON P.ProductID=OD.ProductID  
JOIN [SalesLT].[ProductCategory] AS CAT  
    ON CAT.ProductCategoryID=P.ProductCategoryID  
JOIN [SalesLT].[Customer] AS C  
    ON C.CustomerID=H.CustomerID  
GROUP BY CAT.Name, C.LastName
```

```
ORDER BY CAT.Name;
```

Name	LastName	Total
Bike Racks	Abel	216.000000
Bike Racks	Beck	216.000000
Bike Racks	Byham	216.000000
Bike Racks	Eminhizer	432.000000
Bike Racks	Grande	720.000000
Bike Racks	Kurtz	288.000000
Bike Racks	Liu	144.000000
Bike Racks	Marple	72.000000
Bottles and Cages	Abel	17.964000
Bottles and Cages	Beck	23.952000
Bottles and Cages	Eminhizer	31.199476
Bottles and Cages	Gilbert	2.994000
Bottles and Cages	Grande	31.199476
Bottles and Cages	Kurtz	20.958000
Bottles and Cages	Liu	29.940000
Bottom Brackets	Abel	323.970000
Bottom Brackets	Campbell	218.682000
Bottom Brackets	Laszlo	388.758000
Bottom Brackets	Sunkammurali	388.758000
Brakes	Abel	63.900000

2. Grupy i podgrupy oraz wyliczone dla nich sumy są zapisane w kolejnych wierszach. W rezultacie liczba zwróconych przez zapytanie wierszy szybko rośnie, utrudniając odczytanie i interpretację wyniku. Lepiej by było, gdyby nazwiska były zwracane w kolejnych wierszach, a nazwy kategorii — w kolumnach. Wtedy na przecięciu wiersza i kolumny można by umieścić wartość sprzedaży produktów z danej kategorii danemu klientowi. Ponieważ operator PIVOT wymaga przygotowania danych źródłowych (dane te nie mogą zawierać dodatkowych kolumn), wynik poprzedniego zapytania zapiszemy najpierw do tabeli tymczasowej^[1]:

```
SELECT TOP 20 CAT.Name, C.LastName, SUM(OD.LineTotal) AS Total
```

```

INTO #TabPivot
FROM [SalesLT].[SalesOrderDetail] AS OD
JOIN [SalesLT].[SalesOrderHeader] AS H
    ON OD.SalesOrderID = H.SalesOrderID
JOIN [SalesLT].[Product] AS P
    ON P.ProductID=OD.ProductID
JOIN [SalesLT].[ProductCategory] AS CAT
    ON CAT.ProductCategoryID=P.ProductCategoryID
JOIN [SalesLT].[Customer] AS C
    ON C.CustomerID=H.CustomerID
GROUP BY CAT.Name, C.LastName
ORDER BY CAT.Name;
-----  

(20 row(s) affected)

```

3. Przekształcanie wierszy na kolumny przebiega w trzech etapach:

- W pierwszej kolejności dane są grupowane według wartości tej kolumny tabeli źródłowej, która — odwołując się do terminologii związanej z tabelami przestawnymi — będzie zawierała nagłówki wierszy. **Nazwa tej kolumny nie pojawia się na liście parametrów operatora PIVOT**, dlatego tego typu operacja jest nazywana niejawnym grupowaniem.
- Następnie jawnie wskazane dane są przekształcane w nagłówki kolumn tabeli przestawnej. Do tak utworzonych kolejnych kolumn są kopijowane dane ze wskazanej kolumny tabeli bazowej. Gdybyśmy mogli utworzyć kolumny wyniku na podstawie odczytanych z tabeli bazowej danych, uzyskalibyśmy dynamiczne tabele przestawne, czyli takie, których liczba kolumn jest zmienna i zależy od zwracanych przez zapytanie danych. Niestety, serwer SQL nie pozwala na tworzenie tego typu zapytań.
- Ostatnim etapem jest wywołanie dla każdego pola tak utworzonej tabeli przestawnej określonej funkcji grupującej:

```

SELECT P.LastName, [Bike Racks],[Bottles and Cages] ,[Bottom
Brackets], [Brakes]
FROM #TabPivot
PIVOT (
SUM(Total)
FOR Name IN ([Bike Racks],[Bottles and Cages] ,[Bottom
Brackets], [Brakes]) ) AS P
ORDER BY P.LastName;
-----
```

LastName Brackets	Bike Racks Brakes	Bottles and Cages	Bottom
Abel 63.900000	216.000000	17.964000	323.970000
Beck NULL	216.000000	23.952000	NULL
Byham NULL	216.000000	NULL	NULL
Campbell NULL	NULL	NULL	218.682000
Eminhizer NULL	432.000000	31.199476	NULL
Gilbert NULL	NULL	2.994000	NULL
Grande NULL	720.000000	31.199476	NULL
Kurtz NULL	288.000000	20.958000	NULL
Laszlo NULL	NULL	NULL	388.758000
Liu NULL	144.000000	29.940000	NULL
Marple NULL	72.000000	NULL	NULL
Sunkammurali NULL	NULL	NULL	388.758000

4. Tabela przestawna zawierająca te same dane liczy 12 wierszy (tyle, ile było osób, które kupiły produkty z wybranych kategorii), a odczytanie wartości sprzedaży produktu o danym numerze w wybranym miesiącu sprowadza się do

sprawdzenia wartości pola znajdującego się na przecięciu wiersza z danymi tego produktu i kolumny z danymi o sprzedaży w określonym miesiącu.

Podsumowując:

1. W klauzuli SELECT zdefiniowaliśmy listę kolumn tabeli przestawnej: pierwsza kolumna zawiera identyfikatory produktów, druga — sumę sprzedaży w styczniu, trzecia — w lutym i tak dalej.
2. Następnie za pomocą operatora PIVOT:
 - a. Określiliśmy funkcję grupującą i kolumnę zawierającą jej argumenty SUM(Total).
 - b. Wskazaliśmy kolumnę bazową (FOR Name IN), z której odczytane dane zostały umieszczone w kolejnych kolumnach o jawnie podanych nazwach.
 - c. Niewymieniona kolumna LastName została użyta do pogrupowania danych, a więc kolejne wiersze wyniku zawierają informacje o sprzedaży poszczególnym klientom.

UNPIVOT

Działanie tego operatora polega na odwróceniu wyniku operatora PIVOT, a więc na zamianie kolumny na wiersze i rozbiciu niejawnie utworzonych w tabeli przestawnej grup.

Przekształcanie kolumny na wiersze również przebiega w trzech etapach:

1. Najpierw generowane są duplikaty wartości kolumny wskazanej w bloku IN.
2. Następnie tworzona jest kolumna o nazwie wskazanej w bloku FOR.
3. Na końcu z wyniku usuwane są wartości NULL.

Usunięcie wartości NULL powoduje, że operatory PIVOT i UNPIVOT są



niesymetryczne — po przekształceniu wierszy na kolumny i kolumn z powrotem na wiersze wynik zapytania może zawierać mniej danych.

Działanie operatora UNPIVOT pokazuje kolejny przykład:

1. Najpierw musimy utworzyć tabelę przestawną, z której odczytamy dane i której kolumny przekształcimy z powrotem na wiersze. W tym celu użyjemy wyniku poznanego w poprzednim punkcie zapytania:

```
SELECT P.LastName, [Bike Racks],[Bottles and Cages] ,[Bottom  
Brackets], [Brakes]  
INTO #TabUnpivot  
FROM #TabPivot  
PIVOT (  
SUM(Total)  
FOR Name IN ([Bike Racks],[Bottles and Cages] ,[Bottom  
Brackets], [Brakes]) ) AS P  
ORDER BY P.LastName;;  
-----  
(12 row(s) affected)
```

2. Następnie w klauzuli SELECT, tak jak w standardowych zapytaniach, należy zdefiniować listę kolumn wyniku, a w klauzuli FROM — wskazać tabelę źródłową. Dodatkowo za pomocą operatora UNPIVOT należy zdefiniować kolumnę, w której zostaną umieszczone odczytane z tabeli przestawnej podsumowania (kolumna Total), oraz określić kolumnę, w której zostaną umieszczone nagłówki kolumn tabeli przestawnej (kolumna Name):

```
SELECT Unpiv.Name, Unpiv.LastName, Unpiv.Total  
FROM #TabUnpivot  
UNPIVOT (Total  
FOR Name IN ([Bike Racks],[Bottles and Cages] ,[Bottom  
Brackets], [Brakes]))  
AS Unpiv;  
-----
```

Name	LastName	Total
------	----------	-------

Bike Racks	Abel	216.000000
Bike Racks	Beck	216.000000
Bike Racks	Byham	216.000000
Bike Racks	Eminhizer	432.000000
Bike Racks	Grande	720.000000
Bike Racks	Kurtz	288.000000
Bike Racks	Liu	144.000000
Bike Racks	Marple	72.000000
Bottles and Cages	Abel	17.964000
Bottles and Cages	Beck	23.952000
Bottles and Cages	Eminhizer	31.199476
Bottles and Cages	Gilbert	2.994000
Bottles and Cages	Grande	31.199476
Bottles and Cages	Kurtz	20.958000
Bottles and Cages	Liu	29.940000
Bottom Brackets	Abel	323.970000
Bottom Brackets	Campbell	218.682000
Bottom Brackets	Laszlo	388.758000
Bottom Brackets	Sunkammurali	388.758000
Brakes	Abel	63.900000

Klauzula HAVING

Klauzula HAVING jest wykonywana po klauzuli GROUP BY, a więc umieszcza się w niej dowolną funkcję grupującą lub nazwy kolumn użytych do grupowania, **i umożliwia wyeliminowanie z wyniku zapytania grup niespełniających określonego warunku logicznego.**

Przypomnijmy sobie jeden z wcześniejszych przykładów, w którym chcieliśmy wybrać wiersze na podstawie wyniku funkcji grupującej:

```
SELECT C.Name, COUNT(*), AVG(ListPrice)
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
```

```
WHERE COUNT(*)>10  
GROUP BY C.Name;
```

Ponieważ klauzula `WHERE` jest wykonywana wcześniej niż klauzula `GROUP BY`, nie udało nam się w ten sposób ograniczyć wyniku zapytania do sprzedanych produktów, których średnia liczba przekroczyła 10. Problem ten rozwiążemy, przenosząc ten sam warunek logiczny do klauzuli `HAVING`:

```
SELECT C.Name, COUNT(*) AS Liczba, AVG(ListPrice) AS Średnia  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON P.ProductCategoryID=C.ProductCategoryID  
GROUP BY C.Name  
HAVING COUNT(*)>10;
```

Name	Liczba	Średnia
Mountain Bikes	32	1683,365
Road Bikes	43	1597,45
Touring Bikes	22	1425,2481
Mountain Frames	28	678,2535
Road Frames	33	780,0436
Touring Frames	18	631,4155
Wheels	14	220,9292
Tires and Tubes	11	19,4827

Umieszczony w klauzuli `HAVING` warunek logiczny jest sprawdzany dla każdej grupy i tylko te dane, dla których będzie on prawdziwy, znajdują się w wyniku zapytania. Niemożliwe jest więc odwołanie się w klauzuli `HAVING` do niezgrupowanych wierszy — do jednej grupy mogą przecież należeć zarówno wiersze spełniające dany warunek, jak i wiersze niespełniające go:

```
SELECT C.Name, COUNT(*) AS Liczba, AVG(ListPrice) AS Średnia  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON P.ProductCategoryID=C.ProductCategoryID  
GROUP BY C.Name
```

```
HAVING p.ListPrice>10;
```

```
-----  
Msg 8121, Level 16, State 1, Line 337
```

Column 'SalesLT.Product.ListPrice' is invalid in the HAVING clause because it is not contained in either an aggregate function or the GROUP BY clause.

Wiemy już, że do filtrowania poszczególnych wierszy służy klauzula WHERE:

```
SELECT C.Name, COUNT(*) AS Liczba, AVG(ListPrice) AS Średnia  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON P.ProductCategoryID=C.ProductCategoryID  
WHERE p.ListPrice>200  
GROUP BY C.Name;
```

```
-----  


| Name            | Liczba | Średnia   |
|-----------------|--------|-----------|
| Mountain Bikes  | 32     | 1683,365  |
| Road Bikes      | 43     | 1597,45   |
| Touring Bikes   | 22     | 1425,2481 |
| Cranksets       | 2      | 330,74    |
| Forks           | 1      | 229,49    |
| Mountain Frames | 28     | 678,2535  |
| Road Frames     | 33     | 780,0436  |
| Touring Frames  | 18     | 631,4155  |
| Wheels          | 10     | 274,639   |


```

Powyższe zapytanie zwróciło nazwy kategorii produktów zawierających przynajmniej jeden produkt o cenie przekraczającej 200. Natomiast kolejne zapytanie zwraca nazwy kategorii, dla których średnia cena produktów przekracza 200 — jak widać, są to różne warunki logiczne:

```
SELECT C.Name, COUNT(*) AS Liczba, AVG(ListPrice) AS Średnia  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
    ON P.ProductCategoryID=C.ProductCategoryID
```

```

GROUP BY C.Name
HAVING AVG(p.ListPrice)>200;
-----
```

Name	Liczba	Średnia
Mountain Bikes	32	1683,365
Road Bikes	43	1597,45
Touring Bikes	22	1425,2481
Cranksets	3	278,99
Mountain Frames	28	678,2535
Road Frames	33	780,0436
Touring Frames	18	631,4155
Wheels	14	220,9292

Razem użyte klauzule WHERE i HAVING pozwalają:

- ograniczyć liczbę grupowanych wierszy (w klauzuli WHERE);
- wyeliminować grupy, które nie spełniają określonego warunku logicznego (w klauzuli HAVING):

```

SELECT C.Name, COUNT(*), AVG(ListPrice)
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
WHERE [SellEndDate] IS NOT NULL
GROUP BY C.Name
HAVING COUNT(*)>10;
-----
```

Name	(No column name)	(No column name)
Mountain Bikes	12	2618,3233
Road Bikes	25	1676,2892
Road Frames	11	454,3154
Wheels	14	220,9292

Chociaż najczęściej w klauzuli HAVING umieszcza się funkcje grupujące wymienione w klauzuli SELECT, można też umieścić w niej kolumny lub wyrażenia wymienione w klauzuli GROUP BY. Kolejne zapytanie zwraca nazwy kategorii, przy czym w wyniku zostały uwzględnione jedynie

kategorie o nazwach zaczynających się na literę R:

```
SELECT C.Name, COUNT(*), AVG(ListPrice)
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
    ON P.ProductCategoryID=C.ProductCategoryID
WHERE [SellEndDate] IS NOT NULL
GROUP BY C.Name
HAVING C.Name LIKE 'R%';
-----
```

Name	(No column name)	(No column name)
Road Bikes	25	1676,2892
Road Frames	11	454,3154



Powyższe zapytanie nie tylko jest mało czytelne, ale może też powodować problemy wydajnościowe. Do filtrowania na poziomie pojedynczych wierszy zalecamy używanie wyłącznie klauzuli WHERE.

Podsumowanie

- Funkcje grupujące zwracają jedną wartość, ale są wywoływanie raz dla całej grupy danych.
- Wszystkie funkcje grupujące, z wyjątkiem funkcji COUNT(*), ignorują wartość NULL.
- Domyślnie funkcje grupujące są wywoływanie dla wszystkich, również powtarzających się, wartości w grupie. Możemy to zmienić za pomocą słowa kluczowego DISTINCT.
- Pogrupować wiersze na podstawie wspólnych wartości wskazanych kolumn lub takich samych wyników wyrażeń możemy dzięki klauzuli GROUP BY.
- Klauzula GROUP BY pozwala dowolnie zagnieździć grupy danych.
- Operatory CUBE i ROLLUP pozwalają dodać do wyniku zapytania sumy pośrednie.
- Operator GROUPING SETS pozwala zdefiniować wiele dowolnych grup w ramach jednego zapytania.
- Operatory PIVOT i UNPIVOT pozwalają używać w języku SQL tabel przestawnych.

- Tak jak klauzula WHERE pozwala filtrować wiersze, tak klauzula HAVING pozwala filtrować zwracane przez zapytanie grupy.

Zadania

1. Odczytaj z tabeli [SalesLT].[SalesOrderHeader] wartości zamówień o najwyższych opłatach za wysyłkę zrealizowanych w każdym dniu dla poszczególnych klientów.
2. Odczytaj nazwy produktów, które zostały sprzedane więcej niż trzy razy. Dodaj do wyniku liczbę tych produktów.
3. Uruchom poniższy skrypt tworzący tabelę tymczasową i wstawiający do niej dane o numerach wybranych klientów, miesiącach złożenia przez nich zamówień i wartościach poszczególnych zamówień:

```
CREATE TABLE #Sprzedaz
([ID klienta] INT NOT NULL,
Miesiąc INT NOT NULL,
Wartość MONEY NOT NULL);
GO
INSERT INTO #Sprzedaz
SELECT [CustomerID], DATEPART(MONTH, [OrderDate]), [TotalDue]
FROM [SalesLT].[SalesOrderHeader];
GO
```

Zadanie polega na zbudowaniu tabeli przestawnej zawierającej całkowite wartości zamówień złożonych przez każdego z tych klientów w ciągu kolejnych miesięcy. Nagłówkami kolumn utworzonej tabeli przestawnej mają być numery miesięcy.

[1] W praktyce tabela tymczasowa często jest zastępowana podzapytaniem. Ponieważ lokalne tabele tymczasowe są dostępne w ramach jednej sesji, wszystkie przykładowe instrukcje należy wykonać jedna po drugiej w tym samym oknie SSMSE.

Rozdział 7. Partycjonowanie wierszy oraz funkcje rankingu, analityczne i okienkowe

- Czym partycjonowanie wierszy różni się od ich grupowania?
- Jak działa klauzula OVER?
- Kiedy warto używać funkcji rankingu?
- Czym są okienka danych?
- Jak odwoływać się do wskazanych wierszy za pomocą funkcji okienkowych?
- Do czego służą funkcje analityczne?

Partycjonowanie

Opisane w poprzednim rozdziale grupowanie jest transformacją typu N do 1, co oznacza, że każdy otrzymany w wyniku grupowania wiersz powstaje na podstawie jednego lub więcej wierszy tabeli źródłowej. Tymczasem partycjonowanie jest transformacją typu N do N, co oznacza, że pozwala ono połączyć w wyniku zapytania dane szczegółowe z danymi na różnych poziomach ogólności (rysunek 7.1).

Do partycjonowania danych służy klauzula OVER — najbardziej rozbudowana klauzula języka SQL.

3 wiersze
→ 3 wiersze

2 wiersze
→ 2 wiersze

Kolumna A	Kolumna B	Wartość
A	B	1
A	B	2
A	B	3
C	D	4
C	D	5

3 wiersze

2 wiersze

Kolumna A	Kolumna B	Wartość	SUM(Wartość) OVER (...)
A	B	1	6
A	B	2	6
A	B	3	6
C	D	4	9
C	D	5	9

Rysunek 7.1. Inaczej niż przy grupowaniu, partycjonując dane, nie tracimy szczegółowych informacji, a zyskujemy dostęp do informacji uogólnionych

Klauzula OVER

Serwery bazodanowe pozwalają określić dodatkowy porządek grupowania w klauzuli OVER. Zdefiniowane w tej klauzuli grupy są nazywane partycjami i mogą być użyte do wywołania funkcji grupujących oraz funkcji rankingu, okienkowych i analitycznych.



Wskazówka

Klauzula OVER może wystąpić w zapytaniach niezawierających klauzuli GROUP BY. Dzięki temu zadania, które wymagały użycia podzapytań, mogą być teraz rozwiązane za pomocą klauzuli OVER.

Pokażemy to na prostym przykładzie:

1. Zaczniemy od zapytania, które zwraca identyfikatory i wartości zamówień:

```
SELECT SalesOrderID, TotalDue  
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
SalesOrderID      TotalDue  
71774            972,785
```

71776	87,0851
71780	42452,6519
71782	43962,7901
71783	92663,5609
71784	119960,824
71796	63686,2708
71797	86222,8072
71815	1261,444
71816	3754,9733
71831	2228,0566
71832	39531,6085
71845	45992,3665

...

2. Spróbujmy dodać teraz do tego wyniku kolumnę z wynikiem dowolnej funkcji grupującej, np. z wartością cen wszystkich zamówień:

```
SELECT SalesOrderID, TotalDue, AVG(TotalDue)
FROM [SalesLT].[SalesOrderHeader];
```

Msg 8120, Level 16, State 1, Line 8

Column 'SalesLT.SalesOrderHeader.SalesOrderID' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

3. Ponieważ to zapytanie niejawnie grupuje dane, próba jego wykonania skończyła się błędem — mamy kilkadziesiąt identyfikatorów i wartości poszczególnych zamówień, a więc dwie pierwsze kolumny wyniku zapytania musiałyby liczyć trzydzieści trzy wiersze, a ostatnia tylko jeden. Dodanie klauzuli GROUP BY też nie rozwiąże problemu:

```
SELECT SalesOrderID, TotalDue, AVG(TotalDue) as Średnia
FROM [SalesLT].[SalesOrderHeader]
GROUP BY SalesOrderID, TotalDue;
```

SalesOrderID	TotalDue	Średnia
71774	972,785	972,785

71776	87,0851	87,0851
71780	42452,6519	42452,6519
71782	43962,7901	43962,7901
71783	92663,5609	92663,5609
71784	119960,824	119960,824
71796	63686,2708	63686,2708

...

- Teraz otrzymaliśmy trzydzieści trzy grupy i dla każdej z nich została wywołana funkcja grupująca. Ponieważ każda grupa zawiera dokładnie jeden wiersz, policzona na jego podstawie średnia nie jest średnią wartością wszystkich zamówień, tylko pojedynczego zamówienia. Aby uzyskać poprawny wynik, musielibyśmy wywołać funkcję grupującą tylko raz, czyli utworzyć tylko dla niej grupę zawierającą wszystkie zaklasyfikowane wiersze — pozwala na to właśnie klauzula OVER:

```
SELECT SalesOrderID, TotalDue,
AVG(TotalDue) OVER() AS Średnia
FROM [SalesLT].[SalesOrderHeader];
```

SalesOrderID	TotalDue	Średnia
71774	972,785	29008,3751
71776	87,0851	29008,3751
71780	42452,6519	29008,3751
71782	43962,7901	29008,3751
71783	92663,5609	29008,3751

- Pusta klauzula OVER utworzyła jedną partycję, czyli zadziałała tak samo jak pusta klauzula GROUP BY. Wywołana dla tej partycji funkcja AVG() zwróciła prawidłowy wynik. Klauzula OVER pozwala wywołać każdą funkcję grupującą, a jej wynik może być użyty w wyrażeniach, co pokazuje kolejne zapytanie:

```
SELECT SalesOrderID, TotalDue,
MIN(TotalDue) OVER() AS min,
MAX(TotalDue) OVER() AS max,
AVG(TotalDue) OVER() AS avg,
TotalDue - AVG(TotalDue) OVER() AS diff
```

```
FROM [SalesLT].[SalesOrderHeader];
```

SalesOrderID diff	TotalDue	min	max	avg
71774 -28035,5901	972,785	43,0437	119960,824	29008,3751
71776 -28921,29	87,0851	43,0437	119960,824	29008,3751
71780 13444,2768	42452,6519	43,0437	119960,824	29008,3751
71782 14954,415	43962,7901	43,0437	119960,824	29008,3751
71783 63655,1858	92663,5609	43,0437	119960,824	29008,3751
...				

W poprzednim rozdziale napisaliśmy, że funkcji grupujących nie można zagnieździć. Od tej reguły jest jednak jeden wyjątek — funkcja grupująca wywołana dla klauzuli `OVER` może być użyta jako argument innej funkcji grupującej. Dzięki tej możliwości rozwiążemy typowy problem polegający na obliczeniu procentowej wartości na poziomie poszczególnych grup.

Policzenie sumy zamówień poszczególnych klientów jest łatwe:

```
SELECT [CustomerID], SUM (TotalDue) AS TotalSum  
FROM [SalesLT].[SalesOrderHeader]  
GROUP BY [CustomerID];
```

CustomerID	TotalSum
29485	43962,7901
29531	7330,8972
29546	98138,2131
29568	2669,3183
29584	272,6468
...	

Żeby policzyć procentową wartość zamówień poszczególnych klientów, musielibyśmy podzielić obliczone w ten sposób sumy przez

wartość wszystkich zamówień. Wiemy jednak, że poniższe zapytanie jest niepoprawne:

```
SELECT [CustomerID], SUM (TotalDue) AS TotalSum,  
       SUM (TotalDue*100.) / SUM(TotalDue*1.) OVER () AS Pct  
  FROM [SalesLT].[SalesOrderHeader]  
 GROUP BY [CustomerID];
```

Msg 8120, Level 16, State 1, Line 29

Column 'SalesLT.SalesOrderHeader.TotalDue' is invalid in the select list
because it is not contained in either an aggregate function or the GROUP
BY clause.

Rozwiązaniem jest „zagnieżdżenie” funkcji grupującej, w tym
wypadku umieszczenie całej klauzuli OVER w ramach funkcji SUM:

```
SELECT [CustomerID], SUM (TotalDue) AS TotalSum,  
       SUM (TotalDue*100.) / SUM (SUM(TotalDue*1.)) OVER () AS Pct  
  FROM [SalesLT].[SalesOrderHeader]  
 GROUP BY [CustomerID]  
 ORDER BY Pct DESC;
```

CustomerID	TotalSum	Pct
29736	119960,824	12.531472
30050	108597,9536	11.344472
29546	98138,2131	10.251816
29957	92663,5609	9.679917
29796	86222,8072	9.007096
29929	81834,9826	8.548730
29932	70698,9922	7.385431
29660	63686,2708	6.652861

...

Partycjonowanie danych

Jeżeli w klauzuli OVER zostanie umieszczone słowo kluczowe PARTITION BY, funkcja grupująca lub funkcja rankingu zostaną wywołane dla każdej zdefiniowanej w ten sposób partycji. Klauzula OVER (PARTITION BY) odpowiada więc niepustej klauzuli GROUP BY. Pokażemy to na

przykładzie zapytania zwracającego nazwy kategorii i liczby należących do nich produktów:

1. Skoro w zapytaniu mamy użyć klauzuli `OVER`, a nie `GROUP BY`, zaczniemy od zapytania odczytującego nazwy kategorii z połączonych tabel `ProductCategory` i `Product`.

```
SELECT C.[Name]
  FROM [SalesLT].[ProductCategory] AS C
  JOIN [SalesLT].[Product] AS P
    ON C.ProductCategoryID=P.ProductCategoryID;
```

```
-----
Name
Mountain Bikes
Mountain Bikes
Mountain Bikes
Mountain Bikes
Road Bikes
Road Bikes
Road Bikes
Road Bikes
...

```

2. Nazwa każdej kategorii powtarza się tyle razy, ile jest przypisanych do niej produktów. Te powtórzenia możemy wyeliminować za pomocą słowa kluczowego `DISTINCT`. Jednak zanim to zrobimy, dopiszmy klauzulę `OVER` z funkcją `COUNT` — ponieważ chcemy osobno policzyć liczebność każdej kategorii, musimy utworzyć partycje na podstawie kolumny. Dodatkowo posortujemy wynik zapytania według obliczonej w ten sposób liczby duplikatów:

```
SELECT C.[Name],
       COUNT(*) OVER (PARTITION BY C.ProductCategoryID) AS
       LiczbaDuplikatów
  FROM [SalesLT].[ProductCategory] AS C
  JOIN [SalesLT].[Product] AS P
    ON C.ProductCategoryID=P.ProductCategoryID;
```

Name	LiczbaDuplikatów
Cleaners	1
Fenders	1
Hydration Packs	1
Locks	1
Panniers	1
Pumps	2
Pumps	2
Brakes	2
Brakes	2
Derailleurs	2
Derailleurs	2
Forks	3

...

3. Pozostało nam tylko usunąć powtarzające się wiersze:

```
SELECT DISTINCT C.[Name],  
           COUNT(*) OVER (PARTITION BY C.ProductCategoryID) AS  
LiczbaDuplikatów  
FROM [SalesLT].[ProductCategory] AS C  
JOIN [SalesLT].[Product] AS P  
ON C.ProductCategoryID=P.ProductCategoryID  
ORDER BY LiczbaDuplikatów;
```

Name	LiczbaDuplikatów
Chains	1
Cleaners	1
Fenders	1
Hydration Packs	1
Locks	1
Panniers	1
Brakes	2
Derailleurs	2
Pumps	2

...

Klauzula OVER (PARTITION BY) pozwala pomieszać w jednym zapytaniu dane na różnych poziomach szczegółowości. Zaczniemy od prostego przykładu — do szczegółowych danych (danych na poziomie poszczególnych wierszy) dodamy kolumnę z sumą wyliczoną na poziomie poszczególnych klientów (każdy klient może złożyć wiele zamówień):

```
SELECT CustomerID, SalesOrderID, TotalDue,
       SUM(TotalDue) OVER(PARTITION BY [CustomerID]) AS SumTotalDue
  FROM [SalesLT].[SalesOrderHeader]
 ORDER BY [CustomerID];
```

CustomerID	SalesOrderID	TotalDue	SumTotalDue
29781	71923	117,7276	117,7276
29796	71797	86222,8072	86222,8072
29847	71774	972,785	1945,57
29847	77774	972,785	1945,57

...

Dodanie kolejnego poziomu grupowania sprowadza się do dodania do zapytania kolejnej klauzuli OVER, co pokazuje kolejny przykład:

```
SELECT SalesOrderID, CustomerID, TotalDue,
       CAST (100. * TotalDue / SUM(TotalDue) OVER (PARTITION BY
CustomerID) AS
          NUMERIC (5,2)) AS PctCust,
       CAST (100. * TotalDue / SUM(TotalDue) OVER () AS NUMERIC (5,2))
AS PctTotal
  FROM [SalesLT].[SalesOrderHeader];
```

SalesOrderID PctTotal	CustomerID	TotalDue	PctCust	
71923	29781	117,7276	100.00	0.01
71797	29796	86222,8072	100.00	9.01
71774	29847	972,785	50.00	0.10

77774

29847

972,785

50.00

0.10

Porządkowanie danych

W klauzuli OVER możemy też określić porządek wierszy. Co prawda omówione do tej pory przykłady tego nie wymagały, ale przedstawione w dalszej części rozdziału funkcje rankingu i analityczne wymagają posortowania wierszy za pomocą klauzuli ORDER BY. Porządek wierszy jest też niezbędny do zdefiniowania okienka danych — funkcjonalności wyjątkowo przydatnej w różnorodnych analizach. Zanim jednak nauczysz się porządkować dane, powinieneś poznać pełną składnię klauzuli OVER (rysunek 7.2).



Rysunek 7.2. Klauzula OVER pozwala określić partycje (dla każdej partycji wywołana zostanie określona funkcja), porządek (posortowanie wierszy w partycjach jest wymagane w przypadku użycia funkcji rankingu, okienkowych lub analitycznych) oraz dodatkowo ograniczyć liczbę widocznych w danym momencie wierszy (zdefiniować okienko danych)

Praktyczne przykłady sortowania wierszy w klauzuli OVER zawiera kolejny punkt.

Funkcje rankingu

Funkcje rankingu, w przeciwieństwie do funkcji grupujących, mogą być wywoływane dla utworzonych za pomocą klauzuli OVER partycji, ale nie dla utworzonych za pomocą klauzuli GROUP BY grup. Do najczęściej implementowanych w różnych serwerach bazodanowych funkcji rankingu należą:

1. Zwracająca numer wiersza funkcja `ROW_NUMBER()` — w jej przypadku każdy wiersz wyniku otrzyma kolejny numer.
2. Zwracająca ten sam numer wiersza dla powtarzających się wartości funkcja `DENSE_RANK ()` — zachowuje ona ciągłość numeracji wierszy wyniku. Innymi słowy, funkcja `DENSE_RANK ()` numeruje wartości.
3. Zwracająca ten sam numer dla powtarzających się wartości funkcja `RANK()` — nie zachowuje ona ciągłości numeracji, za to zwraca prawidłową liczbę wierszy wyniku.
4. Dzieląca wiersze wyniku na określoną liczbę bloków funkcja `NTILE()` — w jej przypadku do każdego bloku zostanie zaklasyfikowanych tyle samo (+/- 1) wierszy.

Najprościej jest wytlumaczyć działanie wszystkich czterech funkcji rankingu na przykładzie:

```
SELECT FirstName,
    ROW_NUMBER() OVER(ORDER BY FirstName) AS ROW_NUMBER,
    RANK() OVER(ORDER BY FirstName) AS RANK,
    DENSE_RANK() OVER(ORDER BY FirstName) AS DENSE_RANK,
    NTILE(3) OVER (ORDER BY FirstName) AS NTILE
FROM [SalesLT].[Customer]
WHERE FirstName IN ('Andrew', 'Juanita', 'Christopher');
```

FirstName	ROW_NUMBER	RANK	DENSE_RANK	NTILE
Andrew	1	1	1	1
Andrew	2	1	1	1
Andrew	3	1	1	1
Andrew	4	1	1	1
Andrew	5	1	1	2
Andrew	6	1	1	2
Christopher	7	7	2	2
Christopher	8	7	2	2
Christopher	9	7	2	3
Christopher	10	7	2	3
Juanita	11	11	3	3

Ponieważ wynik wszystkich funkcji rankingu zależy od uporządkowania wierszy, wymagają one posortowania partycji — w powyższym przykładzie klauzule OVER zwracały po jednej partycji i każda z nich była posortowana według imion.

Funkcje rankingu są wywoływane raz dla każdej partycji. Pozwala to np. osobno ponumerować panów i panie:

```
SELECT Title, FirstName,  
ROW_NUMBER() OVER( PARTITION BY [Title] ORDER BY FirstName) AS ROW_NUMBER  
FROM [SalesLT].[Customer]  
WHERE FirstName IN ('Andrew','Juanita');
```

Title	FirstName	ROW_NUMBER
Mr.	Andrew	1
Mr.	Andrew	2
Mr.	Andrew	3
Mr.	Andrew	4
Mr.	Andrew	5
Mr.	Andrew	6
Ms.	Juanita	1
Ms.	Juanita	2

Warto w tym miejscu zauważyc, że bez funkcji rankingu uzyskanie takich samych wyników, np. ponumerowanie wierszy, jest nie tylko dość skomplikowane, ale również bardzo wolne. Wynika to z faktu, że serwery bazodanowe są zoptymalizowane pod kątem przetwarzania zbiorów danych, a dane w zbiorze są z definicji nieposortowane. Klauzula OVER rozwiązuje ten problem, pozwalając określić porządek wierszy na wyłączną potrzebę danej funkcji, a więc pozostałe klauzule zapytania nadal mogą przetwarzać zbiory danych. Poniższe zapytanie ilustruje to zagadnienie — do ponumerowania wartości użyte w nim zostało podzapytanie skorelowane:

```
SELECT  
(  
SELECT COUNT(*) + 1
```

```

        FROM [SalesLT].[SalesOrderHeader] AS t2
        WHERE t2.CustomerID = t1.CustomerID
        AND t2.SalesOrderID < t1.SalesOrderID
    ) AS [DENSE_RANK],
    t1.SalesOrderID,
    t1.OrderDate,
    t1.TotalDue
FROM [SalesLT].[SalesOrderHeader] AS t1
ORDER BY t1.SalesOrderID;
-----
```

...

DENSE_RANK	SalesOrderID	TotalDue
1	71938	98138,2131
1	71946	43,0437
2	77774	972,785

Okienka

Najciekawszą funkcjonalnością klauzuli `OVER` jest możliwość ograniczenia widocznych w danym momencie wierszy do zdefiniowanych okienek. W rezultacie wywołana dla tej klauzuli funkcja „zobacz” w danym momencie tylko część wierszy, np. tylko bieżący wiersz. Podczas kolejnego wywołania okienko przesunie się do następnego wiersza i ta sama funkcja zostanie wywołana dla kolejnego wiersza — i tak dalej. Ilustruje to następny przykład:

1. Zobaczmy najpierw, które zamówienia były realizowane w kolejnych dniach poszczególnych miesięcy:

```

SELECT [SalesOrderID], DAY([DueDate]) AS Dzień, [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];
-----
```

SalesOrderID	Dzień	TotalDue
71776	14	87,0851
71832	14	39531,6085
71774	17	972,785

71895	17	272,6468
71783	17	92663,5609
71920	18	3293,7761
71935	18	7330,8972

...

2. Żeby dodać do wyniku zapytania kolumnę z wartością sprzedaży w danym dniu, wystarczy użyć klauzuli OVER, podzielić w niej wiersze na partycje według dni i posortować je według daty (sortowanie według całej daty jest ważne, inaczej sprzedaż z tych samych dni różnych miesięcy trafiłaby do tej samej partycji):

```
SELECT [SalesOrderID], DAY([DueDate]) AS Dzień, [TotalDue],
       SUM([TotalDue]) OVER(PARTITION BY DAY([DueDate])
                           ORDER BY [DueDate]
                           ) AS DayTotal
  FROM [SalesLT].[SalesOrderHeader]
 ORDER BY [DueDate];
```

SalesOrderID	Dzień	TotalDue	DayTotal
71832	14	39531,6085	39618,6936
71776	14	87,0851	39618,6936
71895	17	272,6468	93908,9927
71774	17	972,785	93908,9927
71783	17	92663,5609	93908,9927
71935	18	7330,8972	10624,6733
71920	18	3293,7761	10624,6733

...

3. W powyższym zapytaniu funkcja SUM() została wywołana dla wszystkich wierszy danej partycji. Żeby zobaczyć wartość sprzedaży z poprzedniego wiersza, należy zdefiniować dla tej funkcji okienko danych, ograniczając widoczne przez nią wiersze do poprzedniego. Do zdefiniowania partycji służą słowa kluczowe ROWS lub RANGE:

```
SELECT [SalesOrderID], DAY([DueDate]) AS Dzień, [TotalDue],
       SUM([TotalDue]) OVER(PARTITION BY DAY([DueDate])
```

```

        ORDER BY [DueDate]
        ROWS BETWEEN 1 PRECEDING
        AND 1 PRECEDING) AS PreviousRow
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];
-----
```

SalesOrderID	Dzień	TotalDue	PreviousRow
71832	14	39531,6085	NULL
71776	14	87,0851	39531,6085
71895	17	272,6468	NULL
71774	17	972,785	272,6468
71783	17	92663,5609	972,785
71935	18	7330,8972	NULL
71920	18	3293,7761	7330,8972

Okienko danych może być zdefiniowane przy użyciu instrukcji:

1. CURRENT ROW — w ten sposób ograniczamy jego wielkość do bieżącego wiersza.
2. PRECEDING — do podanej liczby wierszy poprzedzających bieżący, przy czym konstrukcja UNBOUNDED PRECEDING oznacza wszystkie wiersze od pierwszego do bieżącego w danej partycji.
3. FOLLOWING — do podanej liczby wierszy następujących po bieżącym, przy czym konstrukcja UNBOUNDED FOLLOWING oznacza wszystkie wiersze od bieżącego do ostatniego w danej partycji.

Wynika z tego, że wartości 0 PRECEDING oraz 0 FOLLOWING są równoznaczne z użyciem instrukcji CURRENT ROW. Wiedząc, jak definiuje się okienka, możemy już przeczytać użytkę w ostatnim zapytaniu klauzulę OVER jako zwracającą sumę wartości TotalDue obliczoną dla poprzedniego wiersza, czyli po prostu wartość z tej kolumny odczytaną z poprzedniego wiersza. Ponieważ w klauzuli tej zdefiniowaliśmy partycje, w pierwszym wierszu każdej partycji zwracana jest wartość NULL (wartość poprzedniego wiersza nie istnieje), w drugim — wartość sprzedaży poprzedniej transakcji, w trzecim — wartość sprzedaży drugiej transakcji z tego dnia i tak dalej.

Jednym z typowych zastosowań okienek danych jest obliczanie sum narastających (skumulowanych). W tym celu wystarczy ograniczyć wiersze widoczne dla danej funkcji grupującej (nie musi to być funkcja SUM) do wszystkich poprzedzających bieżący wiersz w ramach zdefiniowanej partycji:

```
SELECT [SalesOrderID], DAY([DueDate]) AS Dzień, [TotalDue],  
       SUM([TotalDue]) OVER(PARTITION BY DAY([DueDate])  
                           ORDER BY [DueDate]  
                           ROWS BETWEEN 1 PRECEDING  
                           AND CURRENT ROW) AS runningTotalDue  
FROM [SalesLT].[SalesOrderHeader]  
ORDER BY DueDate;
```

SalesOrderID	Dzień	TotalDue	runningTotalDue
71832	14	39531,6085	39531,6085
71776	14	87,0851	39618,6936
71895	17	272,6468	272,6468
71774	17	972,785	1245,4318
71783	17	92663,5609	93636,3459
71935	18	7330,8972	7330,8972
71920	18	3293,7761	10624,6733

...

Kolejnym typowym zastosowaniem okienek jest obliczanie średnich ruchomych (tego typu obliczenia są powszechnie używane do „wygładzania” wykresów liniowych). Na przykład możemy ograniczyć okienko do trzech wierszy (poprzedniego, bieżącego i następnego) w danej partycji i dla tych wierszy wywołać funkcję AVG:

```
SELECT [SalesOrderID], DAY([DueDate]) AS Dzień, [TotalDue],  
       AVG([TotalDue]) OVER(PARTITION BY DAY([DueDate])  
                           ORDER BY DueDate  
                           ROWS BETWEEN 1 PRECEDING  
                           AND 1 FOLLOWING) AS movingAvg  
FROM [SalesLT].[SalesOrderHeader]  
ORDER BY DueDate;
```

SalesOrderID	Dzień	TotalDue	movingAvg
71832	14	39531,6085	19809,3468
71776	14	87,0851	34434,9881
71895	17	272,6468	622,7159
71774	17	972,785	31302,9975
71783	17	92663,5609	46818,1729
71935	18	7330,8972	5312,3366
71920	18	3293,7761	5312,3366

...

We wszystkich dotychczasowych przykładach do definiowania okienka używaliśmy słowa kluczowego `ROWS`. Różnica pomiędzy użyciem słowa `ROWS` a `RANGE` przypomina różnicę pomiędzy funkcjami `ROW_NUMBER` a `RANK` — w pierwszym wypadku odwołujemy się do wierszy, w drugim — do wartości. Zakresy (`RANGE`) odwołują się do wartości tych samych względem porządku sortowanego. Różnica ta jest istotna wszędzie tam, gdzie w tabeli źródłowej nie występują pewne wartości. Na przykład poprzedni wynik nie zawiera danych o sprzedaży w 15. i 16. dniu miesiąca (po 14. dniu następuje od razu dzień 17.). Gdybyśmy przedstawili obliczoną w ten sposób średnią ruchomą na wykresie, brak sprzedaży w tych dwóch dniach byłby niewidoczny. Tymczasem gdyby w tabeli znajdowały się wpisy z zerową wartością sprzedaży dla tych dni, wykres wyglądałby inaczej (średnia ruchoma spadłaby dla tych dwóch dni). Problem ten można rozwiązać na dwa sposoby:

1. Uzupełniając brakujące dane. Przedstawienie tego rozwiązania wykracza poza zakres tej książki.
2. Zastępując odwołania do wierszy odwołaniami do wartości (wartością poprzedzającą 17 zawsze byłoby 16). Niestety, zaimplementowana w wersji 2014 serwera SQL klauzula `OVER` pozwala użyć frazy `RANGE` tylko w jednym przypadku — do zdefiniowania okienka zawierającego wszystkie wartości od pierwszej do bieżącej w danej partycji:

```
SELECT [SalesOrderID],DAY([DueDate]) AS Dzień, [TotalDue],
       AVG([TotalDue]) OVER(PARTITION BY DAY([DueDate])
                           ORDER BY DueDate)
```

```

        RANGE BETWEEN 1 PRECEDING
        AND 1 FOLLOWING) AS movingAvg
FROM [SalesLT].[SalesOrderHeader]
ORDER BY DueDate;
-----
Msg 4194, Level 16, State 1, Line 149
RANGE is only supported with UNBOUNDED and CURRENT ROW window frame
delimiters.

```

Funkcje okienkowe

Funkcje okienkowe pozwalają odwołać się do wartości wskazanego pola w okienku danych. Serwer SQL udostępnia cztery funkcje tego typu:

1. Funkcja LAG domyślnie zwraca wartość określonego pola z poprzedniego wiersza partycji, przy czym jeżeli zostanie wywołana dla pierwszego wiersza, zwróci NULL. Opcjonalny parametr tej funkcji pozwala określić przesunięcie:

```

SELECT CAST([DueDate] AS Date) AS Date, [SalesOrderID], [TotalDue],
       LAG([TotalDue]) OVER (PARTITION BY [DueDate] ORDER BY
[DueDate]) as      PreviusTotalDue,
       LAG([TotalDue],2) OVER (PARTITION BY [DueDate] ORDER BY
[DueDate]) as      TwoPrecedingTotalDue
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];
-----
```

Date	SalesOrderID	TotalDue	PreviusTotalDue
TwoPrecedingTotalDue			
2008-06-14	71776	87,0851	NULL
NULL			
2008-06-14	71832	39531,6085	87,0851
NULL			
2008-06-17	71774	972,785	NULL
NULL			
2008-06-17	71895	272,6468	972,785
NULL			
2008-06-17	71783	92663,5609	272,6468

972,785			
2008-06-18	71920	3293,7761	NULL
NULL			
2008-06-18	71935	7330,8972	3293,7761
NULL			

2. Funkcja LEAD domyślnie zwraca wartość określonego pola z następnego wiersza partycji, przy czym jeżeli zostanie wywołana dla ostatniego wiersza, zwróci NULL. Również tę funkcję można wywołać z opcjonalnym parametrem określającym, o ile wierszy chcemy przesunąć odczytywane pole:

```
SELECT CAST([DueDate] AS DATE) AS Date, [SalesOrderID], [TotalDue],
       LEAD([TotalDue]) OVER (PARTITION BY [DueDate] ORDER BY
[DueDate]) as           NextTotalDue,
       LEAD([TotalDue],2) OVER (PARTITION BY [DueDate] ORDER BY
[DueDate]) as           TwoFollowingTotalDue
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];
-----
```

Date	SalesOrderID	TotalDue	NextTotalDue
TwoFollowingTotalDue			
2008-06-14	71776	87,0851	39531,6085
NULL			
2008-06-14	71832	39531,6085	NULL
NULL			
2008-06-17 92663,5609	71774	972,785	272,6468
2008-06-17	71895	272,6468	92663,5609
NULL			
2008-06-17	71783	92663,5609	NULL
NULL			
2008-06-18	71920	3293,7761	7330,8972
NULL			
2008-06-18	71935	7330,8972	NULL
NULL			

3. Funkcja FIRST_VALUE zwraca wartość pola z pierwszego wiersza bieżącej partycji:

```
SELECT CAST([DueDate] AS DATE) AS Date, [SalesOrderID], [TotalDue],
       FIRST_VALUE ([TotalDue]) OVER (PARTITION BY [DueDate] ORDER
```

```

        BY [DueDate])      as FirstTotalDue
    FROM [SalesLT].[SalesOrderHeader]
    ORDER BY [DueDate];

```

Date	SalesOrderID	TotalDue	FirstTotalDue
2008-06-14	71776	87,0851	87,0851
2008-06-14	71832	39531,6085	87,0851
2008-06-17	71774	972,785	972,785
2008-06-17	71895	272,6468	972,785
2008-06-17	71783	92663,5609	972,785
2008-06-18	71920	3293,7761	3293,7761
2008-06-18	71935	7330,8972	3293,7761

4. Funkcja LAST_VALUE zwraca wartość pola z ostatniego wiersza bieżącej partycji:

```

SELECT CAST([DueDate] AS DATE) AS Date, [SalesOrderID], [TotalDue],
       LAST_VALUE ([TotalDue]) OVER (PARTITION BY [DueDate] ORDER BY
[DueDate]) as LastTotalDue
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];

```

Date	SalesOrderID	TotalDue	LastTotalDue
2008-06-14	71776	87,0851	39531,6085
2008-06-14	71832	39531,6085	39531,6085
2008-06-17	71774	972,785	92663,5609
2008-06-17	71895	272,6468	92663,5609
2008-06-17	71783	92663,5609	92663,5609
2008-06-18	71920	3293,7761	7330,8972
2008-06-18	71935	7330,8972	7330,8972

Funkcje analityczne

Serwer SQL udostępnia też cztery funkcje analityczne. Dwie z nich zwracają ranking wiersza w danej partycji, dwie kolejne percentyl dla przekazanego procentu.

Ranking wiersza może być obliczony za pomocą funkcji PERCENT_RANK oraz CUME_DIST:

1. Funkcja PERCENT_RANK zwraca ranking wiersza w danej partycji obliczany według wzoru $(RN - 1) / (RC - 1)$, gdzie RN to kolejny numer wiersza, a RC to liczba wierszy w danej partycji.
2. Funkcja CUME_DIST zwraca ranking wiersza w danej partycji obliczany według wzoru RS/RC , gdzie RS to liczba wierszy o wartości mniejszej od wiersza bieżącego bądź mu równej, a RC to liczba wierszy w danej partycji.

Poniższe zapytanie zwraca obliczony za pomocą tych funkcji ranking produktów w ramach kategorii utworzony na podstawie ich cen (najdroższy produkt z danej kategorii ma najwyższą pozycję, najtańszy — najniższą):

```
SELECT [ProductCategoryID], [ProductNumber], [ListPrice],  
       CUME_DIST () OVER (PARTITION BY [ProductCategoryID] ORDER BY  
[ListPrice] DESC) as CumeDist,  
       PERCENT_RANK () OVER (PARTITION BY [ProductCategoryID] ORDER BY  
[ListPrice] DESC) as PercentRank  
FROM [SalesLT].[Product]  
WHERE [ProductCategoryID] IN (8,9);  
-----
```

ProductCategoryID PercentRank	ProductNumber	ListPrice	CumeDist
8 0	HB-M918	120,27	0,25
8 0	HB-R956	120,27	0,25
8 0,285714285714286	HB-T928	91,57	0,375
8 0,428571428571429	HB-M763	61,92	0,625
8 0,428571428571429	HB-R720	61,92	0,625
8 0,714285714285714	HB-T721	46,09	0,75
8 0,857142857142857	HB-M243	44,54	1

8 0,857142857142857	HB-R504	44,54	1
9 0	BB-9108	121,49	0,333333333333333
9 0,5	BB-8107	101,24	0,666666666666667
9 1	BB-7421	53,99	1

Dwie ostatnie funkcje analityczne zwracają percentyl dla przekazanego procentu (wartości z zakresu od 0.0 do 1.0). Percentyl to wielkość, poniżej której występują wartości zadawanego procentu wierszy z danej partycji. Najczęściej używanym percentilem jest mediana, czyli percentyl 50%:

1. Funkcja PERCENTILE_CONT zwraca wynik, który nie musi pokrywać się z jakąkolwiek wartością w danej partycji.
2. W przypadku funkcji PERCENTILE_DISC zwrócony wynik zawsze pokrywa się z najbliższą (względem pozycji) wartością występującą w danej partycji.

W przypadku obu tych funkcji do określenia kolumny, której wartości będą analizowane, służy dodatkowa klauzula WITHIN GROUP:

```
SELECT [ProductCategoryID], [ProductNumber], [ListPrice],
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY [ListPrice] DESC)
                                         OVER (PARTITION BY [ProductCategoryID] ) as MedianCont,
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY [ListPrice] DESC)
                                         OVER (PARTITION BY [ProductCategoryID] ) as MedianDisc
FROM [SalesLT].[Product]
WHERE [ProductCategoryID] IN (16,17);
```

ProductCategoryID MedianCont	ProductNumber MedianDisc	ListPrice
16 356,425	364,09	FR-M94S-42 1364,50
16 356,425	364,09	FR-M94S-44 1364,50
16 356,425	364,09	FR-M94S-52 1364,50
16		FR-M94S-46 1364,50

356,425	364,09		
16		FR-M94S-38	1364,50
356,425	364,09		
16		FR-M94B-42	1349,60
356,425	364,09		
16		FR-M94B-44	1349,60
356,425	364,09		
16		FR-M94B-48	1349,60
356,425	364,09		
16		FR-M94B-46	1349,60
356,425	364,09		
16		FR-M94B-38	1349,60
356,425	364,09		
16		FR-M63S-40	364,09
364,09			356,425
16		FR-M63S-42	364,09
364,09			356,425
16		FR-M63S-46	364,09
364,09			356,425
16		FR-M63S-38	364,09
364,09			356,425
16		FR-M63B-38	348,76
364,09			356,425
16		FR-M63B-40	348,76
364,09			356,425
16		FR-M63B-44	348,76
364,09			356,425
16		FR-M63B-48	348,76
364,09			356,425
16		FR-M21S-40	264,05
364,09			356,425
16		FR-M21S-42	264,05
364,09			356,425
16		FR-M21S-44	264,05
364,09			356,425
16		FR-M21S-48	264,05
364,09			356,425
16		FR-M21S-52	264,05
364,09			356,425
16		FR-M21B-42	249,79
364,09			356,425

16 364,09	FR-M21B-44	249,79	356,425
16 364,09	FR-M21B-48	249,79	356,425
16 364,09	FR-M21B-52	249,79	356,425
16 364,09	FR-M21B-40	249,79	356,425
17 62,09	PD-R853	80,99	62,09
17 62,09	PD-T852	80,99	62,09
17 62,09	PD-M562	80,99	62,09
17 62,09	PD-M340	62,09	62,09
17 62,09	PD-R563	62,09	62,09
17 62,09	PD-R347	40,49	62,09
17 62,09	PD-M282	40,49	62,09

Podsumowanie

- Klauzula OVER pozwala zdefiniować partycje na wyłączny użytk funkcji grupujących, rankingu, okienkowych oraz analitycznych.
- Funkcje rankingu, okienkowe i analityczne wymagają posortowania partycji.
- Partycje mogą być dzielone na okienka.
- Funkcje rankingu są najprostszym i najwydajniejszym sposobem na ponumerowanie i podzielenie na strony zwracanych przez zapytanie wierszy.
- Okienka i funkcje analityczne pozwalają wyeliminować kosztowne podzapytania powiązane przy porównywaniu wierszy z wartościami z poprzednich lub następnych wierszy.
- Okienka i funkcje okienkowe pozwalają też pozbyć się kursorów przy obliczaniu sum narastających czy średnich ruchomych.

Zadania

- Policz, ile razy sprzedany został każdy produkt, a następnie ponumeruj wiersze wyniku tego zapytania na dwa sposoby: w jednej kolumnie umieść numery od jednego (najczęściej sprzedawany produkt) do ostatniego na podstawie pozycji wiersza z danym produktem, w drugiej wstaw numery na podstawie liczby sprzedaży danego towaru (towary sprzedane tyle samo razy powinny mieć taki sam numer). Częściowy wynik pokazany został poniżej:

ProductNumber column name)	SalesCount	(No column name)	(No
LJ-0192-L	10	1	1
VE-C304-S	10	2	1
SJ-0194-X	9	3	2
CA-1098	9	4	2
SJ-0194-L	8	5	3
RA-H123	8	6	3
...			

- Policz sumę zamówień na poziomie dni, miesiący i lat dat zapłaty za poszczególne zamówienia (wartości kolumny DueDate). Dodaj też do zapytania sumę wartości wszystkich zamówień. Częściowy wynik pokazany został poniżej:

year OverallSales	month	day	SalesPerDay	SalesPerMonth	SalesPerYear
2008 957276,3799	6	28	44688,6943	708766,4186	956303,5949
2008 957276,3799	6	29	119960,824	708766,4186	956303,5949
2008 957276,3799	6	14	103348,0081	708766,4186	956303,5949
2008 957276,3799	7	14	103348,0081	248509,9613	956303,5949
2008 957276,3799	7	1	2669,3183	248509,9613	956303,5949
2008 957276,3799	7	4	3754,9733	248509,9613	956303,5949

3. Policz różnicę wartości pomiędzy dwoma kolejnymi zamówieniami (przyjmij, że kolejność zamówień wyznaczana jest przez wartość SalesOrderID). Częściowy wynik pokazany został poniżej:

SalesOrderID	TotalDue	(No column name)
71774	972,785	NULL
71776	87,0851	-885,6999
71780	42452,6519	42365,5668
71782	43962,7901	1510,1382

Rozdział 8. Podzapytania

- W jaki sposób serwery bazodanowe wykonują podzapytania?
- Jak używać podzapytań w roli zmiennych?
- Czym się różni podzapytanie powiązane od niepowiązanego?
- Co oznacza skrót CTE?
- Co zyskujemy, odczytując dane poprzez podzapytania zamiast bezpośrednio z tabel?
- Jak działają operatory EXISTS, ANY oraz ALL?

Czym są podzapytania?

Język SQL pozwala na zagnieżdżenie zapytania, czyli umieszczenie instrukcji `SELECT` wewnętrznych innej instrukcji `SELECT`. **Tak zagnieżdżone zapytania nazywa się podzapytaniami.**

Serwery bazodanowe w pierwszej kolejności wykonują zapytania wewnętrzne, dzięki czemu ich wyniki mogą być odczytywane w zewnętrznych instrukcjach `SELECT`. Pozwala to na używanie podzapytań do sprawdzania na bieżąco prawdziwości konstruowanych warunków logicznych, posługiwanie się w wyrażeniach faktycznymi danymi (a nie stałymi) czy poprawę czytelności zapytań.

W zależności od typu zwracanych przez zapytania wewnętrznych wartości podzapytania dzieli się na:

1. Podzapytania zwracające pojedynczą wartość skalarną, np. nazwisko sprzedawcy, który sprzedał najczęściej towarów.
2. Podzapytania zwracające listę wartości, np. identyfikatory sprzedanych w danym miesiącu towarów.
3. Podzapytania zwracające dane tabelaryczne, np. dane sprzedawców uzupełnione o liczbę i wartość zrealizowanych przez nich zamówień.

Niezależnie od typu zwracanych wartości podzapytania mogą być powiązane lub niepowiązane:

1. W podzapytaniach niepowiązanych zapytanie wewnętrzne jest wykonywane tylko raz, a więc zwraca jeden wynik.
2. W podzapytaniach powiązanych zapytanie wewnętrzne jest wykonywane dla każdego wiersza zwróconego przez zapytanie zewnętrzne, a więc zwraca tyle wyników, ile wierszy liczy wynik zapytania zewnętrznego.

 Wskazówka	Zapytania wewnętrzne umieszcza się w nawiasach. Pewnym wyjątkiem od tej reguły są omówione w dalszej części rozdziału CTE — w ich przypadku definicja zapytania wewnętrznego znajduje się w nawiasie, ale jego nazwę umieszcza się poza nawiasem.
---	---

Podzapytania jako zmienne

W proceduralnych i obiektowych językach programowania zmienne są pojemnikami o określonych nazwach, w których można przechowywać wartości pewnych typów. Nazwę i typ zmiennej najczęściej określa się podczas jej deklarowania, a wartość początkowa jest nadawana zmiennej w trakcie jej inicjowania.

Dwie najważniejsze wykonywane na zmiennych operacje to:

1. **Przypisanie wartości** pozwalające zmieniać wartość zmiennej w trakcie działania programu.
2. **Odczytanie wartości**, czyli posłużenie się nazwą zmiennej w celu pobrania jej aktualnej wartości.

Podzapytania zwracające pojedynczą wartość są odpowiednikami zmiennych typów prostych, a podzapytania zwracające listę wartości — zmiennych tabelarycznych. Najwyraźniej widać to podobieństwo na przykładzie podzapytań niepowiązanych.

 Wskazówka	Ta część książki jest poświęcona wyłącznie pobieraniu danych, dlatego omawiamy w niej tylko podzapytania odczytujące dane. W większości serwerów bazodanowych podzapytania mogą być też używane do modyfikowania danych, wtedy wewnętrzną instrukcję SELECT umieszcza się w instrukcji INSERT, UPDATE lub DELETE.
---	---

Podzapytania niepowiązane

Przypuśćmy, że chcemy odczytać dane klienta, który złożył określone zamówienie^[1]. W tabeli [SalesLT].[SalesOrderHeader] zapisane są identyfikatory klientów, ale nie znajdziemy w niej ich danych. Żeby wykonać zadanie, musielibyśmy najpierw odczytać z tabeli SalesLT].[SalesOrderHeader] identyfikator klienta, który złożył interesujące nas zamówienie:

```
SELECT [CustomerID]
FROM [SalesLT].[SalesOrderHeader]
WHERE [SalesOrderNumber] = 'S071832';
```

```
-----
```

CustomerID

29922

Znając ten identyfikator, możemy już odczytać dane klienta:

```
SELECT [CompanyName], [FirstName], [LastName], [EmailAddress]
FROM [SalesLT].[Customer]
WHERE [CustomerID] = 29922;
```

```
-----
```

CompanyName	FirstName	LastName	EmailAddress
Closest Bicycle Store	Pamala	Kotc	pamala0@adventure-works.com

Podzapytanie pozwala nam wykonać jednocześnie obie te operacje:

```
SELECT [CompanyName], [FirstName], [LastName], [EmailAddress]
FROM [SalesLT].[Customer]
WHERE [CustomerID] =
      (SELECT [CustomerID]
       FROM [SalesLT].[SalesOrderHeader]
       WHERE [SalesOrderNumber] = 'S071832');
```

```
-----
```

CompanyName	FirstName	LastName	EmailAddress
Closest Bicycle Store	Pamala	Kotc	pamala0@adventure-works.com

Żeby wykonać to podzapytanie, serwer bazodanowy:

1. Odczytał z tabeli [SalesLT].[Customer] dane wszystkich

klientów.

2. Sprawdził warunek w zewnętrznej klauzuli WHERE. W tym celu musiał wykonać zapytanie wewnętrzne i użyć jego wyniku do wybrania klienta, która złożył dane zamówienie.

 Wskazówka	Nazwy kolumn w podzapytaniach należy poprzedzać nazwami lub aliasami nazw tabel. W ten sposób unikniemy trudnych do wykrycia błędów logicznych — gdyby nazwa kolumny klucza jednej z tabel była inna, niż się spodziewaliśmy, np. kolumna klucza obcego w tabeli [SalesLT].[SalesOrderHeader] nazywałaby się ID, serwer bazodanowy nie zgłosiłby błędu. Wynika to ze sposobu sprawdzania nazw przez serwery — jeśli kolumna o podanej nazwie nie istnieje w tabeli odczytywanej przez zapytanie wewnętrzne, sprawdzają one, czy kolumna o tej nazwie istnieje w tabeli odczytywanej przez zapytanie zewnętrzne. Jeżeli tak, zostanie ona użyta do sprawdzenia warunku WHERE. Ponieważ porównywane ze sobą byłyby dane z tej samej kolumny, zapytanie zwróciłoby nazwy wszystkich podkategorii.
--	--

Podzapytania mogą być też umieszczane w klauzuli SELECT. Gdybyśmy chcieli dodać do listy nazw i cen produktów kolumnę zawierającą średnią cenę wszystkich produktów, a nasz serwer bazodanowy nie pozwalałby na użycie klauzuli OVER, najprościej byłoby najpierw policzyć tę średnią:

```
SELECT AVG([ListPrice])
FROM [SalesLT].[Product];
```

```
-----  
742,4952
```

a następnie umieścić otrzymany wynik jako stałą w zapytaniu:

```
SELECT [Name], [ListPrice], [ListPrice] - 742.4952 AS [Różnica do
średniej]
FROM [SalesLT].[Product];
```

```
-----  
Name          ListPrice   Różnica do średniej
HL Road Frame - Black, 58    1431,50    689.0048
HL Road Frame - Red, 58      1431,50    689.0048
Sport-100 Helmet, Red        34,99     -707.5052
Sport-100 Helmet, Black      34,99     -707.5052
Mountain Bike Socks, M        9,50      -732.9952
...
```

Takie rozwiązanie ma jednak bardzo poważną wadę — po każdej zmianie ceny oraz po dodaniu lub usunięciu produktu musimy zmienić nasze zapytanie. Jeżeli tego nie zrobimy, jego wynik będzie nieprawdziwy.

Możemy uniknąć tego problemu, wyliczając w zapytaniu wewnętrznym średnią cenę produktów i dodając otrzymaną wartość do wyniku zapytania zewnętrznego, czyli zastępując zapytanie ze stałą podzapytaniem:

```
SELECT [Name], [ListPrice], [ListPrice] -  
       (SELECT AVG([ListPrice])  
        FROM [SalesLT].[Product]) AS [Różnica do średniej]  
     FROM [SalesLT].[Product];
```

Name	ListPrice	Różnica do średniej
HL Road Frame - Black, 58	1431,50	689.0048
HL Road Frame - Red, 58	1431,50	689.0048
Sport-100 Helmet, Red	34,99	-707.5052
Sport-100 Helmet, Black	34,99	-707.5052
Mountain Bike Socks, M	9,50	-732.9952

...

Jak mogliśmy zauważyc, podzapytania, tak jak zmienne, mogą być też częścią wyrażeń. Na przykład żeby obliczyć różnicę pomiędzy ceną danego produktu a średnią ceną wszystkich produktów, wystarczyło odjąć od ceny danego produktu wynik podzapytania.

Dopuszczalne jest również używanie wyrażeń w zapytaniach wewnętrznych. Kolejne zapytanie zwraca nazwy towarów tańszych o 1094,28 niż towar o kodzie FR-R92R-581:

```
SELECT Z.[Name]  
  FROM [SalesLT].[Product] AS Z  
 WHERE Z.ListPrice =  
       (SELECT W.ListPrice - 1094.28  
        FROM [SalesLT].[Product] AS W  
       WHERE W.ProductNumber = 'FR-R92R-58');
```

```
Name  
LL Road Frame - Black, 58  
LL Road Frame - Black, 60  
LL Road Frame - Black, 62  
LL Road Frame - Red, 44  
LL Road Frame - Red, 48  
LL Road Frame - Red, 52
```

...

Podzapytania zwracające listę wartości

Wróćmy na chwilę do przykładu, w którym wynik zapytania wewnętrznego został użyty do wybrania zwróconych przez podzapytanie wierszy. Podzapytanie zostało wykonane przez serwer bazodanowy tylko dlatego, że wewnętrzna klauzula `WHERE` zawierała warunek prawdziwy dla dokładnie jednego zamówienia (był to warunek `WHERE [SalesOrderNumber] = 'S071832'`), a to samo zamówienie nie może być złożone przez kilku klientów. W rezultacie podzapytanie zwróciło jedną wartość.

Gdybyśmy zmienili ten warunek tak, żeby był prawdziwy dla kilku zamówień, to każde z nich mogłoby być złożone przez różne osoby. W takim przypadku porównanie wyniku podzapytania (numerów kilku zamówień) za pomocą operatora `=` byłoby nielogiczne i takie podzapytanie nie zostałoby wykonane przez serwer bazodanowy:

```
SELECT [CustomerID]  
FROM [SalesLT].[SalesOrderHeader]  
WHERE DueDate = '2008-06-19 00:00:00.000';
```

CustomerID

29653
29975

```
SELECT S.Firma  
FROM dbo.Spedytorzy AS S  
WHERE S.ID =  
(SELECT Z.[ID spedytora]
```

```
FROM dbo.Zamówienia AS Z  
WHERE Z.[ID zamówienia] BETWEEN 35 AND 37);
```

Msg 512, Level 16, State 1, Line 50
Subquery returned more than 1 value. This is not permitted when the
subquery follows =, !=, <, <= , >, >= or when the subquery is used as an
expression.

Próba wykonania zapytania zakończyła się błędem z powodu użycia operatora =. **Operatory porównania, takie jak =, > czy <, nie mogą być używane do sprawdzania listy wartości, ponieważ serwer bazodanowy nie jest w stanie jednoznacznie zinterpretować ich znaczenia (w tym przypadku nie wiadomo, czy numer podkategorii miałby być równy wszystkim zwróconym przez zapytanie wewnętrzne numerom, czy może dowolnemu z nich).**

Jeżeli podzapytanie zwraca listę wartości (czyli jeśli wynik zapytania składa się z wielu wierszy, ale tylko jednej kolumny), do porównywania jego wyniku należy użyć operatora IN. Operator IN zwraca prawdę, jeśli choć jedna ze znajdujących się na liście wartości spełnia umieszczony w nim warunek:

```
SELECT [CompanyName], [FirstName], [LastName], [EmailAddress]  
FROM [SalesLT].[Customer]  
WHERE [CustomerID] IN  
    (SELECT [CustomerID]  
     FROM [SalesLT].[SalesOrderHeader]  
     WHERE DueDate = '2008-06-19 00:00:00.000');
```

CompanyName	FirstName	LastName	EmailAddress
Thrilling Bike Tours works.com	Pei	Chow	pei0@adventure-
Sports Products Store works.com	Walter	Mays	walter1@adventure-

Używając operatora IN, powinniśmy liczyć się z tym, że podzapytanie zwróci więcej niż jeden wiersz (w tym przypadku mogą to być dane wielu klientów). To, że serwer bazodanowy nie zgłosi błędu, nie zawsze jest zaletą. Ponieważ wynik zapytania wewnętrznego nie jest

bezpośrednio widoczny, otrzymane dane mogą być inne, niż się spodziewaliśmy. Dlatego operator `IN` powinien być stosowany tylko wtedy, gdy nie ma logicznego powodu, dla którego zapytanie wewnętrzne nie mogłoby zwrócić listy wartości.

Ponieważ firma AdventureWorks wysyła zamówienia do różnych miast, w poniższym przypadku użycie operatora `IN` jest jak najbardziej uzasadnione:

```
SELECT [City]
FROM [SalesLT].[Address] AS Z
WHERE Z.AddressID IN
    (SELECT [ShipToAddressID]
     FROM [SalesLT].[SalesOrderHeader]
     WHERE [ShipDate] - [OrderDate]>7
    );
```

City

Auburn

Podzapytania niezwracające żadnych wartości

Zapytanie wewnętrzne może nie zwrócić żadnych danych. Wówczas serwer bazodanowy nie zgłosi błędu, niezależnie od tego, czy do sprawdzania wyniku zapytania wewnętrznego został użyty operator porównania, czy operator `IN`. **W obu przypadkach rezultatem porównania wyniku zapytania wewnętrznego będzie wartość nieznana**, która w klauzuli `WHERE` jest traktowana tak samo jak fałsz, a więc całe podzapytanie nie zwróci żadnych danych:

```
SELECT [City]
FROM [SalesLT].[Address] AS Z
WHERE Z.AddressID IN
    (SELECT [ShipToAddressID]
     FROM [SalesLT].[SalesOrderHeader]
     WHERE [ShipDate] - [OrderDate]>365
    );
```

(0 row(s) affected)

Zagnieżdżanie podzapytań

Zanim przejdziemy do omówienia zagnieżdżonych zapytań, pokażemy na przykładzie, dlaczego tak ważne jest używanie aliasów w podzapytaniach. Poniższe zapytanie powinno zwrócić nazwy miast, do których wysyłka zamówienia trwała dłużej niż tydzień:

```
SELECT [City]
FROM [SalesLT].[Address]
WHERE AddressID IN
    (SELECT [AddressID]
     FROM [SalesLT].[SalesOrderHeader]
     WHERE [ShipDate] - [OrderDate]>7
    );
```

```
-----
City
Ottawa
Burnaby
Seattle
...
```

Tymczasem zwróciło ono aż 450 wierszy, czyli wszystkie wiersze z tabeli [SalesLT].[Address]. Powodem jest sposób, w jaki serwery bazodanowe wykonują podzapytania. Ponieważ kolumna AddressID nie występuje w tabeli [SalesLT].[SalesOrderHeader], odczytana została kolumna o tej nazwie znajdująca się w tabeli [SalesLT].[Address]. W rezultacie porównane zostały ze sobą identyfikatory tych samych lokalizacji. Gdybyśmy konsekwentnie posługiwali się aliasami, błąd zostałby natychmiast wykryty:

```
SELECT Z.[City]
FROM [SalesLT].[Address] AS Z
WHERE Z.AddressID IN
    (SELECT W.[AddressID]
     FROM [SalesLT].[SalesOrderHeader] AS W
     WHERE W.[ShipDate] - W.[OrderDate]>7
    );
```

Msg 207, Level 16, State 1, Line 95

Invalid column name 'AddressID'.

Po tym wstępnie możemy już zacząć zagnieźdżać kolejne podzapytania. W języku SQL dopuszczalne jest zagnieźdzanie podzapytań, czyli umieszczanie ich wewnątrz innych podzapytań. Przypuśćmy, że chcemy za pomocą podzapytania odczytać nazwiska i imiona osób, które kiedykolwiek kupiły jakiś produkt z 40-procentową zniżką. W tym celu:

1. Najpierw musimy odczytać identyfikatory zamówień zrealizowanych z taką zniżką.
2. Następnie musimy odczytać identyfikatory osób, które złożyły te zamówienia.
3. Na podstawie identyfikatorów klientów odczytamy ich imiona i nazwiska:

```
SELECT Z.[FirstName],Z.[LastName]
FROM [SalesLT].[Customer] AS Z
WHERE Z.[CustomerID] IN
    (SELECT W1.[CustomerID]
     FROM [SalesLT].[SalesOrderHeader] AS W1
     WHERE W1.[SalesOrderID] IN
         (SELECT W2.[SalesOrderID]
          FROM [SalesLT].[SalesOrderDetail]AS W2
          WHERE [UnitPriceDiscount] = 0.40 ));
```

FirstName	LastName
Donald	Blanton
Pamala	Kotc
Jeffrey	Kurtz
Frank	Campbell
Krishna	Sunkammurali
Roger	Van Houten
Raja	Venugopal

Podzapytania powiązane

We wszystkich przedstawionych do tej pory podzapytaniach **zapytanie wewnętrzne było niezależne od zewnętrznego i mogło być wykonane samodzielnie**. Tymczasem język SQL pozwala na odwołanie się w klauzuli FROM zapytania wewnętrznego do wyniku zapytania zewnętrznego. Otrzymane w ten sposób **podzapytanie powiązane wyróżnia to, że zapytanie wewnętrzne jest wykonywane osobno dla każdego wiersza zwróconego przez zapytanie zewnętrznego i w konsekwencji nie może być wykonane samodzielnie**.

Jeżeli sposób łączenia wierszy przez podzapytania powiązane wydał Ci się znajomy, to bardzo dobrze — przypomina on opisane w rozdziale 5. łączenie tabel. Dlatego w pierwszym przykładzie pokażemy, jak zastąpić złączenie naturalne zwracającym te same dane podzapytaniem powiązanym.

Zaczniemy od zapytania zwracającego nazwiska klientów, którzy złożyli przynajmniej jedno zamówienie o wartości przekraczającej 100 000:

```
SELECT [LastName]
FROM [SalesLT].[Customer] AS C
JOIN [SalesLT].[SalesOrderHeader] AS H
    ON C.CustomerID=H.CustomerID
WHERE [TotalDue]>100000;
```

```
LastName
Eminhizer
Eminhizer
Sunkammurali
```

Ponieważ pan *Eminhizer* zrealizował dwa tak duże zamówienia, jego nazwisko zostało w wyniku zapytania powtórzone. Jest to związane ze sposobem, w jaki serwery bazodanowe łączą tabele — jeżeli nie jesteś pewien, dlaczego tak się stało, przerwij wykonywanie ćwiczenia i wróć do rozdziału 5.

Dane klientów, którzy złożyli chociaż jedno tak duże zamówienie, możemy też odczytać za pomocą następującego podzapytania powiązanego:

```
SELECT [LastName]
FROM [SalesLT].[Customer] AS C
WHERE 100000 <= ANY
    (SELECT [TotalDue]
     FROM [SalesLT].[SalesOrderHeader] AS H
     WHERE C.CustomerID=H.CustomerID);
```

LastName

Eminhizer

Sunkammurali

Kolejność wykonywania podzapytania była następująca:

1. Został odczytany wiersz z tabeli `SalesLT].[Customer]`. Ponieważ na tym etapie nie wiadomo jeszcze, czy trafi on do wyniku podzapytania, nazywa się go wierszem kandydującym.
2. Zostało wykonane zapytanie wewnętrzne:
 - a. Odczytano wartość kolumny `CustomerID` wiersza kandydującego.
 - b. Z tabeli `[SalesLT].[SalesOrderHeader]` zostały wybrane wiersze spełniające warunek `C.CustomerID=H.CustomerID`.
 - c. Z wybranych wierszy (zawierających zamówienia złożone przez danego klienta) została odczytana wartość kolumny `TotalDue`.
3. Na podstawie wyniku zapytania wewnętrznego został sprawdzony warunek `WHERE`, w tym przypadku `100000 <= ANY TotalDue`. Wiersz kandydujący, dla którego był on prawdziwy, został dodany do wyniku podzapytania.
4. Cała operacja była powtarzana aż do odczytania wszystkich wierszy tabeli `[SalesLT].[Customer]`.



Ponieważ operator porównania nie rozróżnia wierszy spełniających określony w nim warunek, wynik podzapytania nie zawiera powtórzonych danych pana Eminhizera.

Kolejny przykład porównuje zapytania powiązane z niepowiązanymi. Punktem wyjścia będzie podzapytanie niepowiązane zwracające nazwy najdroższych produktów:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product]
WHERE [ListPrice] =
    (SELECT MAX([ListPrice])
     FROM [SalesLT].[Product]);
```

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27

Gdybyśmy chcieli poznać nazwę produktu, który jest na drugim miejscu pod względem ceny, moglibyśmy wyeliminować z wyniku dane najdroższego produktu, zagnieżdżając podzapytanie niepowiązane:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product] AS Z
WHERE [ListPrice] =
    (SELECT MAX([ListPrice])
     FROM [SalesLT].[Product] AS W1
     WHERE W1.ListPrice<
         (SELECT MAX([ListPrice])
          FROM [SalesLT].[Product] AS W2
          ));

```

Name	ListPrice
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99
Mountain-100 Silver, 44	3399,99

Mountain-100 Silver, 48 3399,99

W ten sposób nie odczytamy jednak informacji o produktach znajdujących się na dowolnej pozycji pod względem ceny^[2]. Spróbujmy więc zastosować podzapytanie powiązane.

Pozycję w naszej hierarchii produktów, określona na podstawie ich cen, zwraca poniższe zapytanie:

```
SELECT COUNT(DISTINCT [ListPrice])
FROM [SalesLT].[Product]
WHERE [ListPrice] > 3578.27 --3399.99 zwróci 1 itd.;

-----
0
```

Gdybyśmy tylko zamiast wpisywać cenę (w tym przypadku 3578.27), odczytali ją z tabeli SalesLT].[Product], rozwiązałybyśmy zadanie:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product] AS Z
WHERE 0 =
      (SELECT COUNT(DISTINCT [ListPrice])
       FROM [SalesLT].[Product] AS W
       WHERE W.ListPrice>Z.ListPrice);
```

```
-----
Name                     ListPrice
Road-150 Red, 62      3578,27
Road-150 Red, 44      3578,27
Road-150 Red, 48      3578,27
Road-150 Red, 52      3578,27
Road-150 Red, 56      3578,27
```

Otrzymany wynik podzapytania powiązanego możemy łatwo porównać z interesującymi nas pozycjami w hierarchii cen produktów — 0 oznaczałoby, że interesują nas nazwy najdroższych produktów, 1 w naszym przypadku będzie oznaczać, że zapytanie ma zwrócić nazwy produktów, które są na drugiej pozycji pod względem ceny, i tak dalej:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product] AS Z
```

```

WHERE 1 =
    (SELECT COUNT(DISTINCT [ListPrice])
     FROM [SalesLT].[Product] AS W
     WHERE W.ListPrice>Z.ListPrice);
-----
```

Name	ListPrice
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99
Mountain-100 Silver, 44	3399,99
Mountain-100 Silver, 48	3399,99

Podzapytania (zarówno powiązane, jak i niepowiązane), w których zapytania wewnętrzne wywołują funkcje grupujące, są regularnie używane w tych przypadkach, gdy niemożliwe jest użycie klauzuli `OVER`. W kolejnym przykładzie skorzystamy z nich w celu odczytania informacji o największych i najmniejszych zamówieniach złożonych w danym dniu przez poszczególnych sprzedawców.

Zaczniemy od odczytania informacji o zamówieniach. Ponieważ zapytanie zewnętrzne łączy się z wewnętrznym na podstawie daty, z każdego dnia zostaną wybrane zamówienia o największych opłatach związanych z ich wysłaniem:

```

SELECT Z.SalesOrderID, Z.CustomerID, Z.OrderDate, Z.Freight
FROM [SalesLT].[SalesOrderHeader] AS Z
WHERE Z.Freight =
    (SELECT MAX(W.Freight)
     FROM [SalesLT].[SalesOrderHeader] AS W
     WHERE W.OrderDate = Z.OrderDate);
-----
```

SalesOrderID	CustomerID	OrderDate	Freight
77774	29847	2014-04-01 00:00:00.000	22,0087
71784	29736	2008-06-01 00:00:00.000	
2714,0458			

W analogiczny sposób odczytamy informacje o wysłanych w kolejnych dniach zamówieniach o najmniejszych opłatach — wystarczy zastąpić funkcję `MAX()` funkcją `MIN()`. Do połączenia wyników obu podzapytań użyjemy operatora `UNION ALL`, a w celu ułatwienia interpretacji tak

uzyskanego wyniku do obu podzapytań dodamy kolumnę ze stałą informującą, czy dany wiersz został zwrócony przez pierwsze, czy drugie podzapytanie:

```
SELECT Z.SalesOrderID, Z.CustomerID, Z.OrderDate, 'Max Freight: ' +
CAST(Z.Freight AS CHAR(7))

FROM [SalesLT].[SalesOrderHeader] AS Z

WHERE Z.Freight =

(SELECT MAX(W.Freight)

FROM [SalesLT].[SalesOrderHeader] AS W

WHERE W.OrderDate = Z.OrderDate)

UNION ALL

SELECT Z.SalesOrderID, Z.CustomerID, Z.OrderDate, 'Min Freight: ' +
CAST(Z.Freight AS CHAR(7))

FROM [SalesLT].[SalesOrderHeader] AS Z

WHERE Z.Freight =

(SELECT MIN(W.Freight)

FROM [SalesLT].[SalesOrderHeader] AS W

WHERE W.OrderDate = Z.OrderDate);
```

SalesOrderID	CustomerID	OrderDate	(No column name)
77774 22.01	29847	2014-04-01 00:00:00.000	Max Freight:
71784 2714.05	29736	2008-06-01 00:00:00.000	Max Freight:
71946 0.97	29741	2008-06-01 00:00:00.000	Min Freight:
77774 22.01	29847	2014-04-01 00:00:00.000	Min Freight:

Podzapytania a związania

Ważną różnicą pomiędzy podzapytaniem połączonym a złączeniem jest to, że w przypadku podzapytania zwrócone mogą być jedynie kolumny z zewnętrznej tabeli, podczas gdy złączenie daje nam dostęp do kolumn z wszystkich połączonych tabel. Różnicę tę pokazuje kolejny przykład — ponieważ w klauzuli `FROM` zapytania zewnętrznego występuje jedynie tabela `[SalesLT].[Customer]`, tylko jej kolumny możemy odczytać za pomocą zewnętrznej klauzuli `SELECT`:

```

SELECT [LastName], TotalDue
FROM [SalesLT].[Customer] AS C
WHERE 10000 <= ANY
    (SELECT [TotalDue]
     FROM [SalesLT].[SalesOrderHeader] AS H
     WHERE C.CustomerID=H.CustomerID);;
-----
```

Msg 207, Level 16, State 1, Line 198

Invalid column name 'TotalDue'.

```

SELECT [LastName], TotalDue
FROM [SalesLT].[Customer] AS C
JOIN [SalesLT].[SalesOrderHeader] AS H
    ON C.CustomerID=H.CustomerID
WHERE [TotalDue]>=100000
ORDER BY [LastName];
-----
```

LastName	TotalDue
Eminhizer	119960,824
Sunkammurali	108597,9536

Zapytanie zewnętrzne może też odwoływać się do tej samej tabeli co zapytanie wewnętrzne. Najbliższym odpowiednikiem tego typu podzapytań powiązanych są złączenia własne (złączenia tabeli z nią samą). Jednak podzapytania powiązane pozwalają na użycie funkcji grupujących w sposób niedopuszczalny w złączeniach, co pokazuje kolejny przykład.

Spróbujmy za pomocą podzapytania powiązanego wyszukać towary, które zostały sprzedane za cenę dwukrotnie przekraczającą swoją średnią wartość. Ponieważ w klauzuli `FROM` niemożliwe jest porównywanie wierszy (w tym przypadku wartości każdego zamówienia) z grupami (tutaj średnią wartością zamówień dla danego towaru), poniższe podzapytanie nie może być przekształcone w złączenie:

```

SELECT OH.SalesOrderID, OD.ProductID, OD.LineTotal
FROM [SalesLT].[SalesOrderHeader] AS OH
JOIN [SalesLT].[SalesOrderDetail] AS OD
```

```

        ON OD.SalesOrderID = OH.SalesOrderID
WHERE OD.LineTotal >
      (SELECT AVG(OD1.LineTotal)*2
       FROM [SalesLT].[SalesOrderHeader] AS OH1
      JOIN [SalesLT].[SalesOrderDetail] AS OD1
        ON OD1.ProductID = OD.ProductID);
-----
```

SalesOrderID	ProductID	LineTotal
71783	976	19136.137500
71783	974	12568.955308
71783	715	444.036175
71783	711	274.234125
71784	714	269.946000

...

Podsumowując: **podzapytania powiązane są wykonywane podobnie jak złączenia — w obu przypadkach serwer bazodanowy porównuje każdy wiersz jednej tabeli z każdym wierszem drugiej tabeli.** Jednak do porównania są używane inne operatory, przede wszystkim w podzapytaniach można używać operatora IN.

Obie techniki mają też unikatowe zalety:

1. W podzapytaniach można używać funkcji grupujących i porównywać ich wyniki z wartościami pojedynczych wierszy.
2. Wyniki złączeń mogą zawierać dane z wszystkich połączonych tabel.

 Wskaźówka	<p>Ponieważ większość serwerów bazodanowych złączenia wykonuje równie wydajnie jak podzapytania, wybierając pomiędzy nimi, należy kierować się wygodą (jeżeli łatwiej jest nam rozwiązać problem za pomocą podzapytania, wybierzmy tę technikę) oraz przejrzystością kodu (jeżeli złączenie pozwala na napisanie krótszego i czytelniejszego zapytania, wybierzmy złączenie). Wyjątkiem od tej reguły jest sytuacja, w której kilkakrotnie odwołujemy się do wyników tych samych obliczeń — wtedy należy użyć złączenia, bo w przypadku podzapytania serwer bazodanowy może kilkakrotnie wykonywać te same operacje. Przykład ilustrujący to zagadnienie znajduje się w kolejnym rozdziale.</p>
---	---

Podzapytania jako źródła danych

Jeżeli zapytanie wewnętrzne zwraca dane tabelaryczne, możemy odwoływać się do jego wyniku w klauzuli `FROM` zapytania zewnętrznego. Pozwala to:

1. Uprościć zapytanie i poprawić jego czytelność — ponieważ zapytanie wewnętrzne jest wykonywane jako pierwsze, zdefiniowane w nim aliasy kolumn mogą być używane w każdej klauzuli zapytania zewnętrznego.
2. Dynamicznie filtrować wiersze i wyliczać dane bazowe dla zapytań zewnętrznych.

Są dwa rodzaje zapytań wewnętrznych zwracających dane tabelaryczne: tabele pochodne i CTE.

Tabele pochodne

Tabela pochodna to nic innego jak wynik zapytania wewnętrznego, które zostało zdefiniowane w klauzuli `FROM` zapytania zewnętrznego.

Tabele pochodne są dostępne tylko w ramach tych podzapytań, w których zostały zdefiniowane, i po zakończeniu ich wykonywania są automatycznie usuwane.

Zaczniemy od najprostszego przykładu, w którym tabela pochodna zawiera wybrane wiersze i kolumny tabeli bazowej:

```
SELECT [Name],[ListPrice],[Size],[Weight],[Color]
FROM [SalesLT].[Product]
WHERE [SellEndDate] IS NULL
AND [ListPrice]>50;
```

Name	ListPrice	Size	Weight
Color			
HL Road Frame - Black, 58 Black	1431,50	58	1016.04
HL Road Frame - Red, 58	1431,50	58	1016.04
HL Road Frame - Red, 62	1431,50	62	1043.26
...			

Jeżeli umieścimy powyższe zapytanie w klauzuli `FROM` innego zapytania

i nazwiemy otrzymaną w ten sposób tabelę pochodną, nadając jej alias, to będziemy mogli odczytać jej zawartość tak samo, jak odczytuje się zawartość standardowych tabel czy widoków:

```
SELECT [Name],[Color]
FROM (
    SELECT [Name],[ListPrice],[Size],[Weight],[Color]
    FROM [SalesLT].[Product]
    WHERE [SellEndDate] IS NULL
        AND [ListPrice]>50) AS W
WHERE [Color] = 'Black';
```

Name	Color
HL Road Frame - Black, 58	Black
LL Road Frame - Black, 58	Black
LL Road Frame - Black, 60	Black

...

W następnym przykładzie wykorzystamy tabelę pochodną do poprawienia czytelności kodu. Punktem wyjścia będzie zapytanie zwracające informacje o liczbie klientów o różnych tytułach, mieszkających w różnych krajach i miastach:

```
SELECT 'Title: ' +C.[Title], A.CountryRegion + ' ' +A.City, COUNT
(A.AddressID) AS Nr
FROM [SalesLT].[Customer] AS C
JOIN [SalesLT].[CustomerAddress] AS CA
    ON C.CustomerID=CA.CustomerID
JOIN [SalesLT].[Address] AS A
    ON CA.AddressID=A.AddressID
WHERE 'Title: ' +C.[Title] = 'Title: Ms.' GROUP BY 'Title: ' +C.[Title],
A.CountryRegion + ' ' +A.City;
```

(No column name)	(No column name)	Nr
Title: Ms.	Canada Brossard	1
Title: Ms.	Canada Burnaby	1
Title: Ms.	Canada Calgary	3
Title: Ms.	Canada Dorval	1

...

Ponieważ klauzula SELECT jest wykonywana po klauzulach WHERE i GROUP BY, nie możemy posłużyć się w nich aliasami nazw kolumn:

```
SELECT 'Title: ' +C.[Title] AS FullTitle, A.CountryRegion + ' ' +A.City  
AS Adress, COUNT (A.AddressID) AS Nr  
  
FROM [SalesLT].[Customer] AS C  
  
JOIN [SalesLT].[CustomerAddress] AS CA  
    ON C.CustomerID=CA.CustomerID  
  
JOIN [SalesLT].[Address] AS A  
    ON CA.AddressID=A.AddressID  
  
WHERE FullTitle = 'Title: Ms.'  
GROUP BY FullTitle,Adress;
```

Msg 207, Level 16, State 1, Line 248

Invalid column name 'FullTitle'.

Msg 207, Level 16, State 1, Line 249

Invalid column name 'FullTitle'.

Msg 207, Level 16, State 1, Line 249

Invalid column name 'Adress'.

Możemy rozwiązać ten problem za pomocą tabeli pochodnej. Wystarczy przenieść wyrażenia znajdujące się na tym samym poziomie grupowania do zapytania wewnętrznego i w nim zdefiniować odpowiednie aliasy:

```
SELECT FullTitle,Adress, COUNT (Z.AddressID) AS Nr  
  
FROM  
  
    (SELECT 'Title: ' +C.[Title] AS FullTitle, A.CountryRegion + ' '  
+A.City AS Adress, CA.AddressID  
  
FROM [SalesLT].[Customer] AS C  
  
JOIN [SalesLT].[CustomerAddress] AS CA  
    ON C.CustomerID=CA.CustomerID  
  
JOIN [SalesLT].[Address] AS A  
    ON CA.AddressID=A.AddressID  
    ) AS Z  
  
WHERE FullTitle = 'Title: Ms.'
```

```
GROUP BY FullTitle,Adress;
```

```
-----  
FullTitle      Adress          Nr  
Title: Ms.     Canada Burnaby  1  
Title: Ms.     Canada Calgary   3  
Title: Ms.     Canada Dorval    1
```

...

Tabele pochodne są używane nie tylko do poprawiania czytelności i upraszczania zapytań. Możliwość ograniczenia i dowolnego przekształcenia odczytywanych w zapytaniu zewnętrznym danych możemy wykorzystać do przygotowania tych danych, np. do ponumerowania wierszy i późniejszego ich podziału na strony.

Podział wierszy na strony jest często wykonywany przez programy klienckie, np. na stronach WWW z reguły nie umieszcza się całych wyników zapytań, tylko ich określone części, i dodaje się przyciski umożliwiające przeglądanie pozostałych danych. My podobną funkcjonalność uzyskaliśmy za pomocą przedstawionych w rozdziale 4. klauzul `OFFSET` i `FETCH NEXT`, teraz ten sam wynik otrzymamy, korzystając z tabeli pochodnej i funkcji rankingu:

1. W pierwszej kolejności należy posortować dane w wybrany sposób, np. według wartości zamówień, i ponumerować otrzymane wiersze:

```
SELECT [SalesOrderID],[TotalDue],  
       ROW_NUMBER() OVER (ORDER BY [TotalDue] DESC) AS Rnk  
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
SalesOrderID      TotalDue        Rnk  
71784            119960,824    1  
71936            108597,9536   2  
71938            98138,2131    3  
71783            92663,5609    4  
71797            86222,8072    5
```

...

2. Tak otrzymany wynik możemy już wykorzystać do utworzenia

tabeli pochodnej, wybierając w zapytaniu zewnętrznym wiersze na podstawie ich numerów:

```
SELECT *
FROM (
    SELECT [SalesOrderID],[TotalDue],
    ROW_NUMBER() OVER (ORDER BY [TotalDue] DESC) AS Rnk
    FROM [SalesLT].[SalesOrderHeader]) AS W
WHERE Rnk BETWEEN 10 AND 15;
-----
SalesOrderID      TotalDue      Rnk
71782            43962,7901    10
71780            42452,6519    11
71832            39531,6085    12
71858            15275,1977    13
71897            14017,9083    14
78776            10003,00      15
```

CTE

CTE (ang. *Common Table Expressions*) to zdefiniowane w standardzie SQL3 nazwane wyrażenia tabelaryczne. CTE, tak jak tabele pochodne, są wynikami zapytań wewnętrznych, jednak:

1. Nazwę CTE definiuje się za pomocą słowa kluczowego WITH.
2. Do raz zdefiniowanego CTE można się wielokrotnie odwoływać za pomocą jego nazwy.
3. Jeżeli podzapytanie zawiera kilka CTE, w późniejszej zdefiniowanych CTE można odwołać się do tych, które zostały zdefiniowane wcześniej.
4. Zdefiniowane w tym samym podzapytaniu CTE mogą być ze sobą powiązane, tworząc rekurencyjne CTE.

Proste CTE

Pierwsze podzapytanie z CTE będzie zwracało te same dane co wcześniej analizowane podzapytanie z tabelą pochodną — liczbę

klientek z poszczególnych krajów i miast. Żeby zbudować to podzapytanie, należy:

1. Przenieść zapytanie wewnętrzne na początek podzapytania;
2. Poprzedzić je słowem kluczowym **WITH**;
3. Nazwać CTE i ewentualnie nadać aliasy zwracanym przez to CTE kolumnom;
4. Po nazwie CTE użyć słowa kluczowego **AS**;
5. W klauzuli **FROM** zapytania zewnętrznego użyć zdefiniowanej w poprzednim punkcie nazwy:

```
WITH CTE AS (
    SELECT 'Title: ' +C.[Title] AS FullTitle,
    A.CountryRegion + ' ' +A.City AS Adress, CA.AddressID
    FROM [SalesLT].[Customer] AS C
    JOIN [SalesLT].[CustomerAddress] AS CA
        ON C.CustomerID=CA.CustomerID
    JOIN [SalesLT].[Address] AS A
        ON CA.AddressID=A.AddressID
)
SELECT FullTitle,Adress, COUNT (AddressID) AS Nr
FROM CTE
WHERE FullTitle = 'Title: Ms.'
GROUP BY FullTitle,Adress;
-----
FullTitle      Adress          Nr
Title: Ms.     Canada Brossard 1
Title: Ms.     Canada Burnaby 1
Title: Ms.     Canada Calgary  3
...

```

Podobieństwo prostych CTE do tabel pochodnych zobrazujemy na jeszcze jednym przykładzie. Tym razem podzielimy zamówienia sprzedaży na pięć przedziałów (przy czym do pierwszego trafią największe zamówienia, a do ostatniego — najmniejsze) oraz policzymy największą, najmniejszą i średnią wartość zamówienia w

każdym z tych przedziałów.

Zacznijmy od podzielenia zamówień sprzedaży na pięć przedziałów. Najprościej będzie użyć do tego funkcji rankingu NTILE():

```
SELECT [SalesOrderID], [TotalDue],  
NTILE(5) OVER(ORDER BY [SalesOrderID] DESC) AS page  
FROM [SalesLT].[SalesOrderHeader];
```

SalesOrderID	TotalDue	page
71936	108597,9536	1
71935	7330,8972	1
71923	117,7276	2
71920	3293,7761	2
71917	45,1995	2
71915	2361,6403	2
71902	81834,9826	2
71899	2669,3183	2
71898	70698,9922	2
71897	14017,9083	3
71895	272,6468	3

...

Tak otrzymany wynik pogrupujmy na podstawie numerów przedziałów i dla każdej otrzymanej w ten sposób grupy wywołajmy funkcje MIN(), MAX() i AVG():

```
WITH Pages AS  
(SELECT [SalesOrderID], [TotalDue],  
NTILE(5) OVER(ORDER BY [SalesOrderID] DESC) AS page  
FROM [SalesLT].[SalesOrderHeader])  
SELECT page, MIN([TotalDue]) AS Min, MAX([TotalDue]) AS Max,  
AVG([TotalDue] ) AS Avg  
FROM Pages  
GROUP BY page  
ORDER BY page;
```

page	Min	Max	Avg
1	3,00	108597,9536	29386,4865
2	45,1995	81834,9826	23003,0909
3	272,6468	15275,1977	5097,6024
4	1261,444	86222,8072	25957,5237
5	87,0851	119960,824	51969,4239

Użycie CTE pozwala obejść ograniczenia języka SQL (takie jak niemożliwość użycia klauzuli `OVER` w klauzuli `WHERE`) w przejrzysty, generujący czytelny kod, sposób.

Tak jak podzapytania mogą być zagnieżdżane, tak CTE mogą odwoływać się do wcześniej zdefiniowanych w podzapytaniu CTE. W kolejnym podzapytaniu:

1. Pierwsze CTE odczytuje dane z tabeli `[SalesLT].[SalesOrderDetail]` i grupuje je na poziomie produktów, zliczając, ile razy dany produkt został sprzedany.
2. Po przecinku, bez powtarzania słowa kluczowego `WITH`, zostało zdefiniowane drugie CTE. Wylicza ono medianę liczby sprzedaży poszczególnych produktów.
3. Zapytanie zewnętrzne łączy wynik drugiego CTE z tabelą `[SalesLT].[Product]` oraz eliminuje z wyniku wiersze opisujące te produkty, które sprzedawały się rzadziej niż pozostałe:

```

WITH Sales ([ProductID], NumSales) AS
    (SELECT [ProductID], Count(*)
     FROM [SalesLT].[SalesOrderDetail]
     GROUP BY [ProductID]),
    FreqSales AS
        (SELECT [ProductID], NumSales,
        PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY NumSales
DESC)
        OVER () as MedianDisc
     FROM Sales)
    SELECT [Name], NumSales
    FROM FreqSales AS S
  
```

```

JOIN [SalesLT].[Product] AS P
    ON S.ProductID=P.ProductID
WHERE NumSales>MedianDisc
ORDER BY NumSales DESC;
-----
```

Name	NumSales
Classic Vest, S	10
Long-Sleeve Logo Jersey, L	10
AWC Logo Cap	9
Short-Sleeve Classic Jersey, XL	9
Hitch Rack - 4-Bike	8
Short-Sleeve Classic Jersey, L	8
Short-Sleeve Classic Jersey, S	7
Long-Sleeve Logo Jersey, M	7
Racing Socks, L	7
Sport-100 Helmet, Blue	7

...

Rekurencyjne CTE

Łącząc ze sobą wyniki dwóch CTE, uzyskamy bardzo ciekawą i przydatną funkcję — **możliwość wykonywania rekurencyjnych zapytań bez użycia tabel tymczasowych, pętli czy kurSORów.**

 Wskazówka	<p>Wiele definicji matematycznych (np. definicja ciągu arytmetycznego) jest rekurencyjnych. Charakteryzują się tym, że składają się z dwóch części (równań), z których pierwsza określa wartości początkowego elementu, czyli wartości brzegowe, a druga jest ogólnym wzorem pozwalającym wyliczyć wartość dowolnego elementu na podstawie wartości elementu (lub elementów) wcześniejszego. Natomiast w programowaniu rekurencja polega na wywoływaniu jakiejś funkcji przez nią samą tak długo, aż zajdzie warunek brzegowy.</p>
---	--

Pomimo skomplikowanej definicji rekurencja jest naturalnym sposobem rozwiązywania przez nas najróżniejszych zadań. Na przykład jeżeli mamy poukładać porozrzucane na podłodze książki, nie zabieramy się do tego, próbując jednocześnie przenieść je wszystkie. Staramy się raczej podnosić po jednej i układając na swoich miejscach (ogólna reguła postępowania), aż na podłodze nie zostanie

żadna książka (warunek brzegowy).

Rekurencyjne CTE składa się z dwóch zapytań wewnętrznych połączonych operatorem UNION ALL (zatem oba zapytania muszą zwrócić tę samą liczbę kolumn, a odpowiadające sobie kolumny muszą być tego samego typu):

1. Pierwsze zapytanie reprezentuje warunek początkowy rekurencji (ang. *Anchor member*) i jest wykonywane tylko raz.
2. Drugie zapytanie jest wykonywane dopóty, dopóki zwraca jakieś wiersze (ang. *Recursive member*). Zwrócone przez drugie zapytanie wiersze trafiają do wyniku CTE.

Zasadę działania rekurencyjnych CTE ilustruje kolejne zapytanie. Pierwsze zapytanie wykona się raz i zwróci liczbę 1, drugie będzie wykonane 99 razy i zwróci liczby od 2 do 100:

```
WITH Generator AS
    (SELECT i = 1
     UNION All
     SELECT i = i + 1
     FROM Generator
     WHERE i < 100 )
SELECT i
FROM Generator
ORDER BY i;
```

```
-----  
i  
1  
2  
3  
4  
5  
6  
7  
8
```

9
10
11
12
...

Ponieważ rekurencyjne zapytanie mogłoby wykonywać się w nieskończoność (gdyby drugie zapytanie nie przestało zwracać wierszy, czy to wskutek błędu w samym zapytaniu, czy też zapętlonej relacji pomiędzy odczytywanymi przez nie danymi), serwery bazodanowe ograniczają liczbę rekurencyjnych wywołań^[3]. Z tego powodu próba wykonania poniższego zapytania skończyła się błędem:

```
WITH Generator AS
  (SELECT i = 1
  UNION All
  SELECT i = i + 1
  FROM Generator
  WHERE i < 5000 )
  SELECT i
  FROM Generator
  ORDER BY i;
```

```
-----
The statement terminated. The maximum recursion 100 has been exhausted
before
statement completion.
```

Po wyłączeniu limitu rekurencyjnych wywołań dyrektywą MAXRECURSION zapytanie zgodnie z oczekiwaniami zwróciło ciąg 5000 liczb:

```
WITH Generator AS
  (SELECT i = 1
  UNION All
  SELECT i = i + 1
  FROM Generator
  WHERE i < 5000 )
  SELECT i
  FROM Generator
```

```
ORDER BY i  
OPTION (MAXRECURSION 0);
```

```
1  
2  
3  
4  
5  
...
```

Standardowym zastosowaniem rekurencyjnych CTE jest odczytywanie hierarchicznych danych, np. informacji o strukturze pracowników (pracownik X może być podwładnym pracownika Y i jednocześnie przełożonym pracownika Z), o budowie przedmiotów (przedmiot składa się z podzespołów, podzespół z elementów, element z części i tak dalej) lub o przynależności produktów do podkategorii, które też należą do kategorii wyższego poziomu i tak dalej.

Różnicę między prostym i rekurencyjnym CTE pokazuje kolejny przykład. Pierwsze podzapytanie zwraca liczbę kategorii bezpośrednio podlegających pod kategorię główną — do odczytania tych danych wystarcza proste CTE:

```
WITH Categories AS  
(SELECT [ParentProductCategoryID],COUNT([ProductCategoryID]) AS Nr  
     FROM [SalesLT].[ProductCategory]  
     GROUP BY [ParentProductCategoryID])  
  
SELECT *  
FROM Categories  
ORDER BY [ParentProductCategoryID];
```

ParentProductCategoryID	Nr
NULL	4
1	4
2	14
3	8
4	12

W tej tabeli znajdują się cztery kategorie najwyższego poziomu (kategorie o numerach od 1 do 4). Każda z nich zawiera pewną liczbę podkategorii, jednak suma wszystkich zwróconych przez zapytanie kategorii jest mniejsza niż liczba wierszy w tabeli [SalesLT].[ProductCategory]. Różnica ta jest spowodowana nieuwzględnieniem przechodniej natury hierarchii kategorii — podkategoria również może mieć swoje podkategorie.

Żeby odczytać kompletną strukturę kategorii produktów, musimy użyć rekurencyjnego CTE (zwróć uwagę na sposób złączenia drugiego zapytania z pierwszym — porównując ID kategorii nadrzędnej z ID kategorii podzielonej, odczytujemy ich hierarchię):

```
WITH Categories AS
    (SELECT [ParentProductCategoryID], [ProductCategoryID], 0 as lvl, P.[Name]
     FROM [SalesLT].[ProductCategory] AS P
     WHERE [ParentProductCategoryID] IS NULL
UNION ALL
    SELECT e.[ParentProductCategoryID], e.[ProductCategoryID], CTEcat.lvl+1,
    E.[Name]
     FROM [SalesLT].[ProductCategory] AS e
     INNER JOIN Categories AS CTEcat ON
     e.[ParentProductCategoryID] = CTEcat.[ProductCategoryID] )
SELECT * FROM Categories
ORDER BY [ParentProductCategoryID];
```

ParentProductCategoryID	ProductCategoryID	lvl	Name
4	39	1	Panniers
4	40	1	Pumps
4	41	1	Tires and
Tubes			
5	1042	2	Cat1
...			

Tym razem otrzymany wynik prawidłowo oddaje hierarchię kategorii.

Przeanalizujmy kolejność wykonywania rekurencyjnego CTE przez serwer bazodanowy:

1. Najpierw zostały znalezione kategorie najwyższego poziomu (kategorie, dla których nie istnieje kategoria nadzędna):

```
SELECT [ParentProductCategoryID], [ProductCategoryID], 0 as lvl,  
P.[Name]  
  
FROM [SalesLT].[ProductCategory] AS P  
  
WHERE [ParentProductCategoryID] IS NULL;
```

ParentProductCategoryID	ProductCategoryID	lvl	Name
NULL	1	0	Bikes
NULL	2	0	Components
NULL	3	0	Clothing
NULL	4	0	Accessories

2. Następnie zostały wybrane bezpośrednie podkategorie kategorii nadzędnych, a wartość trzeciej kolumny została zwiększona o 1:

```
SELECT e.[ParentProductCategoryID], e.[ProductCategoryID],  
CTEcat.lvl+1, E.[Name]  
  
FROM [SalesLT].[ProductCategory] AS e  
INNER JOIN Categories AS CTEcat ON  
e.[ParentProductCategoryID] = CTEcat.  
[ProductCategoryID];
```

3. Ponieważ drugie zapytanie zwróciło dane, zostało wywołane ponownie, tym razem jednak dla innej wartości: ParentProductCategoryID (przy drugim wywołaniu były to numery kategorii najwyższego poziomu). W rezultacie zwróciło ono numery kategorii znajdujących się na trzeciej pozycji w hierarchii.
4. Operacja ta była powtarzana dopóty, dopóki drugie zapytanie wewnętrzne zwracało dane.

Wyznaczanie trendów

Łącząc poznane do tej pory techniki, będziemy mogli rozwiązać typowy, ale stosunkowo trudny problem, jakim jest wyznaczanie

trendów. Opiera się ono na porównywaniu pewnych wartości na podstawie ich pozycji w sekwencji, np. sprzedaży w bieżącym roku ze sprzedażą w latach ubiegłych, oraz na wyszukiwaniu wartości na podstawie wartości sąsiednich pozycji w sekwencji, np. wybraniu miesięcy, w których wartość sprzedaży przekroczyła określona kwotę.

Zacznijmy od wyznaczenia malejących i rosnących trendów sprzedaży. Żeby poprawić czytelność przykładu i ułatwić jego analizę, utworzymy tabelę tymczasową i wypełnimy ją danymi:

```
CREATE TABLE #SprzedażRoczna(
    rok CHAR(4),
    sprzedaż INT);

INSERT INTO #SprzedażRoczna (rok,sprzedaż)
VALUES('2001',100),('2002',150),('2003',120),('2004',150),('2005',250),
('2006',300),('2007',250);

-----
(7 row(s) affected)
```

W pierwszej kolejności wyznaczamy trendy dla każdego roku oddzielnie. Wymaga to porównania kolejnych wierszy na podstawie ich pozycji w sekwencji, w tym przypadku sekwencji lat:

1. Jeżeli wartość sprzedaży będzie większa niż w roku ubiegłym, zapytanie powinno zwrócić 1.
2. Jeżeli wartość sprzedaży będzie mniejsza niż w roku ubiegłym, zapytanie powinno zwrócić -1.
3. Jeżeli porównanie będzie niemożliwe (co ma miejsce w przypadku pierwszego porównywanego roku), zapytanie powinno zwrócić wartość NULL.

Jednym ze sposobów rozwiązania tak wyznaczonego zadania jest porównywanie kolejno wartości sprzedaży w danym roku z wartością sprzedaży w roku poprzednim, czyli roku o największym, ale mniejszym od bieżącego numerze:

```
SELECT Z.rok, Z.sprzedaż-
    (SELECT W1.sprzedaż
     FROM #SprzedażRoczna AS W1
```

```
WHERE W1.rok =
    (SELECT MAX(W2.rok)
     FROM #SprzedażRoczna AS W2
     WHERE W2.rok < Z.rok)) AS Trend
FROM #SprzedażRoczna AS Z;
```

2001	NULL
2002	50
2003	-30
2004	30
2005	100
2006	50
2007	-50

Żeby wyznaczyć trend dla całego okresu (w tym przypadku sprzedaż rosła przez trzy kolejne lata), powinniśmy najpierw ujednolicić wartości kolumny Trend. My użyliśmy do tego celu funkcji SIGN(), która zwraca -1 dla liczb ujemnych i 1 dla liczb dodatnich:

```
SELECT Z.rok, SIGN(Z.sprzedaż-
    (SELECT W1.sprzedaż
     FROM #SprzedażRoczna AS W1
     WHERE W1.rok =
        (SELECT MAX(W2.rok)
         FROM #SprzedażRoczna AS W2
         WHERE W2.rok < Z.rok))) AS Trend
FROM #SprzedażRoczna AS Z;
```

2001	NULL
2002	1
2003	-1
2004	1
2005	1
2006	1
2007	-1

Kolejnym etapem będzie odróżnienie następujących po sobie lat, w których trend był dodatni, od sumy wszystkich lat z dodatnim trendem. Gdybyśmy tego nie zrobili i pogrupowali otrzymane wiersze według wartości kolumny Trend, otrzymalibyśmy tylko trzy grupy:

```
WITH Lata AS
  (SELECT Z.rok, SIGN(Z.sprzedaż-
    (SELECT W1.sprzedaż
      FROM #SprzedażRoczna AS W1
      WHERE W1.rok =
        (SELECT MAX(W2.rok)
          FROM #SprzedażRoczna AS W2
          WHERE W2.rok < Z.rok))) AS Trend
    FROM #SprzedażRoczna AS Z)
    SELECT MIN(rok), MAX(rok),Trend
    FROM Lata
    GROUP BY Trend;
-----
2001    2001    NULL
2003    2007    -1
2002    2006    1
```

Wyznaczenie czynnika różnicującego grupy jest najtrudniejszą częścią zadania. Jedną z możliwości jest policzenie, w ilu wcześniejszych wierszach wartość rocznej sprzedaży była większa od wartości sprzedaży dla bieżącego roku lub jej równa, ale trend był inny:

```
WITH Lata AS(
  SELECT Z.rok, SIGN(Z.sprzedaż-
    (SELECT W1.sprzedaż
      FROM #SprzedażRoczna AS W1
      WHERE W1.rok =
        (SELECT MAX(W2.rok)
          FROM #SprzedażRoczna AS W2
          WHERE W2.rok < Z.rok))) AS Trend
    FROM #SprzedażRoczna AS Z)
```

```

SELECT *, (SELECT COUNT(*)
            FROM Lata AS T2
           WHERE T2.rok >= T1.rok AND T2.trend <> T1.trend) AS
Liczbawierszy
        FROM Lata AS T1;
-----
```

2001	NULL	0
2002	1	2
2003	-1	3
2004	1	1
2005	1	1
2006	1	1
2007	-1	0

Teraz wystarczy pogrupować otrzymane wiersze, najpierw według czynnika różnicującego, a następnie według trendu, i posortować wynik według początkowych lat wyznaczonych okresów:

```

WITH Lata AS(
    SELECT Z.rok, SIGN(Z.sprzedaż-
        (SELECT W1.sprzedaż
         FROM #SprzedażRoczna AS W1
        WHERE W1.rok =
              (SELECT MAX(W2.rok)
               FROM #SprzedażRoczna AS W2
               WHERE W2.rok < Z.rok))) AS Trend
    FROM #SprzedażRoczna AS Z),
Czynnik AS (
    SELECT *, (SELECT COUNT(*)
                FROM Lata AS T2
               WHERE T2.rok >= T1.rok AND T2.trend <> T1.trend) AS
Liczbawierszy
            FROM Lata AS T1)
    SELECT MIN(rok), MAX(rok),Trend
    FROM Czynnik
    GROUP BY Liczbawierszy,Trend
```

```
ORDER BY MIN(rok);
```

2001	2001	NULL
2002	2002	1
2003	2003	-1
2004	2006	1
2007	2007	-1

Operatory

Wynik zapytania wewnętrznego można porównać za pomocą jednego z trzech specjalnych operatorów:

1. Operator **EXISTS** jest prawdziwy, jeżeli podzapytanie zwróciło jakiekolwiek dane, a fałszywy, jeżeli nie zwróciło ani jednego wiersza.
2. Operator **ANY** lub **SOME** (operatory **SOME** i **ANY** są synonimami, czyli działają dokładnie w taki sam sposób) pozwala sprawdzić wartość dowolnego wiersza wyniku podzapytania.
3. Operator **ALL** pozwala sprawdzić wartość wszystkich wierszy wyniku podzapytania.

Operator **EXISTS**

Operator **EXISTS** jest jednym z dwóch używanych w języku SQL operatorów, które zwracają prawdę lub fałsz, nigdy wartość nieznaną^[4]. Wynika to ze sposobu jego działania — zapytanie wewnętrzne może albo zwrócić przynajmniej jedną wartość (wtedy **EXISTS** będzie prawdziwy), albo nie zwrócić żadnych danych (wtedy **EXISTS** będzie fałszywy).

Operator **EXISTS** jest używany prawie wyłącznie w podzapytaniach powiązanych. Jest wtedy sprawdzany dla każdego wiersza zwróconego przez zewnętrzne zapytanie, tak jak w poniższym przykładzie zostały zwrócone dane klientów, którzy złożyli przynajmniej jedno zamówienie:

```
SELECT [LastName]  
FROM [SalesLT].[Customer] AS C
```

```
WHERE EXISTS
    (SELECT *
     FROM [SalesLT].[SalesOrderHeader] AS H
      WHERE H.CustomerID=C.CustomerID);
```

LastName

Eminhizer

Abel

Booth

Beck

Blanton

Brian

Byham

...

Ponieważ w bazie *AdventureWorksLT2012* każde zamówienie musi mieć przypisanego klienta, ten sam wynik uzyskamy za pomocą operatora IN:

```
SELECT [LastName]
  FROM [SalesLT].[Customer] AS C
 WHERE CustomerID IN
    (SELECT CustomerID
     FROM [SalesLT].[SalesOrderHeader] AS H
      WHERE H.CustomerID=C.CustomerID);
```

LastName

Eminhizer

Abel

Booth

Beck

Blanton

Brian

Byham

...



Wskazówka

Operator EXISTS zwraca wyłącznie prawdę lub fałsz, podczas gdy operator IN może również zwrócić wartość UNKNOWN. Ponieważ wartość nieznana jest w klauzuli WHERE traktowana tak samo jak fałsz, dany wiersz zostanie wyeliminowany z wyniku zapytania. Tak więc operatory EXISTS i IN nie są równoważne.

Informacja o istnieniu lub nieistnieniu wierszy spełniających podany w zapytaniu wewnętrznym warunek logiczny pozwala m.in. wyszukiwać dane nietypowe (a więc wykrywać anomalie mogące świadczyć o niespójności danych lub o podejrzanych operacjach), powtarzające się oraz brakujące.

Zacznijmy od wyszukania nietypowych danych. Pierwsze zapytanie zwraca kody produktów, które choć raz zostały sprzedane po cenie co najmniej dwa razy wyższej niż katalogowa:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product] AS P
WHERE EXISTS
    (SELECT *
     FROM [SalesLT].[SalesOrderDetail] AS I
     WHERE P.ProductID=I.ProductID
     AND P.ListPrice>2*I.UnitPrice
    );
```

Name	ListPrice
Mountain-500 Silver, 40	564,99
Mountain-500 Silver, 42	564,99
Mountain-500 Silver, 44	564,99
Mountain-500 Silver, 48	564,99
Mountain-500 Silver, 52	564,99

Wykonując podzapytanie, serwer bazodanowy:

1. Odczytał wiersz kandydujący z tabeli [SalesLT].[Product].
2. Przeszukał tabelę [SalesLT].[SalesOrderDetail], sprawdzając, czy istnieje w niej wiersz spełniający warunki:
 - a. $P.ProductID=I.ProductID$, czyli wiersz opisujący zamówienie produktu o numerze zwroconym przez

- zapytanie zewnętrzne;
- b. i jednocześnie $P.ListPrice > 2 * I.UnitPrice$, czyli wiersz, w którym ten sam produkt został sprzedany za inną cenę, co najmniej dwa razy wyższą niż jego cena katalogowa.
 3. Jeżeli zapytanie wewnętrzne zwróciło jakieś dane, to znaczy, że dany produkt miał przypisane przynajmniej dwie różne ceny. W takim przypadku operator EXISTS był prawdziwy i wiersz kandydujący został dodany do wyniku podzapytania.
 4. Następnie serwer bazodanowy odczytał kolejny wiersz z tabeli [SalesLT].[Product] i cała operacja powtarzała się aż do odczytania wszystkich wierszy tej tabeli.

W kolejnym zapytaniu raz jeszcze użyjemy operatora EXISTS do wyszukania powtarzających się danych. Tym razem zadanie polega na wybraniu podobnych, a więc możliwe, że przez pomyłkę wpisanych kilkakrotnie, danych klientów. Przyjęliśmy, iż za powtarzające się wpisy uznajemy te, w których nazwiska i tytuły klientów są takie same:

```

SELECT [CustomerID], [Title], [LastName]
FROM [SalesLT].[Customer] AS C
WHERE EXISTS
    (SELECT *
     FROM [SalesLT].[Customer] AS I
     WHERE i.LastName = C.LastName
       AND i.Title = c.Title
       AND i.CustomerID<>c.CustomerID
    )
ORDER BY c.LastName;
-----
```

CustomerID	Title	LastName
582	Ms.	Abel
29485	Ms.	Abel
579	Ms.	Abercrombie
29486	Ms.	Abercrombie
491	Ms.	Adams

...

Tak jak poprzednio zapytanie wewnętrzne zwraca jakieś dane tylko wtedy, gdy w kilku wierszach (kolumna CustomerID jest kluczem podstawowym, a więc na pewno nie zawiera duplikatów) powtórzyły się nazwiska i tytuły.

 Wskazówka	W podzapytaniach z operatorem EXISTS najczęściej stosuje się symbol wieloznaczny *. Nie ma to żadnego negatywnego wpływu na ich wydajność — serwer bazodanowy i tak sprawdza jedynie, czy zapytanie wewnętrzne zwróciło jakieś dane, a nie co to były za dane.
--	--

Operator EXISTS jest bardzo często poprzedzany operatorem NOT — brak pewnych danych z reguły jest bardziej podejrzany niż ich obecność. Na przykład żeby wyszukać klientów, którzy nigdy nie złożyli zamówienia, wystarczy do wcześniejszego zapytania dodać operator NOT:

```
SELECT [LastName]
FROM [SalesLT].[Customer] AS C
WHERE NOT EXISTS
    (SELECT *
     FROM [SalesLT].[SalesOrderHeader] AS H
     WHERE H.CustomerID=C.CustomerID);
```

LastName

Gee

Harris

Carreras

Gates

Harrington

Carroll

...

Bardziej skomplikowane jest wyszukiwanie przerw w sekwencjach, np. w numerach produktów (po lekturze poprzedniego rozdziału wiesz już, że do znalezienia przerw w sekwencjach można też użyć funkcji okienkowych). Przerwa w sekwencji oznacza, że nie istnieje

wartość o jeden większa od wartości bieżącego, zwróconego przez zapytanie zewnętrzne wiersza:

```
SELECT P.ProductID  
FROM [SalesLT].[Product] AS P  
WHERE NOT EXISTS  
    (SELECT I.ProductID  
     FROM [SalesLT].[Product] AS I  
     WHERE P.ProductID = I.ProductID+1);  
-----
```

ProductID

706

680

...

Do otrzymanych w ten sposób numerów musimy dodać 1 (tak znajdziemy brakujący numer, a nie numer, dla którego brakuje wartości o jeden większej). Powinniśmy również usunąć z wyniku zapytania największy numer, który nie świadczy o przerwie w numeracji:

```
SELECT P.ProductID+1  
FROM [SalesLT].[Product] AS P  
WHERE NOT EXISTS  
    (SELECT I1.ProductID  
     FROM [SalesLT].[Product] AS I1  
     WHERE P.ProductID = I1.ProductID+1)  
    AND P.ProductID <  
        (SELECT MAX(ProductID)  
         FROM [SalesLT].[Product] AS I2);  
-----
```

(No column name)

681

707

Operator ANY lub SOME

Operator ANY jest prawdziwy, jeżeli którakolwiek ze zwróconych

przez zapytanie wewnętrzne danych spełnia poprzedzający go warunek logiczny. Jeśli jest to warunek równościowy, operator ANY z reguły jest zastępowany operatorem IN. Przypuśćmy, że chcemy znaleźć te towary, które mają takie same nazwy jak ich modele:

```
SELECT P.[Name]
FROM [SalesLT].[Product] AS P
WHERE P.Name IN
    (SELECT M.[Name]
     FROM [SalesLT].[ProductModel] AS M
    );
```

```
Name
All-Purpose Bike Stand
Cable Lock
Chain
Fender Set - Mountain
Front Brakes
Front Derailleur
Headlights - Dual-Beam
Headlights – Weatherproof
...
```

To podzapytanie zwraca nazwy towarów, które występują też jako nazwy modeli. Ten sam wynik możemy uzyskać, sprawdzając, czy nazwa towaru jest taka sama jak którakolwiek nazwa modelu. Jednak poniższe zapytanie jest niepoprawne:

```
SELECT P.[Name]
FROM [SalesLT].[Product] AS P
WHERE P.Name =
    (SELECT M.[Name]
     FROM [SalesLT].[ProductModel] AS M
    );
```

```
Msg 512, Level 16, State 1, Line 516
Subquery returned more than 1 value. This is not permitted when the
```

subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

Błąd wynika z tego, że zapomnieliśmy zapisać w zapytaniu słowo „którankolwiek” i w rezultacie porównujemy nazwy kolejnych produktów z listą nazw modeli. Dodanie operatora ANY rozwiąże problem:

```
SELECT P.[Name]
FROM [SalesLT].[Product] AS P
WHERE P.Name = ANY
    (SELECT M.[Name]
     FROM [SalesLT].[ProductModel] AS M
    );
```

```
-----  
Name  
All-Purpose Bike Stand  
Cable Lock  
Chain  
Fender Set - Mountain  
Front Brakes  
Front Derailleur  
Headlights - Dual-Beam  
Headlights – Weatherproof
```

...

Operator ANY może być używany także z innymi operatorami porównania, nie tylko z operatorem =. Kolejne zapytanie zwraca kody tych produktów, których cena jest przynajmniej dwukrotnie wyższa od ceny jakiegokolwiek produktu z kategorii 18.:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product]
WHERE [ListPrice] >ANY
    (SELECT [ListPrice]*2
     FROM [SalesLT].[Product]
    WHERE [ProductCategoryID]=18
    );
```

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99

...

Wynik podzapytania zawiera tylko nazwy produktów, których cena była dwukrotnie wyższa od najtańszego towaru z kategorii 18., czyli od ceny zwróconej przez to zapytanie:

```
SELECT TOP 1 [ListPrice]*2
FROM [SalesLT].[Product]
WHERE [ProductCategoryID]=18
ORDER BY ListPrice;
```

(No column name)

674,44

Dlatego podzapytanie zwróciło nazwy produktów droższych niż 674,44.

Każde podzapytanie z operatorem ANY (oraz omówionym w następnym punkcie operatorem ALL) może być przekształcone na podzapytanie z operatorem EXISTS^[5]. W tym przypadku należy wybrać produkty, dla których istnieje cena katalogowa wyższa niż dziesięciokrotność ich kosztu:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product] AS P
WHERE EXISTS
    (SELECT *
     FROM [SalesLT].[Product] AS I
     WHERE I.[ProductCategoryID]=18)
```

```
        AND P.[ListPrice] > I.[ListPrice]*2  
    );
```

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99

...

Według nas pierwsza wersja podzapytania (ta z operatorem ANY) jest czytelniejsza. W dodatku operator EXISTS jest dwuwartościowy, czyli gdyby ceny lub koszty produktów mogły być nieokreślone, podzapytania nie byłyby równoważne. Jeżeli jednak wolisz używać operatorów EXISTS i IS NULL, możesz całkowicie zrezygnować z operatorów ANY i ALL — większość serwerów bazodanowych obie wersje podzapytania wykona według tego samego planu.



Znaczenie operatora ANY nie jest w pełni zgodne z regułami języka angielskiego: > ANY znaczy większy od najmniejszej wartości, a < ANY — mniejszy od największej wartości.

Poniższe zapytanie zwraca numery i wielkość zamówień wybranego klienta:

```
SELECT [SalesOrderID], [TotalDue]  
FROM [SalesLT].[SalesOrderHeader]  
WHERE [CustomerID] = 29847;
```

SalesOrderID	TotalDue
71774	972,785
77774	987,785

Jeżeli za pomocą operatora ANY odczytamy wartości opłat większych niż jakiekolwiek zamówienie tego klienta, wynik podzapytania będzie

zawierał wartości większe od najniższego zamówienia, czyli od 972,785:

```
SELECT [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
WHERE [TotalDue] > ANY
  (SELECT [TotalDue]
   FROM [SalesLT].[SalesOrderHeader]
   WHERE [CustomerID] = 29847)
ORDER BY [TotalDue];
-----
TotalDue
987,785
1170,5376
1261,444
2228,0566
...
...
```

Warto zapamiętać, że jeżeli podzapytanie nie zwróci żadnych danych, operator ANY będzie automatycznie fałszywy:

```
SELECT [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
WHERE [TotalDue] > ANY
  (SELECT [TotalDue]
   FROM [SalesLT].[SalesOrderHeader]
   WHERE [CustomerID] = 123454)
ORDER BY [TotalDue]);
-----
(0 row(s) affected)
```

Ponieważ w tabeli [SalesLT].[SalesOrderHeader] nie ma zamówień klienta o numerze 123454, podzapytanie nie zwróciło żadnych danych i w rezultacie wynik zapytania zewnętrznego również liczy zero wierszy.

Operator ALL

Operator ALL jest prawdziwy, jeżeli wszystkie zwrócone przez zapytanie wewnętrzne dane spełniają poprzedzający go warunek

logiczny. Operator ten pozwala np. odczytać nazwy towarów co najmniej dwukrotnie droższych niż wszystkie towary z 18. kategorii:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product]
WHERE [ListPrice] >ALL
    (SELECT [ListPrice]*2
     FROM [SalesLT].[Product]
     WHERE [ProductCategoryID]=18
    );
```

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99
Mountain-100 Silver, 44	3399,99
Mountain-100 Silver, 48	3399,99
Mountain-100 Black, 38	3374,99
Mountain-100 Black, 42	3374,99
Mountain-100 Black, 44	3374,99
Mountain-100 Black, 48	3374,99

Zapytanie wewnętrzne zwróciło ceny wszystkich produktów z wybranej kategorii. Najwyższą z nich była cena 2863,00:

```
SELECT MAX([ListPrice])*2
FROM [SalesLT].[Product]
WHERE [ProductCategoryID]=18;
```

2863,00

Ponieważ operator ALL był prawdziwy tylko dla produktów droższych niż 2863,00, podzapytanie zwróciło dane tylko trzynastu produktów.

Ten sam wynik możemy też oczywiście uzyskać za pomocą operatora EXISTS:

```
SELECT [Name], [ListPrice]
FROM [SalesLT].[Product] AS P
WHERE NOT EXISTS
    (SELECT *
     FROM [SalesLT].[Product] AS I
     WHERE I.[ProductCategoryID]=18
     AND P.[ListPrice] < 2*I.ListPrice);
```

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27
Mountain-100 Silver, 38	3399,99
Mountain-100 Silver, 42	3399,99
Mountain-100 Silver, 44	3399,99
Mountain-100 Silver, 48	3399,99
Mountain-100 Black, 38	3374,99
Mountain-100 Black, 42	3374,99
Mountain-100 Black, 44	3374,99
Mountain-100 Black, 48	3374,99



Równoważne podzapytania z operatorami ANY, EXISTS i ALL są w większości przypadków tak samo wykonywane przez serwery bazodanowe.

Operator ALL nie jest w praktyce używany do porównań równościowych. Warunek = ALL oznacza równy wszystkim zwróconym przez zapytanie wewnętrzne danym, czyli w rzeczywistości identyczny z jego wynikiem. Używa się go natomiast do wyszukiwania danych różnych od wszystkich danych zwróconych przez zapytanie wewnętrzne. Na przykład poniższe zapytanie

zwraca nazwy produktów o cenach różnych od cen produktów z piątej kategorii:

```
SELECT [Name]
FROM [SalesLT].[Product] AS P
WHERE P.ListPrice <> ALL
    (SELECT ListPrice
     FROM [SalesLT].[Product] AS I
     WHERE I.ProductCategoryID=5
    );
```

```
Name
HL Road Frame - Black, 58
HL Road Frame - Red, 58
Sport-100 Helmet, Red
Sport-100 Helmet, Black
Mountain Bike Socks, M
Mountain Bike Socks, L
```

...

Warto zapamiętać, że jeżeli podzapytanie nie zwróci żadnych danych, operator ALL będzie automatycznie prawdziwy, a więc zapytanie zwróci wszystkie wiersze z tabeli zewnętrznej:

```
SELECT [TotalDue]
FROM [SalesLT].[SalesOrderHeader]
WHERE [TotalDue] > ALL
    (SELECT [TotalDue]
     FROM [SalesLT].[SalesOrderHeader]
     WHERE [CustomerID] = 123454)
ORDER BY [TotalDue];
```

```
30 200,00
```

```
31 5,00
```

```
32 5,00
```

```
TotalDue
```

3,00
43,0437
45,1995
87,0851
117,7276
272,6468
608,1766

...

Ponieważ w tabeli [SalesLT].[SalesOrderHeader] nie ma zamówień klienta o numerze 123454, zapytanie zwróciło wartości wszystkich zamówień, podczas gdy to samo zapytanie z operatorem ANY zwróciło zero wierszy.

Podsumowanie

- W języku SQL podzapytania pełnią funkcję podobną do funkcji zmiennych w innych językach programowania — pozwalają w trakcie wykonywania zapytań używać zmieniających się, odczytywanych z tabel danych.
- W podzapytaniach niepowiązanych zapytanie wewnętrzne jest wykonywane tylko raz, w powiązanych — tyle razy, ile wierszy zwróciło zapytanie zewnętrzne.
- Podzapytania zwracające pojedyncze wartości mogą być używane w klauzulach SELECT, WHERE, HAVING i ORDER BY.
- Podzapytania zwracające listę wartości mogą być używane w klauzuli WHERE, ale wymagają specjalnych operatorów: IN, EXISTS, ANY lub ALL.
- Podzapytania zwracające dane tabelaryczne mogą być używane w klauzuli FROM.
- CTE to nazwane wyniki zapytań, do których można się wielokrotnie odwoływać w podzapytaniu za pomocą ich nazw.
- Rekurencyjne CTE to dwa powiązane ze sobą CTE, z których pierwsze jest wykonywane tylko raz, a drugie — dopóty, dopóki zwraca kolejne dane.
- Wyznaczenie trendów wymaga znalezienia właściwego czynnika grupującego.
- Operator EXISTS zwraca tylko prawdę lub fałsz.

Zadania

1. Odczytaj za pomocą podzapytania numery zamówień złożonych przez klienta o nazwisku Eminhizer.
2. Klauzula TOP pozwala odczytać określona liczbę wierszy, ale zaczynając zawsze od pierwszego lub ostatniego. Chociaż serwer SQL umożliwia określenie nie tylko liczby wierszy, lecz także pozycji pierwszego zaklasyfikowanego wiersza, to niektóre serwery nie mają takiej funkcjonalności. Zadanie polega na odczytaniu przy użyciu niepowiązanego podzapytania, ale bez klauzuli OVER, wartości pięciu zamówień posortowanych według ich wartości i zwróceniu zamówień znajdujących się na pozycjach od 10. do 15.
3. Odczytaj numery (kolumna SalesOrderID), daty zapłaty (kolumna DueDate) i numery klientów (kolumna CustomerID) dla zamówień złożonych ostatniego roboczego dnia każdego miesiąca.

Podpowiedź: ponieważ ostatni roboczy dzień może być inny dla poszczególnych miesięcy, użyj podzapytania niepowiązanego zwracającego ostatnią (największą) datę złożenia zamówienia w każdym miesiącu.

[1] Zadanie to można oczywiście również rozwiązać, łącząc naturalnie tabele SalesOrderHeader i Customer.

[2] Serwery bazodanowe mają różny limit zagnieżdżeń podzapytań, w przypadku serwera SQL wynosi on 32 podzapytania.

[3] Dla serwera SQL domyślny limit rekurencyjnych wywołań CTE wynosi 100.

[4] Drugim jest IS NULL, który służy do sprawdzania, czy dana wartość jest nieznana. Jeżeli tak, zwraca prawdę, w przeciwnym razie zwraca fałsz.

[5] Ale nie każde podzapytanie z operatorem EXISTS da się przekształcić na równoważne zapytanie z operatorami ANY lub ALL.

Rozdział 9. Wydajność zapytań

- W jaki sposób serwery bazodanowe wykonują zapytania?
- W jakiej kolejności wykonywane są poszczególne klauzule zapytań?
- Czym jest plan wykonania zapytania i jak go odczytać?
- Co oznacza akronim SARG?
- Jakie indeksy są przydatne do wyszukiwania wierszy?
- Co to znaczy, że indeks zawiera jakieś zapytanie?
- Które kolumny powinny być poindeksowane w celu poprawy wydajności złączeń?
- Jak poprawić wydajność zapytań grupujących lub partycjonujących wiersze?
- Co oznacza akronim POC?

Wykonywanie zapytań przez serwery bazodanowe

Chociaż szczegóły sposobów, w jakie poszczególne serwery bazodanowe wykonują zapytania, są różne, główne etapy tego procesu wyglądają podobnie. Ponieważ podstawowa wiedza na ten temat jest niezbędna do pisania wydajnych zapytań, poniżej przedstawione zostały poszczególne operacje wykonywane przez SQL Server:

1. Aplikacja kliencka łączy się z serwerem bazodanowym. Po udanym połączeniu nawiązana zostaje dwukierunkowa sesja, w ramach której odbywać się będzie komunikacja pomiędzy serwerem a klientem.
2. Klient wysyła do serwera instrukcję języka SQL.
3. Instrukcja ta zostaje odebrana, a następnie serwer bazodanowy musi opracować plan jej wykonania. Przeprowadzana w tym celu optymalizacja jest bardzo skomplikowanym i zależnym od danego serwera (a nawet od jego wersji) procesem, którego omówienie wykracza poza zakres tej książki. Powinieneś jednak wiedzieć, że optymalizacja jest bardzo czasochłonnym, silnie obciążającym

procesor i wymagającym dużej ilości pamięci procesem, a więc wiele serwerów bazodanowych przechowuje w pamięci raz zoptymalizowane plany wykonania zapytań w celu ich ponownego użycia.

4. Dysponując znalezionym planem wykonania, serwer bazodanowy może przystąpić do wykonywania instrukcji. Prawie zawsze wiąże się to z koniecznością odczytania pewnych danych.
5. Serwery bazodanowe przechowują dane w specjalnych jednostkach, w przypadku serwera SQL są to ośmiokilobajtowe strony. Strony są też jednostkami odczytu i zapisu danych, a więc serwer zapisuje oraz odczytuje jedną lub więcej stron, a nie poszczególne wiersze czy całe tabele. Do wykonania odebranej od klienta instrukcji serwer będzie więc musiał odczytać zawierające niezbędne do wykonania zapytania strony.
6. Jeżeli strony te znajdowały się już w buforze (pamięci RAM), wykonana zostanie operacja logicznego odczytu. W przeciwnym wypadku strony zostaną wczytane z dysku do bufora — taki odczyt nazywany jest odczytem fizycznym. Ponieważ pamięć jest znacznie szybsza od dysków, fizyczne odczyty są bardzo mało wydajne^[1]. W związku z tym, żeby zapewnić jak najwyższą wydajność, należy zminimalizować liczbę fizycznych odczytów poprzez wyposażenie serwera w odpowiednią (wystarczającą do zbuforowania wszystkich danych) ilość pamięci RAM.
7. Wynik wykonania instrukcji języka SQL (w przypadku zapytania będzie to zbiór wierszy) jest wysyłany do aplikacji klienckiej.

Kolejność wykonywania klauzul zapytania

Chociaż serwery bazodanowe optymalizują zapytania przed ich wykonaniem, proces ten nie może mieć wpływu na wynik zapytania (wykonanie zapytania według różnych planów, np. poprzez złączenie tabel w różnej kolejności albo poprzez pogrupowanie wierszy, a następnie złączenie otrzymanych w ten sposób grup czy też złączenie tabel, a następnie pogrupowanie wierszy, musi zawsze skutkować zwroceniem tego samego wyniku).

Poszczególne klauzule instrukcji SELECT są wykonywane zawsze w tej samej kolejności. Jeżeli któraś z opcjonalnych klauzul nie występuje, dany krok jest po prostu pomijany. Rezultatem wykonania każdego

kroku jest zbiór pośredni — wirtualna tabela, która nie jest dostępna dla użytkownika. **Kolejny krok jest wykonywany tylko w oparciu o zbiór pośredni będący rezultatem wykonania poprzedniego kroku**, a zbiór pośredni, rezultat wykonania ostatniego kroku, jest zwracany użytkownikowi.

Tak więc chociaż faktyczny sposób, w jaki serwer bazodanowy wykona nasze zapytanie, może być (i najczęściej będzie) inny niż przedstawiona poniżej ogólna kolejność wykonywania zapytań, warto znać tę logiczną kolejność wykonywania poszczególnych klauzul.

W przypadku zapytań niegrupujących danych ich klauzule wykonywane są w następującej kolejności:

1. Najpierw serwer musi pobrać potrzebne do wykonania zapytania dane, a więc wykona klauzulę **FROM** oraz, jeżeli istnieją, klauzule **JOIN** i operatory **APPLY**.
2. Następnie wybrane zostaną wiersze spełniające warunki klauzuli **WHERE**.
3. Dla otrzymanych w wyniku wykonania dwóch poprzednich punktów wierszy wykonane zostaną wyrażenia zdefiniowane w klauzuli **SELECT**.
4. Na końcu otrzymane wiersze zostaną posortowane, o ile w zapytaniu wystąpiła klauzula **ORDER BY**.

Wykonanie zapytania grupującego wymaga wykonania dodatkowych operacji:

1. Punkty pierwszy i drugi są wykonywane w taki sam sposób jak dla zapytań niegrupujących.
2. Przed wykonaniem klauzuli **SELECT** wiersze kandydujące są grupowane (wykonywana jest klauzula **GROUP BY**).
3. Otrzymane w wyniku grupowania wiersze są filtrowane, o ile w zapytaniu wystąpiła klauzula **HAVING**.
4. Dla otrzymanych w wyniku wykonania poprzednich punktów wierszy wykonane zostaną wyrażenia zdefiniowane w klauzuli **SELECT**.
5. Na końcu otrzymane wiersze zostaną posortowane, o ile w zapytaniu wystąpiła klauzula **ORDER BY**.

Plany wykonania zapytań

Umiejętność czytania i analizowania planów wykonania zapytań jest niezbędna każdemu, kto chce pisać nie tylko poprawne (zwracające właściwe wyniki), ale również wydajne (zwracające te wyniki w najszybszy możliwy sposób) zapytania. Wynika to z faktu, że to samo zapytanie może być wykonane na bardzo dużo różnych sposobów (liczba możliwych sposobów wykonania tego samego zapytania zależy od samego zapytania i od serwera bazodanowego, ale nawet nieskomplikowane zapytania odczytujące dane z kliku tabel mogą być wykonane na kilkadziesiąt, a nawet kilkaset różnych sposobów). Zadaniem serwera bazodanowego jest znalezienie wystarczająco dobrego sposobu (planu) wykonania zapytania. Żeby mu to umożliwić, powinniśmy nie tylko unikać typowych błędów (takich jak używanie argumentów uniemożliwiających efektywne skorzystanie z indeksów), ale również pisać zapytania w sposób ułatwiający znalezienie optymalnych planów ich wykonania. Najprostszym sposobem sprawdzenia, czy nasze zapytanie wykonywane jest wydajnie, jest właśnie analiza planu jego wykonania.

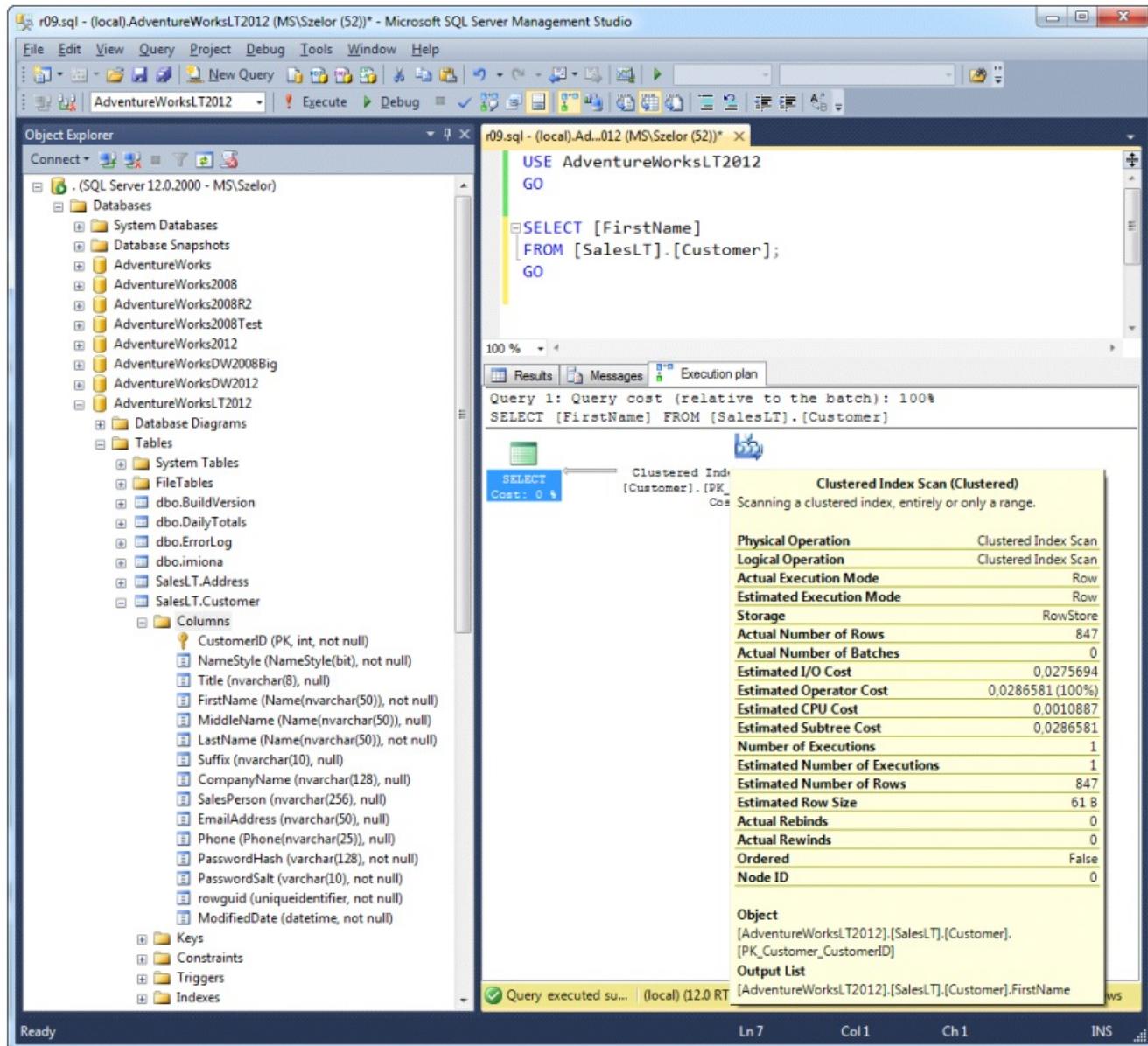
SQL Server pozwala odczytywać plany wykonania zapytań w formie tekstowej oraz graficznej. Dodatkowo możliwe jest poznanie szacowanych oraz faktycznych planów wykonania. Ponieważ szczegółowe przedstawienie kwestii analizy planów wykonania zapytań wykracza poza zakres tej książki, ograniczymy się do przedstawiania faktycznych planów wykonania zapytań w formie graficznej.

Zaczniemy od wyjaśnienia tego, czym jest plan wykonania zapytania. Plan wykonania reprezentuje opracowaną przez serwer bazodanowy strategię odczytania i przetworzenia danych na potrzeby wykonania zapytania. Składa się ona z serii iteratorów, z których każdy wykonuje pojedynczą operację, taką jak odczytanie danych, posortowanie wierszy, złączenie tabel itd.

Plany wykonania zapytań można czytać w dwóch kierunkach:

1. Od lewa do prawa — ten kierunek reprezentuje logikę wykonywania zapytania;
2. Lub od prawa do lewa — ta kolejność repetyuje przepływ danych pomiędzy kolejnymi iteratorami.

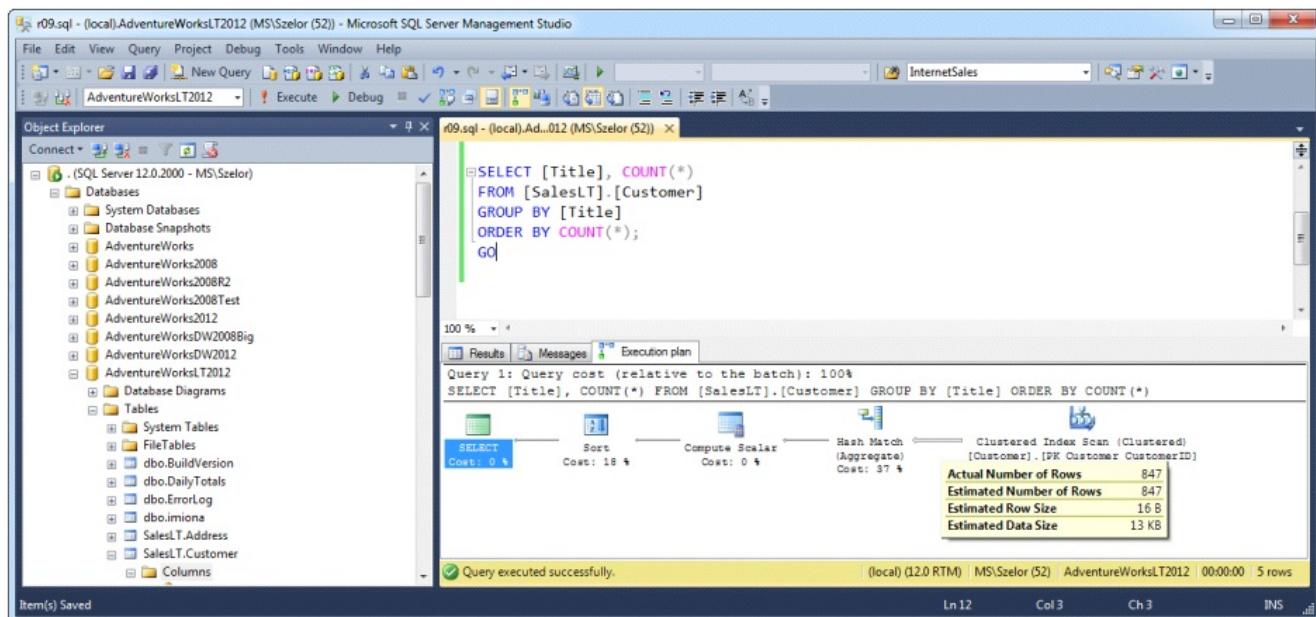
Żeby wyświetlić graficzny plan wykonania zapytania, należy kliknąć znajdujący się na pasku zadań konsoli SSMS przycisk *Include Actual Execution Plan* lub nacisnąć kombinację klawiszy *Ctrl+M*, a następnie uruchomić analizowane zapytanie (rysunek 9.1).



Rysunek 9.1. Najprostszy plan wykonania zapytania — potrzebne dane zostały odczytane za pomocą iteratora Clustered Index Scan (szczegóły operacji możemy poznać, umieszczając kursor myszki nad danym iteratorem), a następnie zwrócone do aplikacji klienckiej

Przyjrzyjmy się nieco bardziej rozbudowanemu planowi wykonania zapytania. Przedstawiony na rysunku 9.2. plan wykonania należy przeczytać następująco:

1. Przeskanowany został indeks zgrupowany (tabela [SalesLT].[Customer]).
2. Odczytanych w ten sposób 847 wierszy (umieszczając kursor myszki nad reprezentującymi przepływ danych strzałkami, wyświetlimy dodatkowe informacje na temat liczby i rozmiaru wierszy) zostało pogrupowanych.
3. Dla każdej grupy wyliczona została funkcja COUNT.
4. Wiersze zostały posortowane.
5. Na końcu wiersze zostały zwrócone do aplikacji klienckiej.



Rysunek 9.2. Składający się z pięciu iteratorów plan wykonania zapytania grupującego dane

SQL Server ocenia i podaje koszt wykonania każdej operacji — np. w poprzednim przykładzie koszt pogrupowania wierszy wyniósł 37% kosztów wykonania całego zapytania. Ta wiedza pozwala nam na wdrożenie względnie prostej, a jednocześnie skutecznej metody optymalizacji zapytań, polegającej na eliminowaniu z planów wykonania najkosztowniejszych operacji.

Co więcej, jeżeli jednocześnie uruchomimy kilka zapytań (np. kilka wersji tego samego zapytania), serwer SQL zwróci informację o procentowym rozkładzie kosztów wykonania całego wsadu (zbiór jednocześnie wysłanych do serwera zapytań nazywa się wsadem). Rysunek 9.3 ilustruje tę funkcjonalność.

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the database 'AdventureWorksLT2012' is selected. The 'Tables' node is expanded, showing various tables like SalesLT.Customer, SalesLT.SalesOrderDetail, etc. The 'Script' tab of the 'i09.sql' query window contains two SELECT statements:

```

SELECT [Title], COUNT(*)
FROM [SalesLT].[Customer]
GROUP BY [Title]
ORDER BY COUNT(*) DESC;
GO

SELECT [SalesOrderID], [LineTotal]
FROM [SalesLT].[SalesOrderDetail];

```

The 'Execution plan' tab shows the cost of each query. Query 1 has a cost of 21% and uses a clustered index scan followed by a compute scalar. Query 2 has a cost of 79% and includes a sort operation before the clustered index scan.

The status bar at the bottom indicates: (local) (12.0 RTM) MS\szel (52) AdventureWorksLT2012 00:00:00 | 1084 rows.

Rysunek 9.3. Drugie zapytanie (to z klauzulą ORDER BY) okazało się cztery razy bardziej kosztowne od pierwszego. Różnica wynika z tego, że wykonując drugie zapytanie, serwer musiał dodać kosztowny iterator SORT

Przekonaliśmy się właśnie, że **sortowanie może wielokrotnie wydłużyć czas wykonania zapytania**, a więc jeżeli wynik zapytania nie musi być posortowany, nie należy umieszczać w nim klauzuli ORDER BY.

Porównajmy jeszcze koszty wykonania zapytania, w którym użyta do sortowania kolumna nie została dołączona do wyniku z zapytaniem zawierającym tę kolumnę (rysunek 9.4) — okaże się, że są one identyczne, a więc **niewymienienie w klauzuli SELECT kolumn użytych do sortowania nie skraca czasu wykonania zapytania**.

```

r09.sql - (local).AdventureWorksLT2012 (MS\Szczepan (S2)) - Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Help
AdventureWorksLT2012 Execute Debug
Object Explorer
Connect
Tables
System Tables
FileTables
dbo.BuildVersion
dbo.DailyTotals
dbo.ErrorLog
dbo.Imprints
SalesLT.Address
SalesLT.Customer
SalesLT.CustomerAddress
SalesLT.Product
SalesLT.ProductCategory
SalesLT.ProductDescription
SalesLT.ProductModel
SalesLT.ProductModelProductDescription
SalesLT.SalesOrderDetail
Columns
SalesOrderID (PK, FK, int, not null)
SalesOrderDetailID (PK, int, not null)
OrderQty (smallint, not null)
ProductID (FK, int, not null)
UnitPrice (money, not null)
UnitPriceDiscount (money, not null)
LineTotal (Computed, numeric(18,2), not null)
rowguid (uniqueidentifier, not null)
ModifiedDate (datetime, not null)
Keys
Constraints
Triggers
Indexes
Results Messages Execution plan
Query 1: Query cost (relative to the batch): 50%
SELECT [SalesOrderID] FROM [SalesLT].[SalesOrderDetail] ORDER BY [LineTotal] DESC
SELECT Cost: 0 % Compute Scalar Cost: 0 % Sort Cost: 73 % Compute Scalar Cost: 0 % Clustered Index Scan (Clustered) ISalesOrderDetail.IPK_SalesOrderDe...
Cost: 26 %
Query 2: Query cost (relative to the batch): 50%
SELECT [SalesOrderID], [LineTotal] FROM [SalesLT].[SalesOrderDetail] ORDER BY [LineTotal] DESC
SELECT Cost: 0 % Compute Scalar Cost: 0 % Sort Cost: 73 % Compute Scalar Cost: 0 % Clustered Index Scan (Clustered) ISalesOrderDetail.IPK_SalesOrderDe...
Cost: 26 %
Query executed successfully.

```

Rysunek 9.4. Chociaż analiza planów wykonania zapytań przez konkretny serwer bazodanowy nie jest tematem tej książki, to warto wiedzieć, że kolumny użyte do sortowania muszą być odczytane tak samo jak kolumny wymienione w klauzuli SELECT. Z tego powodu większość serwerów bazodanowych wykona oba powyższe zapytania w tym samym czasie, a koszt wykonania każdego z nich przez serwer SQL wyniósł dokładnie 50% kosztu wykonania całego wsadu

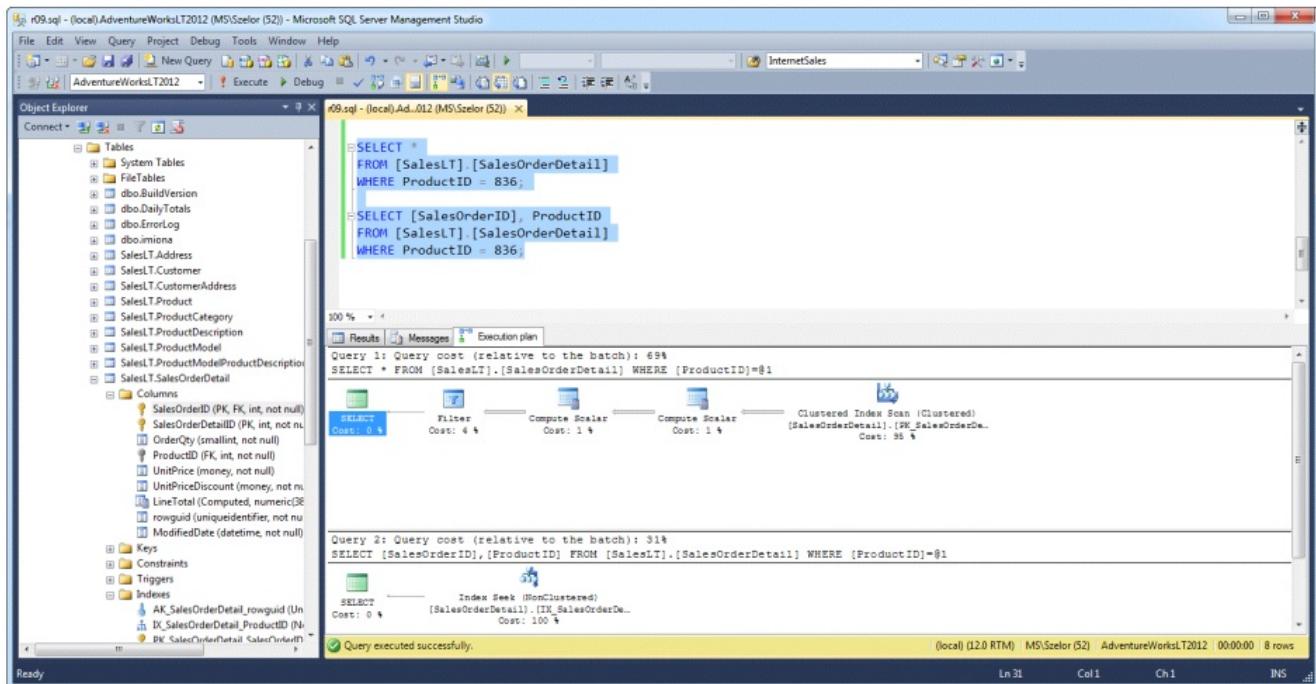
Wydajne wyszukiwanie danych za pomocą argumentów SARG

Dane mogą być zapisane:

1. W tabelach — dla uproszczenia i zgodnie z relacyjnym modelem baz danych przyjmijmy, że kolejność wierszy tabeli jest nieistotna, a więc nie są one w żaden sposób posortowane^[2].
2. W indeksach — aby uniknąć analizowania różnic pomiędzy poszczególnymi serwerami bazodanowymi, przyjmijmy, że indeks przypomina skorowidz książki — dołączaną na jej końcu, alfabetycznie uporządkowaną listę haseł z odnośnikami do numerów stron, na których dane hasło jest opisane. W przypadku baz danych odpowiednikiem hasła będzie wybrana kolumna (lub kolumny) tabeli, czyli tzw. klucz indeksu, a odnośnikiem wskaźnik^[3]. Podsumowując, **każdy klucz indeksu posiada wskaźnik do wiersza tabeli zawierającego wartości**

pozostałych (niekluczowych) kolumn, a klucze indeksu są zawsze posortowane.

Jeżeli odczytywana tabela nie jest poindeksowana, serwer bazodanowy wykonując dowolne odwołujące się do tej tabeli zapytanie, będzie musiał odczytać ją w całości. Tego typu sytuacja zachodzi też, gdy odczytujemy wszystkie kolumny tabeli za pomocą symbolu *. Żeby się o tym przekonać, wystarczy porównać plany wykonania dwóch pokazanych na rysunku 9.5 zapytań:



Rysunek 9.5. Wyszukanie w indeksie kluczy spełniających warunek z klauzuli WHERE i odczytanie tylko czterech wierszy tabeli okazało się w tym przypadku trzy razy szybsze niż odczytanie całej tabeli i wybranie czterech wierszy spełniających podany warunek

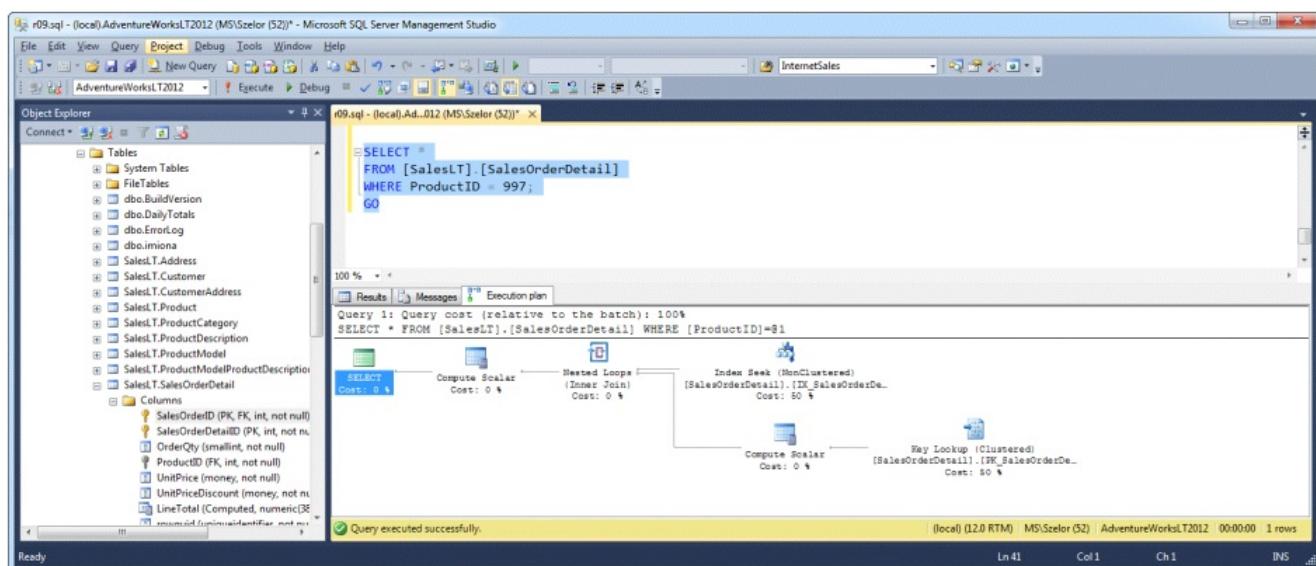
Podsumujmy — tabela [SalesLT].[SalesOrderDetail] ma założony indeks na kolumnie ProductID. Indeks ten zawiera, oprócz identyfikatorów produktów, identyfikatory zamówień. Nie zawiera natomiast pozostałych kolumn tej tabeli. Z tego powodu wykonanie pierwszego, zwracającego tylko cztery wiersze, zapytania wymagało odczytania całej tabeli, podczas gdy do znalezienia odpowiednich danych w indeksie wystarczyło go przeszukać. O takich zapytaniach jak te drugie mówimy, że zawierają się one w indeksie, ponieważ wszystkie potrzebne do ich wykonania dane zapisane są w założonym dla tabeli indeksie.



Wskazówka

Utworzenie zawierającego zapytanie indeksu jest najprostszym i najskuteczniejszym sposobem na poprawę wydajności tego zapytania. Nie oznacza to jednak, że powinniśmy tworzyć indeksy zawierające wszystkie zapytania. Ponieważ serwer bazodanowy automatycznie synchronizuje dane w indeksach z danymi w tabeli, liczba indeksów zdefiniowanych dla pojedynczej tabeli nie powinna przekraczać 10.

Jeżeli istnieje indeks zawierający zapytanie, serwer bazodanowy zawsze go użyje. Zupełnie inaczej wygląda użycie przez serwery bazodanowe indeksów podczas wykonywania zapytań, które się w nich nie zawierają (rysunek 9.6).



Rysunek 9.6. Plan wykonania zapytania poprzez przeszukanie indeksu i pobranie brakujących w tym indeksie danych z tabeli

Na planie wykonania drugiego zapytania widoczny jest operator Key Lookup symbolizujący odczytywanie danych z tabeli [SalesLT].[SalesOrderDetail]. Operacja ta musiała być przeprowadzona, bo indeks zawierał tylko identyfikatory produktów i zamówień, a zapytanie miało zwrócić wszystkie dane wybranego zamówienia. Takie sięganie po dane jest na tyle kosztowne, że **jeżeli zapytanie zwraca więcej niż kilka procent wierszy tabeli, serwery bazodanowe przestają używać indeksów i odczytują całą tabelę, tak jakby użyta do wyszukiwania kolumna nie była zindeksowana**.

Wiedząc, w jaki sposób serwery bazodanowe odczytują dane, możemy teraz uzasadnić uwagę z jednego z poprzedniego rozdziałów, w której napisaliśmy,



Wskazówka

że używając symbolu *, zmuszamy serwer bazodanowy do odczytania wszystkich kolumn tabeli, co może wielokrotnie wydłużyć czas wykonywania zapytania. Ponieważ większość zapytań zwraca więcej niż 1% wierszy tabeli, posługując się symbolem *, powodujemy, że serwery bazodanowe nie korzystają z istniejących indeksów.

Skoro koszt (a więc i czas) wykonania zapytań z użyciem indeksów jest wielokrotnie niższy niż koszt odczytania tych samych danych z tabel, **powinniśmy tak pisać zapytania, żeby serwery bazodanowe korzystały z indeksów przy ich wykonywaniu.**



Wskazówka

Skrót SARG (ang. *Search ARGuments*) oznacza warunki wyszukiwania, czyli takie warunki logiczne, które pozwalają serwerom bazodanowym na wybranie właściwych wierszy poprzez przeszukanie indeksu, a nie odczytanie całej tabeli.

Ponieważ użycie indeksów może być bardzo kosztowne^[4], serwery bazodanowe, zanim się na nie zdecydują, szacują koszt wykonania zapytania, używając do tego danych statystycznych. Jeżeli nie będą w stanie oszacować tego kosztu, wybiorą najbezpieczniejsze rozwiązanie, czyli odczytają tabelę.

W poniższych punktach zostały opisane przypadki, w których serwer bazodanowy może nie mieć możliwości oszacowania kosztu wykonania zapytania, a więc nie skorzysta z indeksów i czas wykonania zapytań będzie wielokrotnie dłuższy:

1. Zanegowanie warunku logicznego (użycie operatora NOT albo operatora <>), o ile serwer bazodanowy nie potrafi automatycznie przekształcić warunku logicznego w równoważny warunek pozytywny. Przykładem takiego przekształcenia może być zastąpienie przez serwer SQL warunku NOT CustomerID >3 warunkiem CustomerID <= 3:

```
SELECT *
FROM [SalesLT].[SalesOrderDetail]
WHERE ProductID <> 999;
```

2. Umieszczenie nazwy kolumny w ramach dowolnego wyrażenia, nawet jeżeli to wyrażenie nie zmienia wyniku testu logicznego. Z tego powodu, choć oba pokazane na rysunku 9.7 zapytania zwracają te same dane, to pierwsze zostanie wykonane

czterokrotnie szybciej.

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the AdventureWorksLT2012 database is selected. In the center pane, there are two queries in the r09.sql script:

```
SELECT [SalesOrderID], ProductID
FROM [SalesLT].[SalesOrderDetail]
WHERE ProductID = 997;

SELECT [SalesOrderID], ProductID
FROM [SalesLT].[SalesOrderDetail]
WHERE ProductID=0 = 997;
GO
```

The execution plan for the first query shows an Index Seek (NonClustered) on the IX_SalesOrderDetail index, with a cost of 42%. The execution plan for the second query shows an Index Scan (NonClustered) on the same index, with a cost of 58%. Both queries return 2 rows. The status bar at the bottom indicates "Query executed successfully." and shows the session details: (local) (12.0 RTM) | MS\Scelor (52) | AdventureWorksLT2012 | 00:00:00 | 2 rows.

Rysunek 9.7. Użycie nazwy kolumny jako części dowolnego wyrażenia oznacza, że serwer bazodanowy będzie musiał odczytać cały indeks (albo nawet całą tabelę), żeby mieć pewność, że zwrócił wszystkie wiersze spełniające tego typu warunek

3. Użycie nazwy kolumny jako argumentu dowolnej funkcji (rysunek 9.8).

```

Object Explorer
File Edit View Query Project Debug Tools Window Help
AdventureWorksLT2012 InternetSales
r09.sql - (local)\Ad...012 (MS\Szczep (52)) - Microsoft SQL Server Management Studio
SELECT *
FROM [SalesLT].[Customer]
WHERE [EmailAddress] = 'keith@adventure-works.com';

SELECT *
FROM [SalesLT].[Customer]
WHERE LOWER([EmailAddress]) = 'keith0@adventure-works.com';
GO

```

Results pane:

```

Query 1: Query cost (relative to the batch): 30%
SELECT * FROM [SalesLT].[Customer] WHERE [EmailAddress]=@1

```

```

Nested Loops (Inner Join)
  Index Seek (NonClustered) [Customer].[IX_Customer_EmailAddress..]
    Cost: 0 %
  Key Lockup (Clustered) [Customer].[PK_CustomerCustomerID]
    Cost: 40 %

```

Query 2: Query cost (relative to the batch): 70%
SELECT * FROM [SalesLT].[Customer] WHERE LOWER([EmailAddress]) = 'keith0@adventure-works.com'

```

Nested Loops (Inner Join)
  Index Scan (NonClustered) [Customer].[IX_Customer_EmailAddress..]
    Cost: 0 %
  Key Lockup (Clustered) [Customer].[PK_CustomerCustomerID]
    Cost: 60 %

```

Query executed successfully.

Rysunek 9.8. Umieszczenie nazwy kolumny jako argumentu dowolnej (również systemowej) funkcji ma dokładnie ten sam wpływ na użycie indeksów co użycie nazwy kolumny w wyrażeniu

- Użycie do wyboru wierszy wzorca zaczynającego się od symbolu wieloznacznego, np. Name LIKE N'%b'.

 Wskazówka	Przykładowa baza danych jest bardzo mała i powyższe zapytania odwołują się do tabel liczących zaledwie kilkaset wierszy każda. Tylko dlatego różnice w wydajności poprawnie napisanych (umożliwiających przeszukiwanie indeksu) i nieoptymalnych zapytań są tak małe. Tymczasem swoją prawdziwą siłę indeksy pokazują w przypadku dużych, liczących tysiące czy miliony wierszy, tabel. Wtedy znalezienie wybranych wierszy w indeksie nadal wymaga odczytania takiej samej ilości danych, podczas gdy odczytanie całej tabeli czy indeksu może zająć nawet kilka minut.
---	--

Poprawa wydajności złączeń

Łączenie tabel, szczególnie tych dużych (liczących wiele wierszy i kolumn), jest kosztowną operacją — wybranie pasujących do warunku złączenia (warunku z klauzuli ON) wierszy w najgorszym wypadku jest operacją o złożoności obliczeniowej rzędu $O(n^2)$, gdzie n jest liczbą wierszy. Oznacza to, że jeżeli koszt złączenia 100 wierszy wyniósł 10 000, to koszt złączenia 200 wierszy wyniesie już 40 000, a więc cztery razy więcej niż w przypadku dwukrotnie

mniejszych tabel.

Zmniejszyć ten koszt możemy, zakładając indeksy na używanych dołączenia tabel kolumnach — jeżeli obie użyte do złączenia kolumny są poindeksowane, złączenie tabel będzie operacją o złożoności obliczeniowej rzędu $O(n)$. Ponieważ prawie zawsze tabele są łączone na podstawie pary kolumn klucz podstawowy – klucz obcy, to na tych kolumnach należy założyć indeksy.

Większość serwerów bazodanowych, w tym SQL Server, automatycznie indeksuje kolumny klucza podstawowego. Oznacza to, że aby poprawić wydajność złączeń, wystarczy założyć indeksy na kolumnach kluczowych wszystkich tabel bazy danych.

Wydajne grupowanie i partycjonowanie danych

Grupowanie, sortowanie i wyszukiwanie danych może być bardzo kosztowne. Najprostszym i najskuteczniejszym sposobem na poprawę wydajności grupowania danych, tak samo jak sortowania czy wyszukiwania na podstawie argumentów SARG, jest użycie indeksów.

W przypadku grupowania zindeksowana powinna być każda lub co najmniej pierwsza kolumna wymieniona w klauzuli GROUP BY. Dzięki temu serwer bazodanowy będzie mógł odczytać odpowiednio posortowane dane, co znacznie uprości i przyspieszy ich pogrupowanie (grupowanie posortowanych danych wymaga wielokrotnie mniej czasu procesora i pamięci niż grupowanie nieposortowanych danych)^[5]. Ponadto klucze indeksu są z reguły znacznie mniejsze niż wiersze tabeli, a więc zmniejszy się liczba odczytywanych z dysku danych.

Żeby się o tym przekonać, wykonamy dwa zapytania zwracające dokładnie tak samo pogrupowane i posortowane dane, za drugim razem zabraniając serwerowi bazodanowemu skorzystania z indeksu^[6] (rysunek 9.9).

```

x09.sql - (local).AdventureWorksLT2012 (MS\Sczelor (S2)) - Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Help
AdventureWorksLT2012 Execute Debug InternetSales
Object Explorer
SELECT [ProductID], COUNT(*)
FROM [SalesLT].[SalesOrderDetail]
GROUP BY [ProductID];

SELECT [ProductID], COUNT(*)
FROM [SalesLT].[SalesOrderDetail] WITH (INDEX(0))
GROUP BY [ProductID];

```

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 16%

```

SELECT [ProductID], COUNT(*) FROM [SalesLT].[SalesOrderDetail] GROUP BY [ProductID]

```

SELECT Compute Scalar Cost: 0 % Stream Aggregate Cost: 8 % Index Scan (NonClustered) Cost: 92 %

Query 2: Query cost (relative to the batch): 84%

```

SELECT [ProductID], COUNT(*) FROM [SalesLT].[SalesOrderDetail] WITH (INDEX(0)) GROUP BY [ProductID]

```

SELECT Compute Scalar Cost: 0 % Stream Aggregate Cost: 2 % Sort Cost: 72 % Clustered Index Scan (Clustered) Cost: 26 %

Query executed successfully.

Rysunek 9.9. Pogrupowanie danych nawet małej, liczącej mniej niż tysiąc wierszy, tabeli bez indeksu okazało się pięć razy bardziej kosztowne niż użycie w tym celu odpowiedniego indeksu

Zapytania partycjonujące dane (a więc zapytania, w których występuje klauzula `OVER`) mają większe wymagania co do indeksów. W ich przypadku najlepszymi indeksami są indeksy POC (ang. *Partitioning, Ordering, Covering*), czyli takie, w których pierwsze kolumny są kolumnami użytymi do partycjonowania danych (w klauzuli `PARTITION BY`), po nich występują kolumny użyte do ustalenia porządku wierszy (w klauzuli `ORDER BY`), a indeks zawiera też wszystkie pozostałe (wymienione w klauzuli `SELECT`) kolumny.

Podsumowanie

- Kolejność, w jakiej wpisuje się poszczególne klauzule zapytania, nie odpowiada kolejności, w której są one wykonywane przez serwery bazodanowe.
- Serwery bazodanowe optymalizują sposoby wykonania zapytań.
- Sposobem na sprawdzenie, czemu dane zapytanie działa wolniej, niż się spodziewaliśmy, jest sprawdzenie planu jego wykonania.
- Indeksy są najprostszym i wyjątkowo skutecznym sposobem na poprawę wydajności zapytań.

- Samo istnienie indeksu nie wystarczy, żeby serwer bazodanowy mógł z niego efektywnie skorzystać — zapytanie musi być poprawnie napisane, np. do wybierania wierszy powinno się używać argumentów typu SARG, a zapytanie nie powinno odczytywać więcej kolumn, niż jest to wymagane.
- Uniwersalna strategia indeksowania polega na tworzeniu indeksów zawierających zapytania, a więc indeksów wielokolumnowych. Dla zapytań grupujących lub partycjonujących dane należy tworzyć indeksy typu POC.

Zadania

1. Poniższe zapytanie zwracające nazwy produktów i kategorii działa zbyt wolno. Jak można poprawić jego wydajność?

```
SELECT [Name]
FROM [SalesLT].[Product]
UNION
SELECT [Name]
FROM [SalesLT].[ProductCategory];
```

2. Wykonanie poniższego zapytania wymaga przeskanowania (odczytania w całości) indeksu założonego na kolumnie **UnitPrice**. Przepisz to zapytanie tak, żeby używało argumentu typu SARG.

```
SELECT [SalesOrderID]
FROM [SalesLT].[SalesOrderDetail]
WHERE [UnitPrice]*.77 > 900;
```

3. Zajdź optymalny indeks dla poniższego zapytania:

```
SELECT [DueDate], [SalesOrderID], [TotalDue],
       LAG([TotalDue]) OVER (PARTITION BY [DueDate] ORDER BY
[DueDate]) as      PreviusTotalDue
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];
```

[1] Automatyczne buforowanie danych jest powodem, dla którego ponowne wykonanie tego samego zapytania może być znacznie szybsze.

[2] Wiele serwerów bazodanowych porządkuje wiersze tabeli według wartości klucza podstawowego. Taka posortowana tabela nazywana jest indeksem zgrupowanym.

[3] Więcej informacji na temat indeksów znajduje się w rozdziale 13.

[4] Z powodu opisanej wcześniej operacji *Key Lookup*, polegającej na odczytywaniu dla każdego klucza indeksu odpowiadającego mu wiersza tabeli.

[5] Niektóre serwery bazodanowe potrafią grupować tylko posortowane dane. Jeżeli brakuje im przydatnego indeksu, przed grupowaniem dodatkowo sortują dane.

[6] Serwery bazodanowe umożliwiają określenie sposobu, w jaki zapytanie ma być wykonane — służą do tego specyficzne dla danego serwera dyrektywy optymalizatora. Dyrektywa `WITH (INDEX(0))` zabrania serwerowi SQL użycia jakiegokolwiek indeksu.

Część III

Modyfikowanie danych, czyli instrukcje INSERT, UPDATE, DELETE oraz MERGE

W porównaniu z rozbudowaną składnią instrukcji SELECT składnia instrukcji modyfikujących dane jest dość prosta^[1]. Jednak sposób ich wykonania może być skomplikowany — serwery bazodanowe, dbając o spójność jednocześnie modyfikowanych przez wielu użytkowników danych, wszystkie zmiany wykonują w ramach transakcji.

Z dwóch następnych rozdziałów książki dowiesz się:

1. W jaki sposób wstawiać i usuwać wiersze;
 2. Jak szybko usunąć całą zawartość tabeli;
 3. W jaki sposób modyfikować zapisane wcześniej w tabelach dane;
 4. Czym są transakcje i na czym polega transakcyjne przetwarzanie danych;
 5. Co oznacza termin „współbieżność” i czym się różni optymistyczny model współbieżności od modelu pesymistycznego.
-

[1] Modyfikacją jest każda zmiana zapisanych w tabelach danych — zarówno wstawianie i usuwanie wierszy, jak i aktualizowanie pól tabeli.

Rozdział 10. Modyfikowanie danych

- Jak wstawić do tabeli nowe wiersze?
- Jak wstawić do tabeli wynik zapytania?
- Jak utworzyć tabelę na podstawie wyniku zapytania?
- Jak usunąć wybrane wiersze?
- Kiedy można wyczyścić tabelę za pomocą instrukcji TRUNCATE TABLE?
- Jak zaktualizować wartości wybranych pól?
- Jak jednocześnie wstawić, usunąć i zaktualizować dane za pomocą instrukcji MERGE?

Wstawianie danych

Wstawić nowe wiersze do tabeli możemy, wykonując instrukcję `INSERT INTO ... VALUES`. W najprostszym przypadku składnia instrukcji wygląda następująco:

```
INSERT INTO [dbo].[ErrorLog]
VALUES (GETDATE(), 'MARCIN');
```

```
-----  
Msg 213, Level 16, State 1, Line 4  
Column name or number of supplied values does not match table definition.
```

Jednak serwer bazodanowy zgłosił błąd, informując nas, że liczba wstawianych wartości (w tym przypadku jeden ciąg znaków) nie zgadza się z liczbą kolumn docelowej tabeli `[dbo].[ErrorLog]`.

Wstawić można tylko cały wiersz, próba wstawienia „niepełnego” wiersza skończy się błędem. W tym przykładzie tabela `[dbo].[ErrorLog]` składa się z dziewięciu kolumn, między innymi: `ErrorTime`, `UserName`, `ErrorNumber` i `ErrorState`, a my próbowaliśmy wstawić do niej wiersz zawierający tylko jedną wartość.

Jeżeli jawnie nie podamy nazw docelowych kolumn, serwery bazodanowe spróbują wstawić podane w klauzuli `VALUES` dane do kolejnych kolumn tabeli, zaczynając od pierwszej. Żeby uniknąć błędów wynikających z takiego sposobu wstawiania podanych

wartości do kolejnych kolumn tabeli, wystarczy wskazać ich nazwy:

```
INSERT INTO FOO ([NAME])  
VALUES ('Marcin');
```

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

Teraz instrukcja jest jednoznaczna i serwer bazodanowy wstawił ciąg znaków Marcin do kolumny Name.

 Wskazówka	Pomijanie nazw kolumn w instrukcji INSERT może prowadzić do błędów i jest złą praktyką, tak samo jak nadużywanie symbolu * w klauzuli SELECT.
---	---

Skoro instrukcja INSERT wstawia wiersz, a nie pojedyncze pola, z reguły wywołuje się ją, przekazując po nazwie tabeli rozdzieloną przecinkami listę nazw kolumn i wymieniając w klauzuli VALUES wstawiane do tych kolumn wartości:

```
INSERT INTO FOO ([AGE], [NAME])  
VALUES(5, 'Bob');
```

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

```
SELECT *  
FROM FOO;
```

```
-----  
NAME          AGE  
Marcin       NULL  
Bob           5
```

Klucze podstawowe

Jeżeli wstawimy wiersz z samą nazwą kategorii do tabeli [SalesLT].[ProductCategory], a następnie go odczytamy, przekonamy się, że dwa pola (a nie tylko jedno wskazane w instrukcji INSERT) zawierają jakieś dane:

```
INSERT INTO [SalesLT].[ProductCategory] ([Name])  
VALUES ('New Category');
```

```
(1 row(s) affected)

SELECT [ProductCategoryID], [ParentProductCategoryID], [Name]
FROM [SalesLT].[ProductCategory]
WHERE [Name] = 'New Category';
```

```
-----
```

ProductCategoryID	ParentProductCategoryID	Name
1045	NULL	New Category

Ta dodatkowa wartość (numer kategorii) została wstawiona przez zdefiniowane dla tej tabeli ograniczenie^[2]. Rola ograniczeń jest dbanie o spójność zapisywanych w tabelach danych — w tym przypadku wstawiony automatycznie numer jest jednoznacznie identyfikującym wiersz kluczem podstawowym.

Większość serwerów bazodanowych automatycznie generuje i wstawia wartości sztucznych kluczy podstawowych. Co więcej, próba samodzielnego wstawienia danych do kolumny klucza podstawowego może wymagać ustawienia specjalnej opcji^[3]:

```
INSERT INTO [SalesLT].[ProductCategory] ([ProductCategoryID], [Name])
VALUES (1234, 'Newest Category');
```

```
-----
```

Msg 544, Level 16, State 1, Line 29

Cannot insert explicit value for identity column in table
'ProductCategory' when IDENTITY_INSERT is set to OFF.



Wskazówka W dobrze zaprojektowanej bazie danych sztuczne klucze podstawowe powinny być wstawiane nie przez użytkowników, tylko przez serwer bazodanowy.

Wartości domyślne

Jeżeli nie podamy wartości domyślnej jakiejś kolumny (wartości zdefiniowanej w ograniczeniu DEFAULT) w instrukcji INSERT, zostanie ona automatycznie wstawiona przez serwer bazodanowy. Na przykład dla dwóch ostatnich kolumn tabeli [SalesLT].[ProductCategory] zdefiniowane są ograniczenia DEFAULT — wstawiają one:

1. do kolumny rowguid losowy identyfikator wiersza;
2. do kolumny ModifiedDate bieżącą datę.

Czyli jeżeli wstawimy do tej tabeli wiersz bez podawania wartości tych kolumn:

```
INSERT INTO [SalesLT].[ProductCategory] ([Name])
VALUES ('Newest Category');
-----
(1 row(s) affected)
```

to identyfikator wiersza i data jego zmodyfikowania zostaną wstawione automatycznie:

```
SELECT *
FROM [SalesLT].[ProductCategory]
WHERE [Name] = 'Newest Category';
-----
ProductCategoryID    ParentProductCategoryID    Name    rowguid
ModifiedDate
1046      NULL      Newest Category      6F006043-6054-4326-8231-BA6358309915
2014-08-16 09:56:53.130
```

Możemy jednak też samodzielnie wstawić datę modyfikacji wiersza:

```
INSERT INTO [SalesLT].[ProductCategory] ([Name], [ModifiedDate])
VALUES ('Spices', '2014-10-10');
-----
(1 row(s) affected)
```

W takim przypadku zostanie wstawiona jawnie podana wartość, a nie zdefiniowana dla kolumny wartość domyślna:

```
SELECT [Name], [ModifiedDate]
FROM [SalesLT].[ProductCategory]
WHERE [Name] = 'Spices';
-----
Name          ModifiedDate
Spices        2014-10-10 00:00:00.000
```

Żeby poprawić czytelność instrukcji `INSERT`, w ramach których wstawiamy wartości domyślne:

1. W klauzuli `INSERT` należy wymienić nazwy wszystkich kolumn, z wyjątkiem nazwy kolumny klucza podstawowego.

2. W klauzuli VALUES należy użyć słowa kluczowego DEFAULT oznaczającego, że do danej kolumny w rzeczywistości nie chcemy wstawić żadnej wartości:

```
INSERT INTO [SalesLT].[ProductCategory] ([Name],[rowguid],  
[ModifiedDate])  
VALUES ('Herbs', DEFAULT,DEFAULT);  
-----  
(1 row(s) affected)
```

W rezultacie do kolumn rowguid i ModifiedDate zostaną wstawione zdefiniowane dla nich wartości domyślne:

```
SELECT [Name],[rowguid], [ModifiedDate]  
FROM [SalesLT].[ProductCategory]  
WHERE [Name] = 'Herbs';  
-----  


| Name  | rowguid                              | ModifiedDate               |
|-------|--------------------------------------|----------------------------|
| Herbs | 938EE5EE-C83F-45DA-B900-F70514CC7D08 | 2014-08-16<br>10:05:22.013 |


```

Wartość NULL

Z poświęconego tworzeniu tabel rozdziału 12. dowiesz się, że dla każdej kolumny tabeli można określić, czy możliwe będzie zapisywanie w niej wartości NULL. Jeżeli tak, to wstawić ją możemy na dwa sposoby:

1. Pomijając nazwę tej kolumny w klauzuli INSERT — ponieważ pogarsza to czytelność instrukcji, odradzamy stosowanie tej techniki:

```
INSERT INTO [dbo].[ErrorLog] ([UserName],[ErrorNumber],  
[ErrorMessage])  
VALUES ('Marcin',50000,'Test message');  
-----  
(1 row(s) affected)
```

Ile wartości NULL zostało wstawionych w rezultacie wykonania tej instrukcji? Niestety, nie można się tego dowiedzieć, analizując samą instrukcję, czyli pozostaje nam odczytanie wstawionego wiersza:

```

SELECT *
FROM [dbo].[ErrorLog]
WHERE [ErrorNumber] = 50000;

-----
ErrorLogID    ErrorTime      UserName     ErrorNumber   ErrorSeverity
ErrorState    ErrorProcedure  ErrorLine     ErrorMessage
1   2014-08-16 10:08:58.260   Marcin     50000       NULL        NULL        NULL
NULL         Test message

```

2. Wymieniając nazwy wszystkich (z wyjątkiem kolumny klucza podstawowego) kolumn w klauzuli INSERT i jawnie wstawiając NULL do odpowiednich kolumn:

```

INSERT INTO [dbo].[ErrorLog]
([ErrorTime],[UserName],[ErrorNumber], [ErrorSeverity],
[ErrorState],[ErrorProcedure],[ErrorLine],[ErrorMessage])
VALUES (DEFAULT,'Marcin',50001,NULL,NULL,NULL,NULL,'Test message');

-----
(1 row(s) affected)

```

Jeżeli natomiast spróbujemy wstawić wartość NULL do kolumny, w której zapisywanie wartości nieznanej jest zabronione^[4], operacja zostanie przerwana i wiersz nie zostanie wstawiony:

```

INSERT INTO [SalesLT].[ProductCategory] ([Name])
VALUES (NULL);

-----
Msg 515, Level 16, State 2, Line 75
Cannot insert the value NULL into column 'Name', table
'AdventureWorksLT2012.SalesLT.ProductCategory'; column does not allow
nulls. INSERT fails.

The statement has been terminated.

```

Konstruktor wierszy

Wstawianie wierszy jeden po drugim jest uciążliwe i mało wydajne. Dlatego niektóre serwery bazodanowe umożliwiają wymienienie w klauzuli VALUES danych, które zostaną jednocześnie wstawione do wielu wierszy:

```
INSERT INTO FOO ([NAME], [AGE])
```

```
VALUES ('Bob', 5), ('Mary', 6);
```

```
-----  
(2 row(s) affected)
```

Wstawianie wyników zapytań

Drugim sposobem na jednorazowe wstawienie do tabeli wielu wierszy jest odczytanie wstawianych danych z innych tabel. Można to zrobić na dwa sposoby:

1. Tworząc tabelę na podstawie wstawianych do niej danych;
2. Wstawiając wynik zapytania do wcześniej utworzonej tabeli.

Instrukcja SELECT ... INTO

Instrukcja `SELECT ... INTO` w rzeczywistości jest zapytaniem. Jednak klauzula `INTO` powoduje, że **wynik zapytania nie będzie zwrócony użytkownikowi, tylko zostanie użyty do utworzenia i wypełnienia danymi nowej tabeli**.

Rezultatem wykonania poniższej instrukcji będzie utworzenie tymczasowej tabeli i wypełnienie jej danymi produktów o cenach przekraczających 100:

```
SELECT *  
INTO #PROD  
FROM [SalesLT].[Product]  
WHERE ListPrice>100;
```

```
-----  
(206 row(s) affected)
```

Instrukcja `SELECT ... INTO` pozwala też modyfikować wstawiane do nowej tabeli dane — możemy np. skopiować tylko nazwy, numery i ceny produktów, zwiększając jednocześnie ceny o 100%:

```
SELECT [ProductID], [Name], [ListPrice]*2 AS DoubledPrice  
INTO #OverPriced  
FROM [SalesLT].[Product];
```

```
-----  
(296 row(s) affected)
```

Możemy również użyć instrukcji `SELECT ... INTO` do utworzenia kopii struktury wybranej tabeli — żeby to zrobić, wystarczy w klauzuli

WHERE wyeliminować wszystkie wiersze. Kolumny tak utworzonej tabeli będą miały właściwe nazwy i typy danych, skopiowane zostaną także informacje o tym, czy dopuszczalne jest w nich zapisywanie wartości NULL. Natomiast pozostałe ograniczenia oraz zdefiniowane dla tabeli źródłowej indeksy i wyzwalacze nie zostaną skopiowane:

```
SELECT *
INTO EmptyTable
FROM [SalesLT].[Product]
WHERE 1=0;;
-----
(0 row(s) affected)
```

Instrukcja INSERT INTO ... SELECT

Instrukcja `INSERT INTO ... SELECT` pozwala wstawić do istniejącej tabeli wynik zapytania. Jako że w poniższym przykładzie wiersze są wstawiane do wcześniej utworzonej tabeli, muszą być zgodne z jej definicją, czyli obowiązują nas te same ograniczenia co w przypadku instrukcji `INSERT INTO ... VALUES`.

Tak więc niemożliwe jest wstawienie danych do kolumny klucza podstawowego:

```
INSERT INTO EmptyTable
SELECT *
FROM [SalesLT].[Product];
-----
Msg 8101, Level 16, State 1, Line 99
An explicit value for the identity column in table 'EmptyTable' can only
be specified when a column list is used and IDENTITY_INSERT is ON..
```

Ponieważ jednak w kolumnie `Name` oryginalnej tabeli nie można było zapisywać wartości NULL, a ograniczenie NOT NULL zostało skopiowane instrukcją `INSERT INTO ... SELECT`, poniższa instrukcja nie zostanie wykonana:

```
INSERT INTO EmptyTable ([Name])
VALUES (NULL);
-----
Msg 515, Level 16, State 2, Line 104
```

```
Cannot insert the value NULL into column 'Name', table  
'AdventureWorksLT2012.dbo.EmptyTable'; column does not allow nulls.  
INSERT fails.
```

The statement has been terminated.

Prawidłowym sposobem wstawienia danych do tej tabeli jest wymienienie nazw docelowych kolumn:

```
INSERT INTO EmptyTable  
([Name]  
, [ProductNumber]  
, [Color]  
, [StandardCost]  
, [ListPrice]  
, [Size]  
, [Weight]  
, [ProductCategoryID]  
, [ProductModelID]  
, [SellStartDate]  
, [SellEndDate]  
, [DiscontinuedDate]  
, [ThumbNailPhoto]  
, [ThumbnailPhotoFileName]  
, [rowguid]  
, [ModifiedDate])  
  
SELECT [Name]  
    , [ProductNumber]  
    , [Color]  
    , [StandardCost]  
    , [ListPrice]  
    , [Size]  
    , [Weight]  
    , [ProductCategoryID]  
    , [ProductModelID]  
    , [SellStartDate]  
    , [SellEndDate]
```

```
, [DiscontinuedDate]
, [ThumbNailPhoto]
, [ThumbnailPhotoFileName]
, [rowguid]
, [ModifiedDate]

FROM [SalesLT].[Product];

-----
(296 row(s) affected)
```

W instrukcji INSERT INTO ... SELECT można używać podzapytań. Pozwala to rozwiązać, bez używania niedostępnej w niektórych serwerach bazodanowych instrukcji MERGE, problem z dopisywaniem do tabeli tylko tych danych, które jeszcze się w niej nie znajdują.

Na przykład gdybyśmy chcieli dodać do tabeli #PROD informacje o wszystkich produktach, ale bez powtarzania znajdujących się w tej tabeli danych (w tym momencie tabela ta liczy 206 wierszy opisujących produkty o cenach powyżej 100), moglibyśmy najpierw sprawdzić, czy zapisany jest w niej produkt o danym numerze:

```
INSERT INTO #PROD
([Name]
, [ProductNumber]
, [Color]
, [StandardCost]
, [ListPrice]
, [Size]
, [Weight]
, [ProductCategoryID]
, [ProductModelID]
, [SellStartDate]
, [rowguid]
, [ModifiedDate])

SELECT [Name]
, [ProductNumber]
, [Color]
, [StandardCost]
```

```

,[ListPrice]
,[Size]
,[Weight]
,[ProductCategoryID]
,[ProductModelID]
,[SellStartDate]
,[rowguid]
,[ModifiedDate]

FROM [SalesLT].[Product] AS P
WHERE NOT EXISTS
(SELECT *
FROM #PROD AS N
WHERE P.ProductNumber = N.ProductNumber);

-----
(90 row(s) affected)

```

Teraz obie tabele, źródłowa ([SalesLT].[Product]) i docelowa (#PROD), zostały zsynchronizowane i obie mają po 296 wierszy.

Usuwanie danych

Do usuwania wierszy służy instrukcja `DELETE`, do czyszczenia całych tabel — instrukcja `TRUNCATE TABLE`[\[5\]](#).

 Wskazówka	Usuwać można tylko całe wiersze. Żeby skasować wartość wybranych pól (np. kolor wybranego produktu), należy je zaktualizować, ustalając nową wartość na <code>NULL</code> .
---	---

Instrukcja `DELETE`

Składnia instrukcji `DELETE` jest bardzo prosta: w klauzuli `FROM` należy podać nazwę tabeli, a w opcjonalnej klauzuli `WHERE` warunek, na podstawie którego będą usuwane wiersze.

Żeby usunąć wszystkie wiersze tabeli `Sosy`, wystarczy wykonać poniższą instrukcję:

```
DELETE FROM [dbo].[EmptyTable];
```

```
(296 row(s) affected)
```

W praktyce bardzo rzadko usuwa się wszystkie wiersze tabeli. Wiersze, które chcemy usunąć, możemy wskazać za pomocą klauzuli **WHERE — usunięte zostaną tylko te wiersze, dla których umieszczony w niej warunek będzie prawdziwy.**

Wykonanie poniższej instrukcji spowoduje skasowanie z tabeli #PROD danych o produktach, dla których wartość SellEndDate była nieznana:

```
DELETE FROM #PROD  
WHERE [SellEndDate] IS NULL;
```

```
(223 row(s) affected)
```

Podczas usuwania danych, tak samo jak podczas ich wstawiania, serwery bazodanowe sprawdzają nałożone na tabele ograniczenia. Gdyby wykonanie instrukcji DELETE miało spowodować utratę spójności danych, serwer zgłosi błąd i nie wykona takiej instrukcji. Sytuacja ta ma miejsce podczas usuwania danych z powiązanych tabel.

Na przykład identyfikatory produktów są zapisane w kolumnach klucza obcego tabeli [SalesLT].[SalesOrderDetail]. Po skasowaniu wiersza, którego identyfikator jest zapisany w którejkolwiek z powiązanych tabel, dane byłyby niespójne. Dlatego próba wykonania kolejnej instrukcji skończyła się niepowodzeniem:

```
DELETE FROM [SalesLT].[Product];
```

```
Msg 547, Level 16, State 0, Line 191
```

```
The DELETE statement conflicted with the REFERENCE constraint  
"FK_SalesOrderDetail_Product_ProductID". The conflict occurred in  
database "AdventureWorksLT2012", table "SalesLT.SalesOrderDetail", column  
'ProductID'.
```

```
The statement has been terminated.
```

Wiemy już, że instrukcja DELETE prawie zawsze zawiera klauzulę WHERE. Niektóre serwery bazodanowe, w tym SQL Server, pozwalają wybrać usuwane wiersze na podstawie danych zapisanych w innej tabeli niż ta, z której chcemy je usunąć bez używania podzapytania. Na przykład żeby usunąć z tabeli [SalesLT].[Product] dane produktów

nieprzypisanych do jakiejkolwiek kategorii, należy wykonać poniższą instrukcję:

```
DELETE P  
FROM [SalesLT].[Product] AS P  
LEFT OUTER JOIN [SalesLT].[ProductCategory] AS C  
ON P.ProductCategoryID = C.ProductCategoryID  
WHERE P.ProductCategoryID IS NULL;
```

(1 row(s) affected)

 Wskazówka	Dane z powiązanych tabel należy usuwać w odpowiedniej kolejności — zaczynając od tabeli, której klucz podstawowy nie występuje w żadnej innej tabeli, a kończąc na tabelach słownikowych [6] .
---	--

Usuwanie wyników podzapytań

Załóżmy, że tabela zawiera duplikaty danych — np. że ktoś ponownie wykonał poniższą instrukcję:

```
INSERT INTO #PROD  
([Name]  
, [ProductNumber]  
, [Color]  
, [StandardCost]  
, [ListPrice]  
, [Size]  
, [Weight]  
, [ProductCategoryID]  
, [ProductModelID]  
, [SellStartDate]  
, [rowguid]  
, [ModifiedDate])
```

```
SELECT [Name]  
, [ProductNumber]  
, [Color]  
, [StandardCost]
```

```
, [ListPrice]
,[Size]
,[Weight]
,[ProductCategoryID]
,[ProductModelID]
,[SellStartDate]
,[rowguid]
,[ModifiedDate]

FROM [SalesLT].[Product] AS P;
```

```
-----  
(295 row(s) affected)
```

Niektóre (295 lub mniej — wcześniej usunęliśmy z tabeli #PROD produkty o nieznanej dacie wycofania ze sprzedaży) wiersze tej tabeli zawierają teraz powtórzone dane, a naszym zadaniem jest naprawienie błędu i usunięcie duplikatów.

Zacznijmy od ich znalezienia. Jeżeli za powtórzone wiersze uznamy te, w których powtarza się numer produktu, to wydajnym sposobem ich znalezienia będzie ponumerowanie wierszy przy użyciu funkcji rankingu. Musimy tylko wywołać ją dla partycji zbudowanych dla numerów nazw produktów (ponumerowanie wierszy wymaga użycia klauzuli ORDER BY, ale w tym przypadku porządek sortowania nie ma znaczenia — nieważne, którą kopię wiersza usuniemy), więc „posortujemy” wiersze według stałej:

```
WITH Duplikaty AS
(SELECT *, ROW_NUMBER() OVER(PARTITION BY [ProductName] ORDER BY
(SELECT 1)) AS Nr
FROM #PROD)
SELECT [ProductName] FROM Duplikaty
WHERE Nr>1;
```

```
-----  
ProductName
```

```
BK-M47B-38
```

```
BK-M47B-40
```

```
BK-M47B-44
```

```
...
```

My jednak nie chcemy odczytywać tych wierszy, tylko chcemy je usunąć. Usunąć z wyniku zapytania wiersze z numerami większymi od 1 (a więc powtórzone wiersze) możemy za pomocą instrukcji `DELETE` — dzięki temu, że może się ona odwoływać do wyniku podzapytania, wystarczy zastąpić zewnętrzną instrukcję `SELECT` instrukcją `DELETE`:

```
WITH Duplikaty AS
  (SELECT *, ROW_NUMBER() OVER(PARTITION BY [ProductNumber] ORDER BY
    (SELECT 1)) AS Nr
   FROM #PROD)
  DELETE FROM Duplikaty
 WHERE Nr>1;
-----
(73 row(s) affected)
```

Instrukcja TRUNCATE TABLE

Najszybciej możemy wyczyścić tabelę, wykonując instrukcję `TRUNCATE TABLE`. Ponieważ jest to operacja przeprowadzana na bardzo niskim poziomie (na poziomie bloków danych, a nie wierszy tabeli), niemożliwe jest wskazanie wierszy, które mają być skasowane. Z tego samego powodu **niemożliwe jest obcięcie tabeli powiązanej z innymi tabelami, nawet jeżeli są one puste**, a więc na pewno nie zawierają odnośników do usuwanych kluczy podstawowych.

Wykonując poniższą instrukcję, obetniemy tabelę `#PROD` — zwróć uwagę, że tym razem komunikat potwierdzenia nie zawiera informacji o liczbie usuniętych wierszy:

```
TRUNCATE TABLE #PROD;
-----
Command(s) completed successfully.
```

Aktualizowanie danych

Zapisane w tabeli dane można zaktualizować, wykonując instrukcję `UPDATE`. W najprostszym przypadku jej składnia jest następująca:

1. W klauzuli `UPDATE` należy podać nazwę tabeli^[7].
2. W klauzuli `SET` należy podać nazwę modyfikowanej kolumny i

przypisać jej nową wartość:

```
UPDATE Foo  
SET [AGE]=18;  
-----  
(4 row(s) affected)
```

Wykonanie powyższej instrukcji spowodowało zmianę wszystkich wierszy docelowej tabeli. Ponieważ z reguły chcemy wybrać, które wiersze mają być zaktualizowane, instrukcja UPDATE w większości przypadków zawiera klauzulę WHERE:

```
UPDATE [SalesLT].[Customer]  
SET [LastName] = 'Nowak'  
WHERE [CustomerID]=1;  
-----  
(1 row(s) affected)
```

Tym razem zaktualizowane zostało nazwisko tylko tych klientów, dla których warunek WHERE był prawdziwy.



Instrukcja UPDATE, tak jak pozostałe instrukcje języka SQL, działa na zbiorach, a nie pojedynczych rekordach. Jeżeli chcemy zaktualizować tylko jedną wartość, w klauzuli WHERE musimy umieścić warunek logiczny, który będzie prawdziwy tylko dla jednego wiersza.

Jednoczesne aktualizowanie wielu kolumn

Instrukcja UPDATE pozwala jednocześnie zmodyfikować dane z wielu kolumn tabeli. **Jeżeli chcemy zmienić kilka wartości, powinniśmy zrobić to w ramach jednej instrukcji UPDATE**, ponieważ:

1. Koszt aktualizacji wielu kolumn jest mniej więcej taki sam jak koszt aktualizacji pojedynczej kolumny — jest jednak znacznie niższy, gdy serwer bazodanowy wykona jedną operację dla aktualizacji wielu kolumn, a nie kilka operacji dla aktualizacji pojedynczych kolumn^[8].
2. Jednoczesne modyfikowanie wszystkich kolumn ułatwia zachowanie spójności danych — w takim przypadku serwer bazodanowy automatycznie dopilnuje, żeby inne osoby nie miały

dostępu do aktualizowanych danych.

Przypuśćmy, że pewna osoba przeprowadziła się. Aktualizując dane adresowe, musimy zmienić nazwę miasta i województwa — obu zmian możemy dokonać jednocześnie:

```
UPDATE Foo  
SET [AGE]=18, [NAME] = 'Tom'  
WHERE [NAME] like 'Bo%';  
-----  
(1 row(s) affected)
```

Wyrażenia

W klauzuli SET można używać wyrażeń — wtedy nowe wartości kolumn będą wyliczane w trakcie wykonywania instrukcji UPDATE. Nie można jednak odwoływać się w tych wyrażeniach do danych z innych tabel, chyba że zostaną one wymienione w dodatkowej klauzuli FROM. Można natomiast aktualizować dane na podstawie ich wcześniejszych wartości:

```
UPDATE [SalesLT].[Product]  
SET [ListPrice] = [ListPrice]-[ListPrice]*0.1  
WHERE [SellEndDate] IS NOT NULL;  
-----  
(98 row(s) affected)
```

Po wykonaniu tej instrukcji cena będących w sprzedaży produktów została obniżona o 10%. W niektórych serwerach bazodanowych ten sam efekt można uzyskać za pomocą specjalnych operatorów $+=$, $-=$, $*=$ i $/=$, np. żeby przecenić produkty o połowę:

```
UPDATE [SalesLT].[Product]  
SET [ListPrice] /= 2  
WHERE [SellEndDate] IS NOT NULL;  
-----  
(98 row(s) affected)
```

Ważną regułą, według której działają serwery bazodanowe, jest reguła jednoczesnego wykonania całej instrukcji języka SQL. Najprościej jest ją wyjaśnić na przykładzie. Zaczniemy od

skopiowania do nowej tabeli nazwisk i imion klientów:

```
SELECT [FirstName], [LastName]
INTO Tab
FROM [SalesLT].[Customer];
-----
(847 row(s) affected)
```

W kolumnie FirstName znajdują się imiona, w kolumnie LastName — nazwiska:

```
SELECT *
FROM Tab;
-----
FirstName          LastName
Orlando            Nowak
Keith              Harris
Donna              Carreras
Janet              Gates
Lucy               Harrington
Rosmarie          Carroll
...
...
```

Reguła jednoczesnego wykonania instrukcji języka SQL pozwala nam zamienić imiona z nazwiskami bez używania jakiejkolwiek zmiennej tymczasowej, co byłoby wymagane w innych językach programowania:

```
UPDATE Tab
SET [FirstName] = [LastName],
    [LastName] = [FirstName];
-----
```

```
(847 row(s) affected)
```

```
SELECT *
FROM Tab;
-----
```

```
FirstName          LastName
Nowak             Orlando
```

Harris	Keith
Carreras	Donna
Gates	Janet
Harrington	Lucy
Carroll	Rosmarie

...

Aktualizowanie danych wybranych na podstawie danych z innych tabel

Niektóre serwery bazodanowe pozwalają na umieszczenie w instrukcji UPDATE klauzuli FROM^[9]. Umożliwia to odwoływanie się w klauzuli WHERE do kolumn z powiązanych, wymienionych w tej klauzuli tabel, np. zmianę kosztu produktów wybranych na podstawie ilości, w jakiej były one zamawiane:

```
UPDATE P
SET [StandardCost] *=0.85
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS D
    ON P.ProductID=D.ProductID
WHERE D.OrderQty>10;
```

(15 row(s) affected)

Aktualizowanie danych za pomocą wyrażeń odwołujących się do innych tabel

Niestandardowa klauzula FROM pozwala również zmienić wartości kolumny jednej tabeli na wartości odczytane z innej tabeli. Możemy np. ustawić datę modyfikacji wierszy w tabeli [SalesLT].[Product] na podstawie daty modyfikacji powiązanych z nimi wierszy w tabeli [SalesLT].[SalesOrderDetail]:

```
UPDATE P
SET P.[ModifiedDate] = D.[ModifiedDate]
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS D
    ON P.ProductID=D.ProductID
```

```
WHERE D.[SalesOrderID] BETWEEN 71500 AND 72000;
```

```
-----  
(142 row(s) affected)
```

Instrukcja MERGE

MERGE pozwala jednocześnie wykonać operacje wstawiania, usuwania lub aktualizacji różnych wierszy, czyli łączy instrukcje INSERT, DELETE i UPDATE. Pozwala to szybko zsynchronizować dane zapisane w różnych tabelach lub wykonać operację warunkowego wstawiania wierszy nieistniejących w tabeli docelowej i aktualizacji danych, które już są w tej tabeli.

Analizę instrukcji MERGE zaczniemy od wstawienia za jej pomocą do utworzonej tabeli tymczasowej^[10] dwóch wierszy:

```
CREATE TABLE #Osoby (ID int identity(1,1), Imię varchar(100));
```

```
-----  
Command(s) completed successfully.
```

```
MERGE INTO #Osoby  
USING (Values ('Danuta',1),('Marcin',2)) Znajomi(Imię,Numer)  
ON #Osoby.ID = Znajomi.Numer  
WHEN NOT MATCHED THEN  
INSERT (imię) values (Znajomi.Imię);
```

```
-----  
(2 row(s) affected)
```

```
SELECT * FROM #Osoby;
```

```
-----  
1      Danuta  
2      Marcin
```

Instrukcja MERGE wymaga podania dwóch tabel: docelowej, której zawartość zostanie zmodyfikowana (#Osoby), oraz źródłowej, z której dane będą odczytane (w tym przypadku zamiast podawać nazwę tabeli, za pomocą klauzuli VALUES podaliśmy bezpośrednio wartości dwóch wierszy i nazwaliśmy tak utworzoną wirtualną tabelę Znajomi).

Ponieważ tabela #Osoby była pusta, warunek złączenia (#Osoby.ID =

`Znajomi.Numer)` będzie zawsze fałszywy. Dlatego zostanie wykonana instrukcja z bloku `WHEN NOT MATCHED THEN`, a więc instrukcja `INSERT`. Jeżeli raz jeszcze wykonamy tę samą instrukcję, oba zbiory danych będą zawierały te same wiersze, a więc warunek z klauzuli `WHEN NOT MATCHED` nie zostanie spełniony i żadne wiersze nie zostaną wstawione do tabeli docelowej:

```
MERGE INTO #Osoby
USING (Values ('Danuta',1),('Marcin',2)) Znajomi(Imię,Numer)
ON #Osoby.ID = Znajomi.Numer
WHEN NOT MATCHED THEN
INSERT (imię) values (Znajomi.Imię);
-----
(0 row(s) affected)
```

Rozbudujmy instrukcję `MERGE` o blok `WHEN MATCHED THEN`. Teraz w tabeli `#Osoby` są już zapisane dwa imiona, a my chcemy zsynchronizować je z nowymi informacjami, przy czym imiona mają być zaktualizowane, jeżeli osoba o tym samym numerze była już zapisana w tabeli, i wstawione, jeżeli tabela docelowa nie zawierała informacji o osobach z takimi numerami:

```
MERGE INTO #Osoby
USING (Values ('Krzyś',3),('Michał',2)) Znajomi(Imię,Numer)
ON #Osoby.ID = Znajomi.Numer
WHEN MATCHED THEN
UPDATE SET #Osoby.Imię = Znajomi.Imię
WHEN NOT MATCHED THEN
INSERT (imię) values (Znajomi.Imię);
-----
(2 row(s) affected)
SELECT * FROM #Osoby;
-----
1      Danuta
2      Michał
3      Krzyś
```

W trakcie wykonywania tej instrukcji wiersze z obu tabel (docelowej

#0soby i źródłowej Znajomi) były porównywane na podstawie ich numerów:

1. Dla wierszy spełniających warunek złączenia (w tym przypadku dla wiersza o numerze 2) został wykonany blok WHEN MATCHED THEN, a więc instrukcja UPDATE.
2. Dla pozostałych wierszy (w tym przypadku dla wiersza o numerze 3) został wykonany blok WHEN NOT MATCHED THEN, czyli instrukcja INSERT.

W rezultacie docelowa tabela liczy teraz trzy wiersze, a imię Marcin zostało zaktualizowane do imienia Michał.

Dodajmy do naszej instrukcji ostatni blok WHEN NOT MATCHED BY SOURCE THEN. Pozwala on wskazać instrukcję, która zostanie wykonana dla wierszy istniejących w tabeli źródłowej, ale nie w docelowej:

```
MERGE INTO #0soby
USING (Values ('Danuta',1),('Marcin',2)) Znajomi(Imię,Numer)
ON #0soby.ID = Znajomi.Numer
WHEN MATCHED THEN
UPDATE SET #0soby.Imię = Znajomi.Imię
WHEN NOT MATCHED THEN
INSERT (Imię) values (Znajomi.Imię)
WHEN NOT MATCHED BY SOURCE THEN
DELETE;
-----
(3 row(s) affected)
SELECT * FROM #0soby;
-----
1      Danuta
2      Marcin
```

Tym razem z tabeli docelowej został usunięty jeden, nieznajdujący się w tabeli źródłowej wiersz (wiersz z numerem 3), a wiersz z numerem 2 został ponownie zaktualizowany.

Podsumowanie

- Do wstawiania nowych wierszy służy instrukcja **INSERT**. Wstawiane dane mogą być jawnie podane w klauzuli **VALUES** albo odczytane z innych tabel.
- Do usuwania wierszy służy instrukcja **DELETE**. Wiersze, które mają być skasowane, wskazuje się w klauzuli **WHERE**, a jej pominięcie powoduje usunięcie wszystkich wierszy tabeli.
- Obciąć (szybko wyczyścić) tabelę za pomocą instrukcji **TRUNCATE TABLE** można tylko wtedy, gdy nie jest ona powiązana z innymi tabelami.
- Do aktualizowania pól służy instrukcja **UPDATE**. Wiersze, których pola mają być zaktualizowane, wskazuje się w opcjonalnej klauzuli **WHERE** — jej pominięcie powoduje zaktualizowanie wszystkich wierszy tabeli.
- Żeby przy użyciu jednej instrukcji **UPDATE** zaktualizować wiele pól, wystarczy nazwy kolumn rozdzielić przecinkami i każdej z nich przypisać nową wartość.
- Pojedyncza instrukcja **MERGE** pozwala wstawić, usunąć lub zaktualizować odpowiednie wiersze tabeli docelowej i doskonale nadaje się do synchronizowania zawartości różnych tabel.
- Wszystkie zmiany danych muszą być zgodne z nałożonymi na tabele ograniczeniami. W przeciwnym razie serwer bazodanowy zgłosi błąd i nie wykona instrukcji **INSERT**, **UPDATE**, **DELETE** czy **MERGE**.
- Wewnątrz instrukcji modyfikujących dane można umieszczać zapytania. Reguły zagnieżdżania zapytań są takie same jak reguły dotyczące podzapytań.

Zadania

1. Przeień o 25% wszystkie produkty z kategorii **Forks**, jednocześnie zwiększając ich koszt standardowy o dwie jednostki.
2. Utwórz tabelę **Panie** zawierającą identyfikatory, imiona i nazwiska wszystkich klientek firmy AdventureWorks. Przyjmij, że tylko imiona pań kończą się na literę a.
3. Wykonaj poniższe instrukcje modyfikujące dane w utworzonej w poprzednim zadaniu tabeli **Panie**, a następnie zsynchronizuj zawartość tej tabeli z danymi pań odczytanymi z tabeli **[SalesLT].[Customer]**:

```
DELETE FROM Panie  
WHERE [CustomerID]<50;  
  
UPDATE Panie  
SET [FirstName]='X'  
WHERE [CustomerID]%2=1;  
  
INSERT INTO Panie ([FirstName], [LastName])  
VALUES ('Ala', 'Nowak');
```

[2] Więcej informacji na temat ograniczeń znajduje się w rozdziale 12.

[3] W serwerze SQL jest to opcja `IDENTITY_INSERT`.

[4] Dotyczy to wszystkich ograniczeń — jeżeli wstawiany wiersz będzie niezgodny z którymkolwiek z nich, serwer bazodanowy zgłosi błąd, a instrukcja `INSERT` nie zostanie wykonana.

[5] Wykonanie instrukcji `TRUNCATE TABLE` nazywa się obcięciem tabeli.

[6] Niektóre serwery bazodanowe umożliwiają kaskadowe usuwanie wierszy z powiązanych tabel. Zdecydowanie odradzamy korzystanie z tej funkcjonalności — w najgorszym przypadku usuwając jeden wiersz, automatycznie skasujemy całą zawartość wszystkich tabel bazy danych.

[7] Niemożliwe jest jednoczesne (przeprowadzone w ramach jednej instrukcji `INSERT`, `UPDATE` czy `DELETE`) zmodyfikowanie danych zapisanych w różnych tabelach.

[8] Wielokrotne aktualizowanie różnych kolumn tych samych wierszy przypomina kupowanie po jednym wielu znaczków pocztowych. Dopóki klientów jest mniej niż otwartych na poczcie okienek, taka strategia, chociaż nieoptymalna, jest do zaakceptowania. Jeżeli jednak do każdego okienka ustawi się kolejka, a my po zakupie jednego znaczka za każdym razem będziemy ustawiali się na jej końcu, kupno znaczków zajmie nam mnóstwo czasu i w dodatku spowoduje wydłużenie kolejki. **Dlatego problemy związane z niewydajnymi instrukcjami SQL najwyraźniej widać w środowiskach produkcyjnych, a nie testowych.**

[9] W pozostałych serwerach tę samą funkcjonalność uzyskamy za

pomocą podzapytań.

[10] W serwerze SQL obiekty (w tym tabele) z nazwami zaczynającymi się znakiem # są obiektami tymczasowymi, czyli przeznaczonymi do automatycznego usunięcia po zakończeniu sesji użytkownika, w ramach której zostały utworzone.

Rozdział 11. Transakcje i współbieżność

- Czym są transakcje?
- Co oznacza skrót ACID?
- Jakie są zalety transakcyjnego przetwarzania danych?
- Na czym polega różnica pomiędzy transakcjami zagnieżdżonymi a zagnieżdżaniem transakcji?
- Co oznacza termin „współbieżność”?
- Po co serwery bazodanowe zakładają blokady?
- Kiedy dochodzi do zakleszczeń?
- Czy warto zmieniać domyślny poziom izolowania transakcji?
- W jakich sytuacjach optymistyczny model współbieżności jest lepszy niż pesymistyczny?

Właściwości transakcji

Transakcje gwarantują spójność modyfikowanych informacji. Typowym przykładem transakcyjnego przetwarzania danych jest przeniesienie pieniędzy z jednego konta na drugie. Taka operacja przebiega w dwóch etapach:

1. zmniejszenie o pewną sumę stanu konta X;
2. dodanie tej sumy do stanu konta Y.

Gdyby po wykonaniu pierwszej operacji wystąpił błąd uniemożliwiający wykonanie drugiej, z systemu zniknęłaby pewna suma pieniędzy. Równie nieprzyjemnym zaskoczeniem dla właściciela byłoby sprawdzenie przez niego stanu obu jego kont już po odjęciu danej sumy z pierwszego, ale przed jej dodaniem do drugiego konta.

Żeby temu zapobiec, transakcje muszą być:

1. Niepodzielne (ang. *Atomicity*). Niepodzielność oznacza, że zatwierdzane są wszystkie wchodzące w skład transakcji instrukcje albo nie jest zatwierdzana żadna z nich. Innymi słowy, wszystkie wchodzące w skład transakcji instrukcje muszą być wykonane poprawnie — jeżeli choć jedna z nich zgłosi błąd, wszystkie przeprowadzone w ramach transakcji zmiany zostaną

wycofane.

2. Spójne (ang. *Consistency*). Ta cecha transakcji gwarantuje, że zarówno ich zatwierdzenie, jak i wycofanie nie doprowadzi do utraty spójności danych. Ponieważ wszystkie zmiany danych są wykonywane w ramach transakcji, przechowywane w bazach informacje zawsze będą spójne.
3. Izolowane (ang. *Isolation*). Izolowanie transakcji wymaga albo zablokowania modyfikowanych w ramach jednej z nich danych, albo utworzenia ich dodatkowej wersji. W zależności od obowiązującego w ramach serwera lub sesji klienckiej poziomu izolowania transakcji może dojść do następujących sytuacji:
 - a. Utrata aktualizacji (ang. *Lost update*) ma miejsce, gdy dwa procesy modyfikują jednocześnie te same dane. Przykładowo jeden użytkownik zmienia cenę towaru na 100 zł, a drugi — na 200 zł. W takim przypadku jedna ze zmian zostanie utracona (zastąpiona drugą modyfikacją). **Domyślnie skonfigurowane serwery bazodanowe nie dopuszczają do utraty aktualizacji.**
 - b. Brudne odczyty (ang. *Dirty reads*) — do takiej sytuacji dochodzi, gdy możliwe jest odczytanie zmian niezatwierdzonych jeszcze przez inny proces. Jeżeli proces odczytujący nie zażąda założenia blokady na odczytywanych danych, uzyska do nich dostęp nawet wtedy, kiedy właśnie będą modyfikowane. Gdyby proces modyfikujący wyczołał wprowadzone zmiany, odczytane dane okazałyby się niespójne. **Domyślnie skonfigurowane serwery bazodanowe nie dopuszczają brudnych odczytów.**
 - c. Niepowtarzalne odczyty (ang. *Non-repeatable reads*) mają miejsce, gdy powtórzenie w ramach transakcji tego samego odczytu daje inny wynik. Różnice w wynikach są spowodowane tym, że natychmiast po zakończeniu odczytu (a nie po zakończeniu całej transakcji) proces odczytujący zdejmuje blokady założone na odczytywane dane. Niezablokowane dane mogą być zmienione przez inny proces, a więc ich powtórne odczytanie da inny (niespójny) wynik. **Domyślnie skonfigurowane serwery bazodanowe dopuszczają niepowtarzalne odczyty.**

- d. Odczyty widma (ang. *Phantom reads*) — sytuacja taka ma miejsce, jeżeli pomiędzy dwoma wykonanymi w ramach transakcji odczytami zmieni się liczba odczytywanych wierszy. Jeśli np. podczas pierwszego odczytu w tabeli Produkty znajdowało się 100 produktów o cenach niższych niż 10 zł, instrukcja `SELECT * FROM Produkty WHERE Cena <10` zwróciłaby 100 wierszy. W trakcie trwania transakcji możliwa jest jednak zmiana pozostałych wierszy tabeli, w tym obniżenie ceny jakiegoś produktu poniżej 10 zł. Możliwe jest również wstawienie do tej tabeli nowego produktu o cenie np. 7 zł. Z tego powodu drugie wykonanie tego samego zapytania zwróciłoby już 102 wiersze. **Domyślnie skonfigurowane serwery bazodanowe dopuszczają odczyty widma.**
4. Trwałe (ang. *Durability*). Trwałość transakcji gwarantuje, że efekty zatwierdzonych transakcji będą zapisane w bazie, nawet w przypadku awarii serwera baz danych. Do przywrócenia spójności danych serwery bazodanowe z reguły używają jakiejś formy dziennika transakcyjnego.

 Wskazówka	Pierwsze litery cech transakcji (A — <i>Atomicity</i> , C — <i>Consistency</i> , I — <i>Isolation</i> , D — <i>Durability</i>) tworzą skrót ACID, powszechnie używany do opisywania reguł przetwarzania danych, których muszą przestrzegać serwery bazodanowe, żeby mogły być nazwane transakcyjnymi lub relacyjnymi.
--	--

Transakcyjne przetwarzanie danych

Serwery bazodanowe mogą działać w trybie niejawnego zatwierdzania transakcji (w serwerze SQL taki tryb jest trybem domyślnym). Oznacza to, że użytkownicy nie muszą samodzielnie rozpoczynać transakcji, bo serwer robi to za nich.

W trybie niejawnego zatwierdzania transakcji wykonanie każdej instrukcji języka SQL składa się z trzech etapów:

1. Serwer bazodanowy automatycznie rozpoczyna nową transakcję.
2. Wykonywana jest pojedyncza instrukcja SQL.
3. Jeżeli instrukcja została wykonana z powodzeniem, transakcja jest zatwierdzana, w przeciwnym razie jest wycofywana.



Wskazówka

Taki sposób działania oznacza, że użytkownicy nie mogą samodzielnie zatwierdzać lub wycofywać automatycznie rozpoczętych transakcji. Dlatego nazywa się on trybem niejawnego zatwierdzania transakcji.

Poniższy przykład ilustruje działanie trybu niejawnego zatwierdzania transakcji za pomocą funkcji systemowej @@TRANCOUNT zwracającej liczbę otwartych, aktywnych w danym momencie transakcji:

```
SELECT @@TRANCOUNT;
UPDATE [SalesLT].[Product]
SET [ListPrice]=[ListPrice]-1
WHERE [ProductID] = 712;
SELECT @@TRANCOUNT;
```

```
-----  
0
```

```
0
```

Jak widać, przed rozpoczęciem i po zakończeniu wykonywania instrukcji UPDATE nie było żadnych otwartych transakcji. Skoro wszystkie modyfikacje danych przeprowadzane są w ramach transakcji, oznacza to, że serwer bazodanowy automatycznie rozpoczął i zatwierdził transakcję.

Tryb jawnego zatwierdzania transakcji

W niektórych serwerach bazodanowych (np. w serwerze Oracle) domyślnym trybem transakcyjnego przetwarzania danych jest tryb jawnego zatwierdzania. W tym trybie wykonanie każdej instrukcji języka SQL przebiega następująco:

1. Serwer bazodanowy automatycznie rozpoczyna nową transakcję.
2. Wykonywana jest pojedyncza instrukcja SQL.
3. Użytkownik samodzielnie musi zatwierdzić lub wycofać otwartą przez serwer transakcję.

Działanie tego trybu można zasymulować w serwerze SQL, ustawiając opcję sesji IMPLICIT_TRANSACTIONS:

```
SELECT @@TRANCOUNT;
UPDATE [SalesLT].[Product]
```

```
SET [ListPrice]=[ListPrice]-1  
WHERE [ProductID] = 712;  
SELECT @@TRANCOUNT;
```

```
-----  
0  
1
```

Tym razem przed rozpoczęciem instrukcji UPDATE również nie było otwartych transakcji, ale niejawnie rozpoczęta transakcja nie została po jej wykonaniu automatycznie zamknięta. Musi to zrobić sam użytkownik — albo zatwierdzając wprowadzone zmiany, albo je wycofując.

Przed przejściem do dalszych ćwiczeń zakończ transakcję i wyłącz omawiany tryb:

```
COMMIT TRAN;  
SET IMPLICIT_TRANSACTIONS OFF;
```



Wskaźówka

Tryb jawnego zatwierdzania transakcji pozwala wycofywać przypadkowe lub błędne modyfikacje, ale zatwierdzanie transakcji, która nie została przez nas rozpoczęta, jest mało intuicyjne.

Rozpoczynanie transakcji

Mechanizm transakcyjnego przetwarzania danych pokażemy, jawnie rozpoczynając i kończąc transakcje. Dzięki temu będziemy mogli wykonać w ramach poszczególnych transakcji dowolną liczbę instrukcji oraz samodzielnie sterować czasem rozpoczęcia i zakończenia poszczególnych transakcji.

Żeby rozpocząć transakcję, należy wykonać instrukcję BEGIN TRAN^[1]:

```
BEGIN TRAN;  
SELECT @@TRANCOUNT;  
-----  
1
```

Jeżeli teraz w ramach tej samej sesji (czyli w tym samym oknie edytora SQL) zaktualizujemy ceny wybranych towarów i sprawdzimy liczbę aktywnych transakcji, dowiemy się, że rozpoczęta przez nas

transakcja nadal jest otwarta:

```
UPDATE [SalesLT].[Product]
SET [ListPrice]=[ListPrice]-1
WHERE [ProductID] = 712;
```

1

Dopóki transakcja, w ramach której przeprowadziliśmy dowolne zmiany, jest otwarta, możemy je albo wycofać, albo zatwierdzić. Ponieważ serwer bazodanowy nie jest w stanie przewidzieć naszej decyzji, a jedną z cech transakcji jest jej odizolowanie, próba odczytania danych z tabeli `dbo.Produkty` w ramach tej samej sesji skończy się zupełnie inaczej niż ta sama próba wykonana przez innego użytkownika.

Żeby się o tym przekonać:

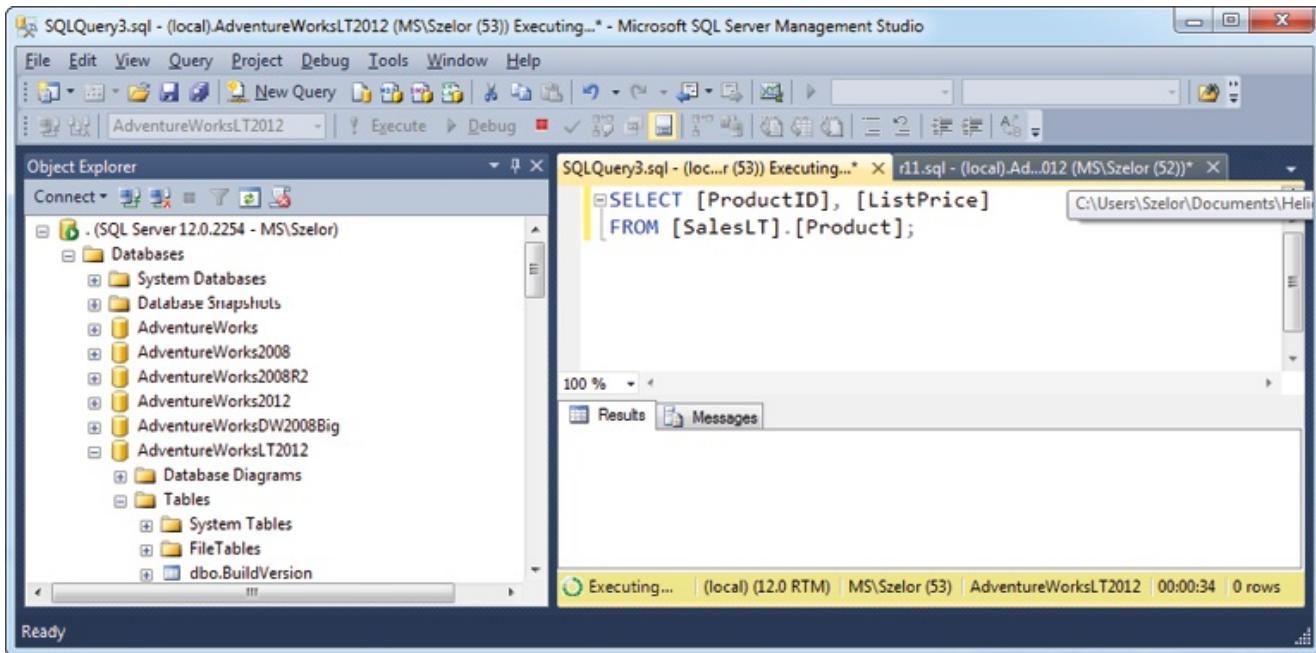
1. W tym samym oknie kodu SQL wykonaj zapytanie:

```
SELECT [ProductID], [ListPrice]
FROM [SalesLT].[Product];
```

ProductID	ListPrice
709	4,275
710	4,275
711	34,99
712	1

...

2. Zostało ono natychmiast wykonane, a cena produktu o numerze 712 wynosi 1.
3. Aby wykonać to samo zapytanie jako inny użytkownik, otwórz nowe okno edytora SQL^[2] i skopiuj do niego powyższą instrukcję SELECT (rysunek 11.1).



Rysunek 11.1. Zapytanie wykonuje się już ponad pół minuty, ale wciąż nie zwróciło żadnych danych

	Transakcyjne przetwarzanie danych polega na takim realizowaniu żądań klientów przez serwery bazodanowe, żeby każdy klient miał wrażenie, iż jest jedynym użytkownikiem serwera. Wymaga to opisanego w dalszej części rozdziału blokowania obiektów, do których w danym momencie odwołują się inni użytkownicy serwera.
--	--

Wycfywanie transakcji

Wycfanie transakcji oznacza przywrócenie danych do stanu sprzed jej rozpoczęcia i zdjęcie wszystkich założonych na potrzeby transakcji blokad. Jeżeli wróćmy do pierwszego okna edytora SQL (tego, w którym zapytanie zwróciło wyniki) i wykonamy w nim instrukcję `ROLLBACK TRAN`^[3], a następnie przełączymy się do drugiego okna edytora SQL, przekonamy się, że zapytanie wreszcie zostało wykonane i w dodatku cena produktu o identyfikatorze 712 wcale nie wynosi 1. Spowodowane jest to wycfaniem transakcji, w ramach której ceny były zmienione, i zdjęciem założonych na jej potrzeby blokad:

```
SELECT [ProductID], [ListPrice]
FROM [SalesLT].[Product];
```

ProductID	ListPrice
-----------	-----------

707	34,99
708	34,99
709	4,275
710	4,275
711	34,99
712	6,99
713	49,99

...

Zatwierdzanie transakcji

Zatwierdzenie transakcji oznacza utrwalenie wprowadzonych w jej trakcie zmian i zdjęcie wszystkich założonych na potrzeby transakcji blokad. Wspomniany na początku rozdziału przykład przelania pieniędzy z jednego konta na drugie mógłby być zaimplementowany w taki sposób:

```
BEGIN TRAN;
EXEC uspDodajDoKonta '123-456-78-90', 500;
EXEC uspOdejmijOdKonta '231-645-87-09', 500;
IF @@ERROR=0
    COMMIT TRAN;
ELSE
    ROLLBACK TRAN;
```

Po jawnym rozpoczęciu transakcji następuje wywołanie dwóch (nieistniejących w przykładowej bazie danych) procedur. Jeżeli żadna z nich nie zgłosi błędu, cała transakcja będzie zatwierdzona (zatwierdzić transakcję możemy, wykonując instrukcję COMMIT TRAN^[41]), w przeciwnym razie zostanie ona wycofana.

Zagnieżdżanie transakcji

Większość serwerów bazodanowych pozwala zagnieżdżać transakcje, czyli wykonać instrukcję BEGIN TRAN w ramach wcześniej rozpoczętej transakcji. **Wynikiem takiej operacji jest zwiększenie licznika otwartych transakcji, a nie rozpoczęcie nowej (atomowej, niepodzielnej, trwałej i spójnej) transakcji.**

Działanie mechanizmu zagnieżdżania transakcji ilustruje poniższy

przykład: wykonanie instrukcji BEGIN TRAN powoduje zwiększenie o jeden licznika otwartych transakcji, wykonanie instrukcji COMMIT TRAN zmniejsza wartość tego licznika o jeden, ale wykonanie instrukcji ROLLBACK TRAN zamyka transakcję i zeruje licznik otwartych transakcji:

```
BEGIN TRAN;  
SELECT @@TRANCOUNT;  
BEGIN TRAN;  
SELECT @@TRANCOUNT;  
BEGIN TRAN;  
SELECT @@TRANCOUNT;  
COMMIT TRAN;  
SELECT @@TRANCOUNT;  
ROLLBACK TRAN;  
SELECT @@TRANCOUNT;
```

```
-----  
1  
2  
3  
2  
0
```



Wskazówka

Powyższy przykład demonstruje ważną cechę transakcyjnego przetwarzania danych przez serwery bazodanowe — chociaż możliwe jest wykonanie instrukcji BEGIN TRANSACTION w ramach już istniejącej transakcji, nie spowoduje ona rozpoczęcie nowej transakcji, a więc wykonanie instrukcji COMMIT nie zatwierdzi żadnych zmian, a jedynie zmniejszy licznik otwartych transakcji o jeden. Ponieważ jednocześnie otwarta może być jedna transakcja, tylko ona cała może być zatwierdzona lub wycofana.

Punkty przywracania

Większość serwerów bazodanowych pozwala wycofać nie tylko całą transakcję, lecz także jej część. W tym celu należy w trakcie transakcji wykonać instrukcję `SAVE TRAN`^[5], a następnie przywrócić ją do danego punktu:

```
BEGIN TRAN;  
INSERT INTO [SalesLT].[ProductCategory](Name)  
VALUES ('TEST1');
```

```
SAVE TRAN PP1;
INSERT INTO [SalesLT].[ProductCategory](Name)
VALUES ('TEST2');
SELECT @@TRANCOUNT;
ROLLBACK TRAN PP1;
SELECT @@TRANCOUNT;
COMMIT TRAN;
```

1

1

Ponieważ przywrócenie stanu transakcji do określonego punktu nie powoduje jej zakończenia (liczba otwartych transakcji nadal wynosi 1), musimy ją zatwierdzić lub wycofać:

```
SELECT [Name]
FROM [SalesLT].[ProductCategory]
WHERE [Name] LIKE 'TEST_';
```

Name

TEST1

Jako że druga instrukcja `INSERT` została wykonana po zdefiniowaniu punktu przywracania `PP1`, instrukcja `ROLLBACK TRAN PP1` przywróciła stan danych do momentu sprzed jej wykonania i w rezultacie tylko pierwszy wiersz został na trwałe wstawiony do tabeli.

Współbieżność

Współbieżność to zdolność systemu do jednoczesnego realizowania wielu operacji, z reguły uzyskiwana poprzez uruchomienie osobnych procesów (robotników) na potrzeby obsługi poszczególnych żądań.

 Wskazówka	Współbieżność ma ogromny wpływ na skalowalność serwerów bazodanowych, czyli ich zdolność do coraz szybszego wykonywania transakcji dzięki rozbudowywaniu komputerów, np. zwiększaniu ich mocy obliczeniowej czy przepustowości dysków twardych.
--	---

Żeby każdy z kilkuset czy nawet kilku tysięcy jednoczesnych

użytkowników serwera bazodanowego mógł pracować tak, jakby był jego jedynym użytkownikiem, konieczne jest odizolowanie od siebie poszczególnych transakcji. Umożliwiają to automatycznie zakładane blokady.

Blokady

Pomijając analizy wewnętrznych mechanizmów działania różnych serwerów bazodanowych, podzielimy blokady ze względu na ich tryb (sposób blokowania) i zakres (typ blokowanych zasobów).

Tryby blokad

Tryb blokady decyduje o tym, czy możliwe będzie jej założenie na zasobie wcześniejszej zablokowanym przez inny proces:

1. **Blokady współdzielone S** (ang. *Shared*) są domyślnie zakładane na odczytywanych obiektach, takich jak tabele czy wiersze. Na obiekt zablokowany w trybie S inne procesy też mogą założyć blokadę S, czyli **odczytujący nie blokują innych odczytujących**. Blokady S domyślnie zakładane są tylko na czas wykonywania zapytania, a nie całej transakcji.
2. **Blokady wyłączne X** (ang. *eXclusive*) są zakładane na modyfikowanych obiektach. Blokady X są niekompatybilne z innymi blokadami, czyli modyfikujący blokują innych użytkowników. W przeciwieństwie do blokad współdzielonych blokady wyłączne domyślnie są utrzymywane do zakończenia całej transakcji, a nie pojedynczej operacji.

Zakresy blokad

Blokady mogą być zakładane na poziomie poszczególnych wierszy, kluczy indeksów, stron, zakresów lub całych tabel. Te obiekty tworzą naturalną hierarchię: tabela składa się z wielu stron, na każdej stronie zapisanych jest wiele wierszy i tak dalej. Z tego powodu serwery bazodanowe muszą analizować wszystkie istniejące blokady, zanim założą nową — jeżeli choć jeden wiersz tabeli jest zablokowany w trybie X, nie można na całej tabeli założyć innej blokady.

	Im większe obiekty są blokowane, tym mniejsza współbieżność (bo użytkownicy muszą dłużej czekać na dostęp do zablokowanych zasobów), ale
--	--



Wskazówka

również tym mniejsza liczba blokad, którymi musi zarządzać serwer bazodanowy (zostanie założona jedna blokada na całej tabeli zamiast miliona blokad na poszczególnych wierszach).

Zakleszczenia

Zakleszczenie (ang. *DeadLock*) ma miejsce, gdy różne procesy blokują się nawzajem w taki sposób, że żaden z nich nie jest w stanie założyć blokad wymaganych do ukończenia już rozpoczętych operacji.

Najczęściej występują dwa typy zakleszczeń:

1. Zakleszczenia cykliczne, wynikające z tego, że dwa procesy w różnych kolejnościach próbują uzyskać dostęp do tych samych zasobów.
2. Zakleszczenia konwersji blokad, związane ze zmianą wcześniej założonej blokady współdzielonej (wiele procesów może jednocześnie zablokować ten sam zasób w trybie S) na blokadę wyłączną (tylko jeden proces może założyć na tym samym obiekcie blokadę X).

Serwery bazodanowe automatycznie wykrywają zakleszczenia i przerywają działanie jednego procesu. Na ofiarę zakleszczenia wybierany jest proces o niższym priorytecie, a jeżeli oba procesy mają ten sam priorytet, ofiarą zakleszczenia zostaje ten, którego wycofanie jest mniej kosztowne.

Mechanizm wykrywania i usuwania zakleszczeń pokazuje poniższy przykład:

Pierwszy użytkownik w ramach jawnie rozpoczętej transakcji modyfikuje kilka danych w tabeli [SalesLT].[Product]:

```
BEGIN TRAN;  
UPDATE [SalesLT].[Product]  
SET Name = UPPER(Name)  
WHERE ProductID<720;
```

```
-----  
(15 row(s) affected)
```

Następnie inny użytkownik w ramach jawnie rozpoczętej przez

siebie transakcji modyfikuje znacznie więcej danych w tabeli^[6]:

```
UPDATE [SalesLT].[SalesOrderDetail]
SET [UnitPriceDiscount] += 1
WHERE [ProductID] < 800;
```

```
-----  
(120 row(s) affected)
```

W dalszej kolejności pierwszy użytkownik próbuje odczytać zawartość tabeli zablokowanej już przez drugą sesję (okno wyników może pokazać kilka wierszy, ale i tak użytkownik będzie musiał czekać na możliwość zablokowania w trybie S pozostałych wierszy tabeli Transakcje magazynowe):

```
SELECT *
FROM [SalesLT].[SalesOrderDetail];
```

W tym momencie nie wystąpiło jeszcze zakleszczenie — wystarczyłoby, żeby drugi użytkownik zakończył swoją transakcję. Ale jeżeli w ramach drugiej sesji użytkownik spróbuje odczytać zawartość tabeli zmodyfikowanej i zablokowanej przez pierwszego użytkownika, oba procesy się zakleszczą:

```
SELECT *
FROM [SalesLT].[Product];
```

```
-----  
Name  
Classic Vest, L  
Classic Vest, M  
Classic Vest, S  
Fender Set - Mountain  
Front Brakes  
Front Derailleur
```

Po chwili drugie zapytanie zostało jednak wykonane, co więcej — nazwy produktów nie zostały przekonwertowane na wielkie litery. Żeby przekonać się, dlaczego tak się stało, wystarczy przełączyć się do okienka pierwszej sesji. Znajdziemy w nim poniższy komunikat błędu:

```
Msg 1205, Level 13, State 51, Line 7
```

Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Jeżeli sprawdzimy liczbę otwartych w ramach pierwszej sesji transakcji, okaże się, że jawnie rozpoczęta przez pierwszego użytkownika transakcja została — zgodnie z komunikatem błędu — wycofana:

```
SELECT @@TRANCOUNT;
```

```
-----  
0
```

Ponieważ wycofanie transakcji wiąże się ze zdjęciem założonych na jej potrzeby blokad, druga sesja mogła z powodzeniem zakończyć operacje i odczytać tabelę [SalesLT].[Product]. Liczba transakcji otwartych w ramach drugiej sesji nadal wynosi 1 — żeby zakończyć ćwiczenie i wycofać zmiany, należy wykonać w tym oknie edytora SQL instrukcję ROLLBACK TRAN.

Poziomy izolowania transakcji

Możemy wpływać na sposób zakładania blokad przez serwery bazodanowe, zmieniając poziom izolowania transakcji. Większość serwerów pozwala ustawić (na poziomie serwera, bazy danych lub poszczególnych sesji) jeden z czterech poziomów izolowania transakcji, przedstawionych przez nas od najmniej restrykcyjnego, w którym maksymalna współbieżność jest okupiona występowaniem największej liczby typów niespójności danych, do najbardziej restrykcyjnego, który kosztem ograniczenia współbieżności gwarantuje najwyższy poziom spójności danych.

Read Uncommitted

W trybie niezatwierdzonego odczytu (ang. *Read Uncommitted*) odczyt danych nie powoduje założenia blokady współdzielonej. **Na tym poziomie występują brudne odczyty, niepowtarzalne odczyty i odczyty widma (jedynym niekorzystnym zjawiskiem niewystępującym na tym poziomie jest utrata aktualizacji).**

Żeby się o tym przekonać:

1. W jednej sesji (oknie edytora SQL) rozpoczęmy transakcję i zaktualizujemy adres e-mail klienta:

```
BEGIN TRAN;  
UPDATE [SalesLT].[Customer]  
SET [EmailAddress] = 'ZmianaWToku'  
WHERE CustomerID=1;  
-----  
(1 row(s) affected)
```

2. W drugiej sesji zmienimy poziom izolowania transakcji na Read Uncommitted i spróbujemy odczytać modyfikowane przez innego użytkownika dane:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
GO  
SELECT [EmailAddress]  
FROM [SalesLT].[Customer]  
WHERE CustomerID=1;  
-----  
ZmianaWToku
```

Udało nam się odczytać dane, mimo że osoba, która je zmieniała, nie zatwierdziła jeszcze transakcji, a więc w każdej chwili może ją wycofać. **W tym trybie** (często wymuszonym na poziomie poszczególnych instrukcji za pomocą specyficznych dla danego serwera bazodanowego dyrektyw optymalizatora) **można odczytywać dane, o których wiemy, że nie będą w tym samym czasie modyfikowane.**

Kończąc ćwiczenie, zamknij bez zatwierdzania otwartej transakcji i na nowo otwórz oba okna edytora SQL — w ten sposób kolejne ćwiczenie rozpocznesz, pracując w domyślnym trybie izolowania transakcji.

Read Committed

Tryb odczytu zatwierzonego (ang. *Read Committed*) **jest domyślnym poziomem izolowania transakcji.** Na tym poziomie odczyt danych wymaga założenia na nich blokady współdzielonej. Ponieważ zakładana na czas zmiany blokada X jest niekompatybilna z innymi blokadami, w tym z blokadą S, eliminuje to brudne odczyty. Jednak **na tym poziomie nadal występują niepowtarzalne**

odczyty i odczyty widma.

Zjawisko niepowtarzalnego odczytu pokazuje poniższy przykład:

1. W pierwszym oknie edytora SQL ustawiamy tryb odczytów zatwierdzonych^[7], jawnie rozpoczynamy transakcję i odczytujemy adres e-mail wybranego klienta:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN TRAN;
```

```
SELECT [EmailAddress]  
FROM [SalesLT].[Customer]  
WHERE CustomerID=1;
```

```
-----  
EmailAddress  
orlando0@adventure-works.com
```

2. W tym momencie transakcja jest nadal otwarta, a my w drugim oknie edytora SQL zmienimy adres tego klienta:

```
UPDATE [SalesLT].[Customer]  
SET [EmailAddress] = 'OdczytWToku'  
WHERE CustomerID=1;
```

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

3. Jeżeli pierwszy użytkownik w ramach tej samej transakcji ponownie odczyta adres tego klienta, uzyska inny wynik:

```
SELECT [EmailAddress]  
FROM [SalesLT].[Customer]  
WHERE CustomerID=1;  
COMMIT TRAN;
```

```
-----  
OdczytWToku
```

Powodem takiego zachowania serwera bazodanowego jest to, że na domyślnym poziomie izolowania transakcji (na poziomie *Read Committed*) blokada współdzielona jest zakładana tylko na czas trwania pojedynczej instrukcji SELECT, a nie całej transakcji. Dlatego

instrukcja UPDATE została wykonana, co w konsekwencji doprowadziło do tego, że ponowny odczyt tego samego wiersza w ramach tej samej transakcji zwrócił inny wynik.

Repeatable Read

W trybie powtarzalnego odczytu (ang. *Repeatable Read*) blokady współdzielone typu S są utrzymywane do czasu zakończenia całej transakcji, a nie pojedynczych zapytań. Dzięki temu inny proces nie może zmodyfikować odczytywanych w jej ramach danych, co eliminuje niepowtarzalne odczyty. Z niekorzystnych zjawisk związanych z izolowaniem transakcji **na tym poziomie występują tylko odczyty widma**.

Zjawisko odczytu widma pokazuje poniższy przykład:

1. W ramach pierwszej sesji zmienimy poziom izolowania transakcji na REPEATABLE READ i w ramach jawnie rozpoczętej transakcji odczytamy nazwy towarów o kodach kończących się cyfrą 6:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN TRAN;
```

```
SELECT [Name]
```

```
FROM [SalesLT].[Product]
```

```
WHERE [ProductNumber] LIKE '%6';
```

```
-----
```

```
Name
```

```
HL Mountain Frame - Black, 46
```

```
HL Mountain Frame - Silver, 46
```

```
HL Road Frame - Red, 56
```

```
HL Road Handlebars
```

```
HL Touring Frame - Blue, 46
```

```
HL Touring Frame - Yellow, 46
```

```
LL Headset
```

```
LL Mountain Seat/Saddle
```

```
ML Fork
```

```
ML Mountain Frame-W - Silver, 46
```

```
Mountain-200 Black, 46
```

```
Mountain-200 Silver, 46
```

- Mountain-400-W Silver, 46
Road-150 Red, 56
Touring-1000 Blue, 46
Touring-1000 Yellow, 46
Touring-2000 Blue, 46
2. Podczas gdy pierwsza transakcja jest wciąż otwarta, w drugim oknie edytora SQL zmienimy kod jednego z pozostałych, niezwróconych przez pierwsze zapytanie produktu na FB-9873, a więc na kod spełniający warunki pierwszego zapytania:
- ```
UPDATE [SalesLT].[Product]
SET [ProductNumber] = 'FB-9876'
WHERE [ProductNumber] ='FB-9873';

(1 row(s) affected)
```
3. Jeżeli pierwszy użytkownik raz jeszcze wykona, w ramach tej samej transakcji, to samo zapytanie, tym razem jego wynik będzie liczył więcej wierszy — pojawi się w nim wiersz widmo z produktem Front Brakes:
- ```
SELECT [Name]
FROM [SalesLT].[Product]
WHERE [ProductNumber] LIKE '%6';;
-----
Name
Front Brakes
HL Mountain Frame - Black, 46
HL Mountain Frame - Silver, 46
HL Road Frame - Red, 56
HL Road Handlebars
HL Touring Frame - Blue, 46
HL Touring Frame - Yellow, 46
LL Headset
LL Mountain Seat/Saddle
ML Fork
ML Mountain Frame-W - Silver, 46
```

Mountain-200 Black, 46
Mountain-200 Silver, 46
Mountain-400-W Silver, 46
Road-150 Red, 56
Touring-1000 Blue, 46
Touring-1000 Yellow, 46
Touring-2000 Blue, 46

4. Jeśli jednak w ramach drugiej sesji spróbujemy zmienić dane odczytywane w ramach nadal otwartej pierwszej transakcji (czyli doprowadzić do zjawiska niepowtarzalnego odczytu), instrukcja będzie oczekwać, aż pierwsza transakcja zostanie zakończona, a założone dla niej blokady zdjęte:

```
UPDATE dbo.Produkty
UPDATE [SalesLT].[Product]
SET [ProductNumber] = 'FR-M94B-41'
WHERE [ProductNumber] = 'FR-M94B-46';
```

5. Żeby powyższa aktualizacja została wykonana, w pierwszym oknie edytora SQL wykonaj instrukcję COMMIT TRAN.

W trybie REPEATABLE READ należy odczytywać te dane, które w ramach transakcji są odczytywane kilkakrotnie i mogą być zmieniane w tym samym czasie przez innych użytkowników. Sytuacja taka ma miejsce np. w różnego rodzaju zestawieniach i raportach zbiorczych, w których odczytując te same dane, za każdym razem musimy otrzymać te same wyniki, inaczej zestawienie lub raport będą niespójne.

Serializable

W trybie szeregowania transakcje odwołujące się do tych samych tabel są wykonywane jedna po drugiej. Blokowanie całych obiektów (albo zakresów kluczy indeksu), a nie tylko odczytywanych danych, na czas trwania transakcji pozwala wyeliminować odczyty widma, ale powoduje, że odczytując nawet jeden wiersz tabeli, możemy uniemożliwić pozostałym użytkownikom zmodyfikowanie przechowywanych w niej danych.

Żeby się o tym przekonać:

1. W pierwszym oknie edytora SQL przełączymy się do trybu szeregowania, jawnie rozpoczęmy transakcję i odczytamy informacje o wybranym towarze:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN;
SELECT [Name]
FROM [SalesLT].[Product]
WHERE [ProductNumber] LIKE '%6';
-----
```

```
Name
Front Brakes
HL Mountain Frame - Silver, 46
HL Road Frame - Red, 56
HL Road Handlebars
HL Touring Frame - Blue, 46
HL Touring Frame - Yellow, 46
LL Headset
LL Mountain Seat/Saddle
ML Fork
ML Mountain Frame-W - Silver, 46
Mountain-200 Black, 46
Mountain-200 Silver, 46
Mountain-400-W Silver, 46
Road-150 Red, 56
Touring-1000 Blue, 46
Touring-1000 Yellow, 46
Touring-2000 Blue, 46
```

2. Jeżeli teraz w drugim oknie edytora SQL spróbujemy zmienić dane dowolnego, również niezwróconego przez pierwsze zapytanie produktu, okaże się, że aktualizacja została zablokowana i będzie wykonana dopiero po zakończeniu pierwszej transakcji:

```
UPDATE [SalesLT].[Product]
SET Name = 'XYZ'
```

```
WHERE [ProductNumber] ='HL-U509-R';
```

3. Kończąc ćwiczenie, zamknij oba okna edytora SQL bez zatwierdzania rozpoczętej w jednym z nich transakcji.

W trybie SERIALIZABLE mamy gwarancję, że odczytywane w ramach transakcji dane zawsze będą takie same — serwer bazodanowy nie dopuści nie tylko do ich zmiany, lecz także do pojawienia się nowych danych. Jednak przez ten czas pozostali użytkownicy nie będą mogli modyfikować zablokowanych tabel. W większości przypadków powoduje to tak znaczne wydłużenie czasu reakcji serwera, że lepiej jest skopiować odczytywane dane^[8], a jeżeli zmian nie jest zbyt dużo, przełączyć się do modelu optymistycznego.

Model optymistyczny

W modelu optymistycznym tylko modyfikujący blokują innych modyfikujących, czyli różni użytkownicy mogą jednocześnie modyfikować i odczytywać te same dane.

Serwery bazodanowe zapewniają spójność modyfikowanych w tym modelu danych poprzez ich wersjonowanie. Zakładając (optymistycznie), że w czasie gdy jeden użytkownik odczytuje dane, inni raczej nie będą ich modyfikować, serwery te są w stanie na bieżąco zarządzać dodatkowymi wersjami danych.

Jeżeli to założenie jest prawdziwe, czyli jeżeli jednocześnie modyfikacje i odczyty tych samych danych nie zachodzą zbyt często, możemy znacznie skrócić czas reakcji serwera^[9], przełączając bazę do optymistycznego modelu współbieżności. Żeby się o tym przekonać:

1. W pierwszym oknie edytora SQL wykonamy poniższe instrukcje, przełączając bazę AdventureWorksLT2012 do modelu optymistycznego:

```
USE master;
ALTER DATABASE [AdventureWorksLT2012]
SET READ_COMMITTED_SNAPSHOT ON
WITH ROLLBACK IMMEDIATE;
```

Nonqualified transactions are being rolled back. Estimated rollback

completion: 0%.

Nonqualified transactions are being rolled back. Estimated rollback completion: 100%.

W tym samym oknie edytora SQL połączymy się z bazą AdventureWorksLT2012 i w ramach jawnie rozpoczętej transakcji zmienimy dane dwóch pracowników:

```
USE AdventureWorksLT2012;
GO
BEGIN TRAN;
UPDATE [SalesLT].[Customer]
SET LastName = 'X'
WHERE CustomerID <3;
-----
(2 row(s) affected)
```

2. W nowym oknie edytora SQL odczytamy dane o kilku pracownikach:

```
SELECT CustomerID, LastName
FROM [SalesLT].[Customer]
WHERE CustomerID <4;
```

```
-----
CustomerID      LastName
1               Nowak
2               Harris
3               Carreras
```

3. Okazuje się, że tym razem zapytanie zostało wykonane natychmiast, ale z zachowaniem wymogów domyślnego trybu izolowania transakcji, czyli trybu READ COMMITTED — **pozostali użytkownicy serwera odczytają ostatnią zatwierdzoną wersję danych**. Gdyby rozpoczęta w ramach pierwszej sesji transakcja została zatwierdzona, to ponowne wykonanie tego samego zapytania zwróciłoby najnowszą, zatwierdzoną wersję ze zmienionymi danymi dwóch pierwszych pracowników.

Model pesymistyczny

W modelu pesymistycznym odczytujący są blokowani przez

modyfikujących (serwer będzie czekał z założeniem blokady S, aż zdjęta zostanie blokada X), a **modyfikujący przez odczytujących** (założenie blokady X wymaga zdjęcia blokady S).

Ponieważ koszt zarządzania wieloma wersjami tych samych danych rośnie wraz ze wzrostem wersjonowanych danych, w tym modelu zakłada się (pesymistycznie), że odczytywane dane będą w tym samym czasie regularnie modyfikowane.

Żeby przywrócić pesymistyczny (domyślny) model współbieżności bazy *Northwind*, należy wykonać poniższe instrukcje:

```
USE master;
ALTER DATABASE AdventureWorksLT2012
SET READ_COMMITTED_SNAPSHOT OFF
WITH ROLLBACK IMMEDIATE;
```

```
-----  
Nonqualified transactions are being rolled back. Estimated rollback  
completion: 0%.
```

```
Nonqualified transactions are being rolled back. Estimated rollback  
completion: 100%.
```

Podsumowanie

- Serwery bazodanowe przeprowadzają wszystkie zmiany danych w ramach jawnie lub niejawnie rozpoczętych transakcji.
- Transakcje powinny być otwierane jak najpóźniej i zamykane jak najwcześniej.
- Transakcje powinny zawierać tylko powiązane ze sobą instrukcje.
- Przerwane (czy to z powodu awarii u klienta, czy też awarii serwera) transakcje będą wycofane.
- Na czas trwania transakcji pewne obiekty bazy danych są automatycznie blokowane.
- Serwery bazodanowe automatycznie wykrywają zakleszczenia i usuwają je poprzez wycofanie jednej z zakleszczonych transakcji.
- Odizolowanie, jedną z czterech cech ACID transakcji, uzyskuje się za pomocą automatycznie zakładanych i zwalnianych blokad.
- Można sterować sposobem zakładania i czasem trwania blokad, zmieniając poziom izolowania transakcji.
- W modelu optymistycznym serwery bazodanowe wersjonują

dane, co poprawia współbieżność kosztem większego obciążenia serwera.

Zadania

1. Twoim zadaniem jest przygotowanie raportu podsumowującego roczną sprzedaż. Wyliczając sumy i średnie wartości sprzedaży produktów, kilkakrotnie musisz odczytać tabelę `SalesOrderDetail`. Jak zagwarantujesz poprawność wyników raportu?
2. Po przerwie na lunch użytkownicy zgłaszają, że próby dalszej pracy z bazą danych kończą się chwilowym zawieszeniem programu i wreszcie komunikatem błędu mówiącym, iż serwer bazodanowy jest niedostępny. Po sprawdzeniu okazuje się, że serwer i sieć działają normalnie, a baza nie została uszkodzona. Co jest najbardziej prawdopodobną przyczyną problemu?
3. W ramach tworzonej procedury modyfikujesz duże ilości danych zapisanych w kilkunastu tabelach oraz wstawiasz jeden wiersz, informujący o wykonaniu wszystkich operacji, do tabeli znajdującej się w bazie danych na zdalnym serwerze. Połączenie między serwerami jest mocno obciążone i zdarza się, że czas nawiązywania sesji i przesyłania danych pomiędzy serwerami wielokrotnie się wydłuża. Co zrobić, aby w przypadku zgłoszenia przez procedurę błędu braku połączenia ze zdalnym serwerem nie trzeba było ponownie wykonywać długotrwałych modyfikacji danych?

[1] W niektórych serwerach bazodanowych transakcje rozpoczyna się instrukcjami `BEGIN TRANSACTION` lub `BEGIN WORK`.

[2] Można to zrobić, naciskając kombinację klawiszy `Ctrl+N` lub klikając przycisk *New Query*.

[3] W niektórych serwerach bazodanowych transakcje wycofuje się instrukcjami `ROLLBACK TRANSACTION` lub `ROLLBACK WORK`.

[4] W niektórych serwerach bazodanowych transakcje zatwierdza się instrukcjami `COMMIT TRANSACTION` lub `COMMIT WORK`.

[5] W niektórych serwerach bazodanowych punkty przywracania tworzy się instrukcjami `SAVE TRANSACTION` lub `SAVE WORK`.

[6] Możemy zasymulować jednoczesną pracę dwóch użytkowników, otwierając nowe okno edytora SQL — każde z okien nawiązuje własną sesję z bazą danych.

[7] Ponieważ ten tryb jest trybem domyślnym, instrukcja SET jest dodana tylko w celach demonstracyjnych.

[8] Niektóre serwery bazodanowe pozwalają utworzyć migawkę (ang. *Snapshot*) danych.

[9] Niektóre serwery bazodanowe, np. serwer Oracle, domyślnie działają w optymistycznym modelu współbieżności.

Część IV

Tworzenie baz danych, czyli instrukcje CREATE, ALTER i DROP

Język SQL pozwala tworzyć, modyfikować i usuwać bazy danych oraz znajdujące się w nich obiekty, takie jak tabele, widoki, indeksy, procedury czy funkcje. Niestety, o ile składnia instrukcji SELECT, INSERT, DELETE oraz UPDATE jest w dużym stopniu ustandaryzowana, o tyle każdy serwer bazodanowy zawiera specyficzne, często wynikające ze sposobu jego działania funkcjonalności. Instrukcje CREATE, ALTER i DROP wykorzystują unikalne cechy danego serwera i w konsekwencji ich składnia jest bardzo różna.

Choć te różnice występują przy tworzeniu wszystkich typów obiektów bazodanowych, największe są dla obiektów wyższego rzędu, takich jak procedury, wyzwalacze i funkcje. Dlatego z lektury tej części książki dowiesz się jedynie:

1. Jak utworzyć i usunąć bazę danych;
2. Jak tworzy się i usuwa tabele;
3. W jakim stopniu można zmienić strukturę istniejącej i wypełnionej danymi tabeli;
4. Jaką funkcję pełnią ograniczenia;
5. Czym są widoki;
6. W jaki sposób serwery bazodanowe wykonują instrukcje odwołujące się do widoków;
7. Jaką rolę odgrywają indeksy;
8. Jak tworzyć, usuwać i porządkować indeksy.

Rozdział 12. Bazy danych i tabele

- Co jest wynikiem instrukcji CREATE DATABASE?
- Kto i kiedy może usunąć bazę danych?
- Jak utworzyć podstawowe obiekty bazy danych, czyli tabele?
- Czemu warto umieszczać tabele (i obiekty innych typów) w osobnych schematach?
- Jak zmienić i usunąć istniejące tabele?
- Na czym polega deklaratywne wymuszanie spójności danych?
- Jak działają poszczególne ograniczenia?
- Czemu warto używać ograniczeń?

Tworzenie i usuwanie baz danych

Tworzyć obiekty dowolnego typu, w tym bazy danych, możemy instrukcją CREATE. Po czasowniku CREATE należy podać typ tworzonego obiektu i jego nazwę.

Serwer SQL przechowuje informacje o bazach danych użytkowników w specjalnej systemowej bazie master i to z jej poziomu powinniśmy zarządzać pozostałymi bazami danych. Żeby połączyć się z wybraną bazą danych, należy wykonać instrukcję USE:

```
USE master;
```

Po połączeniu się z bazą master, o ile tylko posiadamy wystarczające uprawnienia, możemy utworzyć nową bazę danych^[1]:

```
CREATE DATABASE Test;
```

W rezultacie wykonania powyższej instrukcji została stworzona nowa, pusta baza danych. **Jej fizyczna struktura zależy całkowicie od serwera i nie jest ujęta w standardzie SQL.** W przypadku serwera SQL każda baza danych:

1. Musi składać się z co najmniej dwóch plików — w jednym z nich są przechowywane definicje obiektów oraz dane, w drugim przeprowadzone przez użytkowników transakcje oraz informacje potrzebne do ich ewentualnego ponownego wykonania lub wycofania.

2. Tak jak każdy obiekt musi mieć właściciela — domyślnie właścicielem obiektu jest użytkownik, który go utworzył.
3. Znajduje się w określonym stanie — możliwe jest np. samodzielne przełączenie bazy danych w tryb tylko do odczytu oraz automatyczne przełączanie bazy w tryb offline po uszkodzeniu jej plików.
4. Działa na określonym poziomie kompatybilności — wewnętrznym numerem serwera SQL 2014 jest 12 i tylko w bazach działających na tym poziomie można używać nowych funkcji serwera bazodanowego.

Podstawowe informacje o bazie danych poznamy, wywołując procedurę systemową `sp_helpdb`:

```
EXEC sp_helpdb 'Test';
-----
name    db_size    owner    dbid    created    status    compatibility_level
Test      5.23 MB  MS\Szelor  7 Aug 17 2014  Status=ONLINE,
          Updateability=READ_WRITE, UserAccess=MULTI_USER, Recovery=FULL,
          Version=782, Collation=Polish_CI_AS, SQLSortOrder=0,
          IsAutoCreateStatistics, IsAutoUpdateStatistics, IsFullTextEnabled  120
name    fileid    filename   filegroup   size    maxsize    growth    usage
Test    1    C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\Test.mdf  PRIMARY  4288 KB
Unlimited  1024 KB  data only
Test_log  2    C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\Test_log.ldf  NULL    1072 KB
2147483648 KB  10%  log only
```

W tym przypadku:

1. Nazwą bazy danych jest *Test*.
2. Baza ma rozmiar 5 MB, z czego 4 MB przypadają na plik danych o nazwie *Test*, a 1 MB na plik dziennika o nazwie *Test_log*.
3. Baza jest własnością użytkownika *MS\Szelor* i została utworzona 17 sierpnia 2014 roku.
4. Baza jest dostępna (*Status=ONLINE*) w trybie do zapisu i odczytu dla wszystkich użytkowników, działa w trybie *Full Recovery*^[2], dane tekstowe są w niej zakodowane za pomocą strony kodowej *ANSI Polish_CI_AS* i włączone są opcje automatycznego tworzenia i aktualizowania statystyk.
5. Oba pliki bazy danych znajdują się w tym samym folderze i będą

automatycznie zwiększane przez serwer SQL.

Niektóre z tych opcji można podać podczas tworzenia bazy danych (w poprzednim przykładzie zostały użyte domyślne wartości), wiele innych można zmienić, wykonując instrukcję ALTER DATABASE. Poniższy skrypt może być wykonany wyłącznie na serwerze SQL 2014, ponieważ inne serwery w specyficzny sposób zarządzają bazami danych:

```
CREATE DATABASE [TestMS]
CONTAINMENT = NONE
ON PRIMARY
( NAME = N'TestMS', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestMS.mdf' , SIZE = 5120KB ,
FILEGROWTH = 1024KB )
LOG ON
( NAME = N'TestMS_log', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL12.MSSQLSERVER\MSSQL\DATA\TestMS_log.ldf' , SIZE = 1024KB ,
FILEGROWTH = 10%)
GO
ALTER DATABASE [TestMS] SET COMPATIBILITY_LEVEL = 120
GO
ALTER DATABASE [TestMS] SET ANSI_NULL_DEFAULT OFF
GO
ALTER DATABASE [TestMS] SET ANSI_NULLS OFF
GO
ALTER DATABASE [TestMS] SET ANSI_PADDING OFF
GO
ALTER DATABASE [TestMS] SET ANSI_WARNINGS OFF
GO
ALTER DATABASE [TestMS] SET ARITHABORT OFF
GO
ALTER DATABASE [TestMS] SET AUTO_CLOSE OFF
GO
ALTER DATABASE [TestMS] SET AUTO_SHRINK OFF
GO
ALTER DATABASE [TestMS] SET AUTO_CREATE_STATISTICS ON
```

```
GO
ALTER DATABASE [TestMS] SET AUTO_UPDATE_STATISTICS ON
GO
ALTER DATABASE [TestMS] SET CURSOR_CLOSE_ON_COMMIT OFF
GO
ALTER DATABASE [TestMS] SET CURSOR_DEFAULT GLOBAL
GO
ALTER DATABASE [TestMS] SET CONCAT_NULL_YIELDS_NULL OFF
GO
ALTER DATABASE [TestMS] SET NUMERIC_ROUNDABORT OFF
GO
ALTER DATABASE [TestMS] SET QUOTED_IDENTIFIER OFF
GO
ALTER DATABASE [TestMS] SET RECURSIVE_TRIGGERS OFF
GO
ALTER DATABASE [TestMS] SET DISABLE_BROKER
GO
ALTER DATABASE [TestMS] SET AUTO_UPDATE_STATISTICS_ASYNC OFF
GO
ALTER DATABASE [TestMS] SET DATE_CORRELATION_OPTIMIZATION OFF
GO
ALTER DATABASE [TestMS] SET PARAMETERIZATION SIMPLE
GO
ALTER DATABASE [TestMS] SET READ_COMMITTED_SNAPSHOT OFF
GO
ALTER DATABASE [TestMS] SET READ_WRITE
GO
ALTER DATABASE [TestMS] SET RECOVERY FULL
GO
ALTER DATABASE [TestMS] SET MULTI_USER
GO
ALTER DATABASE [TestMS] SET PAGE_VERIFY CHECKSUM
GO
ALTER DATABASE [TestMS] SET TARGET_RECOVERY_TIME = 0 SECONDS
```

```
GO  
ALTER DATABASE [TestMS] SET DELAYED_DURABILITY = DISABLED  
GO  
USE [TestMS]  
GO  
IF NOT EXISTS (SELECT name FROM sys.filegroups WHERE is_default=1 AND  
name = N'PRIMARY') ALTER DATABASE [TestMS] MODIFY FILEGROUP [PRIMARY]  
DEFAULT  
GO
```

Usunąć bazę danych może wyłącznie osoba posiadająca wymagane uprawnienia (np. administrator serwera bazodanowego) i tylko pod warunkiem, że w danym momencie nikt nie jest połączony z tą bazą. W przeciwnym razie próba wykonania instrukcji `DROP DATABASE` skończy się błędem:

```
DROP DATABASE TestMS;
```

```
-----  
Cannot drop database "TestMS" because it is currently in use.
```

W przypadku serwera SQL zamknąć wszystkie sesje klienckie można poprzez wymuszenie przełączenia bazy danych w tryb pojedynczego użytkownika:

```
USE master;  
ALTER DATABASE TestMS  
SET SINGLE_USER  
WITH ROLLBACK IMMEDIATE;  
DROP DATABASE TestMS;
```

```
-----  
Nonqualified transactions are being rolled back. Estimated rollback  
completion: 100%.
```

Tworzenie i usuwanie tabel

Tabele składają się z kolumn określonego typu, przy czym nazwa kolumny musi być niepowtarzalna w skali tabeli. Utworzona instrukcją `CREATE TABLE` tabela są puste (nie zawierają żadnych wierszy).

Tworząc tabele, należy pamiętać, że:

1. Każda tabela musi mieć jednoznaczną (niepowtarzalną w skali schematu) nazwę i właściciela.
2. Nazwy tabel i kolumn (tak samo jak nazwy obiektów innego typu) muszą być poprawnymi, zgodnymi ze standardem SQL identyfikatorami.
3. Każda kolumna musi być określonego typu — dobierając właściwy typ kolumny do przechowywanych w niej danych, poprawimy wydajność zapytań, zmniejszymy wielkość tabeli oraz w pewnym stopniu zabezpieczymy się przed zapisywaniem w kolumnie błędnych danych.
4. W przypadku kolumn typów znakowych należy podawać maksymalną długość przechowywanych w nich ciągów znaków.

Wykonując poniższe instrukcje, utworzymy w bazie `Test` tabelę z czterema kolumnami różnych typów:

```
USE Test;  
CREATE TABLE T1 (  
    ID INT,  
    Nazwa VARCHAR(50),  
    Cena MONEY,  
    DataZakupu DATE);
```

Możemy usunąć tabelę, wykonując instrukcję `DROP TABLE`, przy czym niektóre serwery bazodanowe automatycznie kasują zawartość usuwanej tabeli, a inne wymagają wcześniejszego samodzielnego skasowania jej zawartości:

```
DROP TABLE T1;
```

Schematy

Pojedyncza baza danych może zawierać kilkaset, a nawet kilka tysięcy tabel. Praca i wykonywanie czynności administracyjnych na tak rozbudowanych bazach danych byłaby bardzo trudne, gdyby nie możliwość podzielenia ich obiektów (w tym tabel) pomiędzy schematy.

	Schemat powinien zawierać tylko połączone ze sobą biznesowymi relacjami obiekty, np. tabele przechowujące informacje o produkcji towarów. Ponieważ przynależność do schematu jest sygnalizowana w nazwie obiektu, schematy
--	--



Wskazówka

ułatwiają użytkownikom pracę z bazą danych. A dzięki temu, że schemat też jest obiektem bazodanowym, administratorzy zamiast zarządzać uprawnieniami użytkowników na poziomie poszczególnych tabel, mogą to robić na poziomie całych schematów.

Możemy utworzyć schemat, wykonując w bieżącej bazie danych instrukcję CREATE SCHEMA:

```
CREATE SCHEMA Produkcja;
```

Żeby przypisać tabelę do istniejącego schematu, wystarczy poprzedzić jej nazwę nazwą tego schematu:

```
CREATE TABLE Produkcja.Towary (
    ID INT IDENTITY,
    Nazwa VARCHAR(50),
    Cena MONEY,
    DataZakupu DATE);
```

Zmiana struktury tabeli

Do zmiany struktury istniejącego obiektu służy instrukcja ALTER. Po czasowniku ALTER należy podać typ zmienianego obiektu i jego nazwę.



Wskazówka

Serwery bazodanowe od początku swojego istnienia umożliwiały zmianę struktur obiektów. Natomiast instrukcja ALTER została dodana do standardu języka SQL dopiero w roku 1992. W rezultacie jej składnia do dziś bardziej przypomina pierwotne, charakterystyczne dla danego serwera rozwiązania niż składnię zaproponowaną w standardzie SQL2.

Większość serwerów bazodanowych pozwala:

1. Dodać kolumnę do tabeli (dodawana kolumna będzie ostatnią kolumną tabeli):

```
ALTER TABLE Produkcja.Towary
ADD Kolor VARCHAR(10);
```

2. Usunąć kolumnę z tabeli:

```
ALTER TABLE Produkcja.Towary
DROP COLUMN Kolor
```

3. Dodać lub usunąć ograniczenie.

Ograniczenia

Dla tabel można zdefiniować ograniczenia — warunki, które określają, jakie dane można w nich zapisywać. **Ograniczenia są sprawdzane, zanim dane zostaną wstawione, zmodyfikowane lub usunięte, dzięki czemu serwery bazodanowe nie muszą wycofywać transakcji naruszających spójność danych**^[3].

Ograniczenia mogą być definiowane podczas tworzenia i modyfikowania tabel. W obu przypadkach definicja ograniczenia może być umieszczona bezpośrednio po kolumnie, z którą ma ono być powiązane, lub po wymienieniu wszystkich kolumn, a więc pod koniec instrukcji CREATE lub ALTER.

NOT NULL

NULL jako symbol reprezentujący brakujące, nieistotne lub nieznane wartości nie powinien być wstawiany do większości kolumn. Na przykład służąca do identyfikacji wierszy tabeli kolumna klucza podstawowego nie powinna zawierać wartości NULL — ponieważ niemożliwe jest odróżnienie jednej wartości NULL od drugiej, nieokreślony klucz podstawowy uniemożliwiałby jednoznaczne identyfikowanie wierszy.

Również kolumny przechowujące podstawowe atrybuty obiektu nie powinny zawierać wartości NULL. W innym przypadku informacje zapisane w bazie danych będą niekompletne i mało przydatne, np. niektóre dane towarów bez nazw i cen nie pozwolą przygotować oferty czy zestawienia rocznej sprzedaży.

Możemy zabronić wstawiania do kolumny wartości NULL, dopisując NOT NULL po jej nazwie albo:

1. Tworząc kolumnę, w której od początku nie będzie można zapisywać wartości NULL:

```
ALTER TABLE Produkcja.Towary
```

```
ADD Rabat MONEY NOT NULL;
```

2. Zmieniając definicję istniejącej już kolumny:

```
ALTER TABLE Produkcja.Towary
```

```
ALTER COLUMN Nazwa VARCHAR(50) NOT NULL;
```

3. Natomiast żeby jawnie zezwolić na wstawianie wartości NULL do kolumny, należy po jej nazwie dopisać słowo kluczowe NULL:

```
ALTER TABLE Produkcja.Towary  
ALTER COLUMN DataZakupu DATE NULL;
```

 Wskazówka	Duża liczba kolumn zdefiniowanych jako zezwalające na przechowywanie wartości NULL może świadczyć o nie najlepszym (niewystarczająco znormalizowanym) projekcie bazy danych.
---	--

Klucz podstawowy

Tabela może mieć tylko jeden klucz podstawowy (PK — skrót od ang. *Primary Key*). Ponieważ jest on używany do identyfikowania wierszy tej tabeli, **wartości klucza podstawowego nie mogą być nieokreślone i muszą być niepowtarzalne**.

Żeby dodać ograniczenie klucza podstawowego do istniejącej tabeli, należy użyć instrukcji ALTER TABLE:

```
ALTER TABLE Produkcja.Towary  
ADD CONSTRAINT PK_Towary PRIMARY KEY (ID);
```

Natomiast aby określić klucz podstawowy tworzonej tabeli, wystarczy po nazwie kolumny dopisać PRIMARY KEY^[4]:

```
CREATE TABLE Produkcja.Części (  
ID INT PRIMARY KEY,  
Nazwa VARCHAR(30) NOT NULL);
```

Generowanie wartości kluczy podstawowych

Zdecydowana większość serwerów bazodanowych sprawdza niepowtarzalność wartości kolumn poprzez automatycznie tworzone indeksy. W przypadku serwera SQL kolumny (lub kolumna) klucza podstawowego są używane jako klucze indeksu zgrupowanego. W praktyce oznacza to, że z reguły wartości klucza podstawowego powinny być generowanymi przez serwer liczbami całkowitymi^[5].

Poszczególne serwery bazodanowe mają własne mechanizmy generowania wartości — serwer SQL używa do tego albo wewnętrznych liczników ustawianych słowem kluczowym IDENTITY,

albo sekwencji.

W pierwszej kolejności przyjrzymy się właściwości IDENTITY. Ponieważ niemożliwe jest nadanie właściwości IDENTITY istniejącej kolumnie:

1. Z tabeli Produkcja.Towary usuniemy i ponownie dodamy kolumnę ID^[6]:

```
ALTER TABLE Produkcja.Towary
DROP CONSTRAINT PK_Towary;
ALTER TABLE Produkcja.Towary
DROP COLUMN ID;
ALTER TABLE Produkcja.Towary
ADD ID INT IDENTITY (1,1) CONSTRAINT PK_Towary PRIMARY KEY;
```

2. Żeby pokazać składnię instrukcji CREATE, usuniemy i ponownie utworzymy — tym razem z poprawnym kluczem podstawowym — tabelę Produkcja.Części:

```
DROP TABLE Produkcja.Części;
CREATE TABLE Produkcja.Części (
    ID INT IDENTITY (100,1) CONSTRAINT PK_Części PRIMARY KEY,
    Nazwa VARCHAR(30) NOT NULL);
```

 Wskazówka	Definiując ograniczenia, powinno się samodzielnie określać ich nazwy. Ułatwia to nie tylko ewentualne usuwanie lub wyłączanie tych ograniczeń, lecz także analizowanie komunikatów błędów zawierających ich nazwy.
--	--

Inny sposób automatycznego generowania wartości kluczy podstawowych polega na utworzeniu sekwencji (niezależnego od tabeli obiektu bazy danych):

1. Zaczniemy od utworzenia dwóch bardzo prostych tabel:

```
CREATE TABLE dbo.T1 (C1 INT NOT NULL);
CREATE TABLE dbo.T2 (C2 INT NOT NULL);
```

2. Następnie utworzymy sekwencję o nazwie S1, która będzie generowała kolejne liczby naturalne, zaczynając od 1:

```
CREATE SEQUENCE dbo.S1 AS INT
START WITH 1 INCREMENT BY 1;
```

3. Od teraz wstawiając wiersze do tabel, będziemy mogli posłużyć się utworzoną wcześniej sekwencją:

```
INSERT INTO dbo.T1 (C1)  
VALUES (NEXT VALUE FOR dbo.S1);  
INSERT INTO dbo.T2 (C2)  
VALUES (NEXT VALUE FOR dbo.S1);
```

4. Po odczytaniu danych z naszych tabel testowych okaże się, że zostały do nich wstawione kolejne wygenerowane przez sekwencję liczby naturalne:

```
SELECT * FROM dbo.T1;  
SELECT * FROM dbo.T2;
```

```
1  
2
```

Kompozytowe klucze podstawowe

Decydując się na używanie naturalnych (a nie generowanych przez serwer) wartości klucza podstawowego, musimy zagwarantować ich niepowtarzalność nie tylko dla aktualnych, lecz także dla przyszłych danych. Tak więc kluczem podstawowym nie powinna być kolumna Nazwisko, bo w przyszłości możemy chcieć zapisać w tabeli dane kilku osób o tym samym nazwisku.

W takim przypadku możemy nałożyć ograniczenie klucza podstawowego na kilka kolumn. Na przykład jeżeli przyjmiemy, że kombinacja nazwiska i imienia zawsze będzie niepowtarzalna, możemy nałożyć klucz na kolumny Nazwisko i Imię:

```
CREATE TABLE dbo.Znajomi(  
Nazwisko VARCHAR(50),  
Imię VARCHAR(20),  
DataUr DATE,  
CONSTRAINT PK_Znajomi PRIMARY KEY (Nazwisko, Imię))
```

Niepowtarzalność

W przeciwieństwie do klucza podstawowego, który może być tylko jeden, ograniczenie niepowtarzalności (ang. *Unique*) możemy

zdefiniować dla dowolnej liczby kolumn tabel^[7]. Większość serwerów bazodanowych automatycznie indeksuje kolumny, w których nie można zapisywać duplikatów danych.

Ograniczenie niepowtarzalności, w odróżnieniu od ograniczenia klucza podstawowego, nie uniemożliwia zapisywania wartości NULL. Serwery jednak w różny sposób traktują ją w ograniczeniu niepowtarzalności, np. serwer SQL 2011 uznaje wartości NULL za równe i w konsekwencji w kolumnie z nałożonym ograniczeniem niepowtarzalności wartość NULL też nie może się powtórzyć.

Żeby uniemożliwić powtarzanie się nazw towarów, nałożymy na kolumnę Nazwa odpowiednie ograniczenie:

```
ALTER TABLE Produkcja.Towary  
ADD CONSTRAINT U_NazwaTowaru UNIQUE (Nazwa);
```

natomiast do tabeli Produkcja.Części dodamy kolumnę, w której zapisywane będą niepowtarzalne i wymagane kody części:

```
ALTER TABLE Produkcja.Części  
ADD Kod CHAR(5) NOT NULL CONSTRAINT U_CzęściKod UNIQUE
```

Wartość domyślna

Każdej kolumnie tabeli można nadać jedno ograniczenie wartości domyślnej (ang. *Default*). Tego typu ograniczenia są sprawdzane tylko podczas wstawiania wierszy. Jeżeli użytkownik nie wstawi danych do kolumny z ograniczeniem wartości domyślnej, serwer bazodanowy zrobi to za niego.

Nałożyć ograniczenie wartości domyślnej na istniejącą kolumnę możemy następująco:

```
ALTER TABLE Produkcja.Towary  
ADD CONSTRAINT DF_TowaryData DEFAULT GETDATE() FOR DataZakupu;
```

Możliwe jest też utworzenie kolumny i jednoczesne określenie dla niej wartości domyślnej:

```
ALTER TABLE Produkcja.Części  
ADD Jakość SMALLINT CONSTRAINT DF_CzęściJakość DEFAULT 5
```

Warunek logiczny

Najbardziej uniwersalnym zawężeniem jest warunek logiczny (ang.

Check), który musi być prawdziwy, żeby operacja wstawiania, modyfikowania czy usuwania danych zakończyła się powodzeniem. Dla każdej kolumny tabeli można zdefiniować wiele warunków, można też tworzyć złożone warunki za pomocą operatorów algebry Boole'a: NOT, AND oraz OR.

 Wskazówka	W ramach warunku logicznego niemożliwe jest odwoływanie się do tabel innych niż ta, dla której warunek został zdefiniowany, oraz używanie podzapytań.
--	---

Poniższy przykład pokazuje, jak za pomocą warunku logicznego ograniczyć listę możliwych wartości kolumny do liczb z zakresu od 0 do 5:

```
ALTER TABLE Produkcja.Części  
ADD CONSTRAINT CK_CzęściJakość CHECK (Jakość BETWEEN 0 AND 5);
```

Warunek logiczny może odwoływać się do wielu kolumn tabeli. Możemy np. dodać do tabeli Produkcja.Towary kolumnę DataSprzedaży i ograniczyć zakres możliwych do zapisania w niej znaczników daty do przedziału pomiędzy datą zakupu a bieżącą^[8]:

```
ALTER TABLE Produkcja.Towary  
ADD DataSprzedaży DATE;  
ALTER TABLE Produkcja.Towary  
ADD CONSTRAINT CK_TowaryData CHECK (DataSprzedaży BETWEEN DataZakupu AND  
GETDATE());
```

Klucz obcy

Ograniczenie klucza obcego (ang. *Foreign Key*) pozwala na automatyczne sprawdzanie spójności danych przechowywanych w powiązanych ze sobą tabelach. Jeżeli tabele powiązane są związkiem typu „jeden do wielu” (np. jeden towar składa się z wielu części, ale ta sama część może być użyta tylko w jednym produkcie), do tabeli podrzędnej (w tym przypadku do tabeli Produkcja.Części) należy dodać kolumnę, w której zostaną zapisane identyfikatory towarów. Żeby dane były spójne, wartości klucza obcego:

1. Muszą odpowiadać jednej z wartości powiązanego z nim klucza podstawowego (przy każdej części musi być zapisany

- identyfikator istniejącego towaru, w którym ta część została użyta)
2. lub być nieokreślone (oznacza to, że dana część nie została użyta w żadnym z towarów).

Niektóre serwery bazodanowe, w tym serwer SQL, pozwalają pominąć nazwę kolumny klucza podstawowego tabeli nadzędnej. Jeżeli dodatkowo nie podamy nazwy tworzonego ograniczenia, dodanie kolumny klucza podstawowego może wyglądać następująco:

```
ALTER TABLE Produkcja.Części  
ADD IDTowaru INT REFERENCES Produkcja.Towary;
```

 Wskazówka	Zadaniem klucza obcego nie jest powiązanie ze sobą identyfikatorów, w tym przypadku numerów towarów zapisanych w dwóch tabelach. Klucz obcy ma umożliwić odczytanie kompletnych danych o obiektach z powiązanych tabel (nazwy, ceny czy daty produkcji towaru) na podstawie pary wartości klucz podstawowy - klucz obcy. Powyższa, skrócona składnia gwarantuje, że do połączenia zostanie użyty cały klucz podstawowy tabeli nadzędnej.
---	--

Możemy również jawnie wskazać kolumnę klucza podstawowego tabeli nadzędnej i nazwać ograniczenie klucza obcego:

```
ALTER TABLE Produkcja.Części  
DROP CONSTRAINT FK_Części_IDTowaru_117F9D94[9];  
ALTER TABLE Produkcja.Części  
DROP COLUMN IDTowaru;  
-----  
Msg 5074, Level 16, State 1, Line 210  
The object 'FK_Części_IDTowaru_20C1E124' is dependent on column  
'IDTowaru'.  
Msg 4922, Level 16, State 9, Line 210  
ALTER TABLE DROP COLUMN IDTowaru failed because one or more objects  
access this column.  
ALTER TABLE Produkcja.Części  
DROP CONSTRAINT FK_Części_IDTowaru_20C1E124  
ALTER TABLE Produkcja.Części  
DROP COLUMN IDTowaru;  
GO  
ALTER TABLE Produkcja.Części
```

```
ADD IDTowaru INT CONSTRAINT FK_Towary REFERENCES Produkcja.Towary (ID);
```

Klucz obcy powiązany z kluczem podstawowym tej samej tabeli

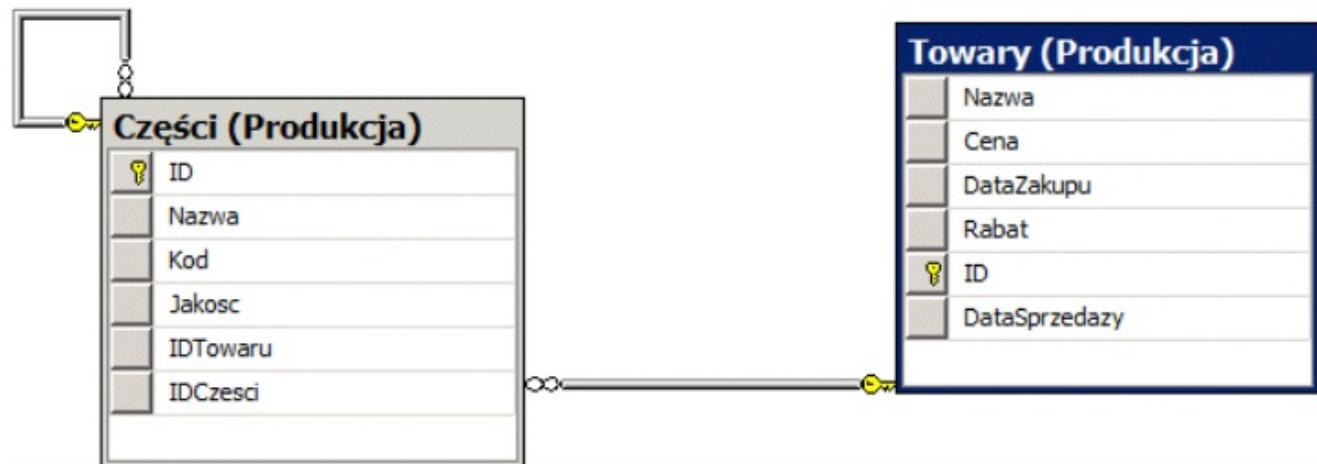
Klucz obcy może być powiązany z dowolnym kluczem podstawowym, również z kluczem podstawowym tej samej tabeli. Otrzymamy w ten sposób związek typu „jeden do wielu”, ale w ramach tej samej tabeli. Takie powiązania pozwalają odzwierciedlać rekurencyjne i hierarchiczne zależności pomiędzy danymi, np. hierarchię pracowników firmy.

Jeżeli przyjmiemy, że poszczególna część może się składać z innych części (tak jak silnik składa się z cylindrów, gaźnika i tak dalej), zapisanie tych informacji będzie możliwe dzięki dodaniu do tabeli Produkcja.Części kolumny klucza obcego powiązanego z kluczem podstawowym tej tabeli:

```
ALTER TABLE Produkcja.Części
```

```
ADD IDCzęści INT CONSTRAINT FK_Części REFERENCES Produkcja.Części;
```

Utworzona w ten sposób tabela oraz istniejące pomiędzy nimi powiązania pokazuje rysunek 12.1.



Rysunek 12.1. Diagram E/R schematu Produkcja utworzonej w tym rozdziale bazy danych Test

Kaskadowe usuwanie i aktualizacja powiązanych danych

Skoro wartości kluczy obcych muszą odpowiadać jednej z wartości powiązanych z nimi kluczy podstawowych, to zmiana lub usunięcie

klucza podstawowego wymaga zastosowania się do pewnych wskazówek. I tak:

1. Aktualizacja klucza podstawowego wymaga zaktualizowania wartości wszystkich powiązanych z nim kluczy obcych. **Ponieważ klucze podstawowe nie powinny w ogóle być aktualizowane** (ich zmiana jest bardzo czasochłonna i prowadzi do fragmentacji danych), kaskadową aktualizację należy definiować tylko w przypadku stosowania naturalnych kluczy podstawowych.
2. Usunięcie wiersza z tabeli nadrzędnej (a więc usunięcie wartości klucza podstawowego) wymaga usunięcia lub zaktualizowania wartości wszystkich powiązanych z nim kluczy obcych:
 - a. Kaskadowe usuwanie może doprowadzić do automatycznego skasowania niewiadomej liczby wierszy z różnych tabel bazy danych (a w skrajnym przypadku do skasowania całej zawartości bazy) po usunięciu z niej jednego wiersza. Dlatego kaskadowe usuwanie należy zdefiniować wyłącznie dla tabel łącznikowych (tabel pozwalających połączyć inne tabele związkiem typu „wiele do wielu”).
 - b. Automatyczna aktualizacja wartości kluczy obcych jest znacznie bezpieczniejsza i może być zdefiniowana dla wszystkich pozostałych tabel. Jeżeli kolumna klucza obcego zezwala na zapisywanie wartości NULL, wartości usuniętego klucza podstawowego mogą być zastąpione wartością NULL, w przeciwnym razie należy zastąpić je specjalnie w tym celu zdefiniowaną wartością domyślną.

Kaskadowe usuwanie i aktualizację definiuje się w klauzulach ON UPDATE i ON DELETE:

1. Domyślna wartość NO ACTION powoduje, że dane w powiązanych tabelach nie będą automatycznie modyfikowane.
2. Wartość CASCADE oznacza, że modyfikacja ma być automatycznie powtórzona we wszystkich powiązanych tabelach.
3. Wartość SET NULL oznacza, że zmodyfikowane wartości klucza podstawowego mają być zastąpione wartością NULL w powiązanych kolumnach klucza obcego.

4. Wartość SET DEFAULT oznacza, że zmodyfikowane wartości klucza podstawowego mają być zastąpione domyślną dla powiązanych kolumn klucza obcego wartością.

Poniższe instrukcje włączają kaskadową aktualizację identyfikatorów towarów dla tabeli Produkcja.Części — podczas aktualizacji w tabeli nadrzędnej zostaną one automatycznie zaktualizowane w kolumnie klucza obcego tabeli podrzędnej, a podczas usuwania towaru jego identyfikator zostanie zastąpiony w tabeli podrzędnej wartością NULL:

```
ALTER TABLE Produkcja.Części
DROP CONSTRAINT FK_Towary;
GO
ALTER TABLE Produkcja.Części
ADD CONSTRAINT FK_Towary FOREIGN KEY (IDCzesci)
REFERENCES Produkcja.Towary
ON UPDATE CASCADE
ON DELETE SET NULL
GO
```

Ograniczenia a wydajność instrukcji modyfikujących i odczytujących dane

Ograniczenia są najwydajniejszym mechanizmem zapewniania przez serwery bazodanowe spójności danych. W przeciwnieństwie do wyzwalaczy (ang. *Triggers*) są sprawdzane przed wykonaniem instrukcji użytkownika, a więc niespójne dane w ogóle nie pojawiają się w tabelach i nie jest potrzebne ewentualne wycofywanie transakcji i przywracanie poprzedniej wersji danych.

Ponadto są one dla serwerów bazodanowych cennym źródłem dodatkowych informacji o przechowywanych w tabelach danych^[10]. **Zapisane w ograniczeniach reguły logiki biznesowej pozwalają znaleźć lepsze (mniej kosztowne) sposoby na wykonanie zapytań.**

Żeby się o tym przekonać:

1. Odczytamy z tabeli [SalesLT].[SalesOrderDetail] wszystkie informacje o zamówieniach z ujemną liczbą produktów. Serwer bazodanowy na potrzeby wykonania zapytania odczytał całą

tabelę, a ponieważ nie było zamówień, w ramach których sprzedano ujemną liczbę produktów, odfiltrował wszystkie wiersze (rysunek 12.2).

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the AdventureWorksLT2012 database is selected. In the center pane, a query window titled 'r12.sql - (local).AdventureWorksLT2012 (MS\Szczepan (51))' contains the following T-SQL code:

```
USE AdventureWorksLT2012
GO

SELECT *
FROM [SalesLT].[SalesOrderDetail]
WHERE [OrderQty] < 0;
```

Below the code, the 'Execution plan' tab is selected, showing a plan with four steps: a 'Clustered Index Scan' on the [SalesOrderDetail].[PK_SalesOrderDetail] clustered index, followed by a 'Compute Scalar' step, another 'Compute Scalar' step, and finally a 'Filter' step where the condition [OrderQty] < 0 is applied. The total cost of the plan is 95. The status bar at the bottom indicates 'Query executed successfully'.

Rysunek 12.2. Nie dysponując dodatkowymi metainformacjami, serwer bazodanowy wykonał zapytanie poprzez odczytanie całej tabeli (w tym przypadku wymagało to odczytania 31 465 wierszy), a następnie przefiltrował i usunął wszystkie odczytane dane

2. Jeżeli jednak dodamy ograniczenie niepozwalające na zapisywanie zamówień z ujemną liczbą produktów, to poinformujemy przy okazji serwer bazodanowy, że takich zamówień w tabeli [SalesLT].[SalesOrderDetail] na pewno nie ma:

```
ALTER TABLE [SalesLT].[SalesOrderDetail]
ADD CONSTRAINT CK_OrderQty CHECK (OrderQty >= 0);
```

3. Teraz to samo zapytanie może być wykonane przez serwer bazodanowy wielokrotnie szybciej (rysunek 12.3).

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the schema 'SalesLT' is expanded, showing tables like SalesOrderDetail, Product, and ProductCategory. A query window titled 'r12.sql - (local).AdventureWorksLT2012 (MS\Szczepan (51)) - Microsoft SQL Server Management Studio' contains the following SQL code:

```

SELECT *
FROM [SalesLT].[SalesOrderDetail]
WHERE [OrderQty] < 0;
GO

```

The results pane shows the query was executed successfully with a cost of 0. The execution plan pane indicates a 'Constant Scan' with a cost of 100%. The status bar at the bottom right shows '(local) (12.0 RTM) MS\Szczepan (51) AdventureWorksLT2012 | 00:00:00 | 0 rows'.

Rysunek 12.3. Bazując na posiadanych metainformacjach, serwer bazodanowy do wykonania tego samego zapytania nie musiał w ogóle odczytywać danych z tabeli SalesOrderDetail, wystarczyło, że sprawdził nałożone na kolumnę OrderQty ograniczenie

Podsumowanie

- Do tworzenia obiektów służy instrukcja CREATE. Po słowie kluczowym CREATE należy podać typ tworzonego obiektu, jego nazwę i definicję.
- Możemy zmienić strukturę istniejących obiektów instrukcją ALTER. Jej składnia w większości przypadków jest identyczna ze składnią instrukcji CREATE.
- Możemy usunąć obiekty instrukcją DROP. Po słowie kluczowym DROP należy podać typ usuwanego obiektu i jego nazwę.
- Fizyczna struktura, w tym opcje, baz danych oraz model ich bezpieczeństwa zależą wyłącznie od serwera bazodanowego.
- Schematy pozwalają podzielić złożone bazy danych na mniejsze, łatwiejsze do zarządzania i prostsze w używaniu moduły.
- Tworząc tabele, należy określić ich nazwę (włącznie z nazwą schematu) i zdefiniować kolejne kolumny. Każda kolumna musi mieć określoną nazwę i typ.
- Ograniczenia pozwalają deklaratywnie wymusić na użytkownikach poprawność zapisywanych w bazie informacji.
- Ograniczenia są również niezastąpionym źródłem metainformacji i pozwalają serwerom bazodanowym efektywniej wykonywać instrukcje użytkowników.

Zadania

1. Na podstawie poniższych informacji utwórz tabelę i nałoż na nią wymagane ograniczenia:

Każda osoba musi podać imię i nazwisko oraz adres e-mail i numer telefonu. Większość osób mieszka w Katowicach. Wiek i płeć, tak jak nazwa miasta, są informacjami opcjonalnymi. Osoby muszą się rejestrować, podając niepowtarzalny 5-znakowy kod promocyjny.

2. Zaimplementuj opracowany w rozdziale 1. projekt tabel, w których będzie można zapisywać informacje o książkach i ich autorach, przy czym jeden autor będzie mógł napisać wiele książek, a książka będzie mogła mieć kilku autorów:

```
Autorzy {IDAutora, Imię, Nazwisko}  
Książki (IDKsiążki, Tytuł, DataWydania}  
AutorzyKsiążki {IDAutora, IDKsiążki}
```

3. Kolega, któremu zlecono taką zmianę struktury bazy danych, aby możliwe było zapisywanie w niej informacji o modelach towarów, zaproponował poniższe rozwiązanie:

```
ALTER TABLE Produkcja.Towary  
ADD Model VARCHAR(5) CONSTRAINT CK_TowaryModel CHECK (Model IN  
('B1', 'A3', 'X54'));
```

Zakładając, że nazwa modelu musi być sprawdzana i odpowiadać jednej z zatwierdzonych nazw, jak wytłumaczysz koledze, że jego pomysł nie jest najlepszy? Znajdź też właściwe rozwiązanie problemu.

[1] Górnny limit uruchomionych na serwerze baz danych zależy od typu i wersji serwera. W niektórych przypadkach (np. w programie Microsoft Access) możliwe jest jednoczesne uruchomienie na serwerze tylko jednej bazy danych, w innych (np. w serwerze SQL 2011) można jednocześnie pracować z kilkudziesięcioma tysiącami baz danych.

[2] Opisanie trybów odtwarzania baz danych wykracza poza zakres książki poświęconej standardowi języka SQL. Warto jednak wiedzieć, że w trybie **FULL** dziennik transakcyjny nie jest automatycznie kasowany i jedynym sposobem kontrolowania wielkości pliku dziennika jest regularne wykonywanie jego kopii zapasowych.

[3] Wycofanie transakcji jest zawsze znacznie bardziej kosztowne od jej zatwierdzenia.

[4] W tym przypadku niepodana nazwa ograniczenia zostanie automatycznie wygenerowana przez serwer bazodanowy.

[5] Analiza sposobu przechowywania, odczytywania i modyfikowania danych w indeksach zgrupowanych wykracza poza zakres tej książki.

[6] Przed usunięciem kolumny należy usunąć wszystkie zdefiniowane dla niej ograniczenia.

[7] Można też utworzyć jedno ograniczenie niepowtarzalności dla kilku kolumn, tak jak to miało miejsce w przypadku kompozytowych kluczy podstawowych.

[8] Bieżącą datę zwraca funkcja GETDATE().

[9] Automatycznie wygenerowana nazwa ograniczenia może być za każdym razem inną. Żeby poznać właściwą nazwę, wystarczy spróbować usunąć kolumnę IDTowaru — komunikat błędu będzie zawierał nazwę ograniczenia uniemożliwiającego jej usunięcie.

[10] Informacje opisujące dane nazywa się metadanymi.

Rozdział 13. Widoki i indeksy

- Czym są widoki?
- Jak utworzyć, zmodyfikować i usunąć widok?
- Jak odczytywać dane poprzez widoki?
- Za pośrednictwem których widoków można modyfikować dane?
- Co możemy zyskać, odizolowując użytkowników od tabel?
- Do czego służą indeksy?
- Które kolumny warto indeksować?
- Jak uporządkować indeksy?

Widoki

Widoki nie przechowują kopii zapisanych w tabelach danych. Widok (ang. *View*) to jedynie zapisane pod podaną nazwą i w określonym schemacie bazy zapytanie (instrukcja `SELECT`). Serwery bazodanowe przechowują definicje widoków i związane z nimi metadane (m.in. uprawnienia oraz informacje o zależnościach pomiędzy widokami i innymi obiektami bazy danych), ale nie kopie zwracanych przez widoki danych.

 Wskazówka	Ponieważ do widoków, tak samo jak do tabel, odwołuje się instrukcja <code>SELECT</code> czy <code>INSERT</code> , nazywa się je tabelami wirtualnymi, a właściwe tabele dla odróżnienia od widoków nazywa się tabelami bazowymi.
--	--

Tworzenie i usuwanie widoków

Żeby zapisać zapytanie w postaci widoku, należy poprzedzić je instrukcją `CREATE VIEW nazwa AS`. Na przykład aby przekształcić poniższe zapytanie w widok:

```
SELECT [ProductID], [ProductNumber], [ListPrice]
FROM [SalesLT].[Product]
WHERE [SellEndDate] IS NULL;
```

wystarczy poprzedzić je instrukcją `CREATE VIEW` i podać nazwę tworzonego widoku:

```
CREATE VIEW [SalesLT].[CurrentProduct]
AS
SELECT [ProductID], [ProductNumber], [ListPrice]
FROM [SalesLT].[Product]
WHERE [SellEndDate] IS NULL;
```

Po wykonaniu instrukcji w bazie *AdventureWorksLT2012* zostanie utworzony nowy widok. Żeby się o tym przekonać, odczytamy zwracane przez niego dane:

```
SELECT TOP 5 *
FROM [SalesLT].[CurrentProduct]
ORDER BY [ListPrice] DESC;
```

ProductID	ProductNumber	ListPrice
792	BK-R89R-58	2443,35
793	BK-R89B-44	2443,35
794	BK-R89B-48	2443,35
795	BK-R89B-52	2443,35
796	BK-R89B-58	2443,35



W tworzących widoki zapytaniach nie powinno się używać symbolu wieloznacznego *. Nie tylko niekorzystnie wpłynie to na wydajność wszystkich odwołujących się do widoku zapytań, lecz także może być powodem niespodziewanych błędów. Jeżeli do tabeli bazowej zostanie dodana kolumna, widok, który wcześniej zwracał trzy kolumny, nagle zwróci cztery. Natomiast w przypadku gdy z tabeli bazowej zostanie usunięta kolumna, próba odwołania się do powiązanych z nią widoków skończy się błędem.

Tworząca widok instrukcja SELECT musi zwracać dane tabelaryczne. Oznacza to, że:

1. Nawet jeżeli dany serwer bazodanowy pozwala zwracać dane w innej postaci (np. dokumentów XML), w definicji widoku nie można skorzystać z tej funkcjonalności.
2. Każda kolumna musi mieć niepowtarzalną nazwę, czyli jeżeli w klauzuli SELECT znajdują się jakieś wyrażenia, trzeba nadać im aliasy.
3. Niemożliwe jest użycie w definicji widoku instrukcji SELECT ... INTO (w przeciwnym razie kolejne odwołanie się do widoku spowodowałoby błąd, bo tabela z kopią danych już by istniała, a

nazwy tabel muszą być niepowtarzalne).

Klauzula ORDER BY

W widokach nie można stosować klauzuli ORDER BY. Wynika to z tego, że widoki, tak samo jak tabele, są reprezentacjami zbiorów, a więc kolejność zwracanych przez nie wierszy powinna być bez znaczenia. **Gdybyśmy posortowali widok, uniemożliwiliśmy serwerom bazodanowym wykonywanie zapytań zgodnie z regułami będącej podstawą ich działania teorii zbiorów.**

Wyjątkiem od tej reguły jest ograniczanie liczby wierszy widoków za pomocą klauzuli TOP. Żeby można było jej poprawnie używać w definicjach widoków, konieczne jest posortowanie zwracanych przez zapytanie wierszy. Producenci serwerów bazodanowych w różny sposób odnieśli się do tego problemu. Na przykład serwer SQL nie gwarantuje, że odczytane poprzez widok dane będą posortowane. Innymi słowy, klauzula ORDER BY jest uwzględniana tylko na potrzeby klauzuli TOP, po jej wykonaniu kolejność wierszy przestaje mieć znaczenie.

Ilustruje to poniższy przykład. Ponieważ wiele osób nie zdawało sobie sprawy, jak niekorzystny wpływ na wydajność zapytania ma przedwczesne wymuszanie sortowania, a użytkownicy spodziewali się otrzymać posortowane wyniki, obchodzili oni zakaz używania klauzuli ORDER BY w definicjach widoków poprzez dopisywanie klauzuli TOP 100 PERCENT. W rezultacie widok zwrócił wszystkie (100%) wiersze, ale czy były one posortowane?

Żeby się o tym przekonać, utwórzmy poniższy widok:

```
CREATE VIEW dbo.OrderedProducts
AS
SELECT TOP 100 PERCENT C.[Name] AS Cat, P.[Name] AS Prod
FROM [SalesLT].[ProductCategory] AS C
JOIN [SalesLT].[Product] AS P
ON C.ProductCategoryID = P.ProductCategoryID
ORDER BY C.[Name];
```

i odczytajmy go:

```
SELECT *
```

```
FROM OrderedProducts;
```

```
-----  
Cat           Prod  
Handlebars    HL Road Handlebars  
Bottom Brackets LL Bottom Bracket  
Bottom Brackets ML Bottom Bracket  
Bottom Brackets HL Bottom Bracket  
Brakes        Front Brakes  
Brakes        Rear Brakes  
Chains         Chain  
Cranksets     LL Crankset  
  
...
```

Okazuje się, że otrzymany wynik nie jest posortowany alfabetycznie według nazw kategorii. **Jeżeli chcemy uzyskać posortowane dane, musimy, odczytując widok, użyć klauzuli ORDER BY:**

```
SELECT *  
FROM OrderedProducts  
ORDER BY Cat;
```

```
-----  
Cat           Prod  
Bib-Shorts   Men's Bib-Shorts, S  
Bib-Shorts   Men's Bib-Shorts, M  
Bib-Shorts   Men's Bib-Shorts, L  
Bike Racks   Hitch Rack - 4-Bike  
Bike Stands  All-Purpose Bike Stand  
Bottles and Cages Water Bottle - 30 oz.  
  
...
```

Modyfikowanie widoków

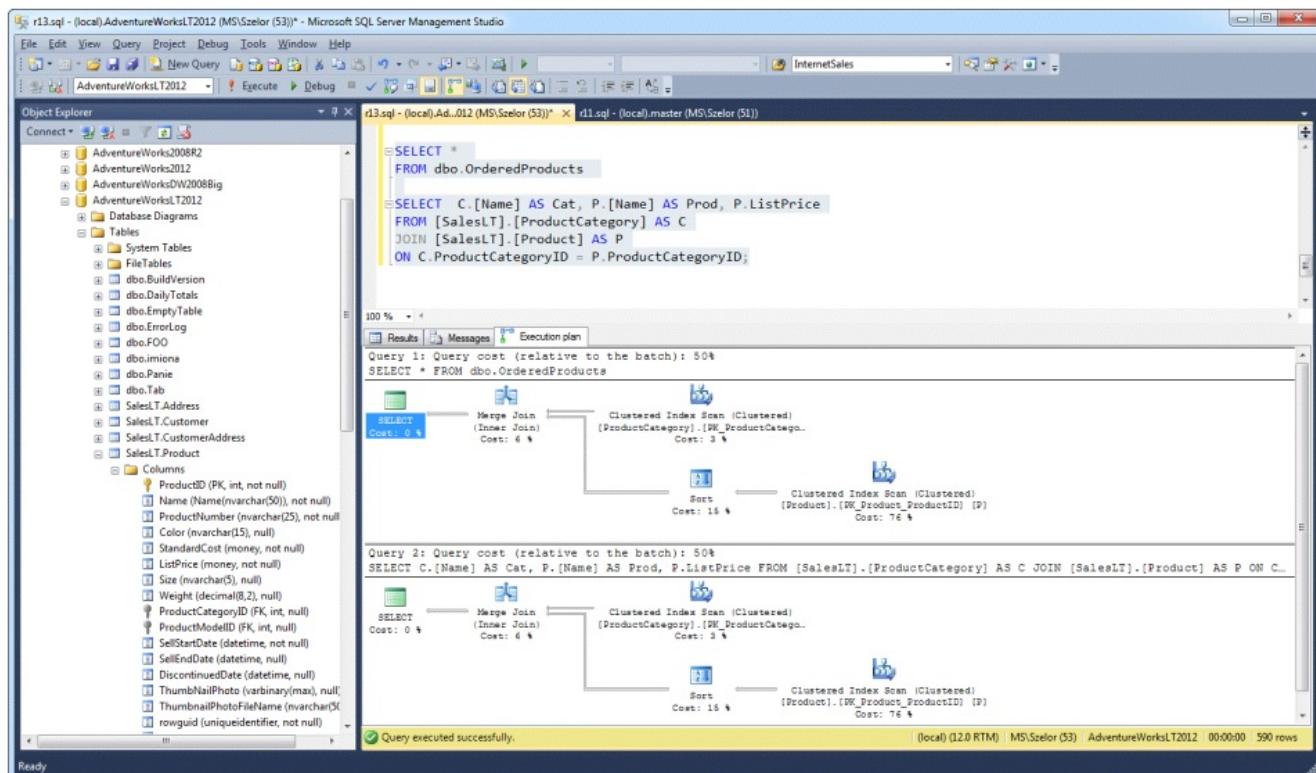
Istniejący widok można zmodyfikować instrukcją `ALTER VIEW`. Modyfikując widok, należy raz jeszcze podać tworzącą go instrukcję `SELECT`. **Główna różnica pomiędzy wprowadzeniem zmian w widoku a usunięciem go i utworzeniem nowego o tej samej nazwie polega na tym, że podczas modyfikowania widoku zachowywane są nadane mu wcześniej uprawnienia.**

Wykonując poniższą instrukcję, usuniemy z widoku OrderedProducts niepotrzebne klauzule ORDER BY i TOP oraz dodamy do niego ceny produktów:

```
ALTER VIEW dbo.OrderedProducts
AS
SELECT C.[Name] AS Cat, P.[Name] AS Prod, P.ListPrice
FROM [SalesLT].[ProductCategory] AS C
JOIN [SalesLT].[Product] AS P
ON C.ProductCategoryID = P.ProductCategoryID;
```

Korzystanie z widoków

Instrukcje, które odwołują się do widoków, są automatycznie rozszerzane (ang. *Expand*) o ich definicje. Tak więc plany wykonania instrukcji bezpośrednio odwołującej się do bazowych tabel widoku i instrukcji, która te same dane odczytuje poprzez widok, będą identyczne (rysunek 13.1).



Rysunek 13.1. Zapytanie odczytujące dane bezpośrednio z tabel bazowych i zapytanie odczytujące te same dane poprzez widok zostały wykonane przez serwer SQL w dokładnie ten sam sposób. Oznacza to, że stosowanie standardowych widoków nie ma żadnego wpływu na wydajność zapytań

Odczytywanie danych poprzez widoki

Dane poprzez widoki są odczytywane w taki sam sposób jak bezpośrednio z tabel. W rzeczywistości użytkownicy baz danych z reguły nawet nie zdają sobie sprawy z tego, że korzystają z widoków, a nie tabel. Z drugiej strony widoki upraszczają zapytania, ułatwiają zarządzanie uprawnieniami użytkowników do danych i umożliwiają zmianę definicji tabel bez konieczności jednoczesnego zmieniania aplikacji klienckich.

Opisywane w książce widoki nazywa się standardowymi — standardowy widok to instrukcja `SELECT` odczytująca dane z jednej bazy. Dwa pozostałe typy widoków: widoki zmaterializowane (widoki, dla których utworzony został indeks) oraz widoki rozproszone (widoki odczytujące dane z różnych, z reguły znajdujących się na osobnych serwerach, baz danych) mają ogromny wpływ na wydajność zapytań, jednak ich przedstawienie wykracza poza zakres tej książki.

Na szczególną uwagę zasługują te widoki standardowe, które odczytują dane z innych widoków, i widoki grupujące dane.

Zagnieżdżone widoki

Widok może odczytywać dane z innych widoków, nie tylko z tabel bazowych. Możliwe jest np. utworzenie poniższego widoku, który łączy tabelę `[SalesLT].[SalesOrderDetail]` z wcześniej utworzonym widokiem `[SalesLT].[CurrentProduct]`:

```
CREATE VIEW [SalesLT].[vOrders]
AS
SELECT P.ProductNumber, OD.LineTotal
FROM [SalesLT].[SalesOrderDetail] AS OD
JOIN [SalesLT].[CurrentProduct] AS P
ON P.ProductID = OD.ProductID;
```

Zagnieżdżając widoki, należy się liczyć z tym, że najprostsze zapytania mogą mieć skompilowane, trudne do analizowania plany wykonań i w rezultacie może się wydawać, że proste zapytanie będzie wykonywane wolniej, niż się spodziewaliśmy. I tak gdybyśmy nie włączyli wyświetlania planów wykonania, poniższe zapytanie prawdopodobnie nie zwróciłoby niczyjej uwagi (rysunek 13.2).

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, there is a tree view of database objects. In the center pane, a query window displays the following T-SQL code:

```

SELECT *
FROM [SalesLT].[vOrders];
GO

```

Below the code, the 'Execution plan' tab is selected, showing a detailed plan for the query. The plan consists of two main operations: a 'Clustered Index Scan (Clustered)' on the 'Product' table (cost: 71 %) and a 'Clustered Index Scan (Clustered)' on the 'SalesOrderDetail' table (cost: 4 %). These two scans are connected by a 'Hash Match (Inner Join)' operation (cost: 23 %), which then feeds into a 'Compute Scalar' operation (cost: 0 %).

At the bottom of the screen, the status bar indicates: '(local) (12.0 RTM) MS\Szelor (53) AdventureWorksLT2012 00:00:00 0 rows'.

Rysunek 13.2. W tym przypadku plan wykonania zapytania nadal jest dość prosty, ale i tak jest bardziej rozbudowany, niż można byłoby się spodziewać po zapytaniu typu `SELECT *` FROM tabeli

 Wskazówka	<p>Głębsza niż trzypoziomowa hierarchia widoków utrudnia wykrywanie i rozwiązywanie problemów związanych z wydajnością zapytań i uprawnieniami użytkowników.</p>
--	--

Widoki grupujące dane

Widoki mogą również grupować dane. W poniższym przykładzie prawie wszystkie kolumny widoku zwracają wynik funkcji grupującej, a więc zamiast nadawać im aliasy, zdefiniowaliśmy je w nagłówku widoku:

```

CREATE VIEW [SalesLT].[DailySales] (OrderDate, Clients, Orders, AvgDue)
AS
SELECT [OrderDate], COUNT(DISTINCT [CustomerID]),
       COUNT (DISTINCT [SalesOrderID]), AVG([TotalDue])
FROM [SalesLT].[SalesOrderHeader]
GROUP BY [OrderDate];
    
```

Łatwiej jest przeszukiwać i analizować dane pogrupowane za pomocą tego typu widoków niż pogrupowane samodzielnie. Na przykład żeby sprawdzić, w których dniach średnia opłata przekroczyła 50, a liczba klientów 2, wystarczy wykonać poniższe zapytanie:

```

SELECT *
FROM [SalesLT].[DailySales]
    
```

```
WHERE AvgDue>10  
AND Clients>2;
```

```
-----  
OrderDate          Clients    Orders     AvgDue  
2008-06-01 00:00:00.000   32        32        29884,4873
```

Modyfikowanie danych poprzez widoki

Dane mogą być poprzez widoki nie tylko odczytywane, lecz także modyfikowane, o ile zostaną spełnione poniższe warunki:

1. Widok nie grupuje danych.
2. W definicji widoku nie występuje klauzula DISTINCT.
3. Definicja widoku nie zawiera wyrażeń.
4. Jednocześnie modyfikowane będą dane tylko z jednej tabeli bazowej^[1].
5. Modyfikowane poprzez widok dane są zgodne z nałożonymi na tabele bazowe ograniczeniami, w szczególności niemożliwe jest wstawienie poprzez widok wierszy niezawierających wszystkich wymaganych kolumn tabeli, chyba że mają one zdefiniowane wartości domyślne.
6. Jeżeli widok odwołuje się do innych widoków, one również muszą spełniać powyższe warunki.



Wskazówka

Modyfikowanie danych poprzez widok jest możliwe tylko wtedy, gdy zmiany będą jednoznaczne (dlatego widok nie może grupować danych czy eliminować powtarzających się wierszy) i poprawne, tj. takie, które mogą być wykonane bezpośrednio w tabelach bazowych.

Poniższy widok spełnia wszystkie warunki, a więc będziemy mogli zmodyfikować poprzez niego tabelę bazową:

```
CREATE VIEW [SalesLT].[Bikes]  
AS  
WITH CategoryCTE([ParentProductCategoryID], [ProductCategoryID], [Name])  
AS  
(  
    SELECT [ParentProductCategoryID], [ProductCategoryID], [Name]  
    FROM SalesLT.ProductCategory  
    WHERE ProductCategoryID =1
```

```

UNION ALL
    SELECT C.[ParentProductCategoryID], C.[ProductCategoryID], C.
    [Name]
        FROM SalesLT.ProductCategory AS C
        INNER JOIN CategoryCTE AS BC ON BC.ProductCategoryID =
    C.ParentProductCategoryID
)
SELECT CCTE.[Name] as [ProductCategoryName], P.Name, P.Color
FROM CategoryCTE AS CCTE
JOIN [SalesLT].[Product] AS P
ON CCTE.ProductCategoryID=p.ProductCategoryID;

```

Spróbujmy zmodyfikować poprzez ten widok kolor wybranego roweru i sprawdzić, czy zmiana będzie widoczna poprzez widok:

```

UPDATE [SalesLT].[Bikes]
SET Color = 'Dark'
WHERE Name = 'Road-750 Black, 44';
SELECT * FROM [SalesLT].[Bikes];
-----
```

ProductCategoryName	Name	Color
Mountain Bikes	Mountain-500 Black, 52	Black
Road Bikes	Road-750 Black, 44	Dark
Road Bikes	Road-750 Black, 48	Black

...

W tym przypadku udało nam się nie tylko zmodyfikować wiersz, lecz także odczytać go poprzez widok. Jeżeli jednak zmienilibyśmy kategorię produktu na inną niż dowolne rowery, zmiana zostałaby wykonana i w konsekwencji wiersz nie byłby dłużej zwracany przez ten widok.

Należy pamiętać, że aplikacje klienckie najczęściej prezentują dane za pomocą kontrolek przypominających arkusze Excela lub formularzy, których poszczególne pola zawierają wartości odpowiednich atrybutów aktywnego rekordu. W obu przypadkach po odświeżeniu ekranu użytkownik, który nie wie, że modyfikuje dane poprzez widok, odniesie wrażenie, iż wiersz nie został zmodyfikowany czy wstawiony. W rezultacie może spróbować

powtórzyć operację i nieświadomie wprowadzić drugi raz te same dane (w przypadku wykonywania instrukcji `INSERT`) lub niepotrzebnie wzywać obsługę techniczną, bo wydawać mu się będzie, że zmodyfikowany wiersz został usunięty (w przypadku wykonywania instrukcji `UPDATE`).

Problem ten zademonstrujemy, używając prostego widoku zwracającego dane klientek z tabeli utworzonej podczas wykonywania jednego z zadań z rozdziału 10. Najpierw utworzymy widok wybierający z tej tabeli dane pań o imionach zaczynających się na literę D:

```
CREATE VIEW vClients
AS
SELECT [CustomerID], [FirstName], [LastName]
FROM [dbo].[Panie]
WHERE FirstName LIKE 'D%';
```

Następnie odczytamy dane poprzez ten widok:

```
SELECT *
FROM vClients;
```

```
-----
CustomerID      FirstName        LastName
30118            Donna           Carreras
84               Della            Demott Jr
334               Dora             Verdad
470               Delia            Toone
29637            Donna           Carreras
29705            Della            Demott Jr
30080            Delia            Toone
30115            Dora             Verdad
```

i wstawimy przez niego do tabeli `[dbo].[Panie]` nowy wiersz (zwróć uwagę, że imię tej pani zaczyna się na inną literę niż D):

```
INSERT INTO vClients ([FirstName],[LastName])
VALUES ('Anna','Nowak');
```

Jeżeli raz jeszcze odczytamy dane poprzez widok (wyobraźmy sobie, że odświeżamy dane w aplikacji klienckiej), nowo wstawionego

wiersza nie znajdziemy:

```
SELECT *
FROM vClients;
-----
CustomerID      FirstName      LastName
30118            Donna          Carreras
84               Della          Demott Jr
334               Dora           Verdad
470               Delia          Toone
29637            Donna          Carreras
29705            Della          Demott Jr
30080            Delia          Toone
30115            Dora           Verdad
```

Następnie zmodyfikujemy poprzez widok imię jednej z pań:

```
UPDATE vClients
SET [FirstName] = 'Rosmarie'
WHERE CustomerID = 334;
```

Tym razem użytkownik odniesie wrażenie, że aktualizując dane, przypadkowo je usunął:

```
SELECT *
FROM vClients;
-----
CustomerID      FirstName      LastName
30118            Donna          Carreras
84               Della          Demott Jr
470               Delia          Toone
29637            Donna          Carreras
29705            Della          Demott Jr
30080            Delia          Toone
30115            Dora           Verdad
```

Możemy zapobiec obu powyższym sytuacjom, stosując klauzulę CHECK OPTION:

```
ALTER VIEW vClients
```

AS

```
SELECT [CustomerID], [FirstName], [LastName]
FROM [dbo].[Panie]
WHERE FirstName LIKE 'D%'
WITH CHECK OPTION;
```

Od teraz modyfikacje danych, które spowodowałyby „zniknięcie” wiersza, będą poprzez ten widok niemożliwe do wykonania:

```
UPDATE vClients
SET [FirstName] = 'Rosmarie'
WHERE CustomerID = 84;
```

Msg 550, Level 16, State 1, Line 146

The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.

The statement has been terminated.



Wskazówka

Klauzula CHECK OPTION powinna być używana we wszystkich umożliwiających modyfikowanie danych widokach. W ten sposób unikniemy wrażenia niespójności i nieprzewidywalności działania aplikacji bazodanowej.

Zalety widoków

Użytkownicy nie powinni mieć bezpośredniego dostępu do tabel. Zamiast tego powinni odczytywać i modyfikować dane poprzez widoki, funkcje lub procedury^[2]. Uzyskamy w ten sposób:

1. Funkcjonalny mechanizm kontroli dostępu do danych. Jeżeli tylko część danych z tabeli powinna być udostępniona wszystkim użytkownikom (np. imiona i nazwiska pracowników, ale nie ich adresy czy numery telefonów), tworząc widok zwracający wyłącznie ogólnodostępne dane i odbierając użytkownikom bezpośredni dostęp do tabeli, zapewnmy poufność pozostałych danych.
2. **Warstwę abstrakcji pozwalającą na zmianę struktury tabel bez konieczności aktualizacji programów klienckich.** Schemat każdej produkcyjnej bazy danych przedzej czy później

- będzie zmieniany w związku z nowymi wymaganiami albo na potrzeby optymalizacji. Jeżeli użytkownicy od początku odczytywali dane poprzez widoki, zmiana struktury tabel może być skompensowana odpowiednią zmianą widoków i nie będzie wymagała aktualizacji wszystkich programów klienckich.
3. Możliwość łatwiejszego i szybszego przesyłania zapytań przez sieć.

Indeksy

Indeksy organizują dane w sposób umożliwiający ich wydajne odczytywanie i modyfikowanie. Chociaż bez indeksów serwery bazodanowe też są w stanie odczytać wszystkie dane wymagane do wykonania instrukcji SQL, to indeksy wielokrotnie skracają czas takich operacji.

 Wskazówka	Budowa i działanie indeksów zależą od konkretnego serwera bazodanowego. Bieżący podrozdział zawiera tylko podstawowe, ogólne informacje na temat indeksów.
--	--

W serwerze SQL wiersze tabel mogą być przechowywane na dwa sposoby^[3]:

1. W postaci sterty (ang. *Heap*), czyli nieuporządkowanego, nieposortowanego zbioru.
2. W postaci uporządkowanej struktury, z reguły drzewa zrównoważonego (ang. *B-tree*). W takim przypadku wiersze są posortowane według klucza indeksu zgrupowanego (ang. *Clustered*). Indeks zgrupowany można porównać do książki telefonicznej, której cała zawartość jest posortowana alfabetycznie według nazwisk (klucza indeksu). Dzięki temu, gdy chcemy znaleźć numer telefonu, wystarczy, że wyszukamy stronę (lub strony), na której znajdują się dane osób o szukanym nazwisku.

Ponadto dla każdej tabeli można utworzyć wiele indeksów dodatkowych (z reguły góry limit serwera wielokrotnie przekracza potrzeby — pojedyncza tabela powinna mieć od kilku do kilkunastu indeksów dodatkowych). Indeks tego typu można porównać do

skorowidza książki — znajdują się w nim posortowane hasła i odnośniki do stron, na których zostały one opisane^[4].

Główna zaletą indeksów jest ograniczenie odczytywanych z dysków danych. Różnice w czasie wykonania tej samej instrukcji bez właściwych indeksów i z nimi mogą być ogromne — w pierwszym przypadku konieczne jest odczytanie wszystkich stron tabeli nawet wtedy, kiedy interesuje nas jeden z milionów wierszy, w drugim przypadku serwer bazodanowy znajdzie potrzebne dane w indeksie.

Jeżeli jednak indeks nie zawiera wszystkich potrzebnych do wykonania instrukcji danych, serwer bazodanowy będzie je musiał odczytać z tabeli. Ta operacja sięgania po dane (ang. *Lookup*) może okazać się bardzo kosztowna. Dlatego serwery bazodanowe używają indeksów niezawierających zapytania tylko wtedy, gdy są one wystarczająco selektywne, czyli zwracają mniej niż kilka procent z wszystkich wierszy tabeli^[5].

Drugą zaletą indeksów jest to, że odczytując je, serwer otrzymuje posortowane dane. Sortowanie dużych zbiorów danych jest czasochłonną i wymagającą dużych ilości pamięci operacją, która jest wykonywana nie tylko na potrzeby klauzuli ORDER BY, lecz także w niektórych typach łączenia i grupowania danych.

Indeksy mają też wady — muszą być na bieżąco aktualizowane i zajmują dodatkowe miejsce. Każde wstawienie, usunięcie czy aktualizacja danych w poindeksowanej tabeli wiąże się z aktualizacją wszystkich zdefiniowanych dla tej tabeli indeksów.

Biorąc pod uwagę zalety i wady indeksów, warto indeksować kolumny, które:

1. Przechowują bardzo mało powtarzających się danych (w innym przypadku selektywność zapytań będzie zbyt mała i serwer bazodanowy i tak nie skorzysta przy ich wykonywaniu z indeksu).
2. Są używane do wybierania wierszy, a więc często używane w klauzulach WHERE lub HAVING.
3. Są używane do łączenia tabel — o ile kolumny kluczy podstawowych są automatycznie indeksowane, o tyle kolumny kluczy obcych należy zindeksować samemu.
4. Są używane do sortowania lub grupowania wierszy.

Tworzenie, modyfikowanie i usuwanie indeksów

Utworzyć indeks możemy za pomocą instrukcji CREATE INDEX. Wymagane jest podanie nazwy indeksu oraz wskazanie tabeli i kolumn, na których ma on być założony^[6]:

```
CREATE INDEX IX_SalesOrderDetailLineTotal  
ON [SalesLT].[SalesOrderDetail] ([LineTotal]);
```

Indeks może być założony na kilku kolumnach tabeli. Wtedy klucz indeksu będzie zawierał dane z wszystkich wybranych kolumn. Takie indeksy zawierają znacznie więcej zapytań, czyli do ich wykonania wystarczy odczytać sam indeks, bez bardzo kosztownego odczytywania brakujących danych z tabeli.

 Wskazówka	Serwery bazodanowe korzystają z indeksów niezawierających zapytań tylko dla wysoce selektywnych instrukcji SELECT, więc tworząc indeksy kompozytowe (złożone), znacznie zwiększymy prawdopodobieństwo ich użycia.
--	---

Przeanalizujemy to na przykładzie liczącej zaledwie 542 wiersze tabeli SalesLT.SalesOrderDetail. Sprawdzimy, czy zapytanie wybierające z niej dane na podstawie kolumny LineTotal będzie wykonane z wykorzystaniem założonego przed chwilą indeksu.

Jeżeli zapytanie nie zawiera się w indeksie (czyli odwołujemy się w nim zarówno do zindeksowanych, jak i niezindeksowanych kolumn), wszystko zależy od jego selektywności:

1. Pierwsze zapytanie zwraca 2 z 542 wierszy, czyli jego selektywność wynosi $2/542 = 0,37\%$. To zapytanie będzie wykonane z użyciem indeksu^[7].

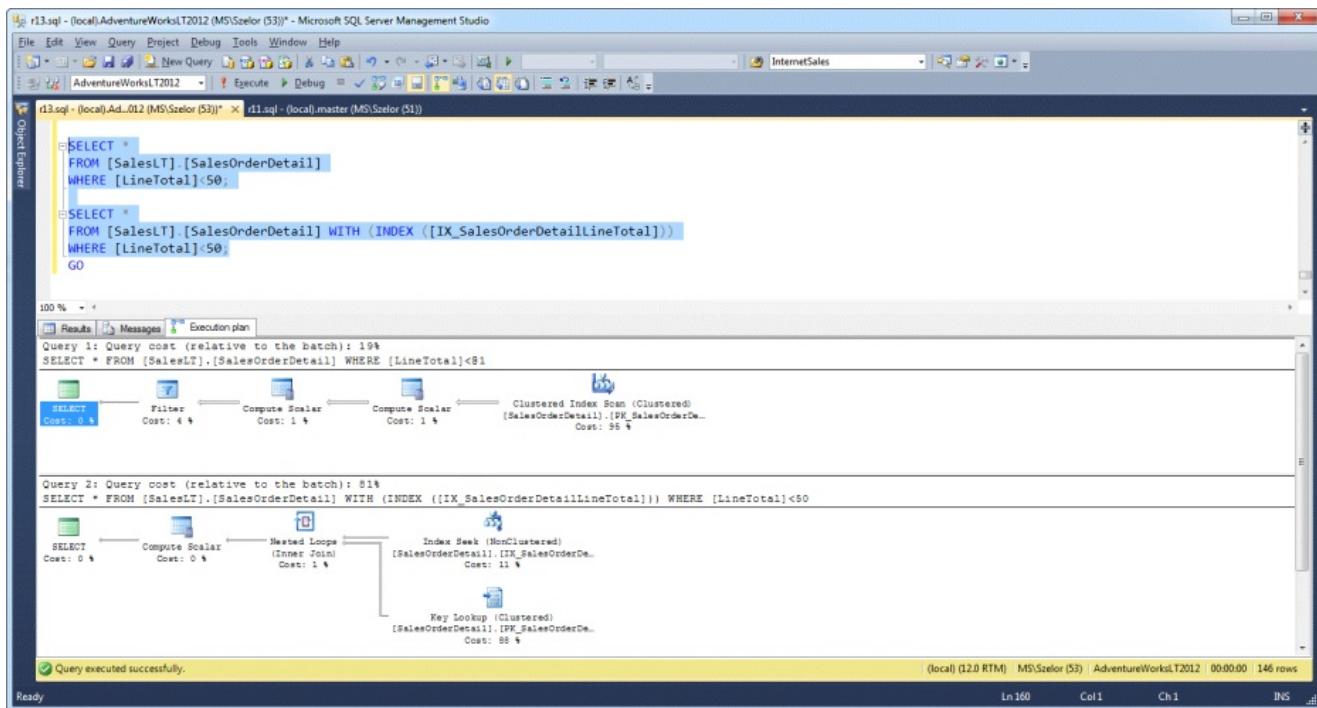
```
SELECT *  
FROM [SalesLT].[SalesOrderDetail]  
WHERE [LineTotal]<5;
```

2. Drugie zapytanie zwraca 73 wiersze, a więc jego selektywność wynosi 13%. Zostało ono wykonane bez użycia indeksu, poprzez odczytanie całej tabeli:

```
SELECT *  
FROM [SalesLT].[SalesOrderDetail]
```

```
WHERE [LineTotal]<50;
```

3. Żeby przekonać się, dlaczego serwer bazodanowy zdecydował się na odczytanie całej tabeli, chociaż selektywność zapytania wynosiła około 1%, wymusimy wykonanie drugiego zapytania z użyciem indeksu (rysunek 13.3).

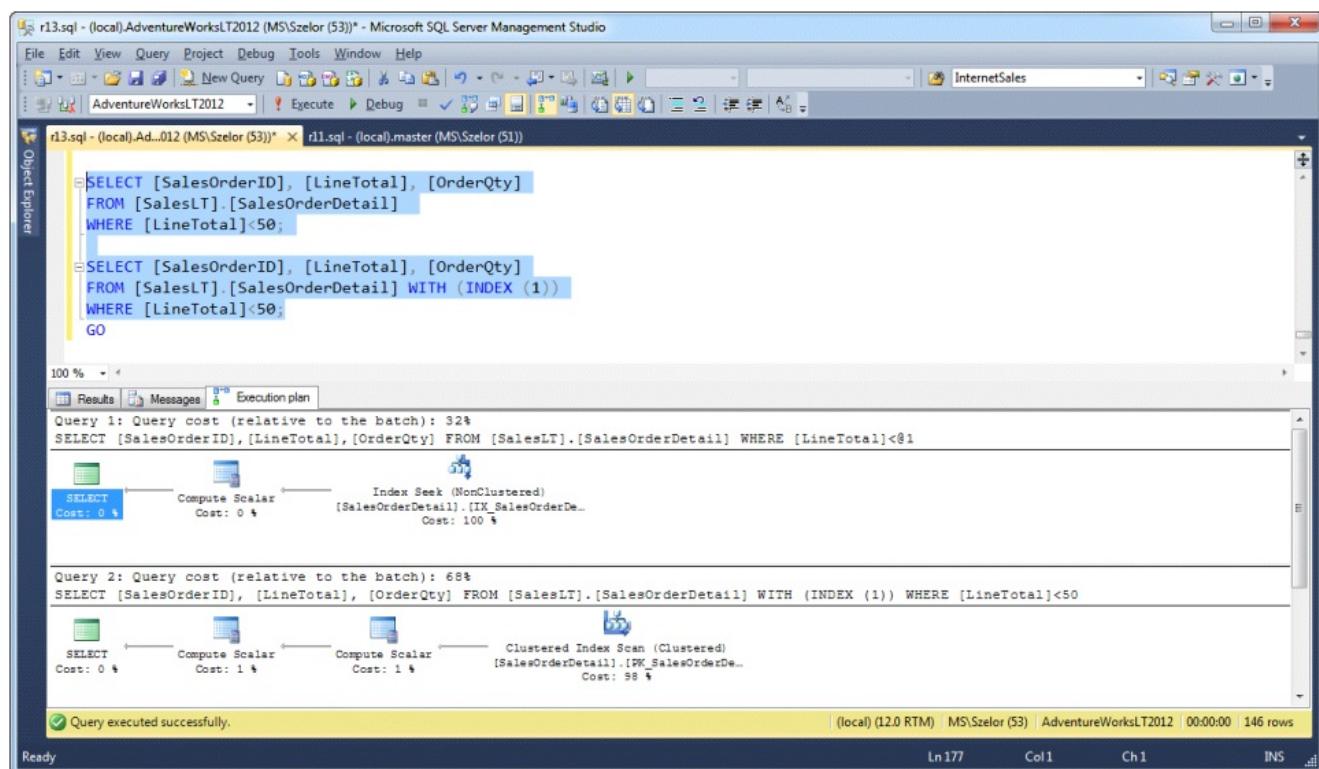
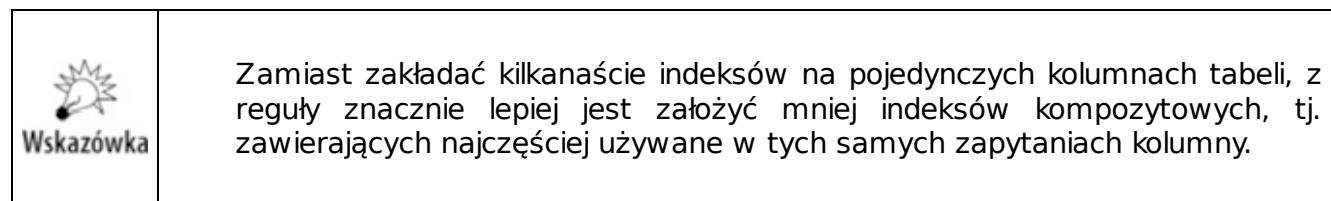


Rysunek 13.3. Okazuje się, że wybranie wierszy na podstawie indeksu i odczytanie dla każdego z nich pozostałych wartości z tabeli trwało pięciokrotnie dłużej niż odczytanie całej tabeli SalesLT.SalesOrderDetail

Gdyby jednak zapytanie zawierało się w indeksie, jego selektywność nie miałaby wpływu na decyzję o użyciu lub nie indeksu — serwer bazodanowy zawsze użyje takiego indeksu. Jeżeli zastąpimy nasz indeks indeksem założonym na obu używanych w poniższym zapytaniu kolumnach^[8]:

```
DROP INDEX [IX_SalesOrderDetailLineTotal]  
ON [SalesLT].[SalesOrderDetail];  
GO  
  
SELECT [SalesOrderID], [LineTotal], [OrderQty]  
FROM [SalesLT].[SalesOrderDetail]  
WHERE [LineTotal]<50;
```

to zapytanie zostanie wykonane poprzez przeszukanie (ang. *Seek*) indeksu, a nie skanowanie (ang. *Scan*) całej tabeli. Żeby przekonać się, o ile skróciło to czas wykonania zapytania, za drugim razem wymusimy na serwerze bazodanowym odczytanie całej tabeli (rysunek 13.4).



Rysunek 13.4. Stworzenie indeksu zawierającego zapytanie trzykrotnie skróciło czas jego wykonywania

Opcje indeksów

Tworząc indeks, możemy m.in. określić, czy wartości jego klucza będą niepowtarzalne^[9]:

```
CREATE UNIQUE INDEX IX_Products
ON [SalesLT].[Product] ([ProductNumber], [StandardCost],
[SellStartDate]);
```

Tego typu indeksy używane są przede wszystkim do wymuszania

pewnych reguł biznesowych (w tym wypadku zapewnienia niepowtarzalności wartości w wymienionych kolumnach), ale również potrafią wielokrotnie poprawić wydajność zapytań.

Drugą, bardzo przydatną opcją jest wybranie indeksowanych wierszy tabeli — domyślnie indeks zawiera tyle kluczy, ile wierszy liczy poindeksowana za jego pomocą tabela. Bardzo często jednak niektóre wartości w kolumnie występują w tak dużej liczbie, że ich umieszczenie w indeksie niewiele daje, a powoduje, że indeks jest znacznie większy (i przez to kosztowniejszy w utrzymaniu), niż mógłby być. Rozwiązaniem tego problemu jest dodanie klauzuli WHERE do instrukcji CREATE INDEX:

```
CREATE INDEX IX_CurrentProducts
ON [SalesLT].[Product] ([ProductNumber], [StandardCost], [SellStartDate],
[SellEndDate])
WHERE [SellEndDate] IS NOT NULL;
```

Porządkowanie indeksów

Klucze indeksów, tak jak pliki na dyskach, z czasem ulegają fragmentacji^[101]. W rezultacie logiczny porządek kluczy indeksu przestaje odpowiadać ich porządkowi fizycznemu (lokalizacji w pliku bazy danych). Wydłuża to czas dostępu do indeksu, przede wszystkim podczas ich skanowania. Dlatego jedną z regularnie wykonywanych (z reguły w nocy, kiedy obciążenie serwera bazodanowego jest najmniejsze) czynności administracyjnych jest porządkowanie indeksów.

Uporządkowanie może polegać na ponownym posortowaniu kluczy indeksu lub na jego usunięciu i odtworzeniu:

1. Przebudować (usunąć i odtworzyć) indeks (lub wszystkie indeksy wskazanej tabeli) możemy instrukcją ALTER INDEX ... REBUILD:

```
ALTER INDEX ALL
ON [SalesLT].[Product]
REBUILD;
```

2. Przebudować wybrany indeks możemy, podając jego nazwę:

```
ALTER INDEX IX_CurrentProducts
ON [SalesLT].[Product]
```

- REBUILD;
3. Uporządkować (posortować jego klucze) indeks (lub wszystkie indeksy wskazanej tabeli) możemy instrukcją ALTER INDEX ... REORGANIZE:

```
ALTER INDEX IX_CurrentProducts  
ON [SalesLT].[Product]  
REORGANIZE;
```

Podsumowanie

- Widoki są najprostszym i najbardziej uniwersalnym sposobem ochrony tabel przed destabilizującymi działaniami ich użytkowników.
- Odczytywanie widoków niczym nie różni się od odczytywania tabel bazowych.
- Żeby odczytane poprzez widok dane były posortowane, w zapytaniu (a nie w definicji widoku) należy użyć klauzuli ORDER BY.
- Jeżeli widok jednoznacznie odzwierciedla dane tabeli bazowej, możliwe jest ich modyfikowanie poprzez widok.
- Widoki, poprzez które można modyfikować dane, powinny zawierać klauzulę CHECK OPTION.
- Indeksy nie wpływają na wyniki instrukcji, ale mają ogromny wpływ na czas ich wykonywania.
- Nie należy indeksować wszystkich kolumn.
- Żeby serwery bazodanowe skorzystały z istniejącego, niezawierającego zapytania indeksu:
 - W zapytaniu muszą być użyte argumenty SARG^[11];
 - Selektyność zapytania musi być bardzo duża.
- Ponieważ nie mamy wpływu na selektynność zapytań, powinniśmy starać się tworzyć widoki zawierające wszystkie kolumny odczytywane w najczęściej lub najdłużej wykonywanych zapytaniach.
- Indeksy należy regularnie porządkować.

Zadania

- Przekształć poniższe zapytanie tak, aby mogło być użyte jako definicja widoku:

```
SELECT c.Name, p.Name, p.ListPrice*1.23  
FROM [SalesLT].[Product] AS P  
JOIN [SalesLT].[ProductCategory] AS C  
ON P.ProductCategoryID = C.ProductCategoryID;
```

- Czy poniższy widok umożliwia modyfikowanie danych? Uzasadnij swoją odpowiedź.

```
CREATE VIEW Zadanie2  
AS  
SELECT DISTINCT [FirstName] + ' ' + [LastName] AS Client,  
[SalesOrderID], [OrderDate]  
FROM [SalesLT].[SalesOrderHeader] AS OH  
JOIN [SalesLT].[Customer] AS C  
ON C.CustomerID = OH.CustomerID;
```

- Załóż unikatowy indeks na kolumnach [SalesOrderID] i [UnitPrice] tabeli [SalesLT].[SalesOrderDetail].

[1] Niektóre serwery bazodanowe są bardziej restrykcyjne i nie pozwalają na modyfikowanie danych poprzez widok, jeżeli odwołuje się on do wielu tabel.

[2] Funkcje i procedury jako zależne od konkretnej wersji serwera bazodanowego nie zostały opisane w tej książce.

[3] Od wersji 2014 serwer SQL umożliwia również przechowywanie wierszy w tabelach pamięciowych. Zagadnienie to wykracza jednak poza zakres tej książki.

[4] Zauważmy, że jeżeli szukamy konkretnego hasła, szybciej znajdziemy je w skorowidzu, znacznie mniejszym niż cała książka. Po jego znalezieniu należy tylko zatrzymać się na odpowiednią stronę lub strony i zapoznać się z opisem interesującego nas hasła. Jeśli jednak szukane hasło występuje na wielu stronach książki, skorowidz okazuje się nieprzydatny. Szybszą metodą zapoznania się z tym hasłem jest przejrzenie po kolejnych wszystkich stron książki, a nie wielokrotne jej kartkowanie według znalezionych w skorowidzu wszystkich odnośników do tego hasła.

[5] Dla serwera SQL próg selektywności wynosi około 1%.

[6] Serwery bazodanowe nie sprawdzają, czy na tych samych kolumnach tabeli nie ma założonego indeksu o innej nazwie. Dlatego przed dodaniem indeksu należy sprawdzić, czy przypadkiem nie tworzymy kopii istniejącego indeksu.

[7] Wiemy już, że aby wyświetlić plany wykonań zapytań, w konsoli SSMSE należy zaznaczyć przycisk *Include Actual Execution Plan*.

[8] Kolejność kolumn klucza indeksu złożonego ma ogromne znaczenie — żeby się o tym przekonać, wystarczy spróbować znaleźć w uporządkowanej według nazwisk książce telefonicznej numer telefonu znajomego na podstawie jego imienia. Pierwszą kolumną indeksu powinna być ta, która zawiera mniej duplikatów.

[9] Lista wszystkich opcji indeksów zależy od konkretnego serwera bazodanowego.

[10] Sposoby odczytywania stopnia nieuporządkowania indeksów zależą od konkretnego serwera bazodanowego.

[11] Argumenty SARG zostały opisane w rozdziale 9.

Część V

Uprawnienia użytkowników, czyli instrukcje GRANT i REVOKE

Serwery bazodanowe nie pozwalają anonimowym użytkownikom na korzystanie ze swoich zasobów. Łącząc się z serwerem, użytkownik musi potwierdzić swoją tożsamość, z reguły poprzez podanie loginu i hasła. **Ten jednorazowy proces nazywa się uwierzytelnieniem**, a jego przebieg zależy od konkretnego serwera bazodanowego i nie został przez nas opisany^[1].

Za każdym razem, gdy użytkownik próbuje wykonać jakąś operację, np. odczytać dane, wstawić wiersze czy wywołać procedurę, serwer sprawdza, czy ma on wystarczające do tego uprawnienia. **Autoryzacja, czyli zabranianie korzystania lub zezwalanie użytkownikowi na korzystanie z pewnych zasobów, jest przeprowadzana automatycznie**, a jej wynik zależy od posiadanych przez użytkownika uprawnień.

Z ostatniej części książki możesz się dowiedzieć:

1. Jak zarządzać kontami użytkowników;
2. Czym są role;
3. Jak nadawać i odbierać uprawnienia;
4. Dlaczego zasada minimalnych uprawnień jest tak ważna dla bezpieczeństwa systemu komputerowego.

[1] Uwierzytelnianie zostało po raz pierwszy ujęte w standardzie SQL2 (a więc w roku 1992), w dodatku w ramach najwyższego poziomu zgodności, którego nie osiągnął żaden serwer bazodanowy.

Rozdział 14. Nadawanie i odbieranie uprawnień

- Jak założyć użytkownikowi konto?
- Dlaczego w przypadku serwera SQL konta użytkowników nie są chronione hasłami?
- Czemu warto łączyć użytkowników w grupy, czyli tzw. role?
- Jak utworzyć nową rolę?
- Jak nadać lub odebrać użytkownikowi bądź roli uprawnienia do wskazanego obiektu i do wykonania danej operacji?
- Jak zastosować w praktyce zasadę minimalnych uprawnień?

Konta użytkowników

Żeby użytkownik mógł się połączyć z bazą danych, musi mieć w niej założone konto. **Serwer SQL uwierzytelnia użytkowników na poziomie serwera, a nie pojedynczej bazy danych.** Z tego powodu, tworząc konto użytkownika, musimy wskazać istniejący na poziomie serwera login. Domyślnie serwer SQL pozwala połączyć się tylko użytkownikom systemu Windows, czyli tworząc login, należy wskazać lokalne lub domenowe konto użytkownika^[2]:

```
CREATE LOGIN [sql\Danka]
FROM WINDOWS;
```

W tym przykładzie `sql` jest nazwą komputera (lub domeny Active Directory), a `Danka` — poprawną nazwą konta użytkownika Windows.

Tworząc login, możemy określić domyślną bazę danych, czyli bazę, z którą posługujący się tym loginem użytkownik automatycznie spróbuje się połączyć^[3]:

```
CREATE LOGIN [SQL\Marcin]
FROM WINDOWS
WITH DEFAULT_DATABASE= AdventureWorksLT2012;
```

Tworząc loginy na podstawie kont systemu Windows, nie podaje się chroniących je haseł. Żeby posłużyć się takim loginem,

użytkownik musi zalogować się do systemu operacyjnego na powiązane z nim konto, czyli musi potwierdzić swoją tożsamość, np. podając poprawne hasło. Serwer SQL ufa systemowi Windows i w takiej sytuacji nie sprawdza ponownie tożsamości użytkowników.

Zakładanie i usuwanie kont użytkowników

Jeżeli użytkownik chce się połączyć z daną bazą danych, musi mieć login powiązany z założonym w tej bazie danych kontem użytkownika albo skorzystać z konta gościa, o ile jest włączone. W przeciwnym razie użytkownik, który połączył się z serwerem SQL na podstawie loginu, nie połączy się z bazą danych:

```
USE AdventureWorksLT2012;
```

```
-----  
The server principal "sql\Danika" is not able to access the database  
"AdventureWorksLT2012" under the current security context.
```

Możemy utworzyć konto użytkownika (o ile mamy odpowiednie uprawnienia), wykonując w docelowej bazie danych instrukcję CREATE USER:

```
USE AdventureWorksLT2012;  
CREATE USER Danka  
FOR LOGIN [sql\Danika];
```

Jeden login można powiązać z kontami użytkowników wielu działających na tym samym serwerze SQL baz danych, a w każdej z tych baz danych nazwa konta użytkownika może być inna^[4]:

```
USE AdventureWorksLT2012;  
CREATE USER Marcin  
FOR LOGIN [SQL\Marcin];  
USE Test;  
CREATE USER Michał  
FOR LOGIN [SQL\Marcin];
```

Role

Utworzony dla użytkowników konta umożliwiają im jedynie połączenie się z bazą danych. Próba odczytania przez nich dowolnej tabeli, nie mówiąc już o próbie zmodyfikowania danych czy

utworzenia nowego widoku, skończy się błędem:

```
SELECT *
FROM [AdventureWorksLT2012].[SalesLT].[Product];
-----
Msg 229, Level 14, State 5, Line 24
The SELECT permission was denied on the object 'Product', database
'AdventureWorksLT2012', schema 'SalesLT'.
```

Wynika to z domyślnej polityki bezpieczeństwa serwerów bazodanowych — **użytkownik, żeby wykonać operację, musi mieć nadane odpowiednie uprawnienia**. Domyślnie użytkownik nie ma żadnych uprawnień, a więc nie może wykonać ani jednej operacji.

Uprawnienia można nadawać użytkownikom lub roliom (grupom użytkowników). Z wyjątkiem kilku specyficznych sytuacji, np. gdy wszystkie loginy są powiązane z grupami użytkowników systemu Windows, a nie z pojedynczymi kontami, **uprawnienia należy nadawać roliom, a nie bezpośrednio użytkownikom**. Zyskamy w ten sposób funkcjonalny, łatwy do zarządzania system zabezpieczeń i będziemy mogli zastosować w praktyce opisaną w dalszej części rozdziału zasadę minimalnych uprawnień.

Tworzenie i usuwanie ról

Utworzyć rolę w bazie danych możemy za pomocą instrukcji CREATE ROLE:

```
CREATE ROLE Dev;
CREATE ROLE Marketing;
```

Natomiast usunąć istniejącą rolę możemy, wykonując instrukcję DROP ROLE:

```
DROP ROLE Marketing;
```

Usuwać można wyłącznie role użytkowników; wbudowane role serwera i bazy danych nie mogą być skasowane:

```
DROP ROLE db_ddadmin;
```

```
-----
Cannot drop the role 'db_ddadmin'.
```

Przypisywanie ról do użytkowników

Do jednej roli możemy przypisać dowolną liczbę kont użytkowników. W ten sposób rola będzie reprezentowała użytkowników o takich samych lub podobnych uprawnieniach, np. pracowników tego samego działu^[5]:

```
EXEC sp_addrolemember 'Dev', 'Marcin';
EXEC sp_addrolemember 'Dev', 'Danka';
```

To samo konto użytkowników może należeć do wielu różnych ról. Przykładowo jeżeli jeden z programistów ma dodatkowo wykonywać kopie zapasowe bazy danych, możemy go dodać do posiadającej odpowiednie uprawnienia predefiniowanej roli db_backupoperator:

```
EXEC sp_addrolemember 'db_backupoperator', 'Danka';
```

Specjalna rola Public

Rola Public ma specjalne znaczenie — automatycznie należy do niej każdy użytkownik bazy danych i nie można nikogo jej pozbawić:

```
EXEC sp_droprolemember 'Public', 'Danka';
-----
```

Membership of the public role cannot be changed.

Pozwala ona nadać lub odebrać określone uprawnienia wszystkim użytkownikom bazy danych.

Uprawnienia

Uprawnienia dzielą się na dwie kategorie:

1. Uprawnienia systemowe pozwalają na wykonanie określonej operacji. Należą do nich m.in.^[6]:
 - a. uprawnienie do modyfikowania ról ALTER ANY ROLE;
 - b. uprawnienie do modyfikowania kont użytkowników ALTER ANY USER;
 - c. uprawnienie do wykonywania kopii zapasowych baz danych BACKUP DATABASE;
 - d. uprawnienie do tworzenia funkcji CREATE FUNCTION;
 - e. uprawnienie do tworzenia procedur CREATE PROCEDURE;
 - f. uprawnienie do tworzenia schematów CREATE SCHEMA;

- g. uprawnienie do tworzenia tabel CREATE TABLE;
 - h. uprawnienie do przejmowania obiektów na własność TAKE OWNERSHIP;
 - i. uprawnienie do odczytywania metadanych obiektów VIEW DEFINITION.
2. Uprawnienia obiektowe pozwalają w określony sposób odwołać się do wskazanego obiektu. Należą do nich m.in.[\[7\]](#):
- a. Do tabel oraz widoków można nadać uprawnienia SELECT, INSERT, UPDATE, DELETE oraz REFERENCE umożliwiające odpowiednio: pobieranie, wstawianie, modyfikowanie, usuwanie danych oraz sprawdzanie i tworzenie kluczy obcych.
 - b. Do kolumn można nadać uprawnienia SELECT oraz UPDATE.
 - c. Do procedur składowanych oraz funkcji można nadać uprawnienie EXECUTE.
 - d. Do schematów można nadać wszystkie powyższe uprawnienia.

Nadawanie i odbieranie uprawnień

Przyjmijmy, że Danka i Marcin, tak jak pozostali programiści, muszą odczytywać i modyfikować dane dotyczące klientów. Zamiast nadawać poszczególnym użytkownikom wymagane uprawnienia, lepiej będzie nadać je roli, do której oni należą:

```
GRANT SELECT,INSERT,UPDATE,DELETE  
ON [SalesLT].[Customer]  
TO Dev;
```

Od teraz wszyscy członkowie roli Dev będą mogli odczytywać i modyfikować zapisane w tej tabeli dane. Gdybyśmy jednak odebrali konkretному użytkownikowi uprawnienia do modyfikowania danych w tabeli dbo.Pracownicy:

```
REVOKE INSERT,UPDATE,DELETE  
ON [SalesLT].[Customer]  
FROM Marcin;
```

to ten użytkownik, jako należący do roli Dev, nadal będzie mógł odczytywać dane:

```
SELECT FirstName  
FROM [SalesLT].[Customer];
```

```
-----  
FirstName  
Keith  
Donna  
Janet
```

ale nie będzie już mógł ich modyfikować. **Wyjątkiem od tej reguły jest serwer SQL, w którym uprawnienia odbiera się instrukcją DENY, a nie REVOKE.**

Odbieranie uprawnień w serwerze SQL

Programiści firmy Microsoft zmienili standardowe znaczenie instrukcji REVOKE — w serwerze SQL nie służy ona do odbierania uprawnień, a do usuwania principium (użytkownika lub roli) z listy kontroli dostępu danego obiektu. Innymi słowy, wykonując instrukcję REVOKE, przywracamy stan neutralny, w którym użytkownik nie ma nadanych żadnych uprawnień.

Wystarczy jednak, żeby użytkownik należał do posiadającej wymagane uprawnienia roli, jak to ma miejsce w przypadku użytkownika Marcin, który należy do roli Dev, by mógł on wykonać daną operację:

```
UPDATE [SalesLT].[Customer]  
SET FirstName='OOPS'  
WHERE CustomerID=1;  
-----  
(1 row(s) affected)
```

Jawnie odebrać użytkownikowi lub roli uprawnienia możemy, wykonując instrukcję DENY:

```
DENY INSERT,UPDATE,DELETE  
ON [SalesLT].[Customer]  
TO Marcin;
```

W ten sposób użytkownik, niezależnie od swojej przynależności do ról, nie będzie mógł skasować wiersza ze wskazanej tabeli:

```
EXECUTE AS USER ='Marcin';  
DELETE FROM dbo.Pracownicy  
WHERE ID=1;  
REVERT;
```

Msg 229, Level 14, State 5, Line 69

The DELETE permission was denied on the object 'Customer', database 'AdventureWorksLT2012', schema 'SalesLT'.

 Wskazówka	Różnica pomiędzy instrukcjami REVOKE i DENY polega na tym, że w pierwszym przypadku użytkownik będzie mógł wykonać daną operację, jeżeli wymagane w tym celu uprawnienia zostały nadane którejś z ról, do których on należy. Natomiast odbierając uprawnienia instrukcją DENY, mamy pewność, że użytkownik nie wykona określonej operacji, nawet jeżeli należy do roli administratorów serwera. Ponieważ wszyscy użytkownicy należą do roli Public, nie powinno się odbierać jej żadnych uprawnień.
---	---

Dziedziczenie uprawnień

Obiekty baz danych tworzą następującą hierarchię:

1. Na szczeblu znajdują się obiekty serwera bazodanowego, takie jak bazy danych, punkty dostępowe czy loginy.
2. Poniżej, na poziomie bazy danych, znajdują się m.in. konta użytkowników, role i schematy.
3. Schematy zawierają tabele, widoki, procedury i funkcje.
4. Tabele składają się z kolumn.

Domyślnie uprawnienia nadane do obiektu znajdującego się wyżej w hierarchii są dziedziczone przez obiekty podzielne, tak jak uprawnienia do folderów są dziedziczone przez znajdujące się w nich pliki. Na przykład użytkownik posiadający uprawnienie UPDATE do danej tabeli może aktualizować dane we wszystkich jej kolumnach.

Uprawnienia mogą być jednak nadawane lub odbierane na wszystkich poziomach. Możliwe jest więc odebranie (w przypadku serwerów zgodnych ze standardem SQL3 za pomocą instrukcji REVOKE, a nie DENY) prawa do aktualizowania wybranych kolumn tabeli:

```
DENY UPDATE  
ON [SalesLT].[Customer]([LastName])  
TO Danka;
```

Od teraz użytkownik Danka będzie mógł aktualizować pozostałe kolumny tabeli [SalesLT].[Customer], ale próba zmiany nazwiska skończy się błędem:

```
EXECUTE AS USER ='Danka';  
UPDATE [SalesLT].[Customer]  
SET [CompanyName]='Test'  
WHERE CustomerID=1;
```

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

```
UPDATE [SalesLT].[Customer]  
SET [LastName] ='Test'  
WHERE CustomerID=1;
```

```
-----  
The UPDATE permission was denied on the column 'LastName' of the object  
'Customer', database 'AdventureWorksLT2012', schema 'SalesLT'.
```

W tym przypadku rola Dev miała nadane uprawnienia do obiektu wyższego poziomu (tabeli), ale odebraliśmy je konkretnemu użytkownikowi do obiektu znajdującego się niżej w hierarchii (kolumny). Ponieważ obiektów wyższego poziomu jest mniej (jedna tabela składa się z wielu kolumn, jeden schemat może zawierać wiele tabel i tak dalej), zmniejszyliśmy w ten sposób liczbę wymaganych uprawnień.

Tę samą strategię zastosujemy do schematów. Na początku podrozdziału założyliśmy, że wszyscy programiści mają mieć możliwość odczytywania i modyfikowania danych pozostałych pracowników. Te dane są zapisane w kilku tabelach i dodatkowo odczytywane są przez kilka widoków. Zamiast nadać roli Dev uprawnienia do poszczególnych obiektów, lepiej będzie nadać je raz do całego schematu SalesLT:

```
GRANT SELECT,INSERT,UPDATE,DELETE  
ON Schema::SalesLT  
TO Dev;
```

W ten sposób zmniejszyliśmy liczbę wymaganych uprawnień do jednego, dzięki czemu:

1. Wszyscy programiści mają takie same uprawnienia.
2. Mamy pewność, że programiści mogą odczytywać i modyfikować wszystkie dane dotyczące pracowników niezależnie od tego, w której tabeli są zapisane i w jaki sposób są odczytywane lub modyfikowane.
3. Możemy szybko i łatwo dostosowywać uprawnienia do zmieniających się sytuacji:
 - a. Jeżeli któryś z programistów odejdzie z firmy, wystarczy usunąć go z roli i automatycznie straci on wszystkie uprawnienia.
 - b. Jeżeli któryś z programistów będzie potrzebował dodatkowych uprawnień, wystarczy dodać go do odpowiedniej roli.
4. Dzięki mechanizmowi dziedziczenia uprawnień do nowo utworzonych obiektów (tabel czy widoków) w schemacie dbo nie musimy nadawać żadnych dodatkowych uprawnień.
5. Jeżeli żaden programista (lub któryś z nich) nie powinien mieć dostępu do wybranej tabeli czy kolumny, wystarczy jawnie odebrać im odpowiednie uprawnienia na poziomie tego obiektu.

Przekazywanie uprawnień

Domyślnie tylko administrator lub właściciel danego obiektu może nadawać do niego uprawnienia innym użytkownikom. W wielu przypadkach tak restrykcyjny model zabezpieczeń jest niepraktyczny — jeden administrator nie byłby w stanie zarządzać uprawnieniami setek użytkowników do tysięcy obiektów, nawet stosując wyżej opisaną strategię zarządzania uprawnieniami na jak najwyższym poziomie hierarchii.

Dlatego uprawnienia mogą być nadawane z prawem do przekazywania dalej. W ten sposób tworzony jest naturalny łańcuch zaufania — jeżeli administrator ufał użytkownikowi Marcin na tyle, że pozwolił mu odczytywać i modyfikować pewne dane, to może mu też pozwolić na nadanie takich samych uprawnień innemu użytkownikowi, np. w celu oddelegowania mu pewnych zadań.

Żeby pozwolić użytkownikowi na przekazywanie uprawnień dalej, należy użyć klauzuli WITH GRANT OPTION:

```
GRANT ALL[8]
ON [SalesLT].[Bikes]
TO Danka
WITH GRANT OPTION;
```

The ALL permission is deprecated and maintained only for compatibility.
It DOES NOT imply ALL permissions defined on the entity.

Od teraz użytkownik Danka może nadawać uprawnienia do widoku Bikes. innym użytkownikom:

```
EXECUTE AS USER = 'Danka'
GRANT SELECT
ON [SalesLT].[Bikes]
TO Marcin;
REVERT;
```

Command(s) completed successfully.

Zasada minimalnych uprawnień

Każde zabezpieczenie może być złamane. Dlatego tak ważne jest stosowanie się do jednej z podstawowych zasad bezpieczeństwa komputerowego: zasady minimalnych uprawnień. **Zgodnie z nią użytkownik powinien mieć nadane tylko takie uprawnienia, jakie są mu potrzebne do wykonywania jego obowiązków.** Pozwala to:

1. Zmniejszyć ryzyko udanego ataku — użytkownikowi o mniejszych uprawnieniach trudniej jest przeprowadzić skuteczny atak.
2. Ograniczyć skutki ewentualnego ataku — jeżeli atakujący podszyje się pod nieuprzywilejowanego użytkownika, będzie mógł wykonać tylko takie operacje, do jakich ten użytkownik miał nadane uprawnienia.
3. Zwiększyć szanse wykrycia i przerwania ataku — operacje wykonywane przez użytkowników powinny być odnotowywane w dziennikach zabezpieczeń, a nietypowa aktywność

nieuprzywilejowanego użytkownika jest łatwa do zauważenia.

Wymaga to jednak zbudowania funkcjonalnego mechanizmu zabezpieczeń, w którym administrator lub oddelegowane przez niego osoby są w stanie na bieżąco zmieniać uprawnienia w związku z zaistniałymi sytuacjami, takimi jak przyjęcie nowego pracownika, przejście pracownika z jednego działu do drugiego czy dodanie do bazy danych nowych tabel i widoków.

Podsumowanie

- Podczas nawiązywania połączenia z serwerem bazodanowym użytkownik jest uwierzytelniany.
- Przed wykonaniem każdej operacji serwer bazodanowy autoryzuje, czyli sprawdza, czy uwierzytelniony użytkownik ma nadane wystarczające uprawnienia.
- Konta użytkowników można łączyć w role.
- Jeden użytkownik może należeć do wielu ról.
- Uprawnienia można nadawać użytkownikom lub roliom.
- Uprawnienia powinno się nadawać do obiektów znajdujących się jak najwyżej w hierarchii, np. schematów, a nie do poszczególnych tabel.
- Odebrać uprawnienia można, wykonując instrukcję REVOKE, którą w serwerze SQL zastąpiła instrukcja DENY.

Zadania

1. Odbierz użytkownikom Danka i Marcin dostęp do zapisanych w kolumnie Phone tabeli [SalesLT].[Customer] numerów telefonów.
2. Pozwól użytkownikowi Marcin na przekazywanie innym użytkownikom wszystkich uprawnień do tabeli [dbo].[BuildVersion].

[2] Możliwe jest też utworzenie loginu reprezentującego całą grupę użytkowników Windows oraz przełączenie serwera SQL w tryb mieszany, w którym pozwala on zalogować się osobom niemającym kont w systemie Windows. Dokładny opis działania i konfigurowania uwierzytelniania tego serwera wykracza jednak poza zakres tej

książki.

[3] Nie oznacza to jednak, że we wskazanej bazie danych zostanie automatycznie utworzone powiązane z loginem konto użytkownika.

[4] Odradzamy zmienianie nazw kont użytkowników. Nie tylko utrudni to monitorowanie aktywności użytkowników (będziemy musieli pamiętać, że login SQL\Marcin jest w bazie *AdventureWorksLT2012* powiązany z kontem Marcin, a w bazie *Test* z kontem Michał), lecz także będzie wprowadzało w błąd samych użytkowników.

[5] W serwerze SQL 2011 konta do ról dodaje się, wywołując specjalną procedurę systemową.

[6] Lista uprawnień systemowych zależy od serwera bazodanowego.

[7] Lista uprawnień obiektowych zależy od serwera bazodanowego.

[8] W standardzie SQL3 słowo kluczowe ALL zastępuje wszystkie możliwe do nadania danemu obiekowi uprawnienia. W serwerze SQL 2011 ten sam efekt uzyskamy, nadając specjalne uprawnienie CONTROL.

Dodatki

Dodatek A Rozwiązania zadań

Zadania z rozdziału 1.

Zadanie 1.

W tabelach słownikowych umieszcza się kolumny przechowujące skończoną i z reguły niewielką liczbę zarówno różnych, jak i powtarzających się wartości. W przypadku tabeli Płyty byłyby to kolumny Nazwa gatunku (liczba gatunków muzycznych jest skończona), Narodowość wykonawcy (lista krajów, z których może pochodzić wykonawca, na pewno jest skończona) i Nazwa wykonawcy (jeden wykonawca jest autorem wielu płyt, a więc jego nazwa będzie się powtarzała). Każda płyta może mieć inną opinię, a zatem kolumna Opinia nie powinna być przeniesiona do osobnej tabeli. Natomiast czas trwania, chociaż się powtarza, to zapisany jest jako liczba minut — tworzenie odrębnej tabeli po to, by zapisać w niej liczby, prawie zawsze jest błędem:

Płyty {ID płyty, ID wykonawcy, ID gatunku, Czas trwania, Opinia}

Wykonawcy {ID wykonawcy, Nazwa wykonawcy, ID kraju}

Gatunki {ID gatunku, Nazwa gatunku}

Kraje {ID kraju, Nazwa kraju}

W tym momencie mamy trzy tabele połączone związkami typu „jeden do wielu”:

1. Wielu wykonawców może pochodzić z tego samego kraju.
2. Wykonawca może nagrać wiele płyt, ale płyta może mieć tylko jednego wykonawcę.
3. Wiele płyt może należeć do tego samego gatunku, ale płyta nie może być jednocześnie zaklasyfikowana do kilku gatunków muzycznych.

Zadanie 2.

Dodając do tabeli Książki kolumnę ID autora, połączymy ją z tabelą Autorzy związkiem typu „jeden do wielu” — autor może napisać wiele książek (wartości kluczy obcych mogą się powtarzać), ale książka

może mieć tylko jednego autora. **Rozwiążanie tego problemu poprzez dodanie do tabeli** Książki kilku kolejnych kolumn, w których zapisywane będą identyfikatory współautorów, jest najgorszym z możliwych:

1. Liczba współautorów książki nie może być większa niż liczba dodatkowych kolumn.
2. Ponieważ większość książek ma jednego autora, duża liczba pól tabeli będzie pusta.
3. Analiza tak zapisanych danych jest bardzo skomplikowana. Na przykład żeby policzyć książki każdego autora, trzeba sprawdzić, czy jego identyfikator nie występuje w różnych kolumnach.

Związki typu „wiele do wielu” (jeden autor może napisać wiele książek, a każda książka może mieć wielu autorów) **implementuje się za pomocą dodatkowej tabeli łącznikowej, w której umieszcza się klucze obce obu łączonych tabel:**

Autorzy {ID autora, Imię, Nazwisko}

Książki {ID książki, Tytuł}

AutorKsiążka {ID autora, ID książki}

Zadanie 3.

Podaną tabelę można przekształcić na kilka sposobów. Żeby doprowadzić ją do 1PN, należy rozbić kolumnę, w której są przechowywane wartości nieatomowe, czyli kolumnę Adres, na kilka kolumn i dodać do tabeli klucz podstawowy:

Uczniowie {Uczeń ID, Imię, Nazwisko, Miasto, Kod pocztowy, Ulica i nr domu, Ocena, Data wystawienia oceny, Uwagi}

Zgodnie z wymogami 2PN należy przenieść do odrębnych tabel te kolumny, w których są przechowywane wartości atrybutów obiektów innych niż uczniowie, i połączyć tak powstałe tabele za pomocą kluczy obcych:

Uczniowie {Uczeń ID, Imię, Nazwisko, Adres ID, Ocena, Data wystawienia oceny, Uwagi}

Adresy {Adres ID, Miasto, Kod pocztowy, Ulica i nr domu}

W ten sposób uczeń może mieć tylko jeden adres, ale pod tym samym adresem może mieszkać wielu uczniów.

Ponieważ data wystawienia oceny jest atrybutem ucznia, który ją dostał, i oceny, którą on otrzymał (a więc występują przechodnie

zależności pomiędzy atrybutami), tabela Uczniowie nie spełnia wymogów 3PN. Żeby doprowadzić ją do tej postaci, należy przenieść informacje o ocenach do osobnej tabeli. Skoro jeden uczeń może otrzymać wiele ocen, a ta sama ocena (np. dobry +) może być wystawiona wielu uczniom (czyli mamy tu do czynienia ze związkiem typu „wiele do wielu”), tabele Uczniowie i Oceny musimy połączyć za pomocą dodatkowej tabeli, w której obok kluczy obcych umieścimy informacje o dacie wystawienia uczniowi danej oceny:

Uczniowie {Uczeń ID, Imię, Nazwisko, Adres ID, Uwagi}

Oceny {Ocena ID, Ocena}

UczniowieOceny {Uczeń ID, Ocena ID, Data wystawienia oceny}

Adresy {Adres ID, Miasto, Kod pocztowy, Ulica i nr domu}

Zadania z rozdziału 2.

Zadanie 1.

Dzięki temu, że pojedyncza instrukcja języka SQL może odczytać wiele danych, prawidłowa implementacja tego algorytmu sprowadza się do poniższego zapytania:

```
SELECT *  
FROM Osoby  
WHERE Nazwisko = 'Nowak';
```

Zadanie 2.

Wartość NULL nie jest ani równa innym wartościom, ani od nich różna, w tym od samej siebie. Wynikiem dowolnych porównań z wartością NULL jest wartość nieznana, a więc pierwsze zapytanie nie zwróci żadnych danych.

Ponieważ wynikiem dowolnej operacji z wartością NULL, w tym dodawania iłączenia ciągów znaków, jest NULL, to niezależnie od tego, jak zostanie zinterpretowany operator +, wynikiem drugiego zapytania będzie NULL.

Zadanie 3.

Ma wpływ na bezpieczeństwo, i to ogromny. Jeżeli aplikacja kliencka nie sprawdzi podanych przez użytkownika danych (np. loginu lub

hasła) i wyśle je do serwera bazodanowego, to serwer zinterpretuje otrzymany ciąg znaków i jeśli będzie on poprawną instrukcją języka SQL, wykona ją. Ta technika ataku nazywa się iniekcją SQL i pozwala na:

1. Poznanie struktury bazy danych — jeżeli błędy zgłasiane przez serwer baz danych zostaną odesłane do przeglądarki, atakujący zdobędzie informacje o nazwach tabel i kolumn, typach poszczególnych kolumn, istniejących procedurach składowanych, funkcjach i tak dalej.
2. Poszerzenie posiadanych uprawnień — wpisana przez atakującego instrukcja SQL zostanie wykonana w kontekście zabezpieczeń innego, często uprzywilejowanego konta użytkownika.
3. Wykonywanie na bazie danych dowolnych instrukcji SQL — jeżeli aplikacja WWW nawiąże połączenie z bazą w kontekście konta administratora, to każda wpisana przez użytkownika i wysłana bez sprawdzania do bazy instrukcja SQL zostanie wykonana.
4. Wywoływanie procedur składowanych — serwery baz danych zawierają procedury składowane, niektóre z nich rozszerzają funkcjonalność serwera (np. wywołują dowolny skrypt powłoki). Atakujący wywołując te procedury, uzyska uprzywilejowany dostęp do zasobów systemowych.

Na przykład jeżeli do sprawdzania tożsamości użytkownika program używa poniższej instrukcji:

```
SELECT *  
FROM Pracownicy  
WHERE login = '' and hasło = '';
```

to gdyby zamiast nazwy użytkownika w zmiennej `Login` został zapisany ciąg znaków `';DROP TABLE Dane --'`, serwer bazodanowy wykonałby następujące instrukcje:

```
SELECT *  
FROM Pracownicy  
WHERE login = '';  
DROP TABLE Dane--' and txtHasło = ''
```

Dzięki apostrofowi wpisane wyrażenie zostanie zinterpretowane, a

nie potraktowane jako ciąg znaków, a więc instrukcja `DROP TABLE Dane` (w języku SQL znak końca wiersza jest ignorowany) będzie pomyślnie wykonana.

Z kolei aby zalogować się bez podawania hasła, wystarczy zamiast nazwy użytkownika wpisać ciąg znaków `'OR 1=1--'`. W efekcie wykonana zostałaby poniższa instrukcja:

```
SELECT *
FROM Pracownicy
WHERE login = ''OR 1=1--' and txtHasło = '';
```

Ponieważ wartość jednego z argumentów operatora `OR` (alternatywy logicznej) będzie prawdą (`1=1`), wartość drugiego (`login = ''`) nie będzie miała wpływu na wynik całego testu logicznego.

Zadania z rozdziału 3.

Zadanie 1.

Żeby rozwiązać zadanie, wystarczy zastanowić się, jakie dane chcemy odczytać, a następnie zapisać zwracające je wyrażenia w klauzuli `SELECT`:

1. Nazwę produktów wystarczy odczytać z kolumny `Name`.
2. Żeby dodać 20% do wartości `ListPrice`, można ją pomnożyć przez 1,2.

Gotowe zapytanie wygląda następująco:

```
SELECT [Name], [ListPrice]*1.2
FROM [SalesLT].[Product];
```

Zadanie 2.

Liczę dni dzielącą dwie daty zwraca funkcja `DATEDIFF(DAY, [ShipDate], [OrderDate])`.

Ponieważ interesuje nas tylko czas, który upłynął pomiędzy złożeniem a wysłaniem zrealizowanych zamówień (a więc takich, które zostały już wysłane), zapytanie może wyglądać następująco:

```
SELECT [SalesOrderID], DATEDIFF (DAY,[OrderDate],[ShipDate])
FROM [SalesLT].[SalesOrderHeader];
```

Trochę trudniejsze byłoby napisanie instrukcji, gdybyśmy chcieli otrzymać liczbę dni, przez które realizowane są wszystkie zamówienia. W przypadku niewysłanych jeszcze zamówień w kolumnie ShipDate znajdowałaby się wartość NULL, a wszystkie wyrażenia z wartością NULL zwracają NULL. Musielibyśmy więc zastąpić wartość NULL bieżącą datą — w serwerze SQL do zastępowania wartości NULL służy funkcja COALESCE(), a bieżącą datę zwraca funkcja GETDATE():

```
SELECT [SalesOrderID], DATEDIFF (DAY,  
[OrderDate],COALESCE([ShipDate],GETDATE()))  
FROM [SalesLT].[SalesOrderHeader];
```

Zadanie 3.

Wynik zapytania ma się składać z trzech kolumn:

1. W pierwszej znajduje się odczytana z tabeli nazwa produktu poprzedzona słowem Produkt.
2. W drugiej kolumnie jest stała kosztuje.
3. W trzeciej — zaokrąglona do jednego miejsca po przecinku cena.

Nazwa odczytywanej tabeli została podana, czyli możemy już utworzyć zapytanie z klauzulami SELECT i FROM:

```
SELECT 'Produkt ' + [Name], 'kosztuje', ROUND([ListPrice],1)  
FROM [SalesLT].[Product];
```

Widzimy również, że wynik zapytania jest posortowany malejąco według cen, czyli należy dodać klauzulę ORDER BY:

```
SELECT 'Produkt ' + [Name], 'kosztuje', ROUND([ListPrice],1)  
FROM [SalesLT].[Product]  
ORDER BY [ListPrice] DESC;
```

Zadanie 4.

Zadanie może wydawać się bardzo łatwe — wystarczy odczytać kolumnę OrderDate tabeli SalesOrderHeader i posortować wynik malejąco według tej właśnie kolumny:

```
SELECT [OrderDate]  
FROM [SalesLT].[SalesOrderHeader]  
ORDER BY [OrderDate] DESC;
```

```
-----  
OrderDate  
2014-04-01 00:00:00.000  
2008-06-01 00:00:00.000  
2008-06-01 00:00:00.000  
2008-06-01 00:00:00.000  
2008-06-01 00:00:00.000
```

...

Jednak otrzymany wynik zawiera znacznik czasu, w dodatku niektóre daty (te, w których przyjęto kilka zamówień) wielokrotnie się powtarzają. Spróbujmy najpierw wyeliminować znacznik czasu.

Zakładając, że możemy używać tylko opisanych w rozdziale funkcji, można spróbować obciąć pierwsze jedenaście znaków kolumny OrderDate:

```
SELECT LEFT([OrderDate],11)  
FROM [SalesLT].[SalesOrderHeader]  
ORDER BY [OrderDate] DESC;
```

(No column name)

```
Apr 1 2014  
Jun 1 2008  
Jun 1 2008  
Jun 1 2008
```

...

Wydaje się, że udało nam się wyeliminować znacznik czasu, ale problem wynikający z zastosowania takiego rozwiązania pojawi się, gdy tylko spróbujemy wyeliminować duplikaty:

```
SELECT DISTINCT LEFT([OrderDate],11)  
FROM [SalesLT].[SalesOrderHeader]  
ORDER BY [OrderDate] DESC;
```

Msg 145, Level 15, State 1, Line 1

```
ORDER BY items must appear in the select list if SELECT DISTINCT is  
specified.
```

Jeżeli w zapytaniu użyte jest słowo kluczowe DISTINCT, to w klauzuli ORDER BY nie możemy umieścić żadnych wyrażeń niewystępujących w klauzuli SELECT. Spróbujmy więc skopiować do klauzuli ORDER BY wywołanie funkcji LEFT():

```
SELECT DISTINCT LEFT([OrderDate],11)
FROM [SalesLT].[SalesOrderHeader]
ORDER BY LEFT([OrderDate],11) DESC;
```

```
-----  
(No column name)
```

```
Jun 28 1905
```

```
Jun 1 2008
```

```
Apr 1 2014
```

Zapytanie jest teraz poprawne składniowo i zwraca właściwy wynik, ale nie używa wymienionych w zadaniu funkcji YEAR(), MONTH() i DAY().

Skorzystajmy zatem z tych funkcji i odczytajmy osobno rok, miesiąc i dzień, łącząc wywołania wszystkich trzech funkcji w jedną kolumnę:

```
SELECT DISTINCT YEAR([OrderDate]) + '-' + MONTH([OrderDate]) + '-' +
DAY([OrderDate])
FROM [SalesLT].[SalesOrderHeader]
ORDER BY YEAR([OrderDate]) + '-' + MONTH([OrderDate]) + '-' +
DAY([OrderDate]) DESC;
```

```
-----  
(No column name)
```

```
2019
```

```
2015
```

```
1939
```

Zapytanie działa, ale co oznaczają zwrócone przez nie liczby?

Tym razem natrafiliśmy na problem niejawniej konwersji typów — od numeru miesiąca (liczby) został odjęty numer dnia (też liczba), a od otrzymanego w ten sposób wyniku odjęto rok (też zapisany jako dane liczbowe). Musimy więc jawnie określić typy tych danych jako znakowe, przy czym rok ma zawsze długość czterech znaków, ale numery dni i miesięcy mogą być jedno- lub dwuznakowe:

```
SELECT DISTINCT CAST(YEAR([OrderDate]) AS CHAR(4))+'-
'+CAST(MONTH([OrderDate]) AS VARCHAR(2))+'-' +CAST(DAY([OrderDate]) AS
```

```
VARCHAR(2))  
FROM [SalesLT].[SalesOrderHeader]  
ORDER BY CAST(YEAR([OrderDate]) AS CHAR(4))+'-' +CAST(MONTH([OrderDate])  
AS VARCHAR(2))+'-' +CAST(DAY([OrderDate]) AS VARCHAR(2)) DESC;  
-----  
(No column name)  
2014-4-1  
2008-6-1  
1905-6-28
```

Zadanie 5.

Zacznijmy od odczytania kolumn ProductNumber i Size oraz posortowania wyniku rosnąco według wartości kolumny Size:

```
SELECT [ProductNumber], [Size]  
FROM [SalesLT].[Product]  
ORDER BY [Size];
```

```
-----  
ProductNumber          Size  
HL-U509-R             NULL  
HL-U509                NULL  
HL-U509-B              NULL
```

...

Przekonaliśmy się właśnie, że podczas sortowania wartość `NULL` jest uznawana za mniejszą od jakiegokolwiek znanej wartości. Gdybyśmy mogli zmienić kolejność sortowania na malejącą, zadanie byłoby rozwiązane. Skoro jednak nie możemy tego zrobić, musimy podczas sortowania zastąpić `NULL` inną wartością. Takie warunkowe zmiany danych umożliwia funkcja `CASE`, co możemy sprawdzić, wykonując poniższe zapytanie:

```
SELECT [ProductNumber], [Size],  
       CASE  
           WHEN [Size] IS NULL THEN 1  
           ELSE 0  
       END  
FROM [SalesLT].[Product];
```

ProductNumber	Size	(No column name)
FR-R92B-58	58	0
FR-R92R-58	58	0
HL-U509-R	NULL	1
HL-U509	NULL	1
...		

Wynikiem naszego wyrażenia jest 1 (jeżeli w kolumnie `Size` była wartość `NULL`) lub 0 (jeżeli ta wartość była określona). Czyli jest to wyrażenie skalarne i jako takie może być użyte w klauzuli `ORDER BY`:

```
SELECT [ProductNumber], [Size]
FROM [SalesLT].[Product]
ORDER BY CASE
    WHEN [Size] IS NULL THEN 1
    ELSE 0
END;
```

ProductNumber	Size
BK-R19B-48	48
BK-R19B-52	52
NW-1S	M
FD-2342	NULL
HB-T721	NULL
...	

Wydaje się, że zadanie zostało rozwiążane. Na pewno?

Rzeczywiście wartości `NULL` znalazły się na końcu wyniku. Ale co z sortowaniem wierszy, w których wartość `Size` była określona? Przecież w żaden sposób ich nie posortowaliśmy.

Na szczęście wynik zapytania może być sortowany według wartości wielu kolumn lub wyrażeń. W tym przypadku chcielibyśmy w pierwszej kolejności przenieść wiersze z wartością `NULL` na początek, a następnie posortować je rosnąco według wartości kolumny `Size`:

```

SELECT [ProductNumber], [Size]
FROM [SalesLT].[Product]
ORDER BY CASE
    WHEN [Size] IS NULL THEN 1
    ELSE 0
END,
[Size];
-----
```

ProductNumber	Size
FR-M94B-38	38
FR-M94S-38	38
BK-M82S-38	38
BK-M82B-38	38

...

Zadania z rozdziału 4.

Zadanie 1.

Sprawdzić, czy koszt jest ponad dwukrotnie niższy od ceny, możemy za pomocą warunku `[StandardCost]*2<[ListPrice]`. Dodatkowo zapytanie ma zwracać tylko nazwy produktów o identyfikatorach kończących się znakami 4 lub 8 — ten warunek możemy zapisać z użyciem operatora `LIKE` lub `IN`.

Zacznijmy od wersji z operatorem `LIKE`: Kod produktu `LIKE '%4'` OR Kod produktu `LIKE '%8'`. Ponieważ muszą być spełnione oba warunki, należy połączyć je operatorem `AND`:

```

SELECT [Name], [ProductNumber], [ListPrice]
FROM [SalesLT].[Product]
WHERE [StandardCost]*2<[ListPrice]
AND [ProductName] LIKE '%4' OR [ProductName] LIKE '%8';
-----
```

Name	ProductNumber	ListPrice
HL Road Frame - Black, 58	FR-R92B-58	1431,50
HL Road Frame - Red, 58	FR-R92R-58	1431,50

AWC Logo Cap	CA-1098	8,99
HL Road Frame - Red, 48	FR-R92R-48	1431,50
LL Road Frame - Black, 58	FR-R38B-58	337,22
...		

Jednak to zapytanie zwraca błędny wynik — znajdziemy w nim nazwy produktów o numerach kończących się na 4, lecz o zbyt wysokim koszcie w stosunku do ich ceny. Spowodowane jest to wykonywaniem operatorów o tym samym priorytecie od lewej do prawej. W tym przypadku kolejność wykonywania wyrażeń i warunków z klauzuli WHERE będzie następująca:

1. Cena jest mnożona przez dwa.
2. Podwojona cena jest porównywana z kosztem.
3. Sprawdzana jest zgodność numeru produktu ze wzorcem '%4'.
4. Przeprowadzana jest koniunkcja obu powyższych warunków.
5. Sprawdzana jest zgodność numeru produktu ze wzorcem '%8'.
6. Wiersze, które spełniły ostatni warunek, są dodawane do wyniku zapytania.

Czyli w niewłaściwej kolejności są wykonywane operacje koniunkcji i alternatywy. Możemy naprawić wykryty błąd za pomocą nawiasów:

```
SELECT [Name], [ProductNumber], [ListPrice]
FROM [SalesLT].[Product]
WHERE [StandardCost]*2<[ListPrice]
AND RIGHT([ProductNumber],1) IN ('4','8');
```

Name	ProductNumber	ListPrice
HL Mountain Handlebars	HB-M918	120,27
LL Road Handlebars	HB-R504	44,54
HL Mountain Front Wheel	FW-M928	300,215
HL Mountain Rear Wheel	RW-M928	327,215
Mountain Pump	PU-M044	24,99

Możemy też zastąpić operator LIKE operatorem IN — ponieważ mamy sprawdzić, czy ostatni znak należy do podanego zbioru wartości, użyjemy też funkcji RIGHT():

```
SELECT [Name], [ProductNumber], [ListPrice]
```

```
FROM [SalesLT].[Product]
WHERE [StandardCost]*2<[ListPrice]
AND (RIGHT([ProductNumber],1) IN ('4','8'));
```

```
-----
```

Name	ProductNumber	ListPrice
HL Mountain Handlebars	HB-M918	120,27
LL Road Handlebars	HB-R504	44,54
HL Mountain Front Wheel	FW-M928	300,215
HL Mountain Rear Wheel	RW-M928	327,215
Mountain Pump	PU-M044	24,99

Zadanie 2.

Zacznijmy od wybrania zamówień z pierwszego kwartału 2008 roku. Ponieważ mamy sprawdzić, czy data należy do określonego przedziału, użyjemy operatora BETWEEN ... AND:

```
SELECT [SalesOrderID], [Freight]
FROM [SalesLT].[SalesOrderHeader]
WHERE [OrderDate] BETWEEN '20080601' AND '20090101';
```

```
-----
```

SalesOrderID	Freight	OrderDate
71774	22,0087	2008-06-01 00:00:00.000
71776	1,9703	2008-06-01 00:00:00.000
71780	960,4672	2008-06-01 00:00:00.000

...

Ten wynik może jednak zawierać również zamówienia z 1 stycznia 2009. Jako że operator BETWEEN ... AND pozwala sprawdzić, czy wartość nie należy do obustronnie domkniętego przedziału, musimy zmienić jego drugi argument:

```
SELECT [SalesOrderID], [Freight],[OrderDate]
FROM [SalesLT].[SalesOrderHeader]
WHERE [OrderDate] BETWEEN '20080601' AND '20081231';
```

Pozostało nam tylko wybranie z otrzymanego zbioru 5% zamówień o najniższym koszcie wysyłki. Możemy to zrobić za pomocą klauzuli TOP, sortując wynik zapytania rosnąco według kolumny Freight:

```
SELECT TOP 5 PERCENT [SalesOrderID], [Freight]
FROM [SalesLT].[SalesOrderHeader]
WHERE [OrderDate] BETWEEN '20080601' AND '20081231'
ORDER BY Freight;
```

To zapytanie wydaje się poprawne, ale będzie zwracało prawidłowe wyniki tylko wtedy, gdy użyta do sortowania kolumna nie będzie zawierała powtórzonych danych. Ponieważ w tym przypadku koszty wysyłki różnych zamówień są takie same, powinniśmy dodać do jego wyniku powtórzone wiersze:

```
SELECT TOP 5 PERCENT WITH TIES [SalesOrderID], [Freight]
FROM [SalesLT].[SalesOrderHeader]
WHERE [OrderDate] BETWEEN '20080601' AND '20081231'
ORDER BY Freight;
```

Zadanie 3.

Wydaje się, że rozwiążaniem tego zadania będzie użycie w klauzuli WHERE warunku, który będzie spełniony przez przypadkowy wiersz tabeli Product. Gdybyśmy np. generowali liczby losowo, moglibyśmy sprawdzać, czy są one równe identyfikatorom produktów, i za każdym razem powinniśmy otrzymać nazwę innego towaru. Ponieważ identyfikatory towarów są liczbami całkowitymi, a funkcja RAND() zwraca liczbę dziesiętną z zakresu od 0 do 1, jej wynik należy pomnożyć przez największą możliwą wartość i przekonwertować tak otrzymaną liczbę na liczbę całkowitą:

```
SELECT [Nazwa produktu]
FROM dbo.Produkty
WHERE ID=CAST(RAND()*99 AS INT);
```

Okazuje się jednak, że nasze zapytanie czasami nie zwraca żadnych danych. Wynika to z tego, iż identyfikatory produktów są niepowtarzalne, ale nie są ciągłe (m.in. brakuje numerów z zakresu od 0 do 600). Skoro nie wiemy, jakie są numery produktów, nie rozwiążemy w ten sposób zadania.

Drugą poznaną klauzulą, pozwalającą ograniczyć liczbę wierszy wyniku zapytania, jest TOP. Zawierające tę klauzulę zapytanie, które zwraca nazwę tylko jednego produktu, wygląda następująco:

```
SELECT TOP 1 [Name]
```

```
FROM [SalesLT].[Product];
```

Name

All-Purpose Bike Stand

To zapytanie zwraca przypadkową, ale zawsze tę samą nazwę produktu. Musimy więc dodać do niego klauzulę ORDER BY i losowo posortować w niej wiersze wyniku:

```
SELECT TOP 1 [Name]  
FROM [SalesLT].[Product]  
ORDER BY RAND();
```

Name

All-Purpose Bike Stand

Niestety, to zapytanie też ciągle zwraca tę samą nazwę produktu. Spowodowane jest to tym, że funkcje bez parametrów są wywoływanie raz dla całego zapytania, a nie dla każdego zwracanego przez nie wiersza, a więc do sortowania jest używana po prostu jedna liczba z zakresu od 0 do 1. Na szczęście jest taka funkcja, którą serwer bazodanowy wywołuje dla każdego wiersza zapytania — użycie jej w klauzuli ORDER BY rozwiąże problem:

```
SELECT TOP 1 [Name]  
FROM [SalesLT].[Product]  
ORDER BY NEWID();
```

Name

LL Touring Handlebars

Zadania z rozdziału 5.

Zadanie 1.

Nazwy produktów odczytamy z kolumny Name tabeli Product, natomiast imię klienta sprawdzimy, porównując je z danymi z kolumny FirstName tabeli Customer. Te tabele jednak nie są ze sobą bezpośrednio połączone, więc poniższe zapytanie zwrócił błędny wynik:

```
SELECT P.Name
```

```
FROM [SalesLT].[Product] AS P,[SalesLT].[Customer] AS C  
WHERE C.FirstName = 'Jeffrey';
```

Name

All-Purpose Bike Stand

AWC Logo Cap

Bike Wash - Dissolver

Cable Lock

Chain

...

Musimy naturalnie złączyć obie potrzebne nam do rozwiązania zadania tabele. W tym celu tabelę Product złączymy z tabelą SalesOrderDetail, a tabelę Customer z tabelą SalesOrderHeader. Na końcu złączymy ze sobą otrzymywane zbiory pośrednie i wyeliminujemy wiersze niespełniające warunku FirstName = 'Jeffrey':

```
SELECT P.Name  
FROM [SalesLT].[Product] AS P  
    JOIN [SalesLT].[SalesOrderDetail] AS OD  
        ON P.ProductID = OD.ProductID  
    JOIN [SalesLT].[SalesOrderHeader] AS OH  
        ON OD.SalesOrderID=OH.SalesOrderID  
    JOIN [SalesLT].[Customer] AS C  
        ON C.CustomerID=OH.CustomerID  
WHERE C.FirstName = 'Jeffrey';
```

Otrzymany wynik może jednak zawierać wiele powtórzeń nazw produktów — każda nazwa powtórzona będzie tyle razy, ile razy ten towar został sprzedany temu klientowi. Pozostało nam więc wyeliminować powtarzające się wiersze:

```
SELECT DISTINCT P.Name  
FROM [SalesLT].[Product] AS P  
    JOIN [SalesLT].[SalesOrderDetail] AS OD  
        ON P.ProductID = OD.ProductID  
    JOIN [SalesLT].[SalesOrderHeader] AS OH  
        ON OD.SalesOrderID=OH.SalesOrderID
```

```
JOIN [SalesLT].[Customer] AS C  
    ON C.CustomerID=OH.CustomerID  
WHERE C.FirstName = 'Jeffrey';
```

Zadanie 2.

Dane klientów są zapisane w kolumnach [FirstName] i [LastName] tabeli Customer. Jednak rozwiążanie zadania wymaga złączenia tej tabeli z tabelą, w której zapisane są operacje sprzedaży — tylko w ten sposób dowiemy się, które zamówienia złożył dany klient, a więc będziemy mogli znaleźć klientów, którzy nie złożyli jeszcze żadnego zamówienia:

```
SELECT [FirstName], [LastName]  
FROM [SalesLT].[Customer] AS C  
    JOIN [SalesLT].[SalesOrderHeader] AS OH  
        ON C.CustomerID=OH.CustomerID  
    WHERE OH.CustomerID IS NULL;
```

To zapytanie nie zwróciło jednak żadnych danych. Powodem jest nieprawidłowy typ złączenia — złączenie wewnętrzne wyeliminowało z wyniku niepasujące wiersze, a zatem warunek z klauzuli WHERE był nieprawdziwy dla każdego otrzymanego wiersza. Zastosowanie lewostronnego złączenia zewnętrznego rozwiąże problem:

```
SELECT [FirstName], [LastName]  
FROM [SalesLT].[Customer] AS C  
    LEFT OUTER JOIN [SalesLT].[SalesOrderHeader] AS OH  
        ON C.CustomerID=OH.CustomerID  
    WHERE OH.CustomerID IS NULL;
```

FirstName	LastName
Orlando	Gee
Keith	Harris
Donna	Carreras
Janet	Gates

...

Zadanie 3.

Skoro wynik zapytania ma zawierać stałą wybraną na podstawie warunku logicznego, a w zapytaniu nie możemy użyć funkcji CASE, pozostaje nam rozbicie tego zapytania na dwa i połączenie ich wyników. Pierwsze zapytanie będzie zwracało dane o dużych zamówieniach:

```
SELECT SalesOrderID, Freight, 'High'  
FROM [SalesLT].[SalesOrderHeader]  
WHERE Freight > 100;
```

```
-----  
SalesOrderID      Freight      (No column name)  
71780            960,4672    High  
71782            994,6333    High  
71783            2096,4607    High
```

...

a drugie — o zamówieniach, których opłata nie przekroczyła 100:

```
SELECT SalesOrderID, Freight, 'Low'  
FROM [SalesLT].[SalesOrderHeader]  
WHERE Freight <= 100;
```

```
-----  
SalesOrderID      Freight      (No column name)  
71774            22,0087     Low  
71776            1,9703      Low  
71815            28,5395     Low
```

Łącząc wyniki obu zapytań, rozwiążemy zadanie (w tym przypadku mamy pewność, że wyniki obu zapytań są rozłączne, a więc należy połączyć je operatorem UNION ALL);

```
SELECT SalesOrderID, Freight, 'High'  
FROM [SalesLT].[SalesOrderHeader]  
WHERE Freight > 100  
UNION ALL  
SELECT SalesOrderID, Freight, 'Low'  
FROM [SalesLT].[SalesOrderHeader]  
WHERE Freight <= 100;
```

SalesOrderID	Freight	(No column name)
71936	2456,9673	High
71938	2220,3216	High
71774	22,0087	Low
71776	1,9703	Low
71815	28,5395	Low

Zadania z rozdziału 6.

Zadanie 1.

Wartość największej opłaty za wysyłkę odczytamy następująco:

```
SELECT MAX([Freight])
FROM [SalesLT].[SalesOrderHeader];
-----
(No column name)
2714,0458
```

Naszym zadaniem jest jednak odczytanie wartości największych opłat za wysyłkę zamówień zrealizowanych w poszczególnych dniach, a więc musimy pogrupować dane według dat zamówień:

```
SELECT MAX([Freight])
FROM [SalesLT].[SalesOrderHeader]
GROUP BY [OrderDate];
-----
(No column name)
1,00
2714,0458
22,0087
```

Dodatkowo dla każdego dnia mamy znaleźć największe zamówienia dla poszczególnych klientów, czyli musimy utworzyć podgrupy i dla nich wywołać funkcję MAX():

```
SELECT MAX([Freight])
FROM [SalesLT].[SalesOrderHeader]
GROUP BY [OrderDate],[CustomerID];
-----
```

(No column name)

1,00

994,6333

165,8574

2220,3216

...

Do wyniku zapytania należy jeszcze dodać użyte do grupowania kolumny:

```
SELECT [OrderDate], [CustomerID], MAX([Freight])
FROM [SalesLT].[SalesOrderHeader]
GROUP BY [OrderDate], [CustomerID];
```

```
-----  
OrderDate          CustomerID      (No column name)  
1905-06-28 00:00:00.000    448           1,00  
2008-06-01 00:00:00.000    29485         994,6333  
2008-06-01 00:00:00.000    29531         165,8574
```

...

 Wskazówka	Spróbuj samodzielnie przedstawić otrzymany wynik w postaci tabeli przestawnej.
---	--

Zadanie 2.

W tabeli SalesOrderDetail nie znajdziemy nazw produktów, czyli musimy połączyć ją z tabelą Product:

```
SELECT P.[Name], OD.[ProductID]
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
ON p.ProductID=od.ProductID;
```

```
-----  
Name          ProductID  
Sport-100 Helmet, Red    707  
Sport-100 Helmet, Red    707  
Sport-100 Helmet, Red    707
```

Sport-100 Helmet, Red	707
Sport-100 Helmet, Black	708

...

Teraz możemy już policzyć za pomocą funkcji COUNT(), ile razy zamawiany był każdy produkt:

```
SELECT P.[Name], COUNT(OD.[ProductID])
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
ON p.ProductID=od.ProductID
GROUP BY P.[Name];
```

Name	(No column name)
AWC Logo Cap	9
Bike Wash - Dissolver	7
Chain	4
Classic Vest, M	6
Classic Vest, S	10

...

Pozostało nam wyeliminowanie z wyniku zapytania tych modeli, dla których funkcja COUNT() zwróciła wartość równą lub mniejszą od 3:

```
SELECT P.[Name], COUNT(OD.[ProductID])
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[SalesOrderDetail] AS OD
ON p.ProductID=od.ProductID
GROUP BY P.[Name]
HAVING COUNT(P.[Name])>3;
```

Name	(No column name)
AWC Logo Cap	9
Bike Wash - Dissolver	7

...

Zadanie 3.

Zadanie polega na użyciu operatora PIVOT. Żeby wyświetlić informacje o wartościach zamówień złożonych przez poszczególnych klientów w ciągu kolejnych miesięcy, należy pogrupować dane według kolumny Miesiąc, a dane odczytane z kolumny ID klienta przenieść do czterech (tylko mamy klientów) dodatkowych kolumn:

```
SELECT *
FROM #Sprzedaz
PIVOT (SUM(Wartość)
FOR [Miesiąc] IN
([4], [6]))
AS piv;
```

ID klienta	4	6
29741	NULL	43,0437
29781	NULL	117,7276
29796	NULL	86222,8072
29847	987,785	972,785

...

Zadania z rozdziału 7.

Zadanie 1.

Policzyć, ile razy sprzedany został każdy produkt, możemy, używając znanej z poprzedniego rozdziału funkcji grupującej COUNT:

```
SELECT [ProductNumber], COUNT([SalesOrderDetailID]) AS SalesCount
FROM [SalesLT].[SalesOrderDetail] AS OD
JOIN [SalesLT].[Product] AS P
ON P.ProductID=OD.ProductID
GROUP BY [ProductNumber];
```

ProductName	SalesCount
-------------	------------

BB-7421	3
BB-9108	4
BK-M18B-40	2
BK-M18B-42	2
BK-M18B-44	2

...

Żeby dodać do wyniku numery wierszy i wartości, wystarczy w klauzuli SELECT wywołać odpowiednie funkcje rankingu (ROW_NUMBER oraz DENSE_RANK). Funkcje rankingu wymagają posortowania wierszy — w tym przypadku mają one być posortowane malejąco według wyników funkcji COUNT:

```
SELECT [ProductNumber], COUNT([SalesOrderDetailID]) AS SalesCount
      ,ROW_NUMBER () OVER (ORDER BY COUNT([SalesOrderDetailID]) DESC)
      ,DENSE_RANK () OVER (ORDER BY COUNT([SalesOrderDetailID]) DESC)
FROM [SalesLT].[SalesOrderDetail] AS OD
JOIN [SalesLT].[Product] AS P
ON P.ProductID=OD.ProductID
GROUP BY [ProductNumber]
ORDER BY COUNT([SalesOrderDetailID]) DESC;
```

ProductName name)	SalesCount	(No column name)	(No column
LJ-0192-L	10	1	1
VE-C304-S	10	2	1
SJ-0194-X	9	3	2
CA-1098	9	4	2
SJ-0194-L	8	5	3
RA-H123	8	6	3

...

Zadanie 2.

Zacznijmy od rozbicia dat zamówień na lata, miesiące i dni:

```
SELECT YEAR([DueDate]) AS year, MONTH([DueDate]) AS month, DAY([DueDate])
AS day
FROM [SalesLT].[SalesOrderHeader];
```

year	month	day
2008	7	2
2008	7	12
2008	6	26
2008	6	15

...

Jeśli dysponuje się powyższymi danymi, rozwiązanie zadania jest już proste — wystarczy wywołać funkcję `SUM` dla klauzuli `OVER`, zmieniając za każdym razem poziom grupowania dla tej klauzuli — raz policzymy sumę na poziomie lat, drugi raz miesięcy, trzeci — dni, a pusta klauzula `OVER` pozwoli nam policzyć sumę dla wszystkich zamówień:

```
SELECT YEAR([DueDate]) AS year, MONTH([DueDate]) AS month, DAY([DueDate])
AS day
    ,SUM ([TotalDue]) OVER (PARTITION BY DAY([DueDate])) AS
SalesPerDay
    ,SUM ([TotalDue]) OVER (PARTITION BY MONTH([DueDate])) AS
SalesPerMonth
    ,SUM ([TotalDue]) OVER (PARTITION BY YEAR([DueDate])) AS
SalesPerYear
    ,SUM ([TotalDue]) OVER () AS OverallSales
FROM [SalesLT].[SalesOrderHeader];
```

year	month	day	SalesPerDay	SalesPerMonth	SalesPerYear
OverallSales					
2008	6	28	44688,6943	708766,4186	956303,5949
957276,3799					
2008	6	29	119960,824	708766,4186	956303,5949
957276,3799					
2008	6	14	103348,0081	708766,4186	956303,5949
957276,3799					
2008	7	14	103348,0081	248509,9613	956303,5949
957276,3799					
2008	7	1	2669,3183	248509,9613	956303,5949
957276,3799					

Zadanie 3.

To zadanie można rozwiązać na kilka sposobów. Najbardziej elegancki (najczytelniejszy i jednocześnie wydajny) polega na zastosowaniu funkcji pozwalającej odwołać się do wartości poprzedniego wiersza, czyli funkcji okienkowej LAG:

```
SELECT [SalesOrderID], [TotalDue],  
       LAG ([TotalDue]) OVER (ORDER BY [SalesOrderID])  
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
SalesOrderID      TotalDue      (No column name)  
71774            972,785      NULL  
71776            87,0851      972,785  
71780            42452,6519    87,0851  
71782            43962,7901    42452,6519
```

Jeśli dysponuje się wartością poprzedniego wiersza, wystarczy już tylko odjąć od wartości sprzedaży z bieżącego wiersza wartość wiersza poprzedniego:

```
SELECT [SalesOrderID], [TotalDue],  
       [TotalDue] - LAG ([TotalDue]) OVER (ORDER BY [SalesOrderID])  
FROM [SalesLT].[SalesOrderHeader];
```

```
-----  
SalesOrderID      TotalDue      (No column name)  
71774            972,785      NULL  
71776            87,0851      -885,6999  
71780            42452,6519    42365,5668  
71782            43962,7901    1510,1382
```

...

Zadania z rozdziału 8.

Zadanie 1.

Numery zamówień są zapisane w tabeli SalesOrderHeader. W tej tabeli jedynymi dotyczącymi klientów informacjami są ich identyfikatory. Gdybyśmy znali identyfikator klienta o podanym nazwisku, moglibyśmy odczytać złożone przez niego zamówienia i rozwiązać

zadanie.

Identyfikator klienta zwraca poniższe zapytanie:

```
SELECT [CustomerID]
FROM [SalesLT].[Customer]
WHERE LastName = 'Eminhizer';
-----
```

CustomerID

448

29736

Tak uzyskane numery moglibyśmy wykorzystać w klauzuli WHERE i w ten sposób odczytać numery zamówień złożonych przez daną osobę. My jednak mamy zbudować podzapytanie. Ponieważ zawsze może być kilku klientów z takim samym nazwiskiem, do porównania wyniku podzapytania użyjemy operatora IN:

```
SELECT [SalesOrderID]
FROM [SalesLT].[SalesOrderHeader] AS OH
WHERE OH.CustomerID IN
      (SELECT [CustomerID]
       FROM [SalesLT].[Customer]
       WHERE LastName = 'Eminhizer'
      );
-----
```

SalesOrderID

77775

71784

Zadanie 2.

To zadanie mamy rozwiązać za pomocą podzapytania niepowiązanego. Zaczniemy od odczytania identyfikatorów dziesięciu największych zamówień:

```
SELECT TOP 10 WITH TIES [SalesOrderID], [TotalDue]
FROM [SalesLT].[SalesOrderHeader] AS OH
ORDER BY [TotalDue] DESC;
-----
```

SalesOrderID	TotalDue
71784	119960,824
71936	108597,9536
77775	100003,00
71938	98138,2131
71783	92663,5609
71797	86222,8072
71902	81834,9826
71898	70698,9922
71796	63686,2708
71845	45992,3665

W ten sam sposób możemy odczytać numery i wartości pięciu zamówień o największych opłatach — ponownie dodanie klauzuli WITH TIES pozwoli uwzględnić fakt, że kilka zamówień może mieć tę samą wartość:

```
SELECT TOP 5 WITH TIES [SalesOrderID], [TotalDue]
FROM [SalesLT].[SalesOrderHeader] AS OH
ORDER BY [TotalDue] DESC;
```

SalesOrderID	TotalDue
71784	119960,824
71936	108597,9536
77775	100003,00
71938	98138,2131
71783	92663,5609

Wystarczy jeszcze połączyć oba zapytania w jedno podzapytanie, przy czym pierwsze zapytanie będzie zapytaniem wewnętrznym, a drugie zewnętrznym.

Zapytanie zewnętrzne zwróciłoby dane o pięciu największych zamówieniach. Jeżeli jednak w klauzuli WHERE z wyniku usuniemy informacje o dziesięciu zamówieniach o największej sumarycznej wartości, podzapytanie zwróci dane o zamówieniach z pozycji od 11. do 15. plus ewentualne zamówienia o takich samych opłatach za wysyłkę jak zamówienie 15.:

```
SELECT TOP 5 WITH TIES [SalesOrderID], [TotalDue]
FROM [SalesLT].[SalesOrderHeader] AS OH
WHERE OH.SalesOrderID NOT IN
    (SELECT TOP 10 WITH TIES [SalesOrderID]
     FROM [SalesLT].[SalesOrderHeader] AS OH
     ORDER BY [TotalDue] DESC)
ORDER BY [TotalDue] DESC;
```

SalesOrderID	TotalDue
71782	43962,7901
71780	42452,6519
71832	39531,6085
71858	15275,1977
71897	14017,9083

Zadanie 3.

W pierwszej kolejności należy znaleźć ostatni dzień (czyli największą datę) każdego miesiąca — jeżeli jednak pogrupujemy daty według numerów miesięcy, otrzymany wynik będzie nieprawidłowy:

```
SELECT MAX([OrderDate])
FROM [SalesLT].[SalesOrderHeader] AS OH
GROUP BY MONTH([OrderDate]);
```

(No column name)

2014-04-01 00:00:00.000

2008-06-01 00:00:00.000

Zapytanie zwróciło dwa wiersze z datami ostatnich dni miesięcy, w których realizowana była sprzedaż, ale w tabeli SalesOrderHeader mogą być zapisane informacje z kilku lat. Natrafiliśmy więc na problem analizowany w przykładach poświęconych wyznaczaniu trendów — żeby go rozwiązać, musimy znaleźć właściwy czynnik grupujący.

W tym przypadku do grupowania wystarczy użyć numerów lat — po dodaniu dodatkowego poziomu grupowania wynik będzie poprawny:

```
SELECT MAX([OrderDate])
```

```
FROM [SalesLT].[SalesOrderHeader] AS OH  
GROUP BY YEAR([OrderDate]), MONTH([OrderDate]);
```

(No column name)

2014-04-01 00:00:00.000
1905-06-28 00:00:00.000
2008-06-01 00:00:00.000

Znając ostatnie dni poszczególnych miesięcy, możemy już odczytać informacje o złożonych w tych dniach zamówieniach:

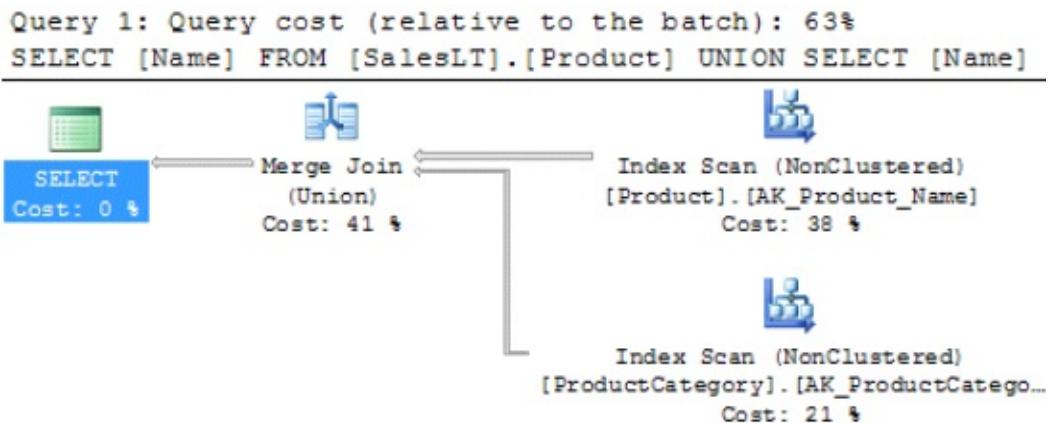
```
SELECT OH.SalesOrderID, OH.OrderDate, OH.CustomerID  
FROM [SalesLT].[SalesOrderHeader] AS OH  
WHERE OH.OrderDate IN  
    (SELECT MAX([OrderDate])  
     FROM [SalesLT].[SalesOrderHeader] AS OH  
     GROUP BY YEAR([OrderDate]), MONTH([OrderDate]));
```

SalesOrderID	OrderDate	CustomerID
71774	2008-06-01 00:00:00.000	29847
71776	2008-06-01 00:00:00.000	30072
71780	2008-06-01 00:00:00.000	30113
71782	2008-06-01 00:00:00.000	29485
71783	2008-06-01 00:00:00.000	29957
...		

Zadania z rozdziału 9.

Zadanie 1.

Jeżeli spojrzymy na plan wykonania tego zapytania, przekonamy się, że najkosztowniejszą operacją jest w nim operacja łączenia MERGE JOIN:



Zastanówmy się, czy można tę kosztowną operację wyeliminować. Jeżeli tylko przyjmiemy, że żaden z produktów nie posiada tej samej nazwy co któraś z kategorii, okaże się, że tak — w takim wypadku nie musimy eliminować duplikatów z wyniku zapytania, wystarczy, że zastąpimy operator UNION znacznie wydajniejszym operatorem UNION ALL:

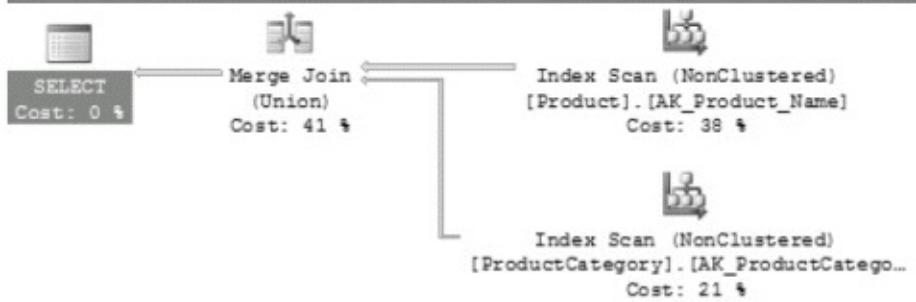
```

SELECT [Name]
FROM [SalesLT].[Product]
UNION ALL
SELECT [Name]
FROM [SalesLT].[ProductCategory];
-----
```

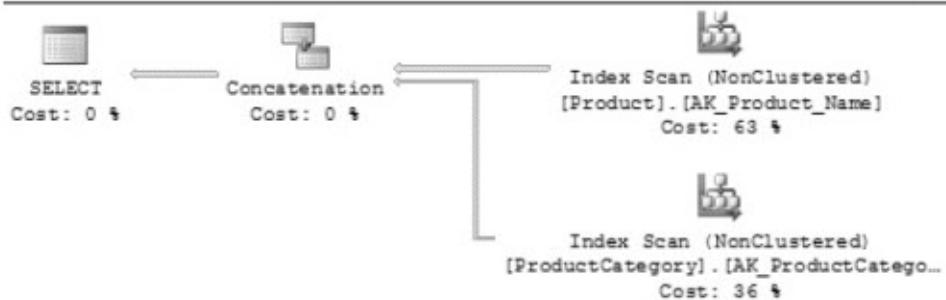
Name
All-Purpose Bike Stand
AWC Logo Cap
Bike Wash - Dissolver
Cable Lock
Chain
Classic Vest, L
Classic Vest, M
Classic Vest, S
...

Wyniki obu zapytań są takie same, ale ponieważ plan wykonania drugiego zapytania nie zawiera operatora MERGE JOIN, jego koszt jest o prawie 20% niższy:

```
Query 1: Query cost (relative to the batch): 63%
SELECT [Name] FROM [SalesLT].[Product] UNION SELECT [Name] FROM
```



```
Query 2: Query cost (relative to the batch): 37%
; SELECT [Name] FROM [SalesLT].[Product] UNION ALL SELECT [Name]
```

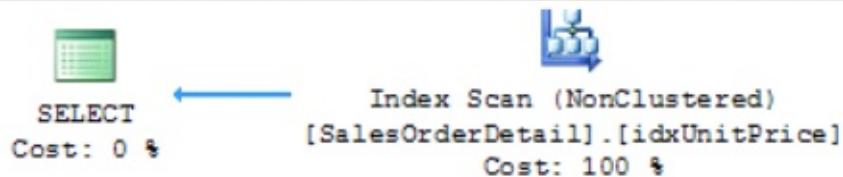


Zadanie 2.

Zacznijmy od sprawdzenia planu wykonania zapytania:

```
SELECT [SalesOrderID]
FROM [SalesLT].[SalesOrderDetail]
WHERE [UnitPrice]*.77 > 900;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [SalesOrderID] FROM [SalesLT].[SalesOrderDetail]
```



Okazuje się, że zostało ono wykonane przez serwer SQL poprzez przeskanowanie indeksu niezgrupowanego. Oznacza to, że przydatny dla tego zapytania indeks nie został efektywnie wykorzystany. Sprawdźmy więc szczegóły operacji przeskanowania tego indeksu:

Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0,0038657
Estimated Operator Cost	0,0046189 (100%)
Estimated Subtree Cost	0,0046189
Estimated CPU Cost	0,0007532
Estimated Number of Executions	1
Estimated Number of Rows	68
Estimated Row Size	19 B
Ordered	False
Node ID	0
Predicate	
CONVERT_IMPLICIT(numeric(19,4), [AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].[UnitPrice], 0)*[@1]>CONVERT_IMPLICIT(numeric(22,6), [@2], 0)	
Object	
[AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].[idxUnitPrice]	
Output List	
[AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].SalesOrderID	

Użyty do wybrania wierszy argument nie był argumentem SARG. Innymi słowy, sprawdzenie poniższego warunku wymagało odczytania wszystkich rekordów indeksu i dopiero potem możliwe było wybranie z nich rekordów spełniających podany warunek:

```
CONVERT_IMPLICIT(numeric(19,4), [AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].[UnitPrice], 0)*[@1]>CONVERT_IMPLICIT(numeric(22,6), [@2], 0)
```

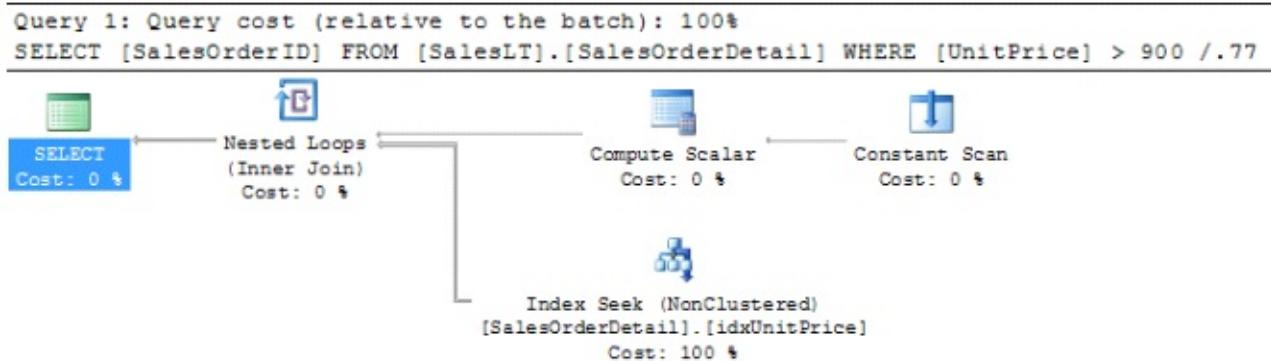
Ponieważ w tym wypadku serwer bazodanowy automatycznie skonwertował typ parametru, tak aby pasował on do typu kolumny `UnitPrice`, powodem nieefektywnego użycia indeksu nie była niejawną konwersją typów.

Przyczyną problemu było występowanie wyrażenia po tej stronie użytego w klauzuli `WHERE` warunku, po której znajduje się nazwa kolumny. Niestety, wiele serwerów bazodanowych, w tym serwer SQL Server, w takiej sytuacji odczytuje wszystkie wiersze, nawet jeżeli są one posortowane według wartości tej kolumny. Przeniesienie wyrażenia na drugą stronę warunku rozwiąże ten problem:

```
SELECT [SalesOrderID]
FROM [SalesLT].[SalesOrderDetail]
WHERE [UnitPrice] > 900 /.77
```

Plan wykonania tego zapytania jest inny:

1. Najpierw, za pomocą operatora COMPUTE SCALAR, wyliczona została wartość podanego wyrażenia.
2. Następnie otrzymany wynik został użyty do przeszukania indeksu (INDEX SEEK):



Jeżeli sprawdzimy właściwości operacji przeszukania indeksu, przekonamy się, że nowy argument jest argumentem typu SARG:

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0,003125
Estimated Operator Cost	0,0033568 (100%)
Estimated Subtree Cost	0,0033568
Estimated CPU Cost	0,0002318
Estimated Number of Executions	1
Estimated Number of Rows	68
Estimated Row Size	11 B
Ordered	True
Node ID	6
Object	
[AdventureWorksLT2012].[SalesLT].[SalesOrderDetail], [idxUnitPrice]	
Output List	
[AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].SalesOrderID	
Seek Predicates	
Seek Keys[1]: Start: [AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].UnitPrice > Scalar Operator ([Expr1004]); End: [AdventureWorksLT2012].[SalesLT].[SalesOrderDetail].UnitPrice < Scalar Operator ([Expr1005])	

Zadanie 3.

Poniższe zapytanie partycjonuje dane:

```
SELECT [DueDate], [SalesOrderID], [TotalDue],
       LAG([TotalDue]) OVER (PARTITION BY [DueDate] ORDER BY [DueDate])
as PreviousTotalDue
FROM [SalesLT].[SalesOrderHeader]
ORDER BY [DueDate];
```

Z lektury rozdziału 9. wiesz, że najlepszym indeksem dla tego typu zapytań jest indeks typu POC. Do wykonania zadania musisz więc określić kolumny użyte do partycjonowania i sortowania wierszy oraz kolumny użyte w pozostałych klauzulach zapytania:

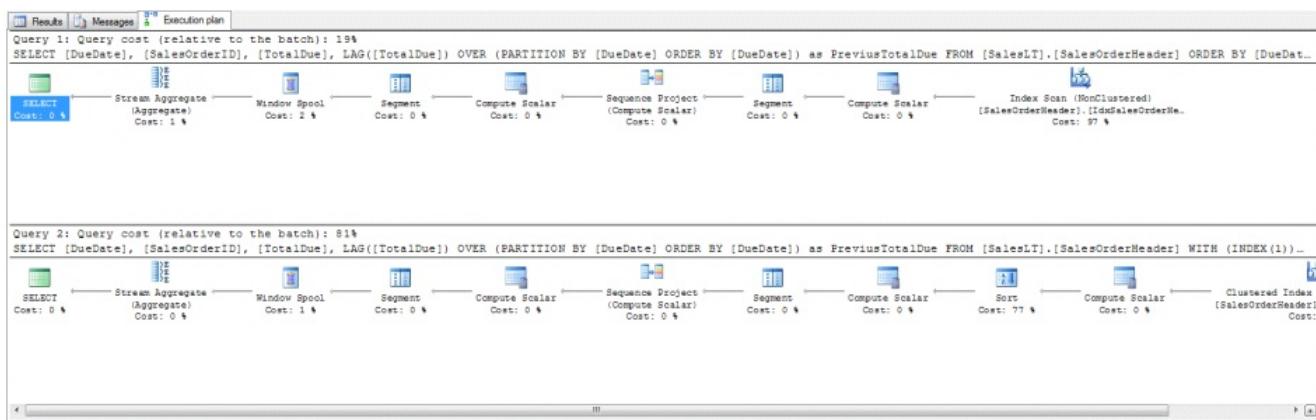
1. Dane są dzielone na partie według wartości kolumny DueDate (w klauzuli PARTITION BY [DueDate]).
2. Ta sama kolumna użyta jest do sortowania (w klauzuli ORDER BY

- [DueDate]).
3. Zapytanie odwołuje się też do kolumn SalesOrderID i TotalDue (w klauzuli SELECT).

Brakujący indeks typu POC może więc mieć następującą definicję:

```
CREATE INDEX IdxSalesOrderHeaderPOC
ON [SalesLT].[SalesOrderHeader]
    ([DueDate]) -- Partitioning and Ordering
    INCLUDE ([SalesOrderID], [TotalDue]) -- Covering
```

Po jego utworzeniu koszt wykonania tej samej instrukcji jest czterokrotnie mniejszy:



Zadania z rozdziału 10.

Zadanie 1.

Język SQL pozwala rozwiązać to zadanie za pomocą jednej instrukcji UPDATE. Zaczniemy od odczytania cen i kosztów widełek:

```
SELECT [ListPrice], [StandardCost]
FROM [SalesLT].[Product] AS P
    JOIN [SalesLT].[ProductCategory] AS C
        ON C.ProductCategoryID=P.ProductCategoryID
WHERE C.Name = 'Forks';
```

ListPrice	StandardCost
148,22	65,8097

175,49	77,9176
229,49	101,8936

Teraz wystarczy zastąpić instrukcję SELECT instrukcją UPDATE i w klauzuli SET określić nowe ceny i koszty wybranych produktów:

```
UPDATE [SalesLT].[Product]
SET [ListPrice] *=.75,
    [StandardCost] +=2
FROM [SalesLT].[ProductCategory] AS C
WHERE C.ProductCategoryID=[SalesLT].[Product].ProductCategoryID
AND C.Name = 'Forks';
-----
(3 row(s) affected)
```

Żeby sprawdzić poprawność rozwiązania, ponownie wykonamy to samo zapytanie:

```
SELECT [ListPrice],[StandardCost]
FROM [SalesLT].[Product] AS P
    JOIN [SalesLT].[ProductCategory] AS C
        ON C.ProductCategoryID=P.ProductCategoryID
WHERE C.Name = 'Forks';
-----
ListPrice      StandardCost
111,165        67,8097
131,6175       79,9176
172,1175       103,8936
```

Zadanie 2.

Zapytanie zwracające identyfikatory, imiona i nazwiska klientek wygląda następująco:

```
SELECT [CustomerID], [FirstName], [LastName]
INTO Panie
FROM [SalesLT].[Customer]
WHERE RIGHT([FirstName],1)='a';
-----
```

CustomerID	FirstName	LastName
------------	-----------	----------

3	Donna	Carreras
22	Linda	Burnett
34	Barbara	German
43	Elsa	Leavitt
...		

Musimy tylko skopiować tak otrzymane dane do tabeli Panie. Najprościej będzie, jeżeli zamiast najpierw utworzyć tabelę z trzema kolumnami, a następnie skopiować do niej wynik powyższego zapytania, utworzymy tabelę na podstawie tego wyniku:

```

SELECT [CustomerID], [FirstName], [LastName]
INTO Panie
FROM [SalesLT].[Customer]
WHERE RIGHT([FirstName],1)='a';
-----
(98 row(s) affected)

```

Zadanie 3.

Po uruchomieniu skryptów z rozdziału 10. dane w tabeli Panie są niekompletne (brakuje informacji o pięciu klientkach), częściowo błędne (niektóre panie mają zmienione imię) i nadmiarowe (znajduje się w niej jeden dodatkowy rekord).

Musimy więc część danych wstawić, część zaktualizować, a część usunąć. Najprościej będzie użyć do tego instrukcji MERGE:

1. Zaczniemy od utworzenia CTE, którego wyniku użyjemy jako tabeli źródłowej (tabeli zawierającej kompletne i aktualne informacje o pracownikach):

```

WITH Klientki AS
(SELECT [CustomerID], [FirstName], [LastName]
FROM [SalesLT].[Customer]
WHERE RIGHT([FirstName],1)='a')
SELECT * FROM Klientki;
-----
```

CustomerID	FirstName	LastName
3	Donna	Carreras

22	Linda	Burnett
34	Barbara	German
43	Elsa	Leavitt
...		

2. Następnie połączmy tabele źródłową i docelową, używając do tego identyfikatorów pracowników:

```
MERGE INTO Panie AS C
USING Klientki AS K
ON (c.[CustomerID] = k.[CustomerID])
```

3. Mając połączone tabele źródłową (Klientki) i docelową (Panie), możemy utworzyć wymagane bloki instrukcji MERGE:

- Jeżeli pracownicy o tych samych numerach, ale różnych imionach znajdują się w obu tabelach, należy zaktualizować ich imiona:

```
WHEN MATCHED AND c.[FirstName] <>k.[FirstName] THEN
    UPDATE SET C.[FirstName] = k.[FirstName]
```

- Jeżeli w tabeli docelowej nie ma informacji o klientce, należy je wstawić:

```
WHEN NOT MATCHED THEN
    INSERT ([FirstName], [LastName])
    VALUES (k.[FirstName], k.[LastName])
```

- Jeżeli w tabeli docelowej są wpisane osoby o numerach nieistniejących w tabeli źródłowej, należy je usunąć:

```
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
```

4. Łącząc wszystkie fragmenty kodu w jedną instrukcję, rozwiążemy zadanie:

```
WITH Klientki AS
    (SELECT [CustomerID], [FirstName], [LastName]
     FROM [SalesLT].[Customer]
     WHERE RIGHT([FirstName],1)='a')
MERGE INTO Panie AS C
USING Klientki AS K
```

```
ON (c.[CustomerID] = k.[CustomerID])
WHEN MATCHED AND c.[FirstName] <>k.[FirstName] THEN
    UPDATE SET C.[FirstName] = k.[FirstName]
WHEN NOT MATCHED THEN
    INSERT ([FirstName], [LastName])
        VALUES (k.[FirstName], k.[LastName])
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
-----
(44 row(s) affected)
```

Zadania z rozdziału 11.

Zadanie 1.

Na domyślnym poziomie izolowania transakcji (Read Committed) blokada współdzielona (blokada typu S) jest zakładana tylko na czas wykonywania poszczególnych zapytań, a nie na czas trwania całej transakcji. Musimy więc liczyć się z możliwością wystąpienia niepowtarzalnych odczytów i odczytów widm.

W przypadku serwera SQL, który domyślnie działa w trybie pesymistycznym, możemy ich uniknąć na cztery sposoby:

1. Zmieniając poziom izolowania transakcji na Serializable — w razie wybrania tego rozwiązania należy liczyć się z tym, że w czasie wykonywania raportu użytkownicy będą mieli ograniczony dostęp do bazy.
2. Kopując potrzebne dane do osobnych tabel — jeżeli mają być one kilkakrotnie odczytywane, takie rozwiązanie może okazać się wydajniejsze niż poprzednie.
3. Przełączając bazę danych do modelu optymistycznego — wykonanie tak poważnej zmiany wymaga jednak dokładnego przetestowania jej wpływu na aplikację bazodanową i nie powinno być przeprowadzane na potrzeby pojedynczych operacji.
4. Tworząc migawkę bazy danych — ta funkcja jest dostępna tylko w edycji Enterprise serwera SQL.

Zadanie 2.

W przeciwieństwie do serwerów bazodanowych aplikacje klienckie nie czekają w nieskończoność, aż uda im się założyć wymagane blokady. Zamiast tego, jeżeli w określonym czasie nie uzyskają od serwera żadnych informacji, przerywają połączenie i zgłaszają błąd typu *Connection Time Out*.

Możliwe więc, że źle napisany program kliencki jawnie rozpoczął transakcję, zanim jeszcze użytkownik wprowadził wszystkie wymagane do jej zakończenia dane. Jeżeli otwarta transakcja dotyczyła regularnie odczytywanych lub modyfikowanych danych, to natychmiast zablokowanych zostało kilka transakcji innych użytkowników. Ponieważ niektóre z nich pewnie zdążyły już zablokować inne zasoby, na ich zakończenie oczekuje kilkudziesiąt innych transakcji.

Jeżeli program kliencki zbyt wcześnie rozpoczęta transakcję i podtrzymuje na czas ich trwania sesję z serwerem bazodanowym, jeden użytkownik, który np. zrobił sobie przerwę w trakcie wpisywania danych nowego produktu, może zablokować całą bazę danych.

Zadanie 3.

Rezultat wykonania procedury musi być zapisany w ramach jawnie rozpoczętej transakcji, w innym przypadku byłby niewiarygodny. Serwer SQL nie obsługuje transakcji zagnieżdzonych, ponadto próba zagnieżdżenia transakcji nie jest sposobem na rozwiązywanie zadania — każdy błąd spowoduje wyzerowanie licznika transakcji i wycofanie kosztownych modyfikacji.

Właściwe rozwiązanie polega na zapisaniu stanu transakcji za pomocą punktu przywracania tuż przed próbą połączenia się ze zdalnym serwerem. Jeżeli ta próba się nie uda, będziemy mogli ją powtórzyć bez konieczności ponownego modyfikowania danych.

Zadania z rozdziału 12.

Zadanie 1.

Z opisu wynika, że:

1. Tabela powinna liczyć osiem kolumn, z których pięć jest wymaganych (imię, nazwisko, adres e-mail, numer telefonu i kod promocyjny), a trzy pozostałe (miasto, wiek, płeć) są opcjonalne.
2. Kluczem podstawowym tabeli może być adres e-mail (jest wymagany i niepowtarzalny), kod promocyjny albo dodatkowa kolumna typu liczbowego. Ponieważ adresy e-mail mogą być dość długie i w dodatku użytkownicy mogą je często zmieniać, nie są one odpowiednie do pełnienia funkcji klucza podstawowego. Albo dodamy do tabeli sztuczny, automatycznie generowany klucz podstawowy, albo użyjemy kodów promocyjnych.
3. Domyślną wartością kolumny **Miasto** powinna być nazwa **Katowice**.
4. Wiek będzie zapisywany jako liczba lat (choć lepiej byłoby zapisać datę urodzin), a więc możemy ograniczyć zakres danych z kolumny **Wiek** do liczb od 5 do 105.
5. Kod promocyjny nie może się powtórzyć.

Zapisując te informacje w postaci instrukcji CREATE TABLE, otrzymamy poniższe rozwiązanie:

```
CREATE TABLE Uczestnicy(  
    ID INT IDENTITY PRIMARY KEY,  
    Imię VARCHAR(30) NOT NULL,  
    Nazwisko VARCHAR(40) NOT NULL,  
    Email VARCHAR(50) NOT NULL,  
    Telefon VARCHAR(15) NOT NULL,  
    Kod CHAR(5) NOT NULL UNIQUE,  
    Miasto VARCHAR(30) NULL DEFAULT 'Katowice',  
    Wiek TINYINT NULL CHECK (Wiek BETWEEN 5 AND 105),  
    Płeć CHAR(1) NULL CHECK (Płeć IN ('K', 'M')));
```

Zadanie 2.

Żeby rozwiązać zadanie, należy utworzyć tabele połączone związkiem typu „wiele do wielu”, co wymaga utworzenia tabeli łącznikowej zawierającej klucze podstawowe obu powiązanych tabel. Ponieważ definiując klucz obcy, trzeba wskazać istniejącą już tabelę, tabela łącznikowa musi być utworzona jako ostatnia:

```
CREATE TABLE Autorzy (
```

```
IDAutora INT IDENTITY PRIMARY KEY,  
Imię VARCHAR(30) NOT NULL,  
Nazwisko VARCHAR (40) NOT NULL);  
  
CREATE TABLE Książki (  
IDKsiążki INT IDENTITY PRIMARY KEY,  
Tytuł VARCHAR(30) NOT NULL,  
DataWydania DATE NOT NULL);  
  
CREATE TABLE AutorzyKsiążki (  
IDAutora INT REFERENCES Autorzy,  
IDKsiążki INT REFERENCES Książki);
```

Zadanie 3.

Główną wadą zaproponowanego przez kolegę rozwiązania jest zapisanie na stałe w ograniczeniu (a więc w definicji obiektu bazodanowego) listy poprawnych nazw modeli. Za każdym razem, gdy zmieni się lista zatwierdzonych modeli (np. gdy dodany zostanie nowy model), konieczna będzie zmiana struktury tabeli.

Dodatkowo próba nałożenia ograniczenia na dodawaną kolumnę skończy się błędem — towary, które już są zapisane, nie mają przecież przypisanego jednego z trzech wymienionych w ograniczeniu `CHECK` modeli.

Ponieważ usuwanie i tworzenie ograniczeń może mieć ogromny wpływ na funkcjonalność i wydajność bazy danych, powinno być przeprowadzane tylko w ramach ustalonej procedury, obejmującej testowanie i dokumentowanie zmian. Nie możemy więc wymagać od administratorów ciągłego zmieniania struktury bazy.

Lepiej będzie zapisać listę modeli w osobnej tabeli słownikowej, a do tabeli `Produkcja`. Towary dodać kolumnę klucza obcego nowej tabeli^[1]. Dodatkowo powinniśmy zabronić zapisywania w tej kolumnie wartości `NULL` — w ten sposób towar będzie musiał należeć do jednego z zapisanych w tabeli słownikowej modeli.

Pozostało nam rozwiązanie problemu już istniejących danych o towarach. Czekanie, aż pracownicy zaktualizują dane i przypiszą wszystkie towary do poszczególnych modeli, doprowadzi raczej do tego, że nowe towary również nie będą klasyfikowane według

określonych modeli. Lepiej będzie wstawić do tabeli słownikowej jeden wiersz symbolizujący brak przypisanego modelu, samodzielnie zaktualizować tabelę Produkcja.Towary i włączyć ograniczenie NOT NULL:

```
CREATE TABLE Produkcja.Modele(
Nazwa VARCHAR (5) PRIMARY KEY);
INSERT INTO Produkcja.Modele
VALUES ('BRAK!');
ALTER TABLE Produkcja.Towary
ADD Model VARCHAR(5) REFERENCES Produkcja.Modele;
UPDATE Produkcja.Towary
SET Model = 'BRAK!';
ALTER TABLE Produkcja.Towary
ALTER COLUMN Model VARCHAR(5) NOT NULL;
```

Zadania z rozdziału 13.

Zadanie 1.

Po pierwsze nazwy wszystkich kolumn widoku muszą być niepowtarzalne. W tym zapytaniu nazwa Name powtarza się dwa razy i chociaż jest to poprawne zapytanie (bo powtarzające się nazwy są poprzedzone nazwami tabeli), to żeby można było je przekształcić na widok, kolumnom tym musimy nadać unikatowe aliasy:

```
SELECT c.Name AS Category, p.Name AS Product, p.ListPrice*1.23
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
ON P.ProductCategoryID = C.ProductCategoryID;
```

Category	Product	(No column name)
Mountain Bikes	Mountain-100 Silver, 38	4181.987700
Mountain Bikes	Mountain-100 Silver, 42	4181.987700
Mountain Bikes	Mountain-100 Silver, 44	4181.987700

Pozostało nam rozwiązanie problemu z ostatnią kolumną widoku. Jako że jest ona wynikiem wyrażenia, nie ma nazwy. Po nadaniu jej dowolnego aliasu zadanie zostanie rozwiązane:

```
CREATE VIEW Zadanie1 AS
SELECT c.Name AS Category, p.Name AS Product, p.ListPrice*1.23 AS WithTax
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
ON P.ProductCategoryID = C.ProductCategoryID;
```

Innym sposobem na rozwiązywanie zadania będzie podanie nazw wszystkich kolumn w nagłówku widoku:

```
CREATE VIEW Zadanie1b (Category, Product, WithTax)
AS
SELECT c.Name, p.Name, p.ListPrice*1.23
FROM [SalesLT].[Product] AS P
JOIN [SalesLT].[ProductCategory] AS C
ON P.ProductCategoryID = C.ProductCategoryID;
```

Zadanie 2.

Nie, poprzez ten widok nie można modyfikować danych:

1. Wstawienie danych poprzez widok uniemożliwia wyrażenie użyte w pierwszej kolumnie: [FirstName] + ' ' + [LastName]. Ponieważ w tabeli bazowej nie ma kolumny Client, serwer bazodanowy nie miałby gdzie zapisać wstawianych przez widok wierszy.
2. Aktualizację danych uniemożliwia słowo kluczowe DISTINCT. Eliminując duplikaty, spowodowaliśmy, że jednemu wierszowi wyniku może odpowiadać wiele wierszy tabeli bazowej, a więc aktualizacja jednego wiersza widoku spowodowałaby aktualizację wszystkich reprezentowanych przez niego wierszy w tabeli bazowej.
3. Usuwanie danych poprzez widok jest niemożliwe, bo odczytuje on dane z wielu tabel. Czyli usunięcie jednego wiersza widoku musiałoby spowodować usunięcie jednego wiersza lub więcej wierszy w kilku tabelach, a instrukcja DELETE nie pozwala jednocześnie usuwać wierszy z wielu tabel.

Zadanie 3.

Założenie takiego indeksu jest niemożliwe, o czym przekonamy się, wykonując poniższą instrukcję:

```
CREATE UNIQUE INDEX IdxSalesOrderPrice  
ON [SalesLT].[SalesOrderDetail]([SalesOrderID],[UnitPrice]);
```

Msg 1505, Level 16, State 1, Line 552

The CREATE UNIQUE INDEX statement terminated because a duplicate key was found for the object name 'SalesLT.SalesOrderDetail' and the index name 'IdxSalesOrderPrice'. The duplicate key value is (71774, 356.90).

The statement has been terminated.

Powodem błędu jest występowanie powtarzających się par wartości w kolumnach SalesOrderID i UnitPrice. Żeby założyć indeks, musimy albo pominąć słowo kluczowe UNIQUE, albo dołączyć do indeksu kolumnę, która zagwarantuje niepowtarzalność jego kluczy.

Zadania z rozdziału 14.

Zadanie 1.

Do poszczególnych kolumn można nadawać i odbierać jedynie uprawnienia SELECT i UPDATE (instrukcje INSERT i DELETE dotyczą całych wierszy). W przypadku serwerów bazodanowych, w których uprawnienia odbiera się instrukcją REVOKE, należy wykonać instrukcję:

```
REVOKE SELECT,UPDATE  
ON [SalesLT].[Customer] (Phone)  
FROM Dev;
```

Serwer SQL oprócz nadawania i odbierania uprawnień pozwala na usuwanie użytkownika lub roli z listy ACL danego obiektu. Wykonując w nim powyższą instrukcję, skasowaliśmy ewentualne uprawnienia roli Dev do kolumny Phone tabeli [SalesLT].[Customer]. Jeżeli jednak członkowie tej roli mają uprawnienia do tej kolumny, czy to odziedziczone po obiekcie nadzorowanym (tabeli, schemacie lub bazie danych), czy też dzięki członkostwu w innych rolach, nadal będą mogli odczytywać i aktualizować numery telefonów.

Dlatego na serwerze SQL zadanie należy rozwiązać, wykonując instrukcję DENY:

```
DENY SELECT,UPDATE  
ON [SalesLT].[Customer] (Phone)
```

```
TO Dev;
```

Zadanie 2.

Nadać użytkownikowi wszystkie uprawnienia do danego obiektu możemy, używając słowa kluczowego ALL, a w przypadku serwera SQL — nadając mu specjalne uprawnienie CONTROL. Żeby ten użytkownik mógł przekazać posiadane uprawnienia innym, należy, nadając mu je, dodać klauzulę WITH GRANT OPTION:

```
GRANT CONTROL  
ON [dbo].[BuildVersion]  
TO Marcin  
WITH GRANT OPTION;;
```

[1] W tym przypadku tabela słownikowa będzie miała tylko jedną kolumnę z nazwami modeli — ponieważ te nazwy są krótkie, lepiej skopiować je do kolumny klucza obcego, dzięki czemu serwer nie będzie musiał łączyć tabel.

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
1. ZAREJESTRUJ SIĘ
 2. PREZENTUJ KSIĄŻKI
 3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA
Helion SA

Spis treści

[Wstęp](#)

[Serwery bazodanowe](#)

[O książce](#)

[SQL Server firmy Microsoft](#)

[Instalacja](#)

[Przykładowa baza danych](#)

[Konwencje i oznaczenia](#)

[Część I Trochę teorii, czyli modele i standardy](#)

[Rozdział 1. Relacyjny model baz danych](#)

[Tabele jako zbiory danych](#)

[Kolumny mają niepowtarzalne nazwy i zawierają określone typy danych](#)

[Wiersze powinny być unikatowe](#)

[Kolejność kolumn jest bez znaczenia](#)

[Kolejność wierszy jest bez znaczenia](#)

[Bazy danych](#)

[Trzy modele baz danych: relacyjny, obiektowy i jednorodny](#)

[Model jednorodny](#)

[Model relacyjny](#)

[Diagram E/R przykładowej bazy danych](#)

[Model obiektowy](#)

[Założenia relacyjnego modelu baz danych](#)

[Postulaty Codda dotyczące struktury danych](#)

[Postulaty Codda dotyczące przetwarzania danych](#)

[Postulaty Codda dotyczące integralności danych](#)

[Normalizacja](#)

[Pierwsza postać normalna](#)

[Druga postać normalna](#)

[Trzecia postać normalna](#)

[Postać Boyce'a-Codda](#)

[Czwarta postać normalna](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 2. Standardy języka SQL](#)

[Strukturalny język zapytań](#)

[Przetwarzanie zbiorów a przetwarzanie pojedynczych danych](#)
[Język strukturalny a język proceduralny](#)
[Język interpretowany a język kompilowany](#)
[Składnia języka SQL](#)
[Identyfikatory](#)
[Literały](#)
[Operatory](#)
[Słowa kluczowe](#)
[Komentarze](#)
[Dialekty języka SQL](#)
[Standardy ANSI](#)
[Historia](#)
[SQL3](#)
[Klasy instrukcji](#)
[Typy danych](#)
[Wartość NULL](#)
[Poziomy zgodności](#)
[Podsumowanie](#)
[Zadania](#)
[Część II Pobieranie danych, czyli instrukcja SELECT](#)
[Rozdział 3. Odczytywanie danych z wybranej tabeli](#)
[Klauzula FROM](#)
[W pełni kwalifikowane nazwy obiektów](#)
[Wybieranie kolumn](#)
[Eliminowanie duplikatów](#)
[Wyrażenia](#)
[Operatory arytmetyczne](#)
[Łączenie danych tekstowych](#)
[Funkcje systemowe](#)
[Funkcje arytmetyczne](#)
[Funkcje znakowe](#)
[Funkcje daty i czasu](#)
[Konwersja typów](#)
[Specjalna funkcja CASE](#)
[Formatowanie wyników](#)
[Aliases](#)
[Stałe \(literały\)](#)
[Sortowanie wyników](#)
[Sortowanie danych tekstowych](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 4. Wybieranie wierszy](#)

[Logika trójwartościowa](#)

[Wartość NULL](#)

[Operatory logiczne](#)

[Operator NOT](#)

[Operator OR](#)

[Operator AND](#)

[Klauzula WHERE](#)

[Standardowe operatory porównania](#)

[Operatory SQL](#)

[Operator IN](#)

[Operator BETWEEN ... AND](#)

[Operator LIKE](#)

[Operator IS NULL](#)

[Złożone warunki logiczne](#)

[Hierarchia operatorów](#)

[Klauzula TOP](#)

[Stronicowanie wierszy](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 5. Łączenie tabel i wyników zapytań](#)

[Złączenia naturalne i nienaturalne](#)

[Klucze obce](#)

[Aliasy](#)

[Złączenia równościowe i nierównościowe](#)

[Złączenia zewnętrzne](#)

[Złączenie lewostronne](#)

[Złączenie prawostronne](#)

[Złączenie obustronne](#)

[Złączenie krzyżowe \(iloczyn kartezjański\)](#)

[Złączenia wielokrotne](#)

[Określanie kolejności złączeń](#)

[Złączenie tabeli z nią samą](#)

[Eliminacja duplikatów](#)

[Klucze obce w obrębie jednej tabeli](#)

[Łączenie wyników zapytań](#)

[Suma](#)

[Część wspólna](#)

[Różnica](#)

[Łączenie wierszy i wyników funkcji tabelarycznych](#)

[Operator APPLY](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 6. Grupowanie wierszy](#)

[Funkcje grupujące](#)

[Funkcja COUNT\(\)](#)

[Zliczanie wierszy](#)

[Funkcje SUM\(\) i AVG\(\)](#)

[Funkcje MIN\(\) i MAX\(\)](#)

[Inne funkcje grupujące](#)

[Wyrażenia](#)

[Zagnieżdżanie funkcji grupujących](#)

[Klauzula GROUP BY](#)

[Kolejność wykonywania klauzuli GROUP BY](#)

[Operatory CUBE i ROLLUP](#)

[Funkcje GROUPING i GROUPING_ID](#)

[Operator GROUPING SETS](#)

[Operatory PIVOT i UNPIVOT](#)

[PIVOT](#)

[UNPIVOT](#)

[Klauzula HAVING](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 7. Partycjonowanie wierszy oraz funkcje rankingu, analityczne i okienkowe](#)

[Partycjonowanie](#)

[Klauzula OVER](#)

[Partycjonowanie danych](#)

[Porządkowanie danych](#)

[Funkcje rankingu](#)

[Okienka](#)

[Funkcje okienkowe](#)

[Funkcje analityczne](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 8. Podzapytania](#)

[Czym są podzapytania?](#)

[Podzapytania jako zmienne](#)

[Podzapytania niepowiązane](#)

[Podzapytania zwracające listę wartości](#)

[Podzapytania niezwracające żadnych wartości](#)

[Zagnieżdżanie podzapytań](#)

[Podzapytania powiązane](#)

[Podzapytania a związki](#)

[Podzapytania jako źródła danych](#)

[Tabele pochodne](#)

[CTE](#)

[Proste CTE](#)

[Rekurencyjne CTE](#)

[Wyznaczanie trendów](#)

[Operatory](#)

[Operator EXISTS](#)

[Operator ANY lub SOME](#)

[Operator ALL](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 9. Wydajność zapytań](#)

[Wykonywanie zapytań przez serwery bazodanowe](#)

[Kolejność wykonywania klauzul zapytania](#)

[Plany wykonania zapytań](#)

[Wydajne wyszukiwanie danych za pomocą argumentów SARG](#)

[Poprawa wydajności złączeń](#)

[Wydajne grupowanie i partycjonowanie danych](#)

[Podsumowanie](#)

[Zadania](#)

[Część III Modyfikowanie danych, czyli instrukcje INSERT, UPDATE, DELETE oraz MERGE](#)

[Rozdział 10. Modyfikowanie danych](#)

[Wstawianie danych](#)

[Klucze podstawowe](#)

[Wartości domyślne](#)

[Wartość NULL](#)

[Konstruktor wierszy](#)

[Wstawianie wyników zapytań](#)

[Instrukcja SELECT ... INTO](#)

[Instrukcja INSERT INTO ... SELECT](#)

[Usuwanie danych](#)

[Instrukcja DELETE](#)

[Usuwanie wyników podzapytań](#)

[Instrukcja TRUNCATE TABLE](#)

[Aktualizowanie danych](#)

[Jednoczesne aktualizowanie wielu kolumn](#)

[Wyrażenia](#)

[Aktualizowanie danych wybranych na podstawie danych z innych tabel](#)

[Aktualizowanie danych za pomocą wyrażeń odwołujących się do innych tabel](#)

[Instrukcja MERGE](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 11. Transakcje i współbieżność](#)

[Właściwości transakcji](#)

[Transakcyjne przetwarzanie danych](#)

[Tryb jawnego zatwierdzania transakcji](#)

[Rozpoczynanie transakcji](#)

[Wycofywanie transakcji](#)

[Zatwierdzanie transakcji](#)

[Zagnieżdżanie transakcji](#)

[Punkty przywracania](#)

[Współbieżność](#)

[Blokady](#)

[Tryby blokad](#)

[Zakresy blokad](#)

[Zakleszczenia](#)

[Poziomy izolowania transakcji](#)

[Read Uncommitted](#)

[Read Committed](#)

[Repeatable Read](#)

[Serializable](#)

[Model optymistyczny](#)

[Model pesymistyczny](#)

[Podsumowanie](#)

[Zadania](#)

[Część IV Tworzenie baz danych, czyli instrukcje CREATE, ALTER i](#)

[DROP](#)

[Rozdział 12. Bazy danych i tabele](#)

[Tworzenie i usuwanie baz danych](#)

[Tworzenie i usuwanie tabel](#)

[Schematy](#)

[Zmiana struktury tabeli](#)

[Ograniczenia](#)

[NOT NULL](#)

[Klucz podstawowy](#)

[Generowanie wartości kluczy podstawowych](#)

[Kompozytowe klucze podstawowe](#)

[Niepowtarzalność](#)

[Wartość domyślna](#)

[Warunek logiczny](#)

[Klucz obcy](#)

[Klucz obcy powiązany z kluczem podstawowym tej samej tabeli](#)

[Kaskadowe usuwanie i aktualizacja powiązanych danych](#)

[Ograniczenia a wydajność instrukcji modyfikujących i odczytujących dane](#)

[Podsumowanie](#)

[Zadania](#)

[Rozdział 13. Widoki i indeksy](#)

[Widoki](#)

[Tworzenie i usuwanie widoków](#)

[Klauzula ORDER BY](#)

[Modyfikowanie widoków](#)

[Korzystanie z widoków](#)

[Odczytywanie danych poprzez widoki](#)

[Zagnieżdżone widoki](#)

[Widoki grupujące dane](#)

[Modyfikowanie danych poprzez widoki](#)

[Zalety widoków](#)

[Indeksy](#)

[Tworzenie, modyfikowanie i usuwanie indeksów](#)

[Opcje indeksów](#)

[Porządkowanie indeksów](#)

[Podsumowanie](#)

[Zadania](#)

[Część V Uprawnienia użytkowników, czyli instrukcje GRANT i](#)

REVOKE

Rozdział 14. Nadawanie i odbieranie uprawnień

Konta użytkowników

Zakładanie i usuwanie kont użytkowników

Role

Tworzenie i usuwanie ról

Przypisywanie ról do użytkowników

Specjalna rola Public

Uprawnienia

Nadawanie i odbieranie uprawnień

Odbieranie uprawnień w serwerze SQL

Dziedziczenie uprawnień

Przekazywanie uprawnień

Zasada minimalnych uprawnień

Podsumowanie

Zadania

Dodatki

Dodatek A Rozwiązania zadań

Zadania z rozdziału 1.

Zadanie 1.

Zadanie 2.

Zadanie 3.

Zadania z rozdziału 2.

Zadanie 1.

Zadanie 2.

Zadanie 3.

Zadania z rozdziału 3.

Zadanie 1.

Zadanie 2.

Zadanie 3.

Zadanie 4.

Zadanie 5.

Zadania z rozdziału 4.

Zadanie 1.

Zadanie 2.

Zadanie 3.

Zadania z rozdziału 5.

Zadanie 1.

Zadanie 2.

[Zadanie 3.](#)

[Zadania z rozdziału 6.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 7.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 8.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 9.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 10.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 11.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 12.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 13.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)

[Zadanie 3.](#)

[Zadania z rozdziału 14.](#)

[Zadanie 1.](#)

[Zadanie 2.](#)