Microsoft

Kompletny przewodnik po DAX

Analiza biznesowa przy użyciu Microsoft Excel, SQL Server Analysis Services i Power BI

Marco Russo i Alberto Ferrari

Przekład:
Marek Włodarz

APN Promise
Warszawa 2016

Kompletny przewodnik po DAX: Analiza biznesowa przy użyciu Microsoft Excel, SQL Server Analysis Services i Power BI

Authorized translation from the English language edition, entitled: The Definitive Guide to DAX: Business intelligence with Microsoft Excel, SQL Server Analysis Services, and Power BI , ISBN 978-0-7356-9835-2, by Marco Russo and Alberto Ferrari, published by Pearson Education, Inc, publishing as Microsoft Press, a Division of Microsoft Corporation.

Polish language edition published by APN PROMISE S.A., Copyright © 2016

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: The Definitive Guide to DAX: Business intelligence with Microsoft Excel, SQL Server Analysis Services, and Power BI , ISBN 978-0-7356-9835-2, by Marco Russo and Alberto Ferrari, opublikowanego przez Pearson Education, Inc, publikującego jako Microsoft Press, oddział Microsoft Corporation.

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

ISBN: 978-83-7541-166-9

Przekład: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Wstęp

Pisaliśmy już wcześniej o języku DAX wielokrotnie: w książkach o Power Pivot i SSAS Tabular, w blogach, artykułach, dokumentacji i wreszcie w książce poświęconej wzorcom DAX. Dlaczego więc zdecydowaliśmy się napisać (z nadzieją, że znajdą się Czytelnicy) jeszcze jedną książkę o DAX? Czy naprawdę jest tak wiele do nauczenia się o tym języku? Oczywiście, naszym zdaniem odpowiedź jest zdecydowanie twierdząca.

Przy pisaniu książki pierwszą rzeczą, którą chce poznać redaktor, jest liczba stron. Istnieją dobre powody, dla których jest to ważne: cena, zarządzanie, alokowanie zasobów i tak dalej. Na koniec niemal wszystko, co zamieścimy w książce, przekłada się na liczbę stron. Dla nas jako autorów jest to dość frustrujące. Ilekroć pisaliśmy kolejną książkę, musieliśmy starannie zaplanować miejsce na opis produktu (na przykład Power Pivot for Microsoft Excel lub SSAS Tabular) i języka DAX. Za każdym razem zostawaliśmy z gorzkim uczuciem, że nie mieliśmy dostatecznie wielu stron, aby opisać wszystko to, co chcieliśmy przekazać na temat języka DAX. Pomijając inne powody, nie można napisać 1000 stron o Power Pivot; książka takich rozmiarów byłaby zniechęcająca dla każdego.

Tak więc przez kilka lat pisaliśmy o SSAS Tabular i Power Pivot, trzymając w szufladzie projekt książki w pełni dedykowanej językowi DAX. Teraz otworzyliśmy tę szufladę i postanowiliśmy nie pomijać niczego w kolejnej książce: chcemy wyjaśnić wszystko o języku DAX, co jest potrzebne, bez kompromisów. Rezultatem tej decyzji jest niniejsza książka.

W książce tej nie będziemy wyjaśniać, jak utworzyć kolumnę obliczaną w konkretnym narzędziu lub którego okna dialogowego użyć w celu ustawienia jakiejś właściwości. To nie jest podręcznik krok po kroku, który nauczy kogoś, jak posługiwać się Microsoft Visual Studio, Power BI czy Power Pivot for Excel. Zamiast tego proponujemy głębokie zanurzenie w język DAX, poczynając od podstaw, aby dojść na koniec do bardzo technicznych szczegółów dotyczących optymalizowania kodu i samego modelu danych.

Ale jest jeszcze jedno pytanie: Dlaczego ktoś w ogóle miałby czytać książkę o DAX?

Niemal każdy ma takie same wrażenia, gdy obejrzy pierwszą demonstrację działania Power Pivot lub Power BI. My również tak myśleliśmy, gdy wypróbowaliśmy

je pierwszy raz: DAX jest taki łatwy! Wygląda tak podobnie do kodu Excela! Co więcej, jeśli ktoś uczył się innych języków programowania lub zapytań, zapewne przyzwyczaił się do nauki nowego języka poprzez przeglądanie przykładów składni i wyszukiwania wzorców, które już zna. My też popełniliśmy ten błąd i chcielibyśmy, aby nasi czytelnicy uniknęli tego samego.

DAX jest potężnym narzędziem, używanym w coraz większej liczbie narzędzi analitycznych. Jest bardzo skuteczny, ale zawiera kilka koncepcji, które trudno zrozumieć intuicyjnie. Na przykład kontekst wykonania jest zagadnieniem, które wymaga podejścia dedukcyjnego: trzeba zacząć od teorii, a następnie przeanalizować kilka przykładów demonstrujących działanie tej teorii. Rozumowanie dedukcyjne jest tym, co prezentujemy w tej książce. Wiemy, że wiele osób nie lubi nauki w tym stylu, gdyż preferują bardziej praktyczne podejście polegające na nauczeniu się, jak rozwiązać określone problemy, a następnie, dzięki doświadczeniu i praktyce, zrozumieć leżącą w tle teorię dzięki rozumowaniu indukcyjnemu. Jeśli ktoś woli takie podejście, ta książka nie będzie dla niego odpowiednia. Napisaliśmy już książkę o wzorcach DAX, pełną przykładów, ale pozbawioną wyjaśnień, dlaczego dana formuła działa ani dlaczego pewien sposób kodowania jest lepszy. Tamta książka jest dobrym źródłem do kopiowania i wklejania formuł DAX. Niniejsza książka ma inny cel: ma pozwolić na prawdziwe opanowanie języka DAX. Wszystkie przykłady mają na celu zademonstrowanie zachowania DAX; nie rozwiązanie określonego problemu. Jeśli Czytelnik znajdzie formuły, których będzie mógł użyć w swoim modelu, to doskonale. Jednak trzeba pamiętać, że to jedynie efekt uboczny, a nie cel podanego przykładu. Warto też zwrócić uwagę na dołączone uwagi, aby się upewnić, że kod użyty w przykładach nie kryje w sobie potencjalnych pułapek. Ze względów edukacyjnych często używamy takich przykładów, które nie stanowią najlepszych praktyk.

Dla kogo jest ta książka

Jeśli Czytelnik jest tylko okazjonalnym użytkownikiem języka DAX, wówczas książka ta zapewne nie jest najlepszym wyborem. Istnieje wiele tytułów, które zapewniają proste wprowadzenie do narzędzi wykorzystujących DAX oraz samego języka DAX, poczynając od podstaw i dochodząc do podstawowego poziomu programowania. Wiemy o tym dobrze, gdyż sami napisaliśmy niektóre z tych książek!

Jeśli jednak Czytelnik intensywnie używa lub zamierza używać języka DAX i chce naprawdę poznać każdy szczegół tego pięknego języka, wówczas jest to właściwa książka. Może być to pierwsza książka o DAX; w takim przypadku nie należy oczekiwać, że będzie można szybko skorzystać z najbardziej zaawansowanych tematów. Sugerujemy przeczytanie książki od deski do deski, a następnie powrót do najbardziej złożonych zagadnień, gdy zdobędzie się już pewne doświadczenie; bardzo możliwe, że niektóre koncepcje staną się wówczas bardziej zrozumiałe.

Język DAX jest przydatny dla różnych ludzi do różnych celów: użytkownicy Excela mogą go wykorzystać w budowaniu modeli danych Power Pivot, profesjonaliści BI mogą potrzebować zaimplementować kod DAX w rozwiązaniach analizy biznesowej dowolnego rozmiaru, okazjonalni użytkownicy Power BI mogą zechcieć tworzyć formuły DAX w ich samoobsługowych modelach. W tej książce staraliśmy się zamieścić informacje przydatne dla wszystkich tych grup. Niektóre tematy (a szczególnie część o optymalizacji) są zapewne bardziej adresowane do profesjonalistów BI, gdyż wiedza niezbędna do optymalizowania miar DAX jest bardzo techniczna. Jesteśmy jednak przekonani, że również użytkownicy Excela powinni rozumieć powody różnic w sprawności działania wyrażeń DAX, aby osiągnąć najlepsze wyniki w swoich modelach.

Na koniec chcieliśmy napisać książkę do studiowania, a nie tylko do czytania. Na początku starliśmy się przedstawiać zagadnienia w jak najprostszy sposób. Jednak gdy omawiane koncepcje stają się bardziej złożone, musieliśmy porzucić prostotę i jest to realistyczne podejście. DAX *nie jest* językiem prostym. Opanowanie go i zrozumienie każdego szczegółu działania silnika zajęło nam wiele lat. Nie należy oczekiwać, że uda się opanować całą treść książki w kilka dni, czytając ją w wolnych chwilach. Książka wymaga wysokiego poziomu skupienia oraz praktycznego realizowania ćwiczeń i eksperymentowania. W zamian oferujemy nieznany wcześniej poziom omawiania wszystkich aspektów języka DAX i możliwość zostania prawdziwym ekspertem w tej dziedzinie.

Założenia

Oczekujemy, że nasz Czytelnik ma podstawową wiedzę na temat tabel przestawnych programu Excel oraz pewne doświadczenie w analizie numerycznej. Jeśli miał już kontakt z językiem DAX, tym lepiej, gdyż szybciej będzie mógł przeczytać początkowe rozdziały, ale znajomość języka DAX nie jest wymagana.

W książce występują również pewne odniesienia do kodu MDX oraz SQL, ale Czytelnik nie musi naprawdę znać tych języków, gdyż pokazuje to po prostu różne sposoby formułowania wyrażeń. Jeśli ktoś nie rozumie tych fragmentów kodu, to jest to w porządku – po prostu ten temat go nie dotyczy.

W najbardziej zaawansowanych częściach książki omawiamy takie zagadnienia, jak przetwarzanie równoległe, dostęp do pamięci, wykorzystanie procesora i inne techniczne tematy, które zapewne nie są dobrze znane wszystkim czytelnikom. Każdy programista odnajdzie się tam jak u siebie, podczas gdy nawet zaawansowany użytkownik Excela może poczuć się nieco przytłoczony. Tym niemniej, przy omawianiu optymalizacji te informacje są niezbędne. Te najbardziej zaawansowane części są kierowane raczej w stronę programistów rozwiązań BI, niż użytkowników Excela. Tym niemniej sądzimy, że każdy może skorzystać po ich przeczytaniu.

Organizacja książki

Książka została zaprojektowana tak, by przechodzić od rozdziałów wprowadzających do bardziej złożonych w logiczny sposób. Każdy rozdział został napisany przy założeniu, że wcześniejsze treści zostały w pełni opanowane; nie ma tu powtarzania koncepcji wyjaśnionych wcześniej. Z tego względu książkę należy czytać we właściwej kolejności i unikać zbyt wczesnego przeskakiwania do bardziej zaawansowanych tematów.

Oto zawartość poszczególnych rozdziałów w skrócie:

- Rozdział 1 stanowi krótkie wprowadzenie do języka DAX, przy czym poszczególne podrozdziały dedykowane są dla Czytelników, którzy mają już pewną wiedzę na temat innych języków, a konkretnie SQL, Excel lub MDX. W tym rozdziale nie są przedstawiane żadne nowe koncepcje, a tylko kilka wskazówek o różnicach pomiędzy DAX a innymi językami znanymi Czytelnikowi.
- Rozdział 2 przedstawia sam język DAX. Omawiamy w nim podstawowe pojęcia, takie jak kolumny obliczane, miary, funkcje obsługi błędów, a także wymieniamy najbardziej podstawowe funkcje języka.
- Rozdział 3 poświęcony jest podstawowym funkcjom tablicowym. Wiele funkcji języka DAX działa na tabelach i zwraca tabele jako wynik. W tym rozdziale omawiamy podstawowe funkcje tego typu, a przedstawienie zaawansowanych zawiera rozdział 9.
- Rozdział 4 dedykowany jest omówieniu kontekstów wykonania. Konteksty wykonania są fundamentem języka DAX i ten rozdział wraz z następnym stanowi zapewne najważniejszą część całej książki.
- Rozdział 5 zajmuje się tylko dwiema funkcjami: `CALCULATE` i `CALCULATETABLE`. Bez przesady można powiedzieć, że są to najważniejsze funkcje języka DAX i ich właściwe stosowanie jest silnie uzależnione od dobrego zrozumienia kontekstów wykonania.
- Rozdział 6 zawiera kilka przykładów kodu DAX wraz z omówieniem ich działania. Nie należy jednak postrzegać ich jako wzorców do ponownego użycia. Zamiast tego pokazujemy w nim, jak rozwiązać kilka typowych scenariuszy przy użyciu podstawowych koncepcji opanowanych do tej pory.
- Rozdział 7 koncentruje się na zagadnieniach analizy czasowej. Wśród omawianych kalkulacji występują takie typowe problemy, jak „od początku roku/miesiąca”, wartości dla odpowiednich okresów z poprzednich lat, kalendarze oparte na tygodniach lub niestandardowe i wiele innych.
- Rozdział 8 jest dedykowany funkcjom statystycznym, takim jak ustalanie rankingu, obliczenia finansowe lub percentyle.

- Rozdział 9 jest logiczną kontynuacją rozdziału 3, w którym wprowadziliśmy podstawowe funkcje tablicowe. W tym rozdziale idziemy dalej, szczegółowo omawiając cały zestaw funkcji do manipulacji tabelami, z których większość jest bardzo użyteczna przy tworzeniu złożonych zapytań
- Rozdział 10 przeniesie wiedzę o kontekstach wykonania na kolejny szczebel. Omawiamy w nim działanie złożonych funkcji, takich jak `ALLSELECTED` i `KEEPFILTERS`, łącznie z teorią tabel rozszerzonych. To trudny rozdział, który odkrywa większość sekretów złożonych wyrażeń DAX.
- Rozdział 11 pokazuje, jak wykonywać obliczenia dla hierarchii i jak obsłużyć struktury drzewiaste w języku DAX.
- Rozdział 12 poświęcony jest realizowaniu nietypowych relacji w języku DAX. W istocie dzięki pomocy DAX w modelu danych można wyrazić dowolny typ relacji. W tym rozdziale pokażemy wiele ich rodzajów. Jest to zarazem ostatni rozdział poświęcony samemu językowi DAX; pozostała część książki zajmuje się technikami optymalizacyjnymi.
- Rozdział 13 zawiera szczegółowy opis silnika bazodanowego VertiPaq; jest to najczęściej stosowany silnik bazodanowy, w którym wykonywane są wyrażenia DAX. Jego zrozumienie jest niezbędne, aby móc nauczyć się pisać wydajny kod w języku DAX.
- Rozdział 14 wykorzystuje wiedzę zdobytą w rozdziale 13, aby pokazać możliwe optymalizacje, które można wprowadzić na poziomie modelu danych. Kiedy normalizować lub denormalizować tabele, jak redukować kardynalność kolumn, jakiego typu relacje definiować – to tylko niektóre z pytań, na które próbujemy odpowiedzieć w tym rozdziale.
- Rozdział 15 pokazuje, jak czytać plany zapytań i jak mierzyć wydajność zapytań przy użyciu takich narzędzi, jak SQL Server Profiler lub DAX Studio.
- Rozdział 16 pokazuje różne techniki optymalizacyjne, oparte na zawartości poprzednich trzech rozdziałów. Pokażemy tu wiele wyrażeń DAX, zmierzymy ich wydajność, po czym pokażemy możliwe poprawki i wyjaśnimy działanie zoptymalizowanych formuł.

Konwencje

W książce wykorzystywane są następujące konwencje typograficzne:

- **Wytłuszczenie** oznacza tekst wpisywany przez Czytelnika lub (w przykładach kodu) szczególnie istotne miejsca.
- *Kursywa* służy do wyróżnienia nowych terminów, a także nazw tabel, kolumn, miar i kolumn obliczanych oraz adresów internetowych.

- `Czcionką stałopozycyjną` złożone są fragmenty kodu oraz nazwy funkcji i dyrektyw języka DAX (nazwy funkcji są pisane `WIELKIMI LITERAMI`).
- Nazwy okien dialogowych, ich elementów i polecenia interfejsu użytkownika są złożone czcionką jednoelementową, na przykład okno dialogowe Save As.
- Skróty klawiszowe są zapisywane ze znakiem plus (+) oddzielającym nazwy klawiszy. Na przykład Ctrl+Alt+Delete oznacza równoczesne naciśnięcie klawiszy Ctrl, Alt i Delete.

Treść dołączona

Przykłady baz danych oraz kod przykładów dostępny jest w materiałach powiązanych z książką. Pliki te można pobrać z następującej strony:

http://aka.ms/GuidetoDAX/files

Zawartość ta obejmuje następujące elementy:

- Kopię zapasową SQL Server bazy danych Contoso Retail DW, której można użyć do samodzielnego zbudowania przykładów. Jest to standardowa demonstracyjna baza danych udostępniana przez firmę Microsoft, którą wzbogaciliśmy o pewne widoki, ułatwiające zbudowanie modelu danych na jej podstawie.
- Model danych Power BI Desktop, który został użyty do wygenerowania wszystkich ilustracji zawartych w książce. Baza danych jest zawsze ta sama, a dla każdego rozdziału został dołączony dokument pokazujący kroki wymagane do zreprodukowania tego samego przykładu, którego użyliśmy w książce.

Errata, aktualizacje i wsparcie dla książki

Dokonaliśmy wszelkich starań, aby zapewnić dokładność tej książki i dołączonej treści. Jeśli jednak Czytelnik odkryje błąd, prosimy o zgłoszenie go poprzez stronę:

http://aka.ms/GuidetoDAX/errata

Kontakt z zespołem Microsoft Press Support możliwy jest poprzez adres mailowy *mspinput@microsoft.com*.

Proszę zwrócić uwagę, że powyższe adresy nie oferują wsparcia technicznego dla produktów programowych i sprzętowych firmy Microsoft. Jeśli potrzebna jest pomoc dotycząca oprogramowania lub sprzętu firmy Microsoft, należy odwiedzić stronę *http://support.microsoft.com*.

Bezpłatne ebooki od Microsoft Press

Począwszy od przeglądów technicznych, po pogłębione informacje na temat szczególnych tematów, darmowe ebooki z Microsoft Press obejmują szeroką tematykę. Ebooki te są dostępne w formatach PDF, EPUB oraz Mobi dla Kindle, gotowe do pobrania ze strony:

http://aka.ms/mspressfree

Warto często sprawdzać, by zobaczyć, co nowego!

Podziękowania

Powinniśmy podziękować tak wielu ludziom za ich pomoc w powstawaniu tej książki, że niemożliwe byłoby zamieszczenie pełnej listy. Zatem po prostu dziękujemy wszystkim, którzy się przyczynili do jej powstania – nawet jeśli nie wiedzieli, że właśnie to robią. Komentarze do wpisów na blogu, posty na forum, dyskusje mailowe, rozmowy na konferencjach technicznych, analizy problemów klientów – wszystko to było bardzo przydatne i pozwoliło znacząco wzbogacić tę książkę.

Tym niemniej, są ludzie, których musimy wymienić osobiście ze względu na ich szczególny udział.

Na pierwszym miejscu musi znaleźć się Edward Melomed: zainspirował nas i zapewne w ogóle nie zaczęlibyśmy zajmować się językiem DAX bez gorącej dyskusji, którą mieliśmy z nim wiele lat temu i która zaowocowała spisem treści naszej pierwszej książki o Power Pivot, naszkicowanym na kawiarnianej serwetce.

Chcemy podziękować zespołowi Microsoft Press i ludziom, którzy przyczynili się do realizacji tego projektu: Carol Dillingham była świetnym redaktorem i bardzo pomogła nam w procesie pisania. Wielu innych również pomagało w tworzeniu książki: dzięki wam wszystkim.

Jedynym zadaniem dłuższym od pisania książki jest przestudiowanie zagadnień, które trzeba poznać wcześniej. Grupa ludzi, którą my nazywamy „ssas-insiders", pomogła nam przygotować się do napisania tej książki. Kilka osób z firmy Microsoft zasługuje na szczególną wzmiankę, gdyż poświęcili swój cenny czas na przekazanie nam ważnych informacji o Power Pivot i DAX: są to Marius Dumitru, Jeffrey Wang, Akshai Mirchandani i Cristian Petculescu. Wasza pomoc była nieoceniona!

Chcemy również podziękować Amirowi Netzowi, Ashvini Sharma, Kasperowi De Jonge oraz T. K. Anand za ich wkład w wiele dyskusji, które prowadziliśmy na temat książki i jej tematyki.

Na koniec szczególne podziękowania należą się recenzentom technicznym: Gerhardowi Brücklowi i Andrea Benedettiemu. Dwukrotnie sprawdzili całą treść naszego oryginalnego tekstu, szukając błędów i takich zdań, które nie były jasne. Ich

nieocenione sugestie pozwoliły znacząco poprawić książkę. Bez ich skrupulatnej pracy byłoby ją znacznie trudniej czytać! Jeśli książka zawiera mniej błędów, niż nasz oryginalny rękopis, to tylko ich zasługa. Jeśli nadal zawiera jakieś błędy, jest to oczywiście nasza wina.

Dziękujemy bardzo!

ROZDZIAŁ 1

Czym jest DAX?

DAX jest językiem programowania wykorzystywanym przez Microsoft SQL Server Analysis Services (SSAS) oraz Microsoft Power Pivot for Excel. Został utworzony w roku 2010, wraz z pierwszym wydaniem narzędzia PowerPivot for Excel 2010 (tak, w roku 2010 *PowerPivot* pisany był bez odstępu; spacja w nazwie *Power Pivot* pojawiła się dopiero w roku 2013). Z upływem czasu DAX zdobył popularność w środowisku zaawansowanych użytkowników Excela, którzy używają go do tworzenia modeli danych Power Pivot w skoroszytach programu Excel, a także w środowisku Business Intelligence (BI), gdzie DAX służy do budowania modeli dla SSAS.

DAX jest językiem prostym. Powiedziawszy to, trzeba zauważyć, że DAX różni się od większości innych języków programowania i przyzwyczajenie się niego może wymagać trochę czasu. Z naszych doświadczeń, zdobytych przy nauczaniu tego języka setek (a może i tysięcy) ludzi, opanowanie podstaw DAX jest proste i naturalne: można zacząć go używać już po paru godzinach. Jednak gdy przyjdzie potrzeba zrozumienia zaawansowanych koncepcji, takich jak konteksty wykonania, iteracje i propagowanie kontekstów, wszystko razem może wydać się bardzo złożone. Nie należy się poddawać! Trochę cierpliwości. Gdy nasz umysł zacznie już ogarniać te pojęcia, odkryjemy, że DAX istotnie jest łatwym językiem.

Potrzeba tylko nieco czasu, aby do niego przywyknąć.

Ten pierwszy rozdział rozpoczyna się krótkim przypomnieniem, czym jest model danych w kontekście tabel i relacji. Zalecamy przeczytanie tego podrozdziału bez względu na posiadane doświadczenie, aby ujednoznacznić terminologię, której będziemy używać w dalszym ciągu tej książki przy odwoływaniu się do tabel, modeli i różnych rodzajów relacji.

W kolejnych podrozdziałach przedstawimy porady dla Czytelników, którzy mają już jakieś doświadczenia z innymi językami programowania – a konkretnie znających Excel, SQL lub MDX. Każdy podrozdział jest przeznaczonych dla Czytelników, którzy już znają dany język i mogą uznać, że przydatne może być przeczytanie bardzo szybkiego wprowadzenia do DAX, zawierającego porównanie z tymi językami. Jeśli ktoś jest np. użytkownikiem Excela i stwierdzi, że część poświęcona MDX jest niemal niemożliwa do zrozumienia, nie ma w tym nic zaskakującego. Można po prostu

pominąć tę część, gdyż zawiera ona informacje zasadniczo nieprzydatne dla takiej osoby i przejść do kolejnego rozdziału, w którym nasza podróż do świata języka DAX rozpocznie się naprawdę.

Istota modelu danych

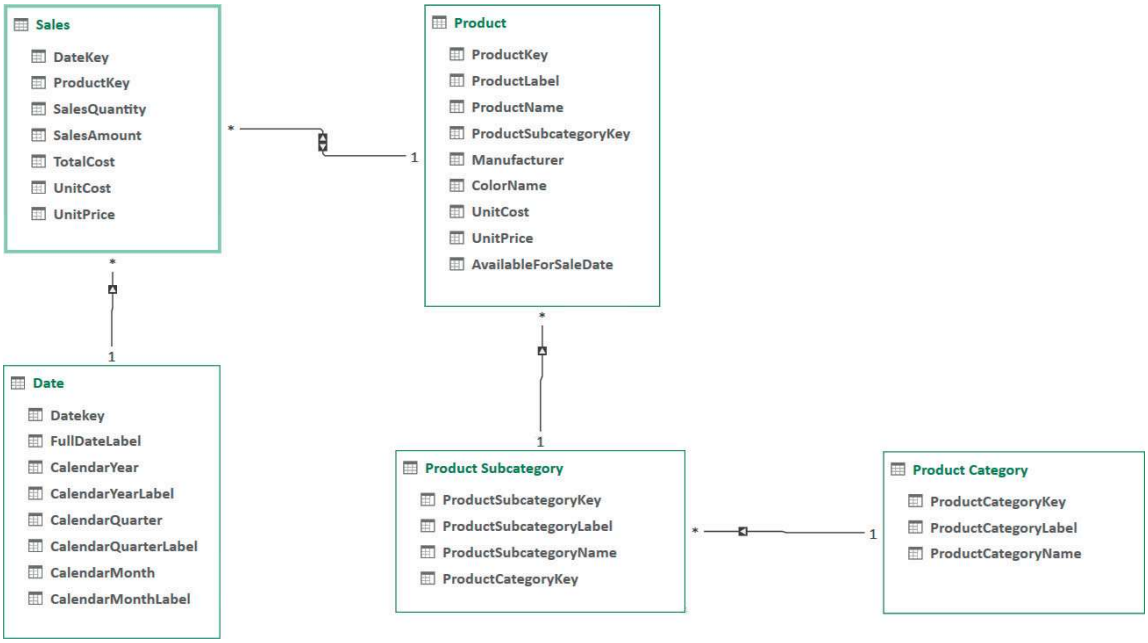
DAX jest językiem zaprojektowanym specjalnie w celu przetwarzania formuł biznesowych w modelu danych. Być może Czytelnik już wie, czym jest *model danych*, ale jeśli nie zna jeszcze dobrze tego pojęcia, warto poświęcić nieco czasu na lekturę kilku kolejnych stron, które zawierają omówienie modeli danych i relacji. W ten sposób można zbudować fundament, na którym będziemy budować dalszą wiedzę o języku DAX.

Model danych to zbiór tabel połączonych relacjami. Tylko tyle i aż tyle.

Wszyscy wiemy, czym jest *tabela*: jest to zbiór wierszy zawierających dane, przy czym każdy wiersz podzielony jest na kolumny. Każda kolumna ma przypisany typ danych i przechowuje pojedynczy element informacji. Zazwyczaj odwołujemy się do wiersza w tabeli jako do rekordu. Tabele są wygodnym sposobem porządkowania danych. W istocie tabela sama w sobie jest modelem danych, choć w najprostszej postaci. Tak więc, gdy wpisujemy nazwy i liczby w skoroszycie Excela, tworzymy model danych.

Jeśli nasz model danych zawiera wiele tabel, najprawdopodobniej będą one połączone ze sobą poprzez relacje. Relacja jest formą związku pomiędzy tabelami. Gdy pomiędzy dwiema tabelami istnieje relacja, mówimy, że są ze sobą powiązane. Graficznie relacja jest zwykle reprezentowana poprzez linię łączącą dwie tabele.

Rysunek 1-1 pokazuje przykład modelu danych.



Rysunek 1-1 Prosty przykład modelu danych złożonego z pięciu tabel

Istnieje kilka ważnych aspektów relacji, które również trzeba poznać:

- Dwie tabele w relacji nie pełnią takich samych ról. Są one określane jako strona jednowartościowa oraz strona wielowartościowa relacji. Zwróćmy uwagę na relację pomiędzy tabelami *Product* i *Product Subcategory* na rysunku 1-1. Pojedyncza podkategoria (*subcategory*) zawiera wiele produktów, podczas gdy każdy produkt należy tylko do jednej podkategorii. Tym samym tabela *Product Subcategory* jest stroną jednoznaczną (jednak podkategoria) relacji, podczas gdy tabela *Product* jest stroną wieloznaczną (zawierającą wiele produktów z wybranej podkategorii).
- Kolumny użyte do utworzenia relacji (które zwykle noszą tę samą nazwę w obu tabelach) są nazywane kluczami relacji. Po stronie jednoznacznej relacji kolumna ta musi zawierać unikatowe wartości w każdym wierszu. Po stronie wieloznacznej ta sama wartość może być (i zazwyczaj jest) powtórzona w wielu różnych wierszach. Gdy kolumna zawiera unikatową wartość dla każdego wiersza, nazywana jest kluczem (lub kolumną klucza) tabeli. Większość tabel zawiera kolumnę, która jest kluczem.
- Relacje mogą tworzyć łańcuch. Każdy produkt ma przypisaną mu podkategorię, a każda podkategoria należy do pewnej kategorii. W ten sposób każdy produkt ma określoną kategorię. Aby odnaleźć kategorię produktu, musimy przejść przez łańcuch dwóch relacji. Rysunek 1-1 zawiera przykład łańcucha złożonego z trzech relacji, zaczynającego się od tabeli *Sales* (sprzedaż) i prowadzącego do tabeli *Product Category*.
- Linia stanowiąca symbol każdej relacji może zawierać jedną lub dwie strzałki. Na rysunku 1-1 można zauważyć dwie strzałki w relacji pomiędzy tabelami *Sales* i *Product*, podczas gdy wszystkie inne relacje mają tylko pojedyncze. Strzałka wskazuje kierunek automatycznego filtrowania relacji. Zagadnienie to omówimy szczegółowo w dalszych rozdziałach, gdyż ustalenie właściwego kierunku filtrowania jest jedną z najważniejszych umiejętności, które trzeba opanować.
- W tabelarycznych modelach danych relacje można tworzyć tylko na podstawie pojedynczych kolumn. Relacje wykorzystujące klucz rozciągający się na wiele kolumn nie są obsługiwane w tym przypadku (choć są możliwe i często stosowane w innych silnikach bazodanowych).

Zrozumienie kierunku relacji

Jak powiedzieliśmy w poprzednim podpunkcie, każda relacja może mieć jeden lub dwa kierunki filtrowania. Filtrowanie zawsze występuje od strony jednoznacznej w kierunku strony wieloznacznej relacji. Jeśli relacja jest dwukierunkowa (czyli jej symbol zawiera dwie strzałki), wówczas filtrowanie może zachodzić również od strony wieloznacznej do jednoznacznej.

Zachowanie to łatwiej będzie przedstawić na przykładzie. Jeśli utworzymy tabelę przestawną opartą na modelu danych pokazanym wcześniej na rysunku 1-1, umieszczając lata (*years*) w wierszach oraz sumy sprzedaży i liczby produktów (*Sum of SalesAmount* oraz *Count of ProductName*, odpowiednio) w obszarze wartości, otrzymamy wynik pokazany na rysunku 1-2.

Row Labels	Sum of SalesAmount	Count of ProductName
2007	$1,010,803,395.04	1,828
2008	$849,671,203.42	2,244
2009	$857,728,031.35	2,504
Grand Total	**$2,718,202,629.81**	**2,517**

Rysunek 1-2 Tabela przestawna pokazuje w działaniu efekt filtrowania poprzez wiele tabel.

Kolumna *Row Labels* (etykiety kolumn) zawiera lata – czyli kolumnę odczytaną z tabeli *Date*. Tabela ta jest jednoznaczną stroną relacji z tabelą *Sales* (sprzedaż). Zatem gdy umieściliśmy *Sum of SalesAmount* w tabeli przestawnej, silnik filtruje tabelę *Sales* według lat. Relacja pomiędzy tabelami *Sales* i *Product* jest dwukierunkowa; umieszczenie zliczania nazw produktów w tabeli przestawnej zwraca jako wynik liczbę produktów sprzedanych w każdym roku. Inaczej mówiąc, filtrowanie według lat propagowane jest do tabeli *Product* poprzez łańcuch relacji.

Jeśli zmodyfikujemy teraz tabelę przestawną, umieszczając pole *Color* w wierszach i dodając *Count of FullDateLabel* (licznik pełnych dat) w obszarze wartości, wynik staje się nieco trudniejszy do zrozumienia, co można zobaczyć na rysunku 1-3.

Row Labels	Sum of SalesAmount	Count of ProductName	Count of FullDateLabel
Azure	$9,432,951.01	14	2556
Black	$544,932,784.76	602	2556
Blue	$221,983,165.79	200	2556
Brown	$112,311,885.56	77	2556
Gold	$39,836,201.72	50	2556
Green	$122,192,345.24	74	2556
Grey	$298,320,774.10	283	2556
Orange	$70,323,434.77	55	2556
Pink	$58,419,345.34	84	2556
Purple	$912,486.31	6	2556
Red	$94,063,709.25	99	2556
Silver	$609,853,067.20	417	2556
Silver Grey	$21,382,969.28	14	2556
Transparent	$241,181.14	1	2556
White	$505,832,783.51	505	2556
Yellow	$8,163,544.83	36	2556
Grand Total	**$2,718,202,629.81**	**2,517**	**2556**

Rysunek 1-3 Ta tabela przestawna pokazuje, że jeśli dwukierunkowe filtrowanie nie jest aktywne, tabele nie są filtrowane.

Filtr dla wierszy to kolumna *Color* z tabeli *Product*. Ponieważ tabela ta jest po jednoznacznej stronie relacji z tabelą *Sales*, wartość *Sum of SalesAmount* jest właściwie filtrowana. Kolumna *Count of ProductNames* również jest filtrowana poprawnie, gdyż wylicza ona wartości z tej samej tabeli, która służy do wybierania wierszy (*Product*). Błędne liczby występują w kolumnie *Count of FullDateLabel*. Można zauważyć, że pokazuje ona tę samą wartość dla wszystkich wierszy – a tak przy okazji, jest to po prostu całkowita liczba wierszy w tabeli *Date*.

Powodem, dla którego filtr pochodzący z kolumny *Color* nie jest propagowany do tabeli *Date*, jest to, że relacja pomiędzy tabelami *Date* a *Sales* jest jednokierunkowa i przebiega od *Date* do *Sales* (co jest pokazane poprzez kierunek strzałki). Tym samym pomimo aktywnego filtru w tabeli *Sales* warunek ten nie przenosi się do tabeli *Date* ze względu na typ istniejącej relacji.

Jeśli zmienimy typ relacji, aby włączyć działanie dwukierunkowe, uzyskamy rezultat pokazany na rysunku 1-4.

Jak można zauważyć, liczby w poszczególnych wierszach są teraz inne, odzwierciedlając liczbę dni, w których sprzedany został przynajmniej jeden produkt określonego koloru. Na pozór mogłoby się wydawać, że wszystkie relacje powinny być definiowane jako dwukierunkowe, pozwalając na propagowanie filtrowania w dowolnym kierunku i zawsze zwracając sensowne wyniki. Jak się jednak okaże w trakcie lektury tej książki, nie zawsze jest to właściwy sposób projektowania modelu danych. W rzeczywistości kierunek relacji powinien zawsze zależeć od tego, jaki scenariusz chcemy wymodelować.

Row Labels	Sum of SalesAmount	Count of ProductName	Count of FullDateLabel
Azure	$9,432,951.01	14	1033
Black	$544,932,784.76	602	1096
Blue	$221,983,165.79	200	1096
Brown	$112,311,885.56	77	1096
Gold	$39,836,201.72	50	1096
Green	$122,192,345.24	74	1096
Grey	$298,320,774.10	283	1096
Orange	$70,323,434.77	55	1096
Pink	$58,419,345.34	84	1096
Purple	$912,486.31	6	600
Red	$94,063,709.25	99	1096
Silver	$609,853,067.20	417	1096
Silver Grey	$21,382,969.28	14	1051
Transparent	$241,181.14	1	221
White	$505,832,783.51	505	1096
Yellow	$8,163,544.83	36	1095
Grand Total	**$2,718,202,629.81**	**2,517**	**2556**

Rysunek 1-4 Przekształcenie relacji na dwukierunkową powoduje filtrowanie tabeli Date poprzez wybór wartości z kolumny Color.

DAX dla użytkowników programu Excel

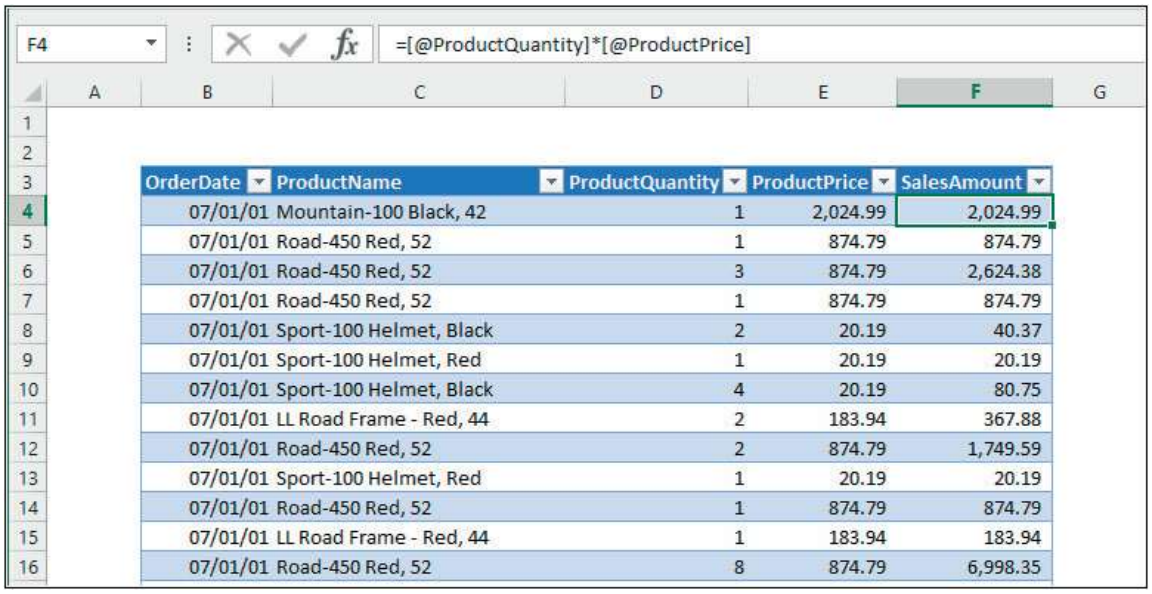
Istnieje spore prawdopodobieństwo, że znasz już język formuł programu Excel, do którego DAX jest w pewnym stopniu podobny. Ostatecznie korzenie języka DAX tkwią w rozszerzeniu Power Pivot for Excel, zaś projektujący je zespół starał się zachować podobieństwo obydwu języków, aby ułatwić użytkownikom przejście do nowego języka. Niemniej jednak istnieje pomiędzy nimi kilka bardzo istotnych różnic.

Komórki kontra tabele

W Excelu obliczenia są wykonywane względem komórek. Komórka wskazywana jest przy użyciu jej współrzędnych. Oznacza to, że pisane przez nas formuły wyglądają podobnie do pokazanej poniżej:

```
= (A1 * 1.25) - B2
```

DAX jest inny. W języku tym nie istnieje pojęcie komórki ani jej współrzędnych. Język DAX odwołuje się do tabel i kolumn, a nie komórek. Tak więc w wyrażeniach DAX pojawią się odniesienia tylko do tabel i zawartych w nich kolumn. Koncepcje tabel i kolumn nie są niczym nowym dla użytkowników Excela. W rzeczywistości, jeśli zdefiniujemy zakres komórek jako tabelę, używając polecenia Format as a Table (Formatuj jako tabelę), można pisać wyrażenia, które odwołują się właśnie do nazw tabel i kolumn. Spoglądając na rysunek 1-5 można zauważyć, że kolumna *SalesAmount* wylicza wyrażenie, które odwołuje się do kolumn w tej samej tabeli, a nie do określonych współrzędnymi komórek arkusza.



OrderDate	ProductName	ProductQuantity	ProductPrice	SalesAmount
07/01/01	Mountain-100 Black, 42	1	2,024.99	2,024.99
07/01/01	Road-450 Red, 52	1	874.79	874.79
07/01/01	Road-450 Red, 52	3	874.79	2,624.38
07/01/01	Road-450 Red, 52	1	874.79	874.79
07/01/01	Sport-100 Helmet, Black	2	20.19	40.37
07/01/01	Sport-100 Helmet, Red	1	20.19	20.19
07/01/01	Sport-100 Helmet, Black	4	20.19	80.75
07/01/01	LL Road Frame - Red, 44	2	183.94	367.88
07/01/01	Road-450 Red, 52	2	874.79	1,749.59
07/01/01	Sport-100 Helmet, Red	1	20.19	20.19
07/01/01	Road-450 Red, 52	1	874.79	874.79
07/01/01	LL Road Frame - Red, 44	1	183.94	183.94
07/01/01	Road-450 Red, 52	8	874.79	6,998.35

Rysunek 1-5 W Excelu również można używać nazw kolumn.

W formułach Excela odwołujemy się do kolumn w tabeli poprzez format `[@NazwaKolumny]`, gdzie *NazwaKolumny* jest nazwą (wpisem w nagłówku) kolumny, której chcemy użyć, zaś symbol @ oznacza „użyj wartości dla bieżącego wiersza". Choć składnia ta nie jest zbyt intuicyjna, zwykle nie musimy pisać takich wyrażeń Pojawiają się one po prostu po kliknięciu komórki, a Excel sam zajmie się wstawieniem do niej odpowiedniego kodu.

Możemy więc myśleć, że Excel udostępnia dwie różne metody wykonywania obliczeń: możemy posługiwać się zwykłymi odwołaniami do komórek (w tym przypadku formuła dla komórki `F4` przyjmie postać `E4*D4`) albo użyć odniesień do kolumn, o ile pracujemy wewnątrz tabeli. Wykorzystanie odniesień do kolumn ma tę zaletę, że można wówczas użyć dokładnie tego samego wyrażenia we wszystkich komórkach kolumny, a Excel obliczy formułę dla różnych wartości w każdym wierszu.

Język DAX pracuje na tabelach, zatem wszystkie formuły muszą odnosić się do kolumn. Na przykład pokazane wcześniej mnożenie zostanie zapisane w języku DAX w następujący sposób:

```
Sales[SalesAmount] = Sales[ProductPrice] * Sales[ProductQuantity]
```

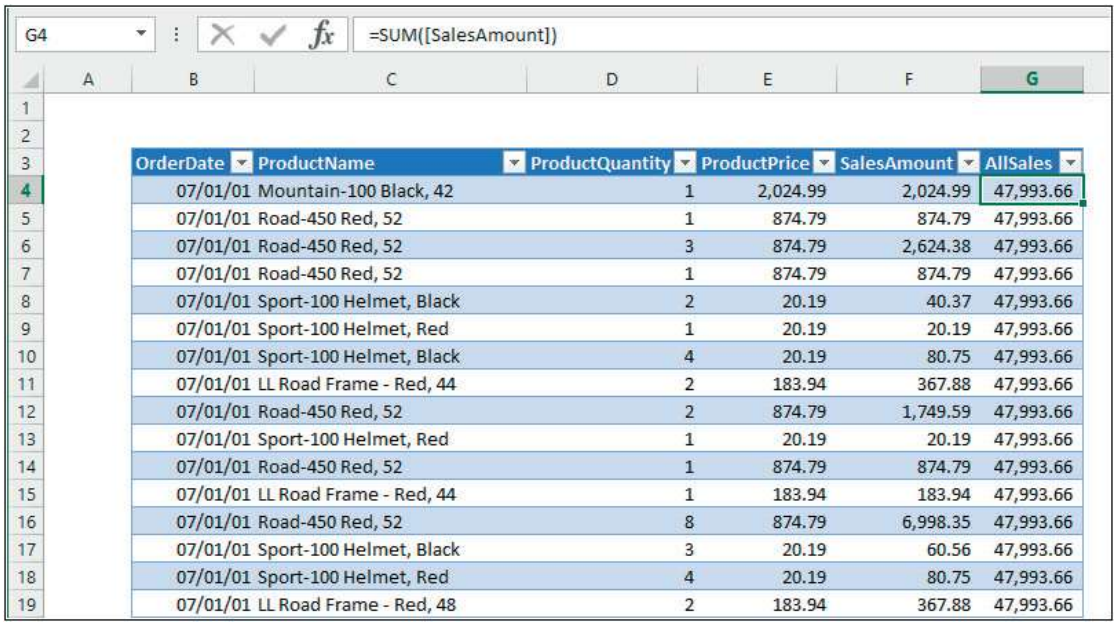
Jak widać, każda kolumna jest prefiksowana nazwą tabeli, do której należy. W Excelu nie podajemy nazwy tabeli, gdyż formuły Excela działają tylko w obrębie pojedynczej tabeli. W języku DAX, przeciwnie, odniesienie musi wskazywać nazwę tabeli, gdyż DAX działa w modelu danych zawierającym wiele tabel, przy czym kolumny z różnych tabel mogą (i często mają) mieć takie same nazwy.

Wiele funkcji DAX działa tak samo, jak odpowiadające im funkcje Excela. Na przykład funkcja IF wygląda dokładnie tak samo w języku DAX i w Excelu[1]:

```
Excel   IF ( [@SalesAmount] > 10, 1, 0)
DAX     IF ( Sales[SalesAmount] > 10, 1, 0)
```

Ważnym aspektem odróżniającym składnię formuły Excel i jej odpowiednik w języku DAX jest sposób odwoływania się do całej kolumny. Można zauważyć, że w zapisie `[@ProductQuantity]` znak `@` oznacza „wartość w bieżącym wierszu". Przy posługiwaniu się językiem DAX nie musimy tego wskazywać. Domyślne zachowanie języka polega na pobraniu wartości z bieżącego wiersza. Jeśli w Excelu chcemy odwołać się do całej kolumny (właśnie tak, do wszystkich wierszy w tej kolumnie), osiągamy to, usuwając symbol `@`, co można zauważyć na rysunku 1-6.

1 Stwierdzenie to jest prawdziwe jedynie w przypadku angielskiej (niezlokalizowanej) wersji programu Excel. W polskiej wersji interfejsu użytkownika formuła dla Excela przybierze postać `JEŻELI([@SalesAmount]>10;1;0)`. Warto zauważyć, że nie tylko zostało przetłumaczone słowo kluczowe `IF`, ale dodatkowo znak separatora zmienił się z przecinka na średnik. To ostatnie zachowanie, czyli zamianę przecinka na średnik, dostrzeżemy również w dodatku Power Pivot do polskiej wersji Excela, choć nazwy funkcji pozostają niezmienione (wszystkie przypisy pochodzą od tłumacza).



G4 =SUM([SalesAmount])

	A	B	C	D	E	F	G
1							
2							
3		OrderDate	ProductName	ProductQuantity	ProductPrice	SalesAmount	AllSales
4		07/01/01	Mountain-100 Black, 42	1	2,024.99	2,024.99	47,993.66
5		07/01/01	Road-450 Red, 52	1	874.79	874.79	47,993.66
6		07/01/01	Road-450 Red, 52	3	874.79	2,624.38	47,993.66
7		07/01/01	Road-450 Red, 52	1	874.79	874.79	47,993.66
8		07/01/01	Sport-100 Helmet, Black	2	20.19	40.37	47,993.66
9		07/01/01	Sport-100 Helmet, Red	1	20.19	20.19	47,993.66
10		07/01/01	Sport-100 Helmet, Black	4	20.19	80.75	47,993.66
11		07/01/01	LL Road Frame - Red, 44	2	183.94	367.88	47,993.66
12		07/01/01	Road-450 Red, 52	2	874.79	1,749.59	47,993.66
13		07/01/01	Sport-100 Helmet, Red	1	20.19	20.19	47,993.66
14		07/01/01	Road-450 Red, 52	1	874.79	874.79	47,993.66
15		07/01/01	LL Road Frame - Red, 44	1	183.94	183.94	47,993.66
16		07/01/01	Road-450 Red, 52	8	874.79	6,998.35	47,993.66
17		07/01/01	Sport-100 Helmet, Black	3	20.19	60.56	47,993.66
18		07/01/01	Sport-100 Helmet, Red	4	20.19	80.75	47,993.66
19		07/01/01	LL Road Frame - Red, 48	2	183.94	367.88	47,993.66

Rysunek 1-6 W Excelu można odwołać się do całej kolumny, pomijając symbol @ przed jej nazwą.

Wartość w kolumnie *AllSales* jest taka sama we wszystkich wierszach, gdyż jest to całkowita suma kolumny *SalesAmount*. Innymi słowy, mamy tu składniowe rozróżnienie wartości dla danej kolumny w bieżącym wierszu oraz wartości kolumny jako całości.

W języku DAX wygląda to inaczej. W tym przypadku wyrażenie *AllSales* pokazane na rysunku 1-6 należy napisać w ten sposób:

```
[AllSales] := SUM ( Sales[SalesAmount] )
```

Jak widać, nie istnieje składniowa różnica pomiędzy odwołaniem się do kolumny w celu pobrania wartości dla określonego wiersza, a użyciem wszystkich wartości z tej kolumny. DAX rozumie, że chcemy zsumować wszystkie wartości z kolumny, gdyż użyliśmy jej nazwy wewnątrz agregatora (w tym przypadku funkcji *SUM*), który wymaga przekazania nazwy kolumny jako parametru. Tak więc, podczas gdy Excel wymaga jawnego rozróżnienia (odpowiedniej składni) pomiędzy dwoma sposobami odczytywania danych, DAX dokonuje tego rozróżnienia w sposób automatyczny. Przynajmniej początkowo, może to być mylące i wymaga zmiany sposobu myślenia.

Excel i DAX: dwa języki funkcyjne

Podobieństwo pomiędzy obydwoma językami wynika z faktu, że zarówno język formuł Excela, jak i DAX są językami funkcyjnymi. Język funkcyjny jest zbudowany z wyrażeń, które są – zasadniczo – wywołaniami funkcji. Ani w Excelu, ani w DAX nie występują koncepcje poleceń, pętli czy skoków, które są powszechne w większości języków programowania. W języku DAX wszystko jest wyrażeniem. Ten aspekt języka

stanowi często wyzwanie dla programistów mających doświadczenie z innymi językami, ale nie powinien być niczym zaskakującym dla użytkowników Excela.

Korzystanie z iteratorów

Jedną z koncepcji, które mogą być nowością, są iteratory. Przy pracy w Excel przyzwyczailiśmy się do wykonywania obliczeń po jednym kroku na raz. W poprzednim przykładzie można zauważyć, że aby wyliczyć całkowitą wartość sprzedaży, najpierw utworzyliśmy kolumnę zawierającą cenę pomnożoną przez ilość sprzedanych elementów, a następnie, w drugim kroku, zsumowaliśmy tę kolumnę. Uzyskaną liczbę można teraz wykorzystać na przykład jako mianownik do wyliczenia procentowego udziału każdego produktu w sprzedaży.

Używając języka DAX można wykonać tę samą operację w jednym kroku poprzez użycie iteratora. Iterator robi dokładnie to, co sugeruje jego nazwa: iteruje tabelę (przechodzi przez kolejne wiersze) i wykonuje obliczenia dla każdego wiersza, agregując rezultaty, aby wytworzyć pojedynczą wartość końcową.

Sumę wszystkich wartości sprzedaży z poprzedniego przykładu można wyliczyć, używając iteratora `SUMX`:

```
[AllSales] :=
SUMX (
    Sales,
    Sales[ProductQuantity] * Sales[ProductPrice]
)
```

W podejściu tym można zauważyć zarówno korzyści, jak i niewygody. Zaletą jest to, że można wykonać wiele złożonych obliczeń w jednym kroku bez konieczności zajmowania się tworzeniem wielu dodatkowych kolumn, które są przydatne tylko do celów wyliczenia pewnych szczególnych formuł. Wadą jest to, że programowanie w DAX jest mniej oczywiste, niż w Excelu. W rzeczywistości nie widzimy kolumny obliczającej cenę pomnożoną przez ilość; istnieje ona tylko wirtualnie, podczas wykonywania obliczeń.

Prawdę mówiąc, nadal mamy opcję utworzenia obliczanej kolumny, która będzie zawierała iloczyn ceny i ilości. Tym niemniej, jak dowiemy się później, rzadko jest to dobra praktyka, gdyż zużywa cenną pamięć i może spowolnić działanie wszystkich obliczeń.

DAX wymaga nieco teorii

Powiedzmy to jasno: nie jest to różnica pomiędzy językami programowania; różnica leży w sposobie myślenia. Podobnie jak każdy inny mieszkaniec tej planety, zapewne często przeglądasz Sieć w poszukiwaniu złożonych formuł i wzorów rozwiązań dla scenariuszy, które próbujesz rozwiązać. Przy korzystaniu z Excela są spore szanse,

że znajdziemy formułę, która robi niemal dokładnie to, co trzeba. Wystarczy skopiować formułę, dostosować ją do swoich potrzeb i następnie jej użyć, nie zastanawiając się zbytnio, jak działa.

Na przykład w jednym z arkuszy, których używam codziennie, mam następującą formułę (trzeba pamiętać, że jest to jeden wiersz kodu, zawinięty ze względu na szerokość drukowanej strony w książce):

```
{ = SUM ( IF ( ('Transactions'!$B$5:$B$991>=M30) *
  ('Transactions'!$B$5:$B$991<=N30),1,0) ) }
```

Nie jestem do końca pewien, jak właściwie działają formuły w nawiasach klamrowych i jak wyliczane jest to wyrażenie `IF`. Mówiąc szczerze, pamiętam tylko, że muszę potwierdzić tę formułę dość dziwną kombinacją klawiszy. Tak więc, to działa, zawsze działało i interesuje mnie wyliczana liczba, a nie to, jak to obliczenie się odbywa. Tak więc, będąc autorem licznych książek i ekspertem od języka DAX, również należę do tej kategorii użytkowników.

Podejście to, które sprawdza się w Excelu, nie działa w przypadku DAX. Konieczne jest przestudiowanie pewnej ilości teorii i dokładne zrozumienie, jak działają konteksty wykonania, zanim posiądzie się umiejętność pisania dobrego kodu DAX. Bez właściwych podstaw teoretycznych DAX albo będzie działać jak magia, albo będziemy otrzymywać dziwne liczby, które nie mają żadnego sensu. Problemem nie jest jednak język DAX, ale fakt, że użytkownik nie zrozumiał jeszcze, jak on działa.

Szczęśliwie teoria leżąca u podstaw DAX ograniczona jest do kilku ważnych koncepcji, które przedstawimy w rozdziale 4, „Istota kontekstów wykonania". Gdy dojdziemy do tego rozdziału, trzeba będzie odnaleźć worek z kapciami i przygotować się na powrót do szkolnej ławki na pewien czas. Gdy jednak opanujesz jego zawartość, DAX nie będzie już miał przed tobą tajemnic i dalsza nauka będzie polegała głównie na zdobywaniu doświadczenia. Jednak nie powinieneś próbować pójść dalej, dopóki nie opanujesz dobrze tego kawałka teorii. Przypomnę: wiedza to połowa zwycięstwa.

DAX dla programistów SQL

Jeśli posługujesz się językiem SQL, pracowałeś już z wieloma tabelami i tworzyłeś między nimi połączenia, aby ustanowić relacje. Z tego punktu widzenia powinieneś poczuć się znajomo w świecie DAX, gdyż przetwarzanie w DAX polega na odpytywaniu zbioru tabel powiązanych relacjami i agregowaniu wartości.

Obsługiwanie relacji

Pierwszą różnicą pomiędzy SQL a DAX jest sposób działania relacji w modelu. W środowisku SQL możemy zdefiniować ograniczenia obcego klucza, aby zadeklarować relacje pomiędzy tabelami, ale silnik bazy danych nigdy nie użyje tych kluczy w zapytaniach, o ile nie wskażemy ich jawnie. Jeśli na przykład mamy tabelę *Customer* oraz tabelę *Sales*, przy czym kolumna *CustomerKey* jest kluczem głównym tabeli *Customer* i kluczem obcym w tabeli *Sales*, możemy napisać zapytanie podobne do poniższego:

```
SELECT
    Customers.CustomerName,
    SUM ( Sales.SalesAmount ) AS SumOfSales
FROM
    Sales
    INNER JOIN Customers
        ON Sales.CustomerKey = Customers.CustomerKey
GROUP BY
    Customers.CustomerName
```

Choć zadeklarowaliśmy relację pomiędzy tabelami w naszym modelu, używając obcych kluczy, nadal musimy jawnie określić warunek złączenia w zapytaniu. Choć to sprawia, że zapytania są nieco bardziej rozbudowane, jest to przydatne, bo pozwala użyć różnych warunków złączenia w różnych zapytaniach, co daje wielką swobodę w sposobie budowania zapytań.

W języku DAX relacje są częścią modelu i wszystkie są typu `LEFT OUTER JOIN` (lewe złączenia zewnętrzne). Po zdefiniowaniu w modelu nie musimy już deklarować typu złączenia w zapytaniu: DAX automatycznie użyje `LEFT OUTER JOIN` w zapytaniu, o ile użyjemy kolumn odwołujących się do tabeli głównej. Tak więc poprzednie zapytanie SQL można w DAX zapisać następująco:

```
EVALUATE
SUMMARIZE (
    Sales, Customers[CustomerName],
    "SumOfSales", SUM ( Sales[SalesAmount] )
)
```

Ponieważ DAX zna istniejącą relację pomiędzy tabelami *Sales* i *Customers*, wykona automatycznie złączenie zgodne z modelem. Na koniec funkcja `SUMMARIZE` potrzebuje wykonać grupowanie według kolumny *Customers[CustomerName]*, ale nie potrzebujemy do tego żadnego specjalnego słowa kluczowego: `SUMMARIZE` automatycznie grupuje dane według wybranych kolumn.

DAX jest językiem funkcyjnym

SQL jest językiem deklaratywnym. To, czego potrzebujemy, definiujemy poprzez deklarowanie zbioru danych do odczytania przy użyciu wyrażeń `SELECT`, nie zastanawiając się, jak silnik faktycznie będzie pobierał potrzebne informacje. Język DAX jest natomiast językiem funkcyjnym.

W DAX każde wyrażenie jest wywołaniem funkcji, a parametry tej funkcji mogą być wywołaniami kolejnych funkcji. Przetwarzanie parametrów może prowadzić do bardzo złożonych planów zapytań, które DAX wykonuje w celu obliczenia wyniku.

Na przykład, aby pobrać tylko dane klientów, którzy mieszkają w Europie, moglibyśmy napisać takie zapytanie w SQL:

```
SELECT
    Customers.CustomerName,
    SUM ( Sales.SalesAmount ) AS SumOfSales
FROM
    Sales
    INNER JOIN Customers
        ON Sales.CustomerKey = Customers.CustomerKey
WHERE
    Customers.Continent = 'Europe'
GROUP BY
    Customers.CustomerName
```

Przy korzystaniu z DAX nie deklarujemy warunku `WHERE` w zapytaniu. Zamiast tego trzeba użyć określonej funkcji (`FILTER`) do ograniczenia zwracanych wyników:

```
EVALUATE
SUMMARIZE (
    FILTER (
        Customers,
        Customers[Continent] = "Europe"
    ),
    Customers[CustomerName],
    "SumOfSales", SUM ( Sales[SalesAmount] )
)
```

Można tu zauważyć, że `FILTER` jest funkcją: zwróci ona tylko tych klientów, którzy mieszkają w Europie, wytwarzając oczekiwany wynik. Kolejność zagnieżdżania kolejnych funkcji i rodzaje użytych funkcji mają wielki wpływ na finalny wynik, a także

na wydajność działania. To samo zdarza się też w SQL, ale w tym przypadku ufamy, że optymalizator zapytań silnika SQL znajdzie optymalny plan zapytania. W języku DAX, choć optymalizator również wykonuje świetną pracę, na programiście spoczywa większa odpowiedzialność za napisanie dobrego kodu.

DAX jako język programowania i zapytań

W świecie SQL istnieje jawne rozróżnienie pomiędzy językiem zapytań a językiem programowania; w tym drugim przypadku chodzi o zbiór instrukcji służących do tworzenia procedur składowanych, widoków, wyzwalaczy i innych części kodu w bazie danych. Każdy dialekt SQL zawiera swoje własne wyrażenia, które pozwalają wzbogacić model danych o własny kod. DAX przeciwnie, nie czyni w zasadzie żadnego rozróżnienia pomiędzy odpytywaniem a programowaniem. Bogaty zbiór funkcji pozwala manipulować tabelami i potrafi w efekcie zwracać tabele. Pokazana przed chwilą funkcja `FILTER` jest dobrym przykładem takiego działania.

Tak więc z tego punktu widzenia DAX jest prostszy niż SQL. Gdy opanuje się go jako język programowania (co zazwyczaj jest pierwszym zastosowaniem), wie się już wszystko, co potrzebne, aby móc używać go również jako języka zapytań

Podzapytania i warunki w DAX i SQL

Jedną z najsilniejszych funkcjonalności SQL jako języka zapytań jest możliwość używania podzapytań. DAX dysponuje kilkoma podobnymi koncepcjami, choć w przypadku podzapytań DAX wynikają one w sposób naturalny z funkcyjnej natury języka.

Na przykład, aby odczytać dane klientów i łączny wynik sprzedaży jedynie tych klientów, którzy dokonali zakupu za więcej niż 100 USD, możemy napisać następujące zapytanie SQL:

```
SELECT
    CustomerName,
    SumOfSales
FROM (
    SELECT
        Customers.CustomerName,
        SUM ( Sales.SalesAmount ) AS SumOfSales
    FROM
        Sales
        INNER JOIN Customers
            ON Sales.CustomerKey = Customers.CustomerKey
    GROUP BY
        Customers.CustomerName
    ) AS SubQuery
WHERE
    SubQuery.SumOfSales > 100
```

Ten sam rezultat możemy uzyskać w DAX, po prostu zagnieżdżając wywołanie funkcji:

```
EVALUATE
FILTER (
    SUMMARIZE (
        Customers,
        Customers[CustomerName],
        "SumOfSales", SUM ( Sales[SalesAmount] )
    ),
    [SumOfSales] > 100
)
```

W tym kodzie podzapytanie odczytujące *CustomerName* oraz *SumOfSales* jest następnie przekazywane do funkcji `FILTER`, która zwraca tylko te wiersze, w których *SumOfSales* jest większa niż 100. Na razie ten kod może wydawać się nieczytelny, ale niedługo, gdy już zaczniemy uczyć się języka DAX, będzie można zauważyć, że korzystanie z podzapytań jest znacznie prostsze, niż w SQL, i że przebiega naturalnie dzięki temu, że DAX jest językiem funkcyjnym.

DAX dla projektantów MDX

Wielu profesjonalistów BI rozpoczyna naukę języka DAX, gdyż jest to nowy język modelu tabelarycznego SSAS (SQL Server Analysis Services Tabular), a w przeszłości używali języka MDX do budowania i odpytywania modeli wielowymiarowych SSAS (SSAS Multidimensional). Jeśli należysz do tej grupy, musisz przygotować się na naukę zupełnie nowego języka: DAX i MDX niewiele mają wspólnego ze sobą. Co gorsze, niektóre koncepcje dostępne w DAX mogą przypominać podobne koncepcje istniejące w MDX, choć w rzeczywistości bardzo się różnią.

W istocie, z naszych doświadczeń wynika, że nauka języka DAX po MDX jest najbardziej wymagającą opcją. Aby móc opanować DAX, trzeba oczyścić umysł ze wszystkiego, co wie się o MDX; trzeba spróbować zapomnieć wszystko na temat wielowymiarowych przestrzeni i nastawić się na poznawanie nowego języka z czystym umysłem.

Model wielowymiarowy kontra tabelaryczny

MDX działa w świecie wielowymiarowym zdefiniowanym przez nasz model. Kształt tej przestrzeni oparty jest na architekturze wymiarów i hierarchiach zdefiniowanych w modelu, które z kolei definiują zbiory współrzędnych przestrzeni wielowymiarowej. Przecięcia tych zbiorów członków w różnych wymiarach definiują punkty przestrzeni wymiarowych. Domyślamy się, że nieco czasu mogło zająć zrozumienie, że członek *[All]* dowolnej hierarchii atrybutów jest naprawdę punktem w wielowymiarowej przestrzeni.

DAX działa w znacznie prostszy sposób. Nie ma tu wymiarów, członków ani punktów przestrzeni. Innymi słowy, w ogóle nie ma tu żadnej wielowymiarowej przestrzeni. Istnieją tu hierarchie, które można zdefiniować w modelu, ale są to struktury bardzo różniące się od hierarchii w MDX. Świat DAX jest zbudowany z tabel, kolumn i relacji. Każda tabela w modelu tabelarycznym nie jest ani grupą miar, ani wymiarem: jest po prostu tabelą i, aby obliczyć jakieś wartości, trzeba ją przeskanować, odfiltrować lub zsumować zawarte w niej liczby. Wszystko oparte jest na dwóch prostych koncepcjach tabel i relacji.

Zapewne niedługo odkryjesz, że z punktu widzenia modelowania wariant tabelaryczny oferuje mniej opcji, niż wariant wielowymiarowy. Jednak mniejsza liczba opcji w tym przypadku nie oznacza mniejszych możliwości, gdyż mamy język programowania (czyli DAX), który pozwala wzbogacić ten model. Prawdziwą siłą modelowania w wariancie tabelarycznym jest zawrotna prędkość DAX. W istocie zapewne nabrałeś nawyku unikania wprowadzania zbyt wiele kodu MDX do swojego modelu, gdyż optymalizowanie prędkości MDX jest zwykle bardzo trudne. DAX natomiast jest zadziwiająco szybki. Tym samym większość złożoności obliczeń nie znajdzie się w samym modelu, ale w formułach języka DAX.

DAX jako język programowania i zapytań

Zarówno DAX, jak i MDX są zarówno językami programowania, jak i językami zapytań. W przypadku MDX rozróżnienie obu ról jest jasne poprzez obecność skryptów MDX. Używamy języka MDX w skryptach MDX wraz z wieloma specjalnymi wyrażeniami, które mogą być użyte tylko w skryptach (na przykład wyrażenia `SCOPE`), ale używamy też języka MDX w zapytaniach, gdy tworzymy wyrażenia `SELECT` odczytujące dane. W języku DAX wygląda to nieco inaczej. Możemy użyć DAX jako języka programowania, aby zdefiniować kolumny obliczane (nowa koncepcja w DAX, która nie występuje w MDX) i miary (podobne do obliczanych członków w MDX). Możemy również użyć DAX jako języka zapytań, na przykład w celu odczytania danych z modelu tabelarycznego przy użyciu Reporting Services. Tym niemniej nie istnieją żadne specjalne funkcje w języku DAX, których można byłoby użyć tylko w jednym z tych dwóch zastosowań języka. Co więcej, możemy odpytywać model tabelaryczny również przy użyciu MDX. Tak więc, część zapytaniowa MDX nadal działa w modelu tabelarycznym, ale DAX jest jedyną opcją, gdy chodzi o programowanie tego modelu.

Hierarchie

Przy posługiwaniu się MDX polegamy na hierarchiach do wykonywania większości obliczeń Jeśli na przykład chcemy wyliczyć sprzedaż w minionym roku, musimy pobrać członka *PrevMember* dla *CurrentMember* w hierarchii *Year* i użyć go do przepisania filtru MDX. Poniższy przykład definiuje takie obliczenie dla poprzedniego roku w MDX:

```
CREATE MEMBER CURRENTCUBE.[Measures].[SamePeriodPreviousYearSales] AS
    (
        [Measures].[Sales Amount],
        ParallelPeriod (
            [Date].[Calendar].[Calendar Year],
            1,
            [Date].[Calendar].CurrentMember
        )
    ),
```

Miara używa funkcji *ParallelPeriod*, która zwraca skojarzonego członka dla *CurrentMember* w hierarchii *Calendar*. Tak więc jest ona oparta na hierarchiach zdefiniowanych w modelu. To samo obliczenie w języku DAX możemy napisać, używając kontekstu filtra i standardowych funkcji analizy czasu:

```
[SamePeriodPreviousYearSales] :=
CALCULATE (
    SUM ( Sales[Sales Amount] ),
    SAMEPERIODLASTYEAR ( 'Date'[Date] )
)
```

To samo obliczenie można napisać na wiele innych sposobów, używając `FILTER` i innych funkcji DAX, ale idea pozostaje niezmieniona: zamiast używania hierarchii, filtrujemy tabele. Ta różnica jest olbrzymia i zapewne będziesz tęsknił za obliczeniami opartymi na hierarchiach, dopóki nie przyzwyczaisz się do języka DAX.

Inna istotna różnica polega na tym, że w MDX odwołujemy się do *[Measures].[Sales Amount]* i funkcji agregującej, która musi być już zdefiniowana w modelu (i do tego trzeba było się przyzwyczaić). W DAX nie ma wstępnie zdefiniowanych agregacji. W istocie, jak można zauważyć, wyrażenie do obliczenia to `SUM(Sales[SalesAmount])`. Model nie zawiera już wstępnie zdefiniowanej agregacji; trzeba ją zdefiniować, gdy zechcemy jej użyć (można oczywiście utworzyć miarę, która będzie przechowywać sumę sprzedaży, ale w tym miejscu nie chcemy się zbytnio nad tym rozwodzić).

Kolejna ważna różnica pomiędzy językiem DAX a MDX to fakt, że ten drugi intensywnie wykorzystuje wyrażenie `SCOPE` do implementacji logiki biznesowej (ponownie przy użyciu hierarchii), podczas gdy ten pierwszy wymaga zupełnie odmiennego podejścia, gdyż obsługa hierarchii jest w tym języku po prostu nieobecna.

Na przykład, jeśli zechcemy wyczyścić miarę na poziomie *Year* (roku), w MDX moglibyśmy napisać takie wyrażenie:

```
SCOPE ( [Measures].[SamePeriodPreviousYearSales], [Date].[Month].[All] )
    THIS = NULL,
END SCOPE,
```

W języku DAX nie istnieje wyrażenie `SCOPE`. Aby uzyskać ten sam wynik, trzeba sprawdzić obecność filtrów w kontekście filtru i rozwiązanie jest znacznie bardziej skomplikowane:

```
[SamePeriodPreviousYearSales] :=
IF (
    ISFILTERED ( 'Date'[Month] ),
    CALCULATE (
        SUM ( Sales[Sales Amount] ),
        SAMEPERIODLASTYEAR ( 'Date'[Date] )
    ),
    BLANK()
)
```

Później wyjaśnimy szczegółowo, co ta formuła oblicza, ale intuicyjnie można zauważyć, że zwraca ona wartość tylko wtedy, gdy użytkownik przegląda hierarchię kalendarza na poziomie miesiąca lub głębiej, w przeciwnym razie zwracając `BLANK`. Formuła ta jest dużo bardziej podatna na błędy, niż jej odpowiednik w kodzie MDX. Uczciwie mówiąc, obsługa hierarchii jest tą funkcjonalnością, której naprawdę brakuje w języku DAX.

Obliczenia na poziomie liści

Na zakończenie jeszcze jedna uwaga: używając MDX zapewne nabrałeś nawyku unikania obliczeń na poziomie liści. Wykonywanie takich obliczeń w MDX okazuje się tak powolne, że zawsze preferowane jest wstępne obliczenie wartości i wykorzystanie agregacji do zwracania wyników. W języku DAX obliczenia na poziomie liści odbywają się niebywale szybko, a wstępne agregacje w ogóle nie istnieją. Będzie to wymagało zmiany sposobu myślenia, gdy przyjdzie czas na budowanie modelu danych. W większości przypadków model, który doskonale pasuje do SSAS Multidimensional, nie jest właściwym wyborem dla modelu tabelarycznego i *vice versa*.

ROZDZIAŁ 2

Wprowadzenie do DAX

Po krótkim przedstawieniu w poprzednim rozdziale przyszedł czas, aby zacząć mówić o języku DAX. W tym rozdziale poznamy składnię tego języka, różnice pomiędzy kolumną obliczaną a miarą (nazywaną również polem obliczanym w terminologii Excela) oraz najczęściej używane funkcje.

Ponieważ jest to rozdział wprowadzający, wiele funkcji nie zostanie omówionych szczegółowo. W dalszych częściach książki zagłębimy się bardziej w poszczególne zagadnienie i wyjaśnimy je szczegółowo. Na razie wystarczy przedstawienie samych funkcji i przyjrzenie się językowi DAX w ogólności.

Istota obliczeń DAX

Aby móc wyrazić złożone formuły, konieczne jest opanowanie podstaw języka DAX, a więc składni, różnych typów danych, które DAX może obsłużyć, podstawowych operatorów oraz sposobów odwoływania się to tabel i kolumn. Koncepcje te zostaną omówione w kilku kolejnych podpunktach.

Język DAX jest używany przede wszystkim do obliczania jakiś wartości na podstawie zawartości kolumn w tabelach. Można wykonywać agregacje, obliczenia i wyszukiwanie liczb, ale ostatecznie wszystkie te obliczenia angażują tabele i kolumny. Zatem pierwsza składnia, którą trzeba poznać, to odwołanie do kolumny w tabeli.

Ogólny format polega na wypisaniu tabeli ujętej w pojedyncze nawiasy, po której następuje nazwa kolumny obramowana nawiasami kwadratowymi, jak poniżej:

```
'Sales'[Quantity]
```

Można pominąć cudzysłowy, jeśli nazwa tabeli nie zaczyna się od cyfry, nie zawiera spacji i nie jest słowem zarezerwowanym (takim jak Date lub Sum).

Uwaga Dobrą praktyką jest unikanie spacji w nazwach tabel. W ten sposób można wyeliminować cudzysłowy z formuł, które zwykle utrudniają odczytywanie kodu. Trzeba jednak mieć na uwadze, że nazwa tabeli jest tą samą nazwą, którą zobaczymy przy przeglądaniu modelu za pomocą tabel przestawnych lub innego narzędzia klienckiego, takiego jak Power View. Zatem, jeśli chcemy mieć spacje w nazwach tabel w naszych raportach, będzie trzeba pogodzić się z obecnością pojedynczych cudzysłowów w naszym kodzie.

Możliwe jest również pominięcie samej nazwy tabeli, o ile odwołujemy się do kolumny lub miary w tej samej tabeli, w której definiujemy formułę. Tak więc `[Quantity]` jest poprawnym odwołaniem do kolumny, jeśli zostanie wpisane w kolumnie obliczanej lub mierze w tabeli *Sales*. Jednak choć technika ta jest składniowo poprawna, a nawet może być zasugerowana przez interfejs użytkownika, gdy wskażemy kolumnę, zamiast ją wpisać, zdecydowanie odradzamy takie postępowanie. Taka składnia sprawia, że kod jest trudniejszy do odczytania i zrozumienia, zatem zawsze lepiej użyć nazwy tabeli przy odwoływaniu się do kolumny w wyrażeniach DAX.

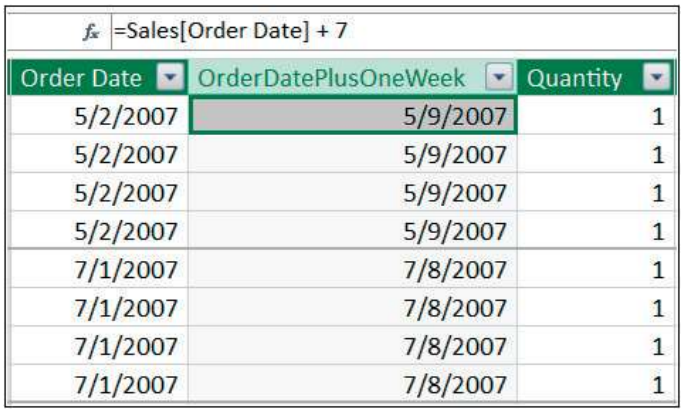
Typy danych DAX

DAX może wykonywać obliczenia na liczbach należących do różnych typów numerycznych (łącznie siedmiu). Poniższa lista pokazuje zarówno nazwę występującą w języku DAX, jak i częściej spotykaną nazwę tego samego typu (w nawiasach), używaną w relacyjnych bazach danych, takich jak SQL Server. Na przykład wartości logiczne (*Boolean*) w terminologii DAX nazywają się *TRUE/FALSE*. Osobiście wolimy się trzymać standardów nazewniczych i będziemy mówić o nich jako o wartościach logicznych.

- Whole Number (*Integer*) (liczba całkowita)
- Decimal Number (*Float*) (liczba dziesiętna)
- Currency (*Currency*), (waluta), liczba dziesiętna o stałej liczbie cyfr ułamkowych, wewnętrznie przechowywana jako liczba całkowita (liczba centów/groszy itp.)
- Date (*DateTime*) (data i czas)
- TRUE/FALSE (*Boolean*) (logiczna)
- Text (*String*) (tekst)
- Binary large object (*BLOB*) (wielki obiekt binarny)

DAX dysponuje wydajnym systemem obsługi typów, dzięki czemu nie musimy się zbytnio martwić typami danych: przy pisaniu wyrażenia DAX wynikowy typ będzie oparty na typie argumentów użytych w wyrażeniu. Trzeba mieć tego świadomość, gdy okaże się, że typ zwrócony przez wyrażenie DAX nie będzie zgodny z oczekiwaniem: trzeba wówczas sprawdzić typy danych argumentów użytych w samym wyrażeniu.

Na przykład, jeśli jeden z argumentów sumy jest datą, wynik również będzie datą; jeśli jednak ten sam operator zostanie użyty do liczb całkowitych, wynik będzie całkowity. Zachowanie takie jest znane jako *przeciążanie operatorów* i można zobaczyć je w przykładzie pokazanym na rysunku 2-1, gdzie kolumna *OrderDatePlusOneWeek* jest obliczana poprzez dodanie liczby całkowitej 7 do wartości w kolumnie *Order Date*. Wynik tego dodawania jest datą.



fx =Sales[Order Date] + 7

Order Date	OrderDatePlusOneWeek	Quantity
5/2/2007	5/9/2007	1
5/2/2007	5/9/2007	1
5/2/2007	5/9/2007	1
5/2/2007	5/9/2007	1
7/1/2007	7/8/2007	1
7/1/2007	7/8/2007	1
7/1/2007	7/8/2007	1
7/1/2007	7/8/2007	1

Rysunek 2-1 Dodanie liczby całkowitej do daty zwraca datę zwiększoną o odpowiednią liczbę dni.

Jako uzupełnienie przeciążania operatorów, DAX automatycznie konwertuje łańcuchy tekstowe na liczby i liczby na łańcuchy, gdy jest to wymagane przez użyty operator. Na przykład jeśli użyjemy operatora &, który powoduje konkatenację łańcuchów, DAX przekształci argumenty na łańcuchy. Dla przykładu formuła

```
= 5 & 4
```

zwróci tekst „54”. Z drugiej jednak strony formuła

```
= "5" + "4"
```

zwróci liczbę całkowitą równą 9.

Wynikowa wartość zależy od użytego operatora, a nie od typów źródłowych kolumn, które są konwertowane zgodnie z wymaganiami operatora. Choć zachowanie to wygląda na bardzo wygodne, w dalszej części rozdziału pokażemy, jakie rodzaje błędów mogą wystąpić podczas takich automatycznych konwersji. Z tych powodów sugerujemy unikanie ich. Jeśli potrzebny jest jakiś rodzaj konwersji, znacznie lepiej będzie, jeśli przejmiemy nad nią kontrolę i wykonamy ją jawnie. Zgodnie z takim podejściem ostatni pokazany przykład powinien wyglądać następująco:

```
= VALUE ( "5" ) + VALUE ( "4" )
```

Typy danych języka DAX zapewne będą wyglądać znajomo dla osób przyzwyczajonych do pracy z Excelem lub innymi językami programowania. Pełną specyfikację typów danych DAX można znaleźć pod adresem *http://msdn.microsoft.com/en-us/library/gg492146.aspx*. Tym niemniej przydatne będzie pokazanie kilku uwarunkowań dotyczących każdego z tych typów.

Liczba całkowita (*Integer*)

Język DAX ma tylko jeden typ całkowitoliczbowy (*Integer*) o rozmiarze 64 bitów – jego odpowiednikiem jest typ *bigint* w języku Transact-SQL. Również wszystkie wewnętrzne obliczenia oparte na liczbach całkowitych używają wartości 64-bitowych.

Liczba dziesiętna (*Float*)

Liczby dziesiętne są zawsze przechowywane jako liczby zmiennoprzecinkowe podwójnej precyzji. Nie należy mylić tego typu danych z typami *decimal* czy *numeric* znanych z języka *Transact-SQL*: odpowiednikiem tego typu jest *Float*. Można zauważyć, że również ten typ zajmuje 64 bity.

Waluta (*Currency*)

Typ danych *Currency* przechowuje liczbę dziesiętną o stałej liczbie miejsc po znaku dziesiętnym. Liczba cyfr dziesiętnych wynosi cztery, zaś sama wartość jest wewnętrznie przechowywana jako 64-bitowa liczba całkowita (dzielona przez 10 000). Wszystkie obliczenia wykonywane na danych typu *Currency* zawsze ignorują część ułamkową wykraczającą poza czwartą cyfrę dziesiętną. Jeśli potrzebna jest większa dokładność, konieczne jest wykonanie konwersji tych liczb do typu *Decimal Number*.

Domyślny format prezentacji typu *Currency* zawiera symbol waluty. Można również zastosować formatowanie walutowe do liczb całkowitych i dziesiętnych, a także użyć formatu bez symbolu waluty dla typu danych *Currency*.

Data (*DateTime*)

Język DAX przechowuje daty i czas w postaci typu *DateTime*, analogicznego do SQL. Format ten wewnętrznie używa liczby zmiennoprzecinkowej, w której część całkowita odpowiada liczbie dni, które upłynęły od 30 grudnia 1899 roku, zaś część ułamkowa identyfikuje fragment dnia (od północy). Godziny, minuty i sekundy są przekształcane na tę część ułamkową. Tym samym poniższe wyrażenie zwraca bieżącą datę powiększoną o jeden dzień(dokładnie o 24 godziny):

```
= NOW () + 1
```

Wynikiem będzie jutrzejsza data o tej samej godzinie, co czas wykonania obliczenia. Jeśli chcemy otrzymać tylko część daty dla typu *DateTime*, trzeba pamiętać o użyciu `TRUNC` w celu odrzucenia części ułamkowej.

Bug roku przestępnego

Lotus 1-2-3, popularny arkusz kalkulacyjny opublikowany w roku 1983, zawierał błąd dotyczący obsługi lat przestępnych dla typu *DateTime*. Uznawał on rok 1900 za rok przestępny, choć w rzeczywistości nim nie był (ostatni rok stulecia jest rokiem przestępnym tylko wtedy, gdy dwie pierwsze cyfry – liczba setek lat – dzielą się przez 4 bez reszty). Zespół projektowy pierwszej wersji Excel z premedytacją powielił ten błąd, aby zachować kompatybilność z Lotus 1-2-3. Od tego czasu każda nowa wersja Excela kontynuuje utrzymanie tego błędu ze względu na kompatybilność z poprzednimi wersjami.

Obecnie, w roku 2015, bug ten nadal jest obecny w DAX z tych samych powodów – dla utrzymania kompatybilności. Obecność tego błędu (może powinniśmy go nazywać właściwością?) może powodować błędy w obliczeniach dotyczących okresów poprzedzających datę 1 marca 1900 r. Z tego względu pierwszą datą oficjalnie obsługiwaną przez DAX jest właśnie 1 marca 1900. Obliczenia dat wykonane dla okresów przed tą datą mogą prowadzić do błędów i powinny być uważane za niedokładne.

Jeśli konieczne jest wykonywanie obliczeńdla dat położonych przed rokiem 1900, należy przeliczyć te daty na położone po roku 1900, wykonać konieczne obliczenia, po czym ponownie przeliczyć daty na wcześniejsze.

Logiczne (*TRUE/FALSE*)

Typ danych *Boolean* (logiczny) służy do wyrażania warunków logicznych. Na przykład kolumna obliczana definiowana przez poniższe wyrażenie ma typ *Boolean*:

```
= Sales[Unit Price] > Sales[Unit Cost]
```

Wartości typu *Boolean* można również uważać za liczby, przy czym *TRUE* jest równe 1, a *FALSE* jest równe 0. Jest to niekiedy przydatne przy sortowaniu, gdyż *TRUE* > *FALSE*.

Tekst (*String*)

Każdy łańcuch tekstowy w języku DAX jest przechowywany jako łańcuch *Unicode*, w którym każdy znak jest przechowywany w dwóch bajtach (16 bitach). Domyślnie porównywanie łańcuchów tekstowych odbywa się bez rozróżniania wielkości liter, zatem teksty „Power Pivot” i „POWER PIVOT” są uważane za równe.

Wielki obiekt binarny (*BLOB*)

Typ danych *BLOB* jest używany w modelach danych do przechowywania obrazów lub innych plików binarnych i nie jest dostępny w języku DAX. Jest on używany głównie przez Power View i inne, podobne narzędzia klienckie do prezentowania obrazów (filmów, nagrańitp.) przechowywanych wprost w modelu danych.

Operatory języka DAX

Zwróciliśmy już uwagę na ważność operatorów jako determinujących wynikowy typ wyrażenia. Możemy teraz zaprezentować pełną listę operatorów dostępnych w DAX (tabela 2-1).

Tabela 2-1 Operatory

Rodzaj operatora	Symbol	Użycie	Przykład
Nawiasy	()	Zmiana kolejności wykonywania działań oraz grupowanie argumentów	(5+2) * 3
Arytmetyczne	+	Dodawanie	4 + 2
	-	Odejmowanie	5 – 3
	*	Mnożenie	4 * 2
	/	Dzielenie	4 / 2
Porównania	=	Równe	[CountryRegion] = "USA"
	<>	Nierówne	[CountryRegion] <> "USA"
	>	Większe niż	[Quantity] > 0
	>=	Większe lub równe niż	[Quantity] >= 100
	<	Mniejsze niż	[Quantity] < 0
	<=	Mniejsze lub równe niż	[Quantity] <= 100
Złączanie tekstów	&	Scalanie łańcuchów znaków (tekstów)	"Value is: " & [Amount]
Logiczne	&&	Koniunkcja dwóch wyrażeń logicznych	[Country] = "USA" && [Quantity] > 0
	||	Alternatywa dwóch wyrażeń logicznych	[Country] = "USA" || [Quantity] > 0
	!	Zaprzeczenie	! [Country] = "USA"

Co więcej, operatory logiczne są również dostępne jako funkcje DAX, o składni bardzo podobnej do używanej w Excelu. Na przykład możemy napisać:

```
AND ( [CountryRegion] = "USA", [Quantity] > 0 )
OR  ( [CountryRegion] = "USA", [Quantity] > 0 )
```

co jest równoważne poniższym warunkom (odpowiednio):

```
[CountryRegion] = "USA" && [Quantity] > 0
[CountryRegion] = "USA" || [Quantity] > 0
```

Posługiwanie się tymi funkcjami zamiast operatorami logicznymi może być bardzo użyteczne, gdy zachodzi potrzeba napisania bardziej złożonych warunków. W istocie, gdy trzeba sformatować obszerne fragmenty kodu, funkcje okazują się łatwiejsze do zapisu i formatowania, niż operatory. Wiąże się to jednak ze znaczącym ograniczeniem: do funkcji można przekazać tylko dwa parametry na raz. Oznacza to konieczność zagnieżdżania funkcji, gdy trzeba ocenić więcej niż dwa warunki.

Kolumny obliczane i miary

Po poznaniu podstaw składni języka DAX zajmiemy się poznawaniem jednej z najważniejszych koncepcji występującej w tym języku: rozróżnienia pomiędzy obliczanymi kolumnami a miarami. Mimo że na pierwszy rzut oka mogą wydawać się podobne, gdyż można pewne obliczenia wykonać obiema metodami, w rzeczywistości bardzo się różnią i zrozumienie tej różnicy jest kluczem do odblokowania pełnej mocy języka DAX.

Kolumny obliczane

Każdy, kto używał Excela w trochę bardziej zaawansowanym stopniu, wie, jak dodać w nim kolumnę obliczaną. Wystarczy po prostu przejść do ostatniej kolumny tabeli, noszącej zachęcającą nazwę *Add Column* (Dodaj kolumnę) i zacząć wpisywać formułę. Naturalnie, inne narzędzia implementujące język DAX mogą używać innego interfejsu użytkownika. W każdym przypadku wyrażenie wpisujemy w pasku formuły, a mechanizm IntelliSense pomoże we wpisaniu poprawnej składni wyrażenia.

Kolumna obliczana po zdefiniowaniu zachowuje się jak każda inna kolumna tabeli i można odwoływać się do niej w obszarach wierszy, kolumn, filtrów czy wartości tabeli przestawnej czy innego raportu. Można też użyć kolumny obliczanej do zdefiniowania relacji, jeśli zajdzie taka potrzeba. Wyrażenie DAX zdefiniowane dla kolumny obliczanej zawsze operuje w kontekście bieżącego wiersza tabeli, do której należy. Dowolne odniesienie do kolumny zwraca wartość z tej kolumny w bieżącym wierszu. Nie jest możliwe bezpośrednie odwołanie się do wartości zawartych w innych wierszach.

Uwaga Jak zobaczymy później, istnieją funkcje DAX, które agregują wartości kolumny dla całej tabeli. Jedyną metodą uzyskania wartości dla pewnego podzbioru wierszy jest posłużenie się funkcjami DAX zwracającymi tabelę (filtrujących) i następnie operowanie na tej tabeli pośredniej (wirtualnej). W ten sposób można agregować wartości kolumny dla zakresu wierszy, a także odwołać się do pojedynczej wartości z innego wiersza, odfiltrowując tabelę zawierającą jedynie ten właśnie wiersz. Więcej informacji na ten temat zawiera rozdział 4, „Istota kontekstów wykonania".

Ważną cechą kolumn obliczanych, którą trzeba zapamiętać, jest to, że są one wyliczane podczas przetwarzania bazy danych i następnie przechowywane w modelu. Może się to wydawać dziwne komuś przyzwyczajonemu do kolumn obliczanych występujących w systemach SQL (nietrwałych), które są obliczane podczas wykonywania zapytań i nie zajmują pamięci. W modelu tabelarycznym jednak wszystkie kolumny obliczane są przechowywane w pamięci i są wyliczane podczas przetwarzania tabeli.

Zachowanie to jest pomocne, jeśli zechcemy utworzyć bardzo złożone kolumny obliczane. Czas wymagany na ich obliczanie jest zużywany podczas przygotowywania bazy danych (tzw. czas procesowy), a nie podczas wykonywania zapytań, co daje lepsze wrażenia dla użytkowników. Trzeba jednak pamiętać, że kolumny obliczane zajmą cenną pamięć RAM. Jeśli na przykład mamy bardzo złożoną formułę definiującą kolumnę obliczaną, może pojawić się pokusa rozdzielenia kroków tego obliczenia na kilka pośredniczących kolumn. Taka technika może być użyteczna w fazie projektowania, ułatwiając debugowanie, ale jest bardzo złym nawykiem w rozwiązaniu produkcyjnym, gdyż każda pośrednia kolumna będzie przechowywana w pamięci RAM, co oznacza zmarnowanie cennego miejsca.

Miary

Istnieje też druga metoda definiowania obliczeń w modelu DAX, przydatna w sytuacjach, gdy nie chcemy wyliczać jakiejś wartości dla każdego wiersza danych, ale chcemy zagregować wartości z wielu wierszy tabeli. Takie obliczenia nazywamy *miarami*.

Pokażemy to na przykładzie. Możemy zdefiniować w tabeli Sales kolumnę obliczaną *GrossMargin* (marża), aby obliczyć wielkość marży brutto:

```
Sales[GrossMargin] = Sales[SalesAmount] – Sales[TotalProductCost]
```

Co jednak się stanie, gdy zechcemy pokazać wielkość marży jako procentowy udział w wielkości sprzedaży? Spróbujmy utworzyć kolumnę obliczaną z poniższą formułą:

```
Sales[GrossMarginPct] = Sales[GrossMargin] / Sales[SalesAmount]
```

Formuła ta wyliczy poprawne wartości na poziomie indywidualnych wierszy, co widać na rysunku 2-2.

Unit Discount	Unit Cost	Net Price	Quantity	GrossMargin	GrossMarginPct
€ 60.00	€ 152.94	€ 239.99	2	294.1	49.02 %
€ 39.38	€ 90.55	€ 157.52	4	425.4	54.01 %
€ 119.80	€ 275.46	€ 479.20	2	647.08	54.01 %
€ 133.19	€ 220.64	€ 532.75	2	890.6	66.87 %
€ 24.20	€ 55.64	€ 96.80	3	196.08	54.02 %
€ 65.80	€ 167.73	€ 263.20	4	645.08	49.02 %
€ 4.00	€ 10.19	€ 15.99	4	39.2	49.02 %
€ 20.00	€ 50.98	€ 79.99	4	196.04	49.01 %
€ 25.98	€ 66.23	€ 103.92	4	254.68	49.01 %

Rysunek 2-2 Kolumna *GrossMarginPct* pokazuje udział procentowy wartości *GrossMargin*, wyliczony wiersz po wierszu.

Jeśli jednak zechcemy obliczyć całkowitą marżę procentową, oczywiste się stanie, że nie możemy po prostu zagregować wartości procentowych! Po zastanowieniu się zauważymy, że przy obliczaniu zagregowanej wartości procentowej nie możemy polegać na kolumnach obliczanych. W rzeczywistości potrzebujemy wyliczyć wartość zagregowaną jako sumę marżbrutto podzieloną przez sumę sprzedaży. Musimy zatem obliczyć proporcję agregacji. Mówiąc inaczej, trzeba obliczyć iloraz sum, a nie sumę ilorazów.

Właściwą implementacją wielkości *GrossMarginPct* jest miara:

```
Sales[GrossMarginPct] := SUM ( Sales[GrossMargin] ) / SUM
(Sales[SalesAmount] )
```

Jednak, jak już powiedzieliśmy, nie możemy umieścić tej formuły w kolumnie obliczanej. Jeśli potrzebujemy użyć wartości zagregowanych, a nie z indywidualnych wierszy, konieczne jest użycie miar. Czytelnik mógł zauważyć, że w formule powyżej użyliśmy zapisu `:=` zamiast samego znaku równości (`=`). Jest to standard, który przyjęliśmy w całej książce: definicje kolumn obliczanych zapisywane są przy użyciu samego znaku równości, podczas gdy definicje miar używają zapisu z dwukropkiem.

Miary i kolumny obliczane wykorzystują bardzo podobne wyrażenia DAX; różnica leży w kontekście wykonania. Miara jest obliczana w kontekście wyznaczanym przez komórkę tabeli przestawnej lub zapytanie DAX, podczas gdy kolumna obliczana jest przetwarzana na poziomie pojedynczych wierszy tabeli, do której należy. Kontekst wyznaczany przez komórkę (w dalszej części książki dowiemy się, że jest *kontekst filtru* i dowiemy się więcej na ten temat) zależy od tego, co użytkownik zaznaczył w tabeli przestawnej albo od kształtu zapytania DAX. Zatem, gdy używamy formuły `SUM(Sales[SalesAmount])` w mierze, rozumiemy przez to sumę wszystkich wartości, które są agregowane dla tej komórki tabeli przestawnej, podczas gdy użycie `Sales[SalesAmount]` w kolumnie obliczanej oznacza użycie wartości przechowywanej w kolumnie *SalesAmount* w bieżącym wierszu.