



Quick answers to common problems

# MDX with Microsoft SQL Server 2008 R2 Analysis Services: Cookbook

80 recipes for enriching your Business Intelligence solutions with high-performance MDX calculations and flexible MDX queries

**Tomislav Piasevoli**

**[PACKT]** enterprise  
PUBLISHING professional expertise distilled 

# MDX with Microsoft SQL Server 2008 R2 Analysis Services: Cookbook

80 recipes for enriching your Business Intelligence  
solutions with high-performance MDX calculations and  
flexible MDX queries

**Tomislav Piasevoli**



BIRMINGHAM - MUMBAI

# MDX with Microsoft SQL Server 2008 R2 Analysis Services: Cookbook

Copyright © 2011 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2011

Production Reference: 2010811

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-849681-30-8

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Artie Ng ([artherng@yahoo.com.au](mailto:artherng@yahoo.com.au))

# Credits

**Author**

Tomislav Piasevoli

**Project Coordinator**

Zainab Bagasrawala

**Reviewers**

Greg Galloway

Darren Gosbell

Deepak Puri

Marco Russo

Chris Webb

**Proofreader**

Josh Toth

**Indexer**

Rekha Nair

**Production Coordinator**

Arvindkumar Gupta

**Acquisition Editor**

Kerry George

**Cover Work**

Arvindkumar Gupta

**Development Editor**

Chris Rodrigues

**Technical Editors**

Ajay Chamkeri

Merwine Machado

# About the Author

**Tomislav Piasevoli** ([tomislav@piasevoli.com](mailto:tomislav@piasevoli.com)) is a Business Intelligence Specialist with years of experience in Microsoft SQL Server Analysis Services (SSAS). He lives in Croatia and works for SoftPro Tetral d.o.o., a company specializing in development of SSAS frontends and implementation of BI solutions.

His main interests are dimensional modeling, cube design, and MDX about which he blogs at <http://tomislav.piasevoli.com>. Tomislav also writes articles and speaks at regional conferences and user groups. For his contribution to the community, Microsoft awarded him with the Microsoft SQL Server MVP title.

# Acknowledgement

I wish to thank everyone who contributed to this book, in one way or another.

First and foremost, a big thank you goes to my wife Kristina and our three children—one of them was born while I was beginning to write this book—for having enormous patience throughout this period.

Secondly, I'd like to thank Chris Webb for vouching for me to the publisher and providing valuable information as one of the reviewers of this book. Greg Galloway, Marco Russo, Darren Gosbell, and Deepak Puri were the other precious reviewers of this book who corrected my mistakes and provided additional information relevant to the chapter topics.

Next, I'd like to thank all the people at Packt Publishing who worked on this book, to help make it better by assisting me during the book-writing process: Kerry George, Zainab Bagasrawala, Chris Rodrigues, and Merwine Machado.

There were also people who knowingly and unknowingly helped me with their ideas, articles, and helpful tips. My younger brother Hrvoje Piasevoli and Willfried Färber are representatives of the first group. The other group consists of bloggers and subject-matter experts whose articles I found useful for many recipes of this book. Here they are: Mosha Pasumansky, Teo Lachev, Jeffrey Wang, Jeremy Kashel, Vidas Matelis, Thomas Kejser (and other bloggers at SQLCAT site), Jeff Moden, Michael Coles, Itzik Ben-Gan, Irina Gorbach, Vincent Rainardi and in particular my reviewers again. Additionally, I acknowledge countless contributors to MSDN SQL Server Analysis Services forum—whom I had the luck to meet—for the solutions they shared with the rest of us there.

Last but not least, a thank you goes to my current and former colleagues at SoftPro Tetral company who played a part by exchanging ideas with me during the time spent together all these years.

# About the Reviewers

**Greg Galloway** is a recipient of Microsoft's Most Valuable Professional, mainly for his work with Analysis Services. He has been a BI architect with Artis Consulting in Dallas, Texas, since 2003. He is a coordinator on several Codeplex projects including the well-known BIDS Helper, the Analysis Services Stored Procedure project, and OLAP PivotTable Extensions. Greg blogs at <http://www.artisconsulting.com/blogs/GregGalloway>.

**Darren Gosbell** is a Solution Architect with James & Monroe, a consulting firm whose specialties include the areas of Business Intelligence, Data Warehousing, and Business Performance Management (BPM). He has over ten years of practical experience in building and implementing data warehousing and business intelligence solutions on the Microsoft platform. Darren is a Microsoft MVP in SQL Server, having been a recipient of this award since 2006 in recognition of his contribution to the community specifically in the areas of Analysis Services and MDX. He lives in Melbourne, Australia with his wife and two children.

**Deepak Puri** is a Business Intelligence Consultant who has been working with SQL Server Analysis Services since 2000. He is an active contributor to the MSDN Analysis Services forum and a regular presenter at Ohio North SQL Server User Group meetings. Deepak has also presented at the recent Cleveland and Honolulu SQL Saturday events.

**Marco Russo** ([marco.russo@sqlbi.com](mailto:marco.russo@sqlbi.com)) is a consultant and trainer involved in several Business Intelligence projects, making data warehouse relational and multidimensional designs, with particular experience in sectors such as banking and financial services, manufacturing, and commercial distribution. He speaks at several international conferences and actively contributes to the SQL community. Marco is the founder of SQLBI (<http://www.sqlbi.com>) and his blog is available at [http://sqlblog.com/blogs/marco\\_russo](http://sqlblog.com/blogs/marco_russo).

Marco has written several books: *Expert Cube Development with Microsoft SQL Server 2008 Analysis Services*, *Microsoft PowerPivot 2010 for Excel: Give Your Data Meaning*, and *The many-to-many revolution*, a mini-book about many-to-many dimension relationships in Analysis Services. He has also co-authored the *SQLBI Methodology* with Alberto Ferrari and has written several books about .NET and three books about LINQ published by Microsoft Press.

**Chris Webb** ([chris@crossjoin.co.uk](mailto:chris@crossjoin.co.uk)) is an independent consultant specializing in Microsoft SQL Server Analysis Services and the MDX query language. He is the co-author of the book *Expert Cube Development with SQL Server Analysis Services 2008* and blogs about Microsoft BI topics at <http://cwebbbi.wordpress.com/>.

# www.PacktPub.com

## **Support files, eBooks, discount offers and more**

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

### Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

### Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

## **Instant Updates on New Packt Books**

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter, or the *Packt Enterprise* Facebook page.





*I dedicate this book to my children Petra, Matko, and Nina, and to my wife Kristina.  
Without their devoted support, I could not have completed this project.*



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Elementary MDX Techniques</b>	<b>7</b>
Introduction	8
Skipping axis	8
Handling division by zero errors	10
Setting special format for negative, zero and null values	13
Applying conditional formatting on calculations	17
Setting default member of a hierarchy in MDX script	20
Implementing NOT IN set logic	23
Implementing logical OR on members from different hierarchies	26
Iterating on a set in order to reduce it	30
Iterating on a set in order to create a new one	34
Iterating on a set using recursion	37
Dissecting and debugging MDX queries	41
Using NON_EMPTY_BEHAVIOR	45
Optimizing MDX queries using the NonEmpty() function	48
Implementing logical AND on members from the same hierarchy	51
<b>Chapter 2: Working with Time</b>	<b>59</b>
Introduction	59
Calculating the YTD (Year-To-Date) value	60
Calculating the YoY (Year-over-Year) growth (parallel periods)	66
Calculating moving averages	71
Finding the last date with data	74
Getting values on the last date with data	79
Hiding calculation values on future dates	83
Calculating today's date using the string functions	87
Calculating today's date using the MemberValue function	94
Calculating today's date using an attribute hierarchy	97

*Table of Contents* \_\_\_\_\_

<b>Calculating the difference between two dates</b>	<b>100</b>
<b>Calculating the difference between two times</b>	<b>103</b>
<b>Calculating parallel periods for multiple dates in a set</b>	<b>106</b>
<b>Calculating parallel periods for multiple dates in a slicer</b>	<b>110</b>
<b>Chapter 3: Concise Reporting</b>	<b>115</b>
<b>Introduction</b>	<b>115</b>
<b>Isolating the best N members in a set</b>	<b>116</b>
<b>Isolating the worst N members in a set</b>	<b>122</b>
<b>Identifying the best/worst members for each member of another hierarchy</b>	<b>125</b>
<b>Displaying few important members, others as a single row, and the total at the end</b>	<b>130</b>
<b>Combining two hierarchies into one</b>	<b>134</b>
<b>Finding the name of a child with the best/worst value</b>	<b>138</b>
<b>Highlighting siblings with the best/worst values</b>	<b>143</b>
<b>Implementing bubble-up exceptions</b>	<b>149</b>
<b>Chapter 4: Navigation</b>	<b>155</b>
<b>Introduction</b>	<b>155</b>
<b>Detecting a particular member in a hierarchy</b>	<b>156</b>
<b>Detecting the root member</b>	<b>160</b>
<b>Detecting members on the same branch</b>	<b>164</b>
<b>Finding related members in the same dimension</b>	<b>172</b>
<b>Finding related members in another dimension</b>	<b>178</b>
<b>Calculating various percentages</b>	<b>183</b>
<b>Calculating various averages</b>	<b>189</b>
<b>Calculating various ranks</b>	<b>194</b>
<b>Chapter 5: Business Analytics</b>	<b>205</b>
<b>Introduction</b>	<b>205</b>
<b>Forecasting using the linear regression</b>	<b>206</b>
<b>Forecasting using the periodic cycles</b>	<b>212</b>
<b>Allocating the non-allocated company expenses to departments</b>	<b>219</b>
<b>Calculating the number of days from the last sales to identify the slow-moving goods</b>	<b>227</b>
<b>Analyzing fluctuation of customers</b>	<b>233</b>
<b>Implementing the ABC analysis</b>	<b>241</b>
<b>Chapter 6: When MDX is Not Enough</b>	<b>247</b>
<b>Introduction</b>	<b>247</b>
<b>Using a new attribute to separate members on a level</b>	<b>249</b>
<b>Using a distinct count measure to implement histograms over existing hierarchies</b>	<b>254</b>

---

*Table of Contents*

<b>Using a dummy dimension to implement histograms over non-existing hierarchies</b>	<b>257</b>
<b>Creating a physical measure as a placeholder for MDX assignments</b>	<b>266</b>
<b>Using a new dimension to calculate the most frequent price</b>	<b>271</b>
<b>Using a utility dimension to implement flexible display units</b>	<b>275</b>
<b>Using a utility dimension to implement time-based calculations</b>	<b>281</b>
<b>Chapter 7: Context-aware Calculations</b>	<b>295</b>
<b>Introduction</b>	<b>295</b>
<b>Identifying the number of columns and rows a query will return</b>	<b>297</b>
<b>Identifying the axis with measures</b>	<b>301</b>
<b>Identifying the axis without measures</b>	<b>304</b>
<b>Adjusting the number of columns and rows for OWC and Excel</b>	<b>307</b>
<b>Identifying the content of axes</b>	<b>310</b>
<b>Calculating row numbers</b>	<b>316</b>
<b>Calculating the bit-string for hierarchies on an axis</b>	<b>319</b>
<b>Preserving empty rows</b>	<b>324</b>
<b>Implementing utility dimension with context-aware calculations</b>	<b>327</b>
<b>Chapter 8: Advanced MDX Topics</b>	<b>337</b>
<b>Introduction</b>	<b>337</b>
<b>Displaying members without children (leaves)</b>	<b>338</b>
<b>Displaying members with data in parent-child hierarchies</b>	<b>343</b>
<b>Implementing the Tally table utility dimension</b>	<b>347</b>
<b>Displaying random values</b>	<b>352</b>
<b>Displaying a random sample of hierarchy members</b>	<b>356</b>
<b>Displaying a sample from a random hierarchy</b>	<b>359</b>
<b>Performing complex sorts</b>	<b>366</b>
<b>Using recursion to calculate cumulative values</b>	<b>373</b>
<b>Chapter 9: On the Edge</b>	<b>381</b>
<b>Introduction</b>	<b>381</b>
<b>Clearing the Analysis Services cache</b>	<b>382</b>
<b>Using Analysis Services stored procedures</b>	<b>387</b>
<b>Executing MDX queries in T-SQL environments</b>	<b>394</b>
<b>Using SSAS Dynamic Management Views (DMV) to fast-document a cube</b>	<b>400</b>
<b>Using SSAS Dynamic Management Views (DMVs) to monitor activity and usage</b>	<b>406</b>
<b>Capturing MDX queries generated by SSAS front-ends</b>	<b>411</b>
<b>Performing custom drillthrough</b>	<b>416</b>
<b>Conclusion</b>	<b>423</b>

*Table of Contents* —————

<b>Appendix: Glossary of Terms</b>	<b>427</b>
Parts of an MDX query	427
MDX query in action	433
Cube and dimension design	437
MDX script	440
Query optimization	442
Types of query	445
<b>Index</b>	<b>449</b>

# Preface

MDX-related books often dedicate a significant part of their content to explaining the concepts of multidimensional cubes, the MDX language and its functions, and other specifics related to working with Analysis Services. And that's perfectly fine, there should be books like that, the tutorials that teach the concepts. However, that also means that when it comes to examples, there's usually not enough space to provide all the details about them and their variations, otherwise the book would become huge or oftentimes lose its focus. The result of that is that making a step further from the provided calculations and queries might not be an easy task for an average reader.

The other problem with tutorials is that the solution to a particular problem might be scattered throughout the book, which is where the cookbook style of books like this one come into play. Similar to data warehouses where we consolidate many tables of the relational database into a few and then organize those dimension tables in a topic-based star schema, in cookbooks we aggregate the information about a particular problem in form of one or more recipes and present that topic-based knowledge in full detail.

Both the relational databases and the data warehouses have their purpose; it's not uncommon to see them together in one place. The same is true about books and their approaches. What we also know is that there are far too few data warehouses than relational databases. Again, the same is with MDX-related cookbooks in respect to MDX tutorials, particularly those dealing with advanced topics.

As a writer, I hope you recognize my intention and the value this book could bring you. As a reader, we rarely have enough time to start reading a book, not to mention finish it. This is another advantage of the cookbook format. You can browse through the contents and look for the solution to a particular problem. As the recipes are relatively short and organized in chapters, that task won't take much of your time. All the information will be in one place. In addition to that, you'll see which recipes are related to that one, so that you can learn even more.

---

Preface

The next time you encounter a problem, you might start on a completely different part of the book. Little by little, you'll read it all and hopefully become confident not only in using these recipes, but also those not present in this book. In other words, the moment you start creating your own solutions by combining principles laid out in this book, my goal of teaching through examples is accomplished. Feel free to drop me a note about it.

## What this book covers

This book is a cookbook style of book, a book full of solutions with tips and techniques for writing advanced MDX calculations and queries. The recipes are organized in chapters, each covering one specific area to help you navigate the contents, find a recipe to the problem that you currently have and focus entirely on a particular topic by reading related or even all recipes in that chapter.

Here's a brief overview of the book's contents.

*Chapter 1, Elementary MDX Techniques*, illustrates several ways of how to perform common MDX-related tasks such as iterating on sets or applying Boolean logic on them, how to check for errors, and how to format, optimize, and debug your MDX calculations.

*Chapter 2, Working with Time*, covers the often-required time calculations such as the year-to-date, year-over-year, and the moving averages calculations, shows how to get the last date with data or calculate the difference between two events, and also presents several solutions to the problem of using today's date in your calculations.

*Chapter 3, Concise Reporting*, is all about how to do more with less - how to make your queries return only what matters by identifying important members and then either isolating them from the rest of the set, combining them with other members or other hierarchies, highlighting them, extracting information about them, or creating custom groups.

*Chapter 4, Navigation*, explores the benefits of having multilevel hierarchies and correct attribute relationships with recipes that guide you how to identify the current context, find related members, and implement typical calculations like relative percentages, averages, or ranks.

*Chapter 5, Business Analytics*, presents some of the typical business requests and a way to solve them in your cubes using MDX calculations and queries.

*Chapter 6, When MDX is Not Enough*, suggests rethinking where to solve the problem, using MDX or by modifying the cube by implementing relatively simple improvements such as adding a new attribute, new measure, new utility dimension, or using the new granularity.

*Chapter 7, Context-aware Calculations*, deals with calculations that can work on any cube, dimension, hierarchy, level, or measure and ends with a recipe how to implement them using a utility dimension.

*Chapter 8, Advanced MDX Topics*, is a collection of recipes that cover parent-child hierarchies, random values, and complex sorts and iterations.

*Chapter 9, On the Edge*, looks at expanding your horizons with recipes covering data management views (DMVs), stored procedures, drillthrough, and other exotic areas.

*Conclusion*, is a wrap-up chapter with suggestions for additional contents on the web you might find interesting after reading this book.

*Appendix*, contains the glossary of terms covered in this book or relevant to MDX in general.

## What you need for this book

To follow the examples in this book you should have the following installed on your computer:

- ▶ Microsoft Windows XP or later for the workstation or Microsoft Windows Server 2003 or later for the server.
- ▶ Microsoft SQL Server 2008 R2 (full installation or at least the database engine, Analysis Services and the client components).
- ▶ Adventure Works sample database, built specifically for the Microsoft SQL Server 2008 R2.

Having an older version of Microsoft SQL Server (2005 or 2008) is also fine, most of the recipes will work. However, some of them will need adjustments because the Date dimension in respective older versions of the Adventure Works database has different set of years. To solve that problem simply shift the date-specific parts of the queries few years back in time, for example, turn the year 2006 into the year 2002 and Q3 of the year 2007 to Q3 of 2003.

Also, use the Developer, Enterprise, or the Trial Edition of Microsoft SQL Server together with the enterprise version of the Adventure Works cube project. Using the Standard Edition is also fine, but a few recipes might not work. If they are important to you, consider installing the above mentioned Trial Edition because it's free for 180 days.

Here's the link for the Trial Edition of Microsoft SQL Server 2008 R2:

<http://tinyurl.com/SQLServer2008RTrial>

Here's the link for the Adventure Works sample database built specifically for Microsoft SQL Server 2008 R2:

<http://tinyurl.com/AdventureWorks2008R2>

## Who this book is for

If you are a SQL Server Analysis Services developer who wants to take your high-performance cubes further using MDX, then this book is for you. The book assumes you have a working knowledge of MDX and a basic understanding of dimensional modeling and cube design.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
CreateMemberCurrentCube.[Measures].[Gross Profit formatted]
As [Measures].[Sales Amount] -
    [Measures].[Total Product Cost],
Format_String = "#,##0;- #,##0;0;n/a",
Associated_Measure_Group = 'Sales Summary';
```

When we wish to draw your attention to a particular part of a code block, the relevant lines, or items are set in bold:

```
MEMBER [Measures].[Cumulative Sum] AS
    Sum( null : [Product].[Subcategory].CurrentMember,
        [Measures].[Gross Profit] )
MEMBER [Measures].[Cumulative Product] AS
    Exp( Sum( null : [Product].[Subcategory].CurrentMember,
        Log( [Measures].[Gross Profit Margin] )
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on [www.packtpub.com](http://www.packtpub.com) or e-mail [suggest@packtpub.com](mailto:suggest@packtpub.com).

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

### Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

### Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1

## Elementary MDX Techniques

In this chapter, we will cover:

- ▶ Skipping axis
- ▶ Handling division by zero errors
- ▶ Setting special format for negative, zero, and null values
- ▶ Applying conditional formatting on calculations
- ▶ Setting default member of a hierarchy in MDX script
- ▶ Implementing NOT IN set logic
- ▶ Implementing logical OR on members from different hierarchies
- ▶ Iterating on a set in order to reduce it
- ▶ Iterating on a set in order to create a new one
- ▶ Iterating on a set using recursion
- ▶ Dissecting and debugging MDX queries
- ▶ Using NON\_EMPTY\_BEHAVIOR
- ▶ Optimizing MDX queries using the NonEmpty() function
- ▶ Implementing logical AND on members from the same hierarchy

## Introduction

This chapter presents common MDX tasks and one or more ways to solve them or deal with them appropriately. We'll cover basic principles and approaches such as how to skip an axis and prevent common errors, how to set the default member of a hierarchy, and how to format cell foreground and background colors based on the value in cells.

Then we will tackle the logical operations **NOT** and **OR** while leaving the most complex **AND** logic for the end of the chapter.

The second half of the chapter concentrates on iterations and ways to perform them, followed by optimization of calculations using two of the most common approaches, **NonEmpty()** function and **NON\_EMPTY\_BEHAVIOR** property. Finally, we will cover how to dissect and debug MDX queries and calculations.

Be sure to read the recipes thoroughly.

## Skipping axis

There are situations when we want to display just a list of members and no data associated with them. Naturally, we expect to get that list on rows, so that we can scroll through them nicely. However, the rules of MDX say we can't skip axes. If we want something on rows (which is **AXIS(1)** by the way), we must use all previous axes as well (columns in this case, which is also known as **AXIS(0)**).

The reason why we want the list to appear on axis 1 and not axis 0 is because a horizontal list is not as easy to read as a vertical one.

Is there a way to display those members on rows and have nothing on columns? Sure! This recipe shows how.

### Getting ready

Follow these steps to set up the environment for this recipe:

1. Start **SQL Server Management Studio (SSMS)** or any other application you use for writing and executing MDX queries and connect to your **SQL Server Analysis Services (SSAS)** 2008 R2 instance (`localhost` or `servername\instancename`).
2. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

## How to do it...

Follow these steps to get a one-dimensional query result with members on rows:

1. Put an empty set on columns (**AXIS(0)**). Notation for empty set is this: {}.
2. Put some hierarchy on rows (**AXIS(1)**). In this case we used the largest hierarchy available in this cube – Customer hierarchy of the same dimension.
3. Run the following query:

```
SELECT
    {} ON 0,
    {[Customer].[Customer].[Customer].MEMBERS} ON 1
FROM
    [Adventure Works]
```

## How it works...

Although we can't skip axes, we are allowed to provide an empty set on them. This trick allows us to get what we need – nothing on columns and a set of members on rows.

## There's more...

Notice that this type of query is very convenient for parameter selection of another query as well as for search. See how it can be modified to include only those customers whose name contains the phrase "John":

```
SELECT
    {} ON 0,
    { Filter(
        [Customer].[Customer].[Customer].MEMBERS,
        InStr(
            [Customer].[Customer].CurrentMember.Name,
            'John'
            ) > 0
        )
    } ON 1
FROM
    [Adventure Works]
```

In the final result, you will notice the "John" phrase in various positions in member names:

Jasmine S. Johnson
Jennifer F. Johnson
Jeremiah A. Johnson
Jerome Johnsen
Jeny Johnsen
Jessica Johnson
Joanna L. Johnston
John A. Johnson
John A. Martin

### The idea behind

If you put a cube measure or a calculated measure with a non-constant expression on axis 0 instead, you'll slow down the query. Sometimes it won't be so obvious, sometimes it will. It will depend on the measure's definition and the number of members in the hierarchy being displayed. For example, if you put the Sales Amount measure on columns, that measure will have to be evaluated for each member in the rows. Do we need those values? No, we don't. The only thing we need is a list of members; hence we've used an empty set. That way, the SSAS engine doesn't have to go into cube space. It can reside in dimension space which is much smaller and the query is therefore more efficient.

### Possible workarounds

In case of a third-party application or a control which has problems with this kind of MDX statement (i.e. expects something on columns and is not working with an empty set), we can define a constant measure (a measure returning null, 0, 1 or any other constant) and place it on columns instead of that empty set. For example, we can define a calculated measure in the MDX script whose definition is 1, or any other constant value, and use that measure on the columns axis. It might not be as efficient as an empty set, but it is a much better solution than the one with a regular (non-constant) cube measure like the Sales Amount measure.

## Handling division by zero errors

Another common task is handling errors, especially division by zero type of errors. This recipe offers a way to solve that problem.

 Not all versions of Adventure Works database have the same date range. If you're not using the recommended version of it, the one for the SSAS 2008 R2, you might have problems with queries in this book. Older versions of Adventure Works database have dates up to the year 2006 or even 2004. If that's the case, make sure you adjust examples by offsetting years in the query with a fixed number. For example, the year 2006 should become 2002 and so on.

## Getting ready

Start a new query in SQL Server Management Studio and check that you're working on Adventure Works database. Then write and execute this query:

```
WITH
MEMBER [Date].[Calendar Year].[CY 2006 vs 2005 Bad] AS
    [Date].[Calendar Year].[Calendar Year].&[2006] /
    [Date].[Calendar Year].[Calendar Year].&[2005],
FORMAT_STRING = 'Percent'
SELECT
    { [Date].[Calendar Year].[Calendar Year].&[2005],
        [Date].[Calendar Year].[Calendar Year].&[2006],
        [Date].[Calendar Year].[CY 2006 vs 2005 Bad] } *
    [Measures].[Reseller Sales Amount] ON 0,
    { [Sales Territory].[Sales Territory].[Country].MEMBERS }
ON 1
FROM
    [Adventure Works]
```

This query returns 6 rows with countries and 3 rows with years, the third row being the ratio of the previous two, as its definition says.

The problem is that we get **1.#INF** on some cells. To be precise, that value (the formatted value of infinity), appears on rows where the **CY 2005** is null. Here's a solution for that.

## How to do it...

Follow these steps to handle division by zero errors:

1. Copy the calculated member and paste it as another calculated member. During that, replace the term **Bad** with **Good** in its name, just to differentiate those two members.
2. Copy the denominator.
3. Wrap the expression in an outer **IIF( )** statement.

4. Paste the denominator in the condition part of the **IIF()** statement and compare it against 0.
5. Provide null value for the True part.
6. Your initial expression should be in the False part.
7. Don't forget to include the new member on columns and execute the query:

```
MEMBER [Date].[Calendar Year].[CY 2006 vs 2005 Good] AS
    IIF ([Date].[Calendar Year].[Calendar Year].&[2005] = 0,
        null,
        [Date].[Calendar Year].[Calendar Year].&[2006] /
        [Date].[Calendar Year].[Calendar Year].&[2005]
    ),
    FORMAT_STRING = 'Percent'
```

8. The result shows that the new calculated measure corrects the problem – we don't get errors (the rightmost column, compared to the one on its left):

	CY 2005	CY 2006	CY 2006 vs 2005 Bad	CY 2006 vs 2005 Good
	Reseller Sales Amount	Reseller Sales Amount	Reseller Sales Amount	Reseller Sales Amount
France	(null)	\$857,123.18	1.#INF	(null)
Germany	(null)	(null)	(null)	(null)
United Kingdom	(null)	\$841,757.76	1.#INF	(null)
Canada	\$1,513,359.46	\$4,822,999.20	318.69%	318.69%
United States	\$6,552,075.85	\$17,622,549.51	268.96%	268.96%
Australia	(null)	(null)	(null)	(null)

### How it works...

A division by zero error occurs when the denominator is null or zero and the numerator is not null. In order to prevent this error, we must test the denominator before the division and handle the case when it is null or zero. That is done using an outer **IIF()** statement.

It is enough to test just for zero because `null = 0` returns True.

### There's more...

SQLCAT's SQL Server 2008 Analysis Services Performance Guide has lots of interesting details regarding the **IIF()** function:

<http://tinyurl.com/PerfGuide2008>

Additionally, you may find Jeffrey Wang's blog article useful in explaining the details of the **IIF()** function:

<http://tinyurl.com/IIFJeffrey>

## Earlier versions of SSAS

If you're using a version of SSAS prior to 2008 (that is, 2005), the performance will not be as good. See Moshav Pasumansky's article for more info:  
<http://tinyurl.com/IIFMosha>

# Setting special format for negative, zero and null values

The **FORMAT\_STRING** property is used for specifying how a regular or calculated member (usually a measure) should be presented and what its **FORMATTED\_VALUE** should look like. There are some predefined formats such as *Standard*, *Currency* or *Percent*, but we can also specify our own using the right combination of characters.

Most people are unaware of the fact that there are actually 4 sections of the format string for numeric values and 2 sections for strings. In this recipe we'll see how to set the format for negative, zero, or null values.

## Getting ready

Follow these steps to set up the environment for this recipe:

1. Open Business Intelligence Development Studio (BIDS) and then open the **Adventure Works DW 2008** solution.
2. Double-click the **Adventure Works** cube and go to the **Calculations** tab.
3. Choose **Script View**.
4. Position the cursor at the end of the script and create a new calculated measure as follows:

```
Create Member CurrentCube.[Measures].[Gross Profit formatted]
As [Measures].[Sales Amount] -
[Measures].[Total Product Cost],
Associated_Measure_Group = 'Sales Summary';
```

That's actually the same definition as the definition of the *Gross Profit* measure. We have created a clone of that measure using a different name so that we can compare them later and see the formatting effect in action.

## How to do it...

Follow these steps to set a custom format:

1. Add the **Format\_String** property for that measure.
2. Be sure to specify it as below, using 4 sections separated by a semi-colon (;):

```
Create Member CurrentCube.[Measures].[Gross Profit formatted]
As [Measures].[Sales Amount] -
[Measures].[Total Product Cost],
```

```
Format_String = "#,##0;- #,##0;0;n/a",
```

```
Associated_Measure_Group = 'Sales Summary';
```
3. Save and deploy (or just press the **Deploy MDX Script** icon if you're using **BIDS Helper**).
4. Go to the **Cube Browser** tab. Reconnect and click on the **Show Empty Cells** button.
5. Put **Promotion Category** level on rows. Put **Gross Profit** and the new **Gross Profit formatted** measures on columns. Both measures have the same values, but they are formatted differently, as shown in the following image:

Drop Filter Fields Here		Drop Column Fields Here	
Promotion Category ▾	Gross Profit	Gross Profit formatted	
Customer		n/a	
No Discount	\$13,501,872.02	13,501,872	
Reseller	(\$950,505.77)	- 950,506	
Grand Total	\$12,551,366.25	12,551,366	

6. Notice that the new measure shows **n/a** for null values and uses minus sign for negative ones. In addition to that, it removes decimal part of numbers.

## How it works...

The four sections of the format string for numeric values are as follows: the first is for positive numbers, the second is for negative numbers, the third is for the zero value, and the fourth is for null values.

Usually we provide a format string for one section only. However, when required, we can use additional sections and provide formatting there as well. In the example, we eliminated decimals from the value, emphasized negative values with a space before a minus sign, and provided **n/a** for null values.

## There's more...

Another way of specifying format string is by using the `Format_String()` function inside the MDX script:

```
Format_String( [Measures].[Gross Profit formatted] ) =
    "#,##0;;;-";
```

By the way, the **Adventure Works DW 2008R2** database has a similar example at the end of the MDX script.

We can omit the format of one section by closing it with a semicolon. In the expression above, the format has been specified for positive and null values only. The negative and zero values will retain their default format.

In fact, the format string can be set for several measures that way.

```
Format_String( { [Measures].[Gross Profit],
                 [Measures].[Gross Profit formatted] } ) =
    "#,##0;;;-";
```

Finally, the set of members doesn't have to be explicit. It can be obtained using MDX calculations. That might be a nice way to handle and maintain formats for all measures in a cube if there's an order with prefixes/suffixes/special characters inside their names. For example, this code shows how to search for a specific pattern in measures names and apply an appropriate format in case of a hit.

```
Format_String( Filter( Measures.ALLMEMBERS,
                      InStr( Measures.CurrentMember.Name,
                             'Ratio' ) > 0 )
    ) = 'Percent';

Format_String( Filter( Measures.ALLMEMBERS,
                      InStr( Measures.CurrentMember.Name,
                             'Amount' ) > 0 )
    ) = '#,##0.00';
```

## Tips and tricks

MDX expressions can be used inside the `FORMAT_STRING` property. The calculation should return a string value which will then be used as a format string. Note that only the values of cells are cached by Analysis Services. When complex expressions are used for the `FORMAT_STRING` property, it may improve performance if you wrap that expression in a calculated measure. You can then use the calculated measure in the assignment to the `FORMAT_STRING` property.

### A friendly warning

Remember never to put zero values inside your cube instead of empty values or your measure will become dense, always returning a non-null value and the performance will suffer because many MDX functions will not be able to operate in bulk mode by taking advantage of the sparsity of the measure.

If it is required that null values are shown as zeros, then leave them as nulls and use the format string instead. Setting the **NullHandling** property to **Preserve** on the measure's source column preserves the null value instead of converting it to zero.

### Troubleshooting formatted values

By default, a query with no CELL PROPERTIES returns VALUE and FORMATTED\_VALUE, which means the data is returned in two shapes – raw and formatted. And such are almost all queries. However, if we explicitly omit the FORMATTED\_VALUE property from cell properties, we won't have formatted values. The success of this recipe depends therefore on the SSAS front end you're using and its way of displaying values.

In case of a problem, be sure to check whether there's an option in the program's connection or its display properties which can turn this on or help in another way. Sometimes you'll be lucky, other times you won't. For example, only the first three sections work in Excel 2007 and Excel 2010. The fourth section is simply ignored.

You can find more information about it on MSDN, on the *Using Cell Properties (MDX)* page:  
<http://tinyurl.com/CellProperties>

### Formatting options in detail

There are various options for using format strings but we cannot cover them all in this book. If you require additional information regarding this topic, follow these links:

- ▶ FORMAT\_STRING Contents (MDX):  
<http://tinyurl.com/Format-String1>
- ▶ LANGUAGE and FORMAT\_STRING on FORMATTED\_VALUE  
<http://tinyurl.com/Format-String2>

### See also

The next recipe, *Applying conditional formatting on calculations*, shows how to additionally emphasize data by using properties such as BACK\_COLOR and FORE\_COLOR.

## Applying conditional formatting on calculations

Sometimes we want to emphasize cells based on their values. That way data analysis becomes easier because we combine best of charts and tables – we still see the values while we have additional color information about the value in each cell.

A typical case is highlighting the background of cells with green or red color for good or bad results, respectively. It's a pretty trivial operation in Excel and it's not that hard in MDX either. This recipe demonstrates how to do this.

### Getting ready

Open BIDS and verify you have the Gross Profit formatted measure defined in the previous recipe, *Setting special format for negative, zero and null values*. If not, define the measure like this:

```
Create Member CurrentCube.[Measures].[Gross Profit formatted]
As [Measures].[Sales Amount] -
    [Measures].[Total Product Cost],
Format_String = "#,##0;;;-",
Associated_Measure_Group = 'Sales Summary';
```

### How to do it...

Follow these steps to apply conditional formatting:

1. Locate the measure's definition and position of the cursor at the beginning of any line between the definition and the end of the measure.
2. Add this part inside.

```
Fore_Color = RGB(255, 255, 255),
Back_Color = case
    when [Measures].CurrentMember > 0
        then RGB(0, 128, 0)
    when [Measures].CurrentMember < 0
        then RGB(128, 0, 0)
    else    RGB(100, 100, 100)
end,
```

3. Save and deploy (or just press **Deploy MDX Script** icon if you're using **BIDS Helper**).
4. Go to **Cube Browser** tab. Reconnect and click on the **Show Empty Cells** button.

5. Put the **Promotion** attribute hierarchy on rows. Put **Gross Profit** and the new **Gross Profit formatted** measures on columns. Notice the effect of the format string in the following image. Positive values are colored green, negative ones have become maroon and everything else is grey.

	Drop Column Fields Here	
Promotion	Gross Profit	Gross Profit formatted
No Discount	\$13,501,872.02	13,501,872
Volume Discount 11 to 14	\$794,013.51	794,014
Volume Discount 15 to 24	(\$109,260.96)	-109,261
Volume Discount 25 to 40	(\$21,098.59)	-21,099
Volume Discount 41 to 60	(\$25.76)	-26
Volume Discount over 60		-
Mountain-100 Clearance Sale	(\$617,513.77)	-617,514
Sport Helmet Discount-2002	\$620.75	621
Road-650 Overstock	(\$97,972.72)	-97,973
Mountain Tire Sale		-
Sport Helmet Discount-2003	\$202.22	202
LL Road Frame Sale		-
Touring-3000 Promotion	(\$271,453.02)	-271,453
Touring-1000 Promotion	(\$536,177.33)	-536,177
Half-Price Pedal Sale		-
Mountain-500 Silver Clearance Sale	(\$91,840.10)	-91,840
Grand Total	\$12,551,366.25	12,551,366

### How it works...

The `FORE_COLOR` property can be applied to any calculated member, the same as the `BACK_COLOR` property. Just like with the `FORMAT_STRING` property, we can provide a constant value or an expression. If we provide the expression, the color varies based on what we specified in that expression. If it's a constant, the color remains on all the cells.

In this example, we specified a green background (lighter shade) for positive values, dark red for negative values (dark shades) and grey for the rest (zero and null). Since the background became dark, we changed the font color (or foreground) to white, to keep the contrast high.

The value to be assigned is a number derived by multiplying the factors of 3 colors, components and their values. RGB function found in VBA assembly comes to the rescue here, so we don't have to do that math. That assembly is available by default and registered on every SSAS server.

The arguments of `RGB()` function are values of intensity of red, green and blue color, in that order. Higher values represent lighter colors, lower values represent darker ones. For example, the black color is (0, 0, 0), whereas white is (255, 255, 255). Everything in between is given to you to experiment with. Take some time and try it!

## There's more...

The foreground and background colors can be changed for regular cube measures as well. Since that property is not directly exposed in the GUI, we have to use the `Fore_Color()` and `Back_Color()` functions in MDX script. Here's one example:

```
Back_Color( [Measures].[Order Quantity] ) =  
    RGB(200, 200, 100);
```

### Tips and tricks

There's a color picker in the **Calculation** tab of the cube editor. When you operate in the **Form View**, two color pickers are available, one for the **Fore color** property of the measure and the other for the **Back color** property of the measure. Both are found under the **Color Expressions** section.

When you're using the **Script View**, the color picker is available in the toolbar.

The color picker returns a single value for a particular color but it also specifies the RGB values in the comment. You may find using the RGB values more convenient. If so, simply delete the single value and apply the **RGB()** function on RGB values instead.

Advanced graphical programs have color-picker features which lets you point to any dot on the screen and pick its color. As before, RGB values can be read and used in the calculations. This saves you from having to find a particular color if that color is already available somewhere, that is in the company brochure, an Excel table, chart, and so on.

Finally, here are a few sites to help you with colors and their names:

500+ Named Colours with rgb and hex values:  
<http://tinyurl.com/RGBColors1>

RGB to Color Name Mapping:  
<http://tinyurl.com/RGBColors2>

### Warning

Cells have various properties like value, font, color, etc., but only some of them are returned by default, to make the result more compact and faster: `VALUE`, `FORMATTED_VALUE` and `CELL_ORDINAL`. You can verify this by double-clicking on any cell with the value in the result of an MDX query in SQL Server Management Studio.

Other properties, `FORE_COLOR` and `BACK_COLOR`, can be retrieved on demand, using the `CELL_PROPERTIES` MDX keyword, an optional part of the query. When that keyword is used, only the explicitly stated properties are returned. In other words, if we want to include the color properties, we should specify those default properties explicitly or they won't be included in the result.

The success of this recipe ultimately depends on the SSAS front end you're using and its way of displaying values. It may be that in its default mode the front end omits cell properties in order to display results faster. If so, be sure to check whether there's an option in the program's connection or display properties which can turn them on when requested. If not, see if you can edit the MDX and insert the FORE\_COLOR and the BACK\_COLOR keywords in the CELL PROPERTIES part of the query. If that part is not there at all, you must add it at the end of the query by specifying the CELL PROPERTIES keyword and then enlisting all the properties you want to be returned. The complete list of cell properties can be found here: <http://tinyurl.com/CellProperties>

## See also

The Recipe entitled *Setting special format for negative, zero and null values* shows how to provide more than one format based on the value of the calculation using the FORMAT\_STRING property and its sections.

## Setting default member of a hierarchy in MDX script

Setting a default member is a tempting option which looks like it can be used on any dimension we would like. The truth is far from that. Default members should be used as exceptions and not as a general rule when designing dimensions.

The reason for that is not so obvious. The feature looks self-explanatory and it's hard to anticipate what could go wrong. If we're not careful enough, our calculations can become unpredictable, especially on complex dimensions with many relations among attributes.

Default members can be defined in three places. The easy-to-find option is the dimension itself, using the **DefaultMember** property found on every attribute. The second option is the role, on **Dimension Data** tab. Finally, default members can be defined in MDX script. One of the main benefits of this place is easy maintenance of all default members in the cube because everything is in one place and in the form of an easy-to-read text. That is also the only way to define the default member of a role-playing dimension.

In this recipe we'll show the most common option, that is, the last one or how to set a default member of a hierarchy in MDX script. More information on setting the **DefaultMember** is available here:

<http://tinyurl.com/DefaultMember>

## Getting ready

Follow these steps to set up the environment for this recipe:

1. Start SSMS and connect to your SSAS 2008 R2 instance.
2. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**. Then execute the following query:

```
WITH
  MEMBER [Measures].[Default account] AS
    [Account].[Accounts].DefaultMember.Name
SELECT
  { [Measures].[Amount],
  [Measures].[Default account] } ON 0
FROM
  [Adventure Works]
```

3. The results will show that the default member is **Net Income** account and its value in this context is a bit more than 12.6 million USD.
4. Next, open **Adventure Works DW 2008** solution in BIDS.
5. Double-click on the **Adventure Works** cube and go to the **Calculations** tab. Choose **Script View**.
6. Position the cursor at the beginning of the script, just below the CALCULATE command.

## How to do it...

Follow these steps to set a new default member:

1. Enter the following expression to set a new default account:

```
ALTER CUBE CurrentCube
  UPDATE DIMENSION [Account].[Accounts],
  Default_Member = [Account].[Accounts].&[48];
  //Operating Profit
```

2. Save and deploy (or just press the **Deploy MDX Script** icon if you're using **BIDS Helper**).
3. Run the previous query again.
4. Notice the result has changed. The new default account is **Operating Profit**, the one we specified in the MDX script using ALTER CUBE command. The value changed as well – now it's above 16.7 million USD.

## How it works...

The ALTER CUBE statement changes the default member of a hierarchy specified in the UPDATE DIMENSION part of the statement. The third part is where we specify which member should be the default member of that hierarchy.

Don't mind that it says "UPDATE DIMENSION". SSAS 2005 and above interpret that as hierarchy.

## There's more...

Setting the default member on a dimension with multiple hierarchies can lead to unexpected results. Due to attribute relations, related attributes are implicitly set to corresponding members while the non-related attributes remain on their default members, that is, the *All* member (also known as the root member). Certain combinations of members from all available hierarchies can result in a non-existing coordinate. In that case, the query will return no data. Other times, the intersection will only be partial. In that case, the query will return the data, but the values will not be correct which might be even worse than no data at all.

Here's an example illustrating that:

Enter the following expression in the MDX script, deploy it, and then analyze the result in the **Cube Browser** tab.

```
ALTER CUBE CurrentCube
    UPDATE DIMENSION [Date].[Calendar],
        Default_Member = [Date].[Calendar]
            .[Calendar Year].&[2007];
            -- "current" year on the user hierarchy
```

The expression sets the year 2007 as the default member of the [Date].[Calendar] user hierarchy.

The analysis of the **Sales Amount** measure in the **Cube Browser** shows good results in almost all cases except in few. Fiscal hierarchies that have the fiscal year level in them return empty or incomplete results when used in slicer. Empty, because the intersection between the fiscal year 2006 and the calendar year 2007 (the latter being the default member in the calendar hierarchy) is a non-existing combination. Remember, the calendar year 2007 doesn't get overwritten by the fiscal 2006. It gets combined (open the **Date** dimension in BIDS and observe the relations in the corresponding tab). Moreover, when you put the fiscal year 2007 in slicer, you only get a portion of data, the portion which matches the intersection of the calendar and the fiscal year. That's only one half of the fiscal year, right? In short, you have a potential problem with this approach.

Can we fix the result? Yes, we can. The correct results will be there when we explicitly select the **All** member from the **Date.Calendar** hierarchy in the slicer. Only then we get good results using fiscal hierarchies. The question is – will the end-users remember that every time?

The situation is similar when the default member is defined on an attribute hierarchy, for example, on the **Date.Calendar Year** hierarchy. By now, you should be able to modify the previous expression so that it sets the year 2007 as the default member on the `[Date].[Calendar Year]`. Test this to see it for yourself.

Another scenario could be that you want to put the current date as the default member on the **Date.Date** hierarchy. Try that too, and see that when you use the year 2006 from the **Date.Calendar Year** hierarchy in slicer, you get an empty result. Again, the intersection formed a non-existing coordinate.

To conclude, you should avoid defining default members on complex dimensions. Define them where it is appropriate: on dimensions with a single non-aggregatable attribute (that is when you set the **IsAggregatable** property of an attribute to False) or on dimensions with one or more user hierarchies where that non-aggregatable attribute is the top level on each user hierarchy and where all relations are well made.

The **Account** dimension used in this example is not such a dimension. In order to correct it, two visible attributes should be made hidden because they can cause empty results when used in slicer. Experimenting with scope might help too, but that adds to the complexity of the solution and hence the initial advice of keeping things simple when using default members should prevail.

Take a look at other dimensions in the **Adventure Works DW 2008** database. There you will find good examples of using default members.

### Helpful tips

When you're defining the default members in MDX script, do it in the beginning of the script. This way the calculations that follow can reference them.

In addition, provide a comment explaining which member was chosen to be the default member and optionally why. Look back at the code in this recipe to see how it was done.

## Implementing NOT IN set logic

There are times when we want to exclude some members from the result. We can perform this operation using a set of members on an axis or using a set of members in slicer, that is, the **WHERE** part of an MDX query. This recipe shows how to do the latter, how to exclude some members from a set in slicer. The principle is the same for any part of an MDX query.

## Getting ready

Start a new query in SSMS and check that you're working on the right database. Then type in the following query and execute it:

```
SELECT
    { [Measures].[Reseller Order Count] } ON 0,
    NON EMPTY
    { [Promotion].[Promotion].MEMBERS }
    DIMENSION PROPERTIES
        [Promotion].[Promotion].[Discount Percent]
    ON 1
FROM
    [Adventure Works]
```

The above query returns 12 promotions. The **DIMENSION PROPERTIES** keyword is used to get additional information about members – their discount percent. That property can be seen among other properties by double-clicking each member on rows.

Our task is to exclude promotions with a discount percent of 0, 2 and 5.

## How to do it...

Follow these steps to reverse the set:

1. Navigate to the **Promotion** dimension and expand it.
2. Expand the **Discount Percent** hierarchy and its level.
3. Take the first 3 members (with the names **0**, **2** and **5**) and drag them one by one below the query, then form a set of them using curly brackets.
4. Expand the query by adding the **WHERE** part.
5. Add the set with those 3 members using a minus sign in front of the set.

```
SELECT
    { [Measures].[Reseller Order Count] } ON 0,
    NON EMPTY
    { [Promotion].[Promotion].MEMBERS }
    DIMENSION PROPERTIES
        [Promotion].[Promotion].[Discount Percent]
    ON 1
FROM
    [Adventure Works]
WHERE
    ( - { [Promotion].[Discount Percent].&[0],
        [Promotion].[Discount Percent].&[2.E-2],
        [Promotion].[Discount Percent].&[5.E-2] } )
```

6. Execute the query and see how the results change. Double-click each promotion and verify that no promotion has discount percent equal to 0, 2 or 5 anymore.

	Reseller Order Count
All Promotions	363
Volume Discount 25 to 40	71
Volume Discount 41 to 60	2
Mountain-100 Clearance Sale	24
Sport Helmet Discount-2002	36
Road-650 Overstock	61
Sport Helmet Discount-2003	30
Touring-3000 Promotion	101
Touring-1000 Promotion	101
Mountain-500 Silver Clearance Sale	62

## How it works...

The initial query is not sliced by discount percentages. We can think of it as if all the members of that hierarchy are being there in the slicer:

```
WHERE ( { [Promotion].[Discount Percent]
          .[Discount Percent].MEMBERS } )
```

Of course, we don't have to write such expressions; the SSAS engine takes care of it by default. In other words, we're fine until the moment we want to change the slicer by either isolating or removing some members from that set. That's when we have to use that hierarchy in slicer.

Isolation of members is simply done by enumerating them in slicer. Reduction, the opposite operation, is performed using the `Except()` function:

```
WHERE ( Except( { [Promotion].[Discount Percent]
                  .[Discount Percent].MEMBERS },
                  { [Promotion].[Discount Percent].&[0],
                    [Promotion].[Discount Percent].&[2.E-2],
                    [Promotion].[Discount Percent].&[5.E-2] }
                )
      )
```

The alternative for the `Except()` function is a minus sign, which brings us to the shorter version of the previous expression, the version that was used in this recipe.

When a minus sign is used between two sets, it performs the same difference operation between those sets as `Extract( )` does. When the first set is missing, set of all members is implicitly added as the first set. The difference between all members and the members of any set is the opposite set of that set. This is how you can perform the **NOT IN** logic on sets. Both variants work, but the one with the minus sign in front of the set is hopefully easier to remember.

### There's more...

If we open the **Promotion** dimension inside BIDS, we'll notice that the **Discount Percent** attribute has the `MemberValue` property defined. The value of that property is equal to a discount percentage and therefore in this particular case we could write an equivalent syntax:

```
WHERE
( { Filter( [Promotion].[Discount Percent]
    .[Discount Percent].MEMBERS,
    [Promotion].[Discount Percent]
    .CurrentMember.MemberValue >= 0.1 ) } )
```

The advantage of this expression is that it should filter out additional members with a percentage less than 10% if they ever appear on that hierarchy. If we're not expecting such a case or if we strictly want to exclude certain, not necessarily consecutive, members from the hierarchy (Unknown Member, NA member, and so on), we should use the first example: the one with explicit members in the set.

### See also

The next recipe, *Implementing logical OR on members from different hierarchies* is based on a similar theme as this recipe.

## Implementing logical OR on members from different hierarchies

The nature of a multidimensional database and its underlying structures has a direct consequence on how we should write combinations of members. Some combinations are there by design, others require a bit of imagination.

For example, a set of two members of the same hierarchy (colors black and white) placed in a slicer automatically applies OR logic on the result. This means that the result will have data where the first, the second or both of the members (or at least one of their descendants to be precise) occurred in the underlying fact table. In other words, where the product sold was either black or white. The emphasis is on two things: the set and the **OR** word. In other words, OR logic manifests in sets.

The other example is a tuple formed by two members from different hierarchies (i.e. color black and size XL). Once placed in slicer, this tuple guarantees that the resulting rows will have data on that exact slice, meaning, on both members (or at least one of the descendants of each to be precise). Here, the emphasis is again on two things: the tuple and the **AND** word. In other words, AND logic manifests in tuples.

Let's summarize. In MDX, a set is by default the equivalent of logical OR while a tuple is by default the equivalent of logical AND. So where's the problem?

The problem is we can only put members of different hierarchies in a tuple and of the same hierarchy in a set. Which means we're missing two combinations, different hierarchies using OR and the same hierarchy using AND.

This recipe shows how to implement OR logic using members from different hierarchies. The last recipe in this chapter shows how to perform AND logic using members from the same hierarchy. It is recommended that you read both recipes.

## Getting ready

Start a new query in SSMS and check that you're working on the right database. Then type in the following query and execute it:

```
SELECT
    { [Measures].[Reseller Order Quantity],
      [Measures].[Reseller Order Count] } ON 0,
  NON EMPTY
    { [Product].[Subcategory].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Color].&[Black] )
```

The query displays 10 product subcategories containing black products.

Next, open a new query window and execute the following query.

```
SELECT
    { [Measures].[Reseller Order Quantity],
      [Measures].[Reseller Order Count] } ON 0,
  NON EMPTY
    { [Product].[Subcategory].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Size Range].&[XL] )
```

It's a query similar to the previous one, but this one returns only product subcategories containing XL size range products. There's only one product subcategory in the result - **Jersey**.

The task is to combine these queries so that we get the result of OR operation on those two conditions, in a single query, of course.

## How to do it...

Follow these steps to apply the OR logic with members from different hierarchies:

1. Copy-paste the first query in the new window.
2. Copy-paste the member in slicer from the second query and put it in slicer as well.
3. Notice that those two members originate from different hierarchies and that we cannot put them in a set.
4. Wrap each member in brackets so that they become tuples.
5. Put a comma between those two tuples and wrap them in curly brackets so that they form a multidimensional set of two tuples.
6. Add the root member of the opposite hierarchy in each tuple while preserving the order of hierarchies in them.
7. You're done. The query should look as follows. Notice how the color and size range hierarchies have the same order in those tuples. That's the way it should be.

```
SELECT
    { [Measures].[Reseller Order Quantity],
      [Measures].[Reseller Order Count] } ON 0,
    NON EMPTY
    { [Product].[Subcategory].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
(
    { ( [Product].[Color].&[Black],
        [Product].[Size Range].[All Products] )

    ,
        ( [Product].[Color].[All Products],
        [Product].[Size Range].&[XL] ) }
)
```

8. Next, execute that query and check that you have 12 rows including the total on top:

	Reseller Order Quantity	Reseller Order Count
All Products	78,035	3,017
Cranksets	1,107	261
Gloves	11,553	991
Helmets	4,447	922
Jerseys	6,022	967
Mountain Bikes	12,771	1,119
Mountain Frames	5,604	736
Road Bikes	14,304	1,237
Road Frames	3,456	769
Shorts	8,946	758
Tights	4,562	470
Wheels	5,263	716

### How it works...

Logical OR represents a set. Since we have members of different dimensionality, we must first convert them to tuples of the same dimensionality. That is done by expanding each with the other one's root member and enclosing the expression in brackets (which is how we convert a member to a tuple). Once we have compatible tuples, we can convert them into a set by separating them with a comma and adding curly brackets around the whole expression. This is the standard way that we enumerate members in single-dimensional sets. Multi-dimensional sets are no different except it's the tuples that we're enumerating this time.

### There's more...

We can also use the `UNION( )` function instead of enumerating members in the set. The union has an extra feature, an option to remove or preserve duplicates in the resulting set. While that feature is of little interest when the slicer is concerned, it might be interesting when the same logic is applied in calculations. Keep that in mind.

### A special case of a non-aggregatable dimension

In case your dimension has no root member (eliminated by setting the property `IsAggregatable` to False) use its default member instead.

## Elementary MDX Techniques

---

### A very complex scenario

In this recipe we used two hierarchies of the same dimension because this is often the case in real life. However, that's not a limitation. The solution is applicable to any dimension and its hierarchies. For example, when you need to combine 3 different hierarchies you can apply the same solution, thereby expanding each member into tuple with N-1 root members (here N=2) and creating a set of N such tuples.

In case you need to combine many members using OR logic, sometimes with even more than one of them on the same hierarchy and others on different hierarchies, you need to apply the knowledge about dimensionality – members of the same hierarchy should be enlisted in a set, members of different dimensions should be combined with root members of other hierarchies. You just need to be careful with various brackets. The `AsymmetricSet()` function from the Analysis Services Stored Procedure Project may help construct complex sets:  
<http://tinyurl.com/AsymmetricSet>

### See also

The *Implementing NOT IN set operation* recipe is based on a similar theme to this recipe.

For more information on default members, take a look at the *Setting default member of a hierarchy in MDX script* recipe.

## Iterating on a set in order to reduce it

Iteration is a very natural way of thinking for us humans. We set a starting point, we step into a loop, and we end when a condition is met. While we're looping, we can do whatever we want: check, take, leave, and modify items in that set. Being able to break down the problems in steps makes us feel that we have things under control. However, by breaking down the problem, the query performance often breaks down as well. Therefore, we have to be extra careful with iterations when data is concerned.

If there's a way to manipulate the collection of members as one item, one set, without cutting that set into small pieces and iterating on individual members, we should use it. It's not always easy to find that way, but we should at least try.

This and the next two recipes show how to perform iteration. They deal with those cases when there's no other way but to iterate. However, some of the recipes also point out which calculation patterns we must recognize and thereby give up on using classic naïve iteration and use a better approach.

## Getting ready

Start a new query in SSMS and check that you're working on the right database. Then write the following query:

```
SELECT
    { [Measures].[Customer Count],
      [Measures].[Growth in Customer Base] } ON 0,
    NON EMPTY
    { [Date].[Fiscal].[Month].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Product Categories].[Subcategory].&[1] )
```

The query returns fiscal months on rows and two measures: a count of customers and their growth compared to the previous month. Mountain bikes are in slicer.

Now let's see how we can get the number of days the growth was positive for each period.

## How to do it...

Follow these steps to reduce the initial set:

1. Create a new calculated measure in the query and name it Positive growth days.
2. Specify that you need descendants of current member on leaves.
3. Wrap around the FILTER( ) function and specify the condition which says that the growth measure should be greater than zero.
4. Apply the COUNT( ) function on a complete expression to get count of days.
5. The new calculated member's definition should look as follows, verify that it does.

```
WITH
MEMBER [Measures].[Positive growth days] AS
  FILTER(
    DESCENDANTS([Date].[Fiscal].CurrentMember, , leaves),
    [Measures].[Growth in Customer Base] > 0
  ).COUNT
```

6. Add the measure on columns.

7. Run the query and observe if the results match the following image:

	Customer Count	Growth in Customer Base	Positive growth days
July 2005	31	(null)	2
August 2005	31	0.00%	7
September 2005	13	-58.06%	0
October 2005	27	107.69%	2
November 2005	32	18.52%	3
December 2005	39	21.88%	7
January 2006	36	-7.69%	3
February 2006	18	-50.00%	2
March 2006	39	116.67%	4
April 2006	41	5.13%	7
May 2006	45	9.76%	7
June 2006	44	-2.22%	8

### How it works...

The task says we need to count days for each time period and use only positive ones. Therefore, it might seem appropriate to perform iteration, which, in this case, can be performed using the `FILTER( )` function.

But, there's a potential problem. We cannot expect to have days on rows, so we must use the `DESCENDANTS( )` function to get all dates in the current context.

Finally, in order to get the number of items that came up upon filtering, we use the `COUNT` function.

### There's more...

Filter function is an iterative function which doesn't run in block mode, hence it will slow down the query. In the introduction, we said that it's always wise to search for an alternative if available. Let's see if something can be done here. A keen eye will notice a "count of filtered items" pattern in this expression. That pattern suggests the use of a set-based approach in the form of **SUM-IF** combination. The trick is to provide 1 for the True part of the condition taken from the `FILTER( )` statement and null for the False part. The sum of one will be equivalent to the count of filtered items.

In other words, once rewritten, that same calculated member would look like this:

```
MEMBER [Measures].[Positive growth days] AS
    SUM(
        Descendants([Date].[Fiscal].CurrentMember, , leaves),
        IIF( [Measures].[Growth in Customer Base] > 0, 1, null)
    )
```

Execute the query using the new definition. Both the `SUM()` and the `IIF()` functions are optimized to run in the block mode, especially when one of the branches in `IIF()` is null. In this particular example, the impact on performance was not noticeable because the set of rows was relatively small. Applying this technique on large sets will result in drastic performance improvement as compared to the `FILTER-COUNT` approach. Be sure to remember that in future.

More information about this type of optimization can be found in Moshav Pasumansky's blog:

<http://tinyurl.com/SumIIF>

### Hints for query improvements

There are several ways you can avoid the `FILTER()` function in order to improve performance.

- ▶ When you need to filter by non-numeric values (i.e. properties or other metadata), you should consider creating an attribute hierarchy for often-searched items and then do one of the following:
  - Use a tuple when you need to get a value sliced by that new member
  - Use the `EXCEPT()` function when you need to negate that member on its own hierarchy (`NOT` or `<>`)
  - Use the `EXISTS()` function when you need to limit other hierarchies of the same dimension by that member
  - Use the `NONEMPTY()` function when you need to operate on other dimensions, that is, subcubes created with that new member
  - Use the 3-argument `EXISTS()` function instead of the `NONEMPTY()` function if you also want to get combinations with nulls in the corresponding measure group (nulls are available only when the **NullProcessing** property for a measure is set to `Preserve`)
- ▶ When you need to filter by values and then count a member in that set, you should consider aggregate functions like `SUM()` with `IIF()` part in its expression, as described earlier.

## See also

The next recipes, *Iterating on a set in order to create a new one* and *Iterating on a set using recursion*, deal with other methods of iteration.

# Iterating on a set in order to create a new one

There are situations when we don't want to eliminate certain members from a set, but instead execute a for-each type of loop. This is done using the **GENERATE()** function. In this recipe we'll show you how to create a new set of members from the existing one.

## Getting ready

Start a new query in SSMS and check that you're working on the right database. Then write the following query:

```
SELECT  
    NON EMPTY  
    { [Date].[Calendar].[Calendar Year].MEMBERS *  
      [Measures].[Sales Amount] } ON 0,  
    NON EMPTY  
    { [Sales Territory].[Sales Territory Country].MEMBERS }  
    ON 1  
FROM  
    [Adventure Works]
```

The query returns 4 years on columns and 6 countries on rows. This recipe shows how to get a set of best months, one for each year.

## How to do it...

Follow these steps to create a new set from the initial one:

1. Cut the years from columns and define a named set using them.
2. Name that set `Best month per year`.
3. Wrap that set in the `Generate()` function so that the set of years becomes its first argument.

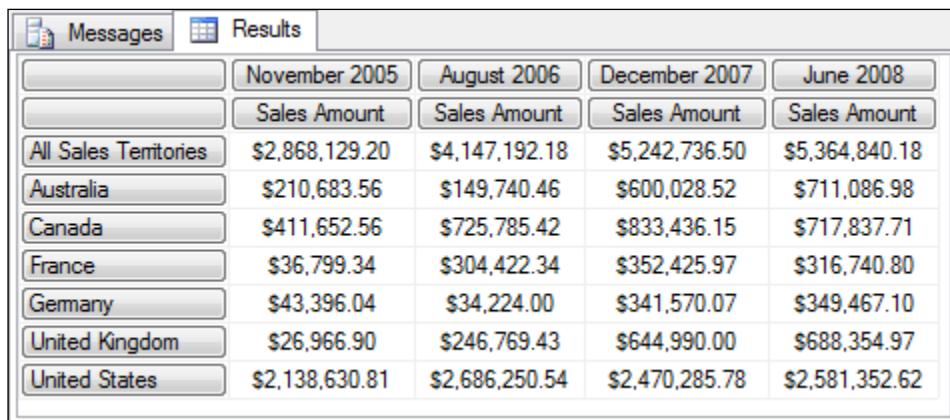
4. The second argument should be the TopCount() function which uses the descendants of each year on the Month level and finds the best month according to the value of the measure Sales Amount.
5. Put the name of the new set on columns.
6. The final query should look as follows:

```

WITH
SET [Best month per year] AS
    Generate( [Date].[Calendar].[Calendar Year].MEMBERS,
        TopCount(
            Descendants( [Date].[Calendar].CurrentMember,
                [Date].[Calendar].[Month],
                SELF ),
            1,
            [Measures].[Sales Amount] )
        )
SELECT
    NON EMPTY
    { [Best month per year] *
        [Measures].[Sales Amount] } ON 0,
    NON EMPTY
    { [Sales Territory].[Sales Territory Country].MEMBERS } ON 1
FROM
    [Adventure Works]

```

7. Execute the query. Notice that each year is replaced with a single month, the month with the best sales result in that year:



The screenshot shows the SSMS Results tab displaying a table of sales data. The table has five columns: November 2005, August 2006, December 2007, and June 2008. The rows list territories and their corresponding sales amounts for those specific months. The data is as follows:

	November 2005	August 2006	December 2007	June 2008
	Sales Amount	Sales Amount	Sales Amount	Sales Amount
All Sales Territories	\$2,868,129.20	\$4,147,192.18	\$5,242,736.50	\$5,364,840.18
Australia	\$210,683.56	\$149,740.46	\$600,028.52	\$711,086.98
Canada	\$411,652.56	\$725,785.42	\$833,436.15	\$717,837.71
France	\$36,799.34	\$304,422.34	\$352,425.97	\$316,740.80
Germany	\$43,396.04	\$34,224.00	\$341,570.07	\$349,467.10
United Kingdom	\$26,966.90	\$246,769.43	\$644,990.00	\$688,354.97
United States	\$2,138,630.81	\$2,686,250.54	\$2,470,285.78	\$2,581,352.62

## How it works...

The `Generate( )` function can be thought of as a for-each loop. This means that we will iterate through each member of the initial set and assign another set instead of each member. That new set can have zero, one, or many members and this can vary during the iteration. In our example we're assigning a set with one member only, the best month in each year. That member is obtained using the `TopCount( )` function where the first argument is months of the current year in iteration, the second argument is 1 (only one member to be returned), and the third argument is the `Sales Amount` measure – the criterion for deciding which month is the best. Months are obtained the standard way, using the `Descendants( )` function.

## There's more...

The **CURRENTORDINAL** function is a special MDX function valid only in iterations. It returns the position of the current member (or tuple, to be precise) in the set in iteration (from 0 to N, where N is the total number of members in a set). In addition to that, there's also the **CURRENT** function. The **CURRENT** function returns the current tuple in a set being iterated. Again, it's only applicable during iterations.

Both of these functions can be used to detect the current tuple and to create various calculations with the current tuple and other tuples in that set. Reversing any initial set is one example of these manipulations. Comparing the value of the current tuple with the value of the previous tuple in the set (or any one before or after) in order to isolate certain tuples is another example.

Here's how you could reverse the set of months from the previous example.

```
SET [Best month per year reversed] AS
    Generate( [Date].[Calendar].[Calendar Year].MEMBERS
        AS MySetAlias,
        TopCount(
            Descendants(
                MySetAlias.Item( MySetAlias.Count -
                    MySetAlias.CurrentOrdinal
                    - 1 ).Item(0),
                [Date].[Calendar].[Month],
                SELF ),
            1,
            [Measures].[Sales Amount] )
    )
```

A set alias (`MySetAlias` in this example) is defined for the initial set. That set alias is later used for navigation. The combination of `Count` and `CurrentOrdinal` gives us members from the end of the set to its beginning, progressively, while the `Item( )` function serves as a pointer on members in that set.

Yes, the same operation could be done simply by sorting the months by their member key, in descending order. Nevertheless, the idea of that example was to show you the principle which can be applied on any set, especially those that can't be reversed easily.

The other example mentioned above uses the `Filter()` function, not the `Generate()` function. There, tuples can be compared to each other progressively in order to see which one has the value higher than both of its neighboring members, which would signal that the current member is a relative peak. Or the opposite, whatever is more interesting in a particular case. However, the `Filter()` function doesn't add new members, it only limits its initial set and for that reason it is out of the scope of this recipe.

To summarize, `Current()` and `CurrentOrdinal()` are powerful functions that allow us to perform self-joining type of operations in MDX or make use of the existing relations between dimensions and measure groups. These functions are useful not only in the `Generate()` function, but in other iterating functions as well, namely, the `Filter()` function.

### Did you know

In MDX, there's no concept of the `FOR` loop. Iterations cannot be based on numbers (as in other languages or on other systems). They must always be based on a set. If we need to loop exactly N times, there are two basic ways we can achieve this. One is with the existing cube structure, the other is by expanding a cube with a utility dimension. The former means that we can use date dimension and take N members from its start. Or it could be some other dimension, as long as it has enough members to loop on. The latter, using the utility dimension, will be explained later in *Chapter 8, Implementing the Tally table utility dimension*

### See also

The recipes *Iterating on a set using recursion* and *Iterating on a set in order to reduce it* show other methods of iteration.

## Iterating on a set using recursion

Recursion is sometimes the best way to iterate a collection. Why? Because iterations using set functions (including the `GENERATE()` function) require that we loop through the whole set. But what if that set is big and we only need to find something in it? Wouldn't it be great to be able to stop the process when we've found what we wanted? Recursion enables just that – to stop when we're done.

In this recipe we're going to see how to calculate the average of an average using recursion.

## Getting ready

To get started, start a new query in SSMS and check that you're working in the right database. Then write the following query:

```
SELECT
    { [Measures].[Order Count] } ON 0,
    NON EMPTY
    { Descendants( [Date].[Fiscal Weeks].[All Periods],
        1 , SELF_AND_BEFORE) } ON 1
FROM
    [Adventure Works]
```

It returns 4 fiscal years and their total on top for the Order Count measure. Now let's see how to calculate the average daily value on the week level and the average weekly level on the year level, but based on the week level, not on the date level. In other words, each level will have the average value of members on the level immediately below.

## How to do it...

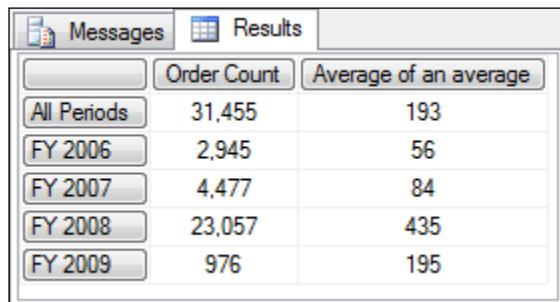
Follow these steps to perform recursion over a set:

1. Define a new calculated measure and name it Average of an average.
2. Use the IIF() function and specify its True parameter as the initial measure (Order Count).
3. The value should be returned for the leaf level, so the condition in IIF() should test exactly that using the ISLEAF() function.
4. In the False parameter we should provide the calculation we want to repeat recursively. In this case it is the AVG() function used on children of the current member.
5. The measure expression inside AVG() function should be the measure being defined.
6. Check if the measure is defined as follows:

```
WITH
MEMBER [Measures].[Average of an average] AS
    iif( IsLeaf( [Date].[Fiscal Weeks].CurrentMember ),
        [Measures].[Order Count],
        Avg( [Date].[Fiscal Weeks].CurrentMember.Children,
            [Measures].[Average of an average] )
    )
, FORMAT_STRING = '#,#'
```

7. Don't forget to include that measure as the second measure on columns.

8. Run the query. The results will look as follows. The first row, the one with the **All Periods** member, will have the average yearly value as result, that is  $(56+84+435+195)/4=193$ . In turn, every year will have the average weekly value. The weekly values are not visible in this screenshot, but we can divide the **Order Count** values by 53, that is, by the number of weeks per year. That should give us the values for the **Average of an average** measure shown in the second column.



	Order Count	Average of an average
All Periods	31,455	193
FY 2006	2,945	56
FY 2007	4,477	84
FY 2008	23,057	435
FY 2009	976	195

### How it works...

Recursions are the most difficult iteration concept to apply. Their logic is very condensed. However, once you conquer them, you'll appreciate their power and efficiency. Let's see how that solution worked.

In order to start the recursive process, we have to specify an expression that uses the same calculated measure we're defining, thereby providing a different input parameter than the one which was being used in the current pass of recursive process. In order to stop the process, we must have a branch without the reference to that measure. On top of all that, we must perform some operation to collect values on the way. Complicated? Let's analyze our query.

Fiscal years on rows are not the leaf level of the Fiscal Weeks user hierarchy. Therefore, the expression inside the `IIF( )` statement evaluates as False. This leads us to the part where we have to calculate the average value for each child of the current member. With a small detail, the calculation should be performed using the same measure we're evaluating!

The evaluation for the current year member cannot be completed and is therefore delayed until the calculation for all its child members (weeks in this case) is performed. One by one, each week of the year in context is passed inside the definition of this measure and evaluated.

In case of a leaf member, the **Order Count** measure would be evaluated and returned to the outer evaluation context. Otherwise, another turn of the child member's evaluation would occur. And so on until we would finally hit leaf-level members.

In this example, weeks are the leaf level of the hierarchy being used in the query. They would be evaluated using the True part of the condition. The True parameter is without reference to the measure we're calculating, which means the recursive path would be over. The value of the **Order Count** measure starting from the Week **1 of FY 2006** would be collected and saved in a temporary buffer. The same process would be repeated for all weeks of that year. Only then the average of them would be calculated and returned as a value for FY 2006. After which the process would repeat for subsequent years on rows.

Let's also mention that the value for the root member (All years) is calculated with the recursion depth of 2, meaning each year it is first evaluated as an average of its weeks and then the average of its years is calculated and returned as the final result.

### There's more...

You might be wondering how does one recognize when to use recursion and when to use other types of iteration? Look for some of these pointers: relative positions, relative granulation for calculation and stop logic. If there's a mention of going back or forth from the current member in a set, but there's no fixed span, then that might be a good lead to use recursion. If there's a relative stopping point, that's another sign. Finally, if there's no explicit requirement to loop through the whole set, but moreover a requirement to stop at some point in the process, that's a definite sign to try to apply recursion as a solution to the problem.

In case no such signs exist, it's perhaps better and easier to use simple types of iterations we covered in previous recipes. The other case when you should consider straightforward iteration is when the recursion would span over more than half of the members on a particular hierarchy, that pushes the SSAS engine into the slow cell-by-cell mode.

### Earlier versions of SSAS

SSAS 2008 and later have better support for recursion than previous versions of SSAS. Optimizations have been added to the code in form of unlimited recursion depth. Versions prior to that may suffer from memory limitations in some extreme cases.

### See also

The recipes *Iterating on a set in order to create a new one* and *Iterating on a set in order to reduce it* illustrate other ways of iteration.

## Dissecting and debugging MDX queries

If you write a query involving complex calculations, you might have a hard time trying to debug it in, not if case there is a problem inside, but there is a way. By breaking complex sets and calculations into smaller ones and/or by converting those sets and members into strings, we can visually represent the intermediate results and thereby isolate the problematic part of the query.

True, there's no real debugger in the sense that you can pause the calculation process of the query and evaluate the variables. What you can do is to simulate that using this approach.

### Getting ready

For this recipe we'll use the final query in the previous recipe, *Iterating on a set using recursion*. We have chosen this as our example because it's a relatively complex calculation and we want to check if we're doing the right thing.

### How to do it...

Follow these steps to create a calculated measure that shows the evaluation of another calculation:

1. Start SSMS and execute the following query:

```
WITH
MEMBER [Measures].[Average of an average] AS
    iif( IsLeaf( [Date].[Fiscal Weeks].CurrentMember ),
        [Measures].[Order Count],
        Avg( [Date].[Fiscal Weeks].CurrentMember.Children,
            [Measures].[Average of an average] )
    )
    , FORMAT_STRING = '#,#'
SELECT
    { [Measures].[Order Count],
        [Measures].[Average of an average] } ON 0,
    NON EMPTY
    { Descendants( [Date].[Fiscal Weeks].[All Periods],
        1 , SELF_AND_BEFORE) } ON 1
FROM
    [Adventure Works]
```

2. Create a new calculated measure and name it `Proof`.
3. Copy-paste to measure we're validating inside the definition of the `Average of an average` measure.
4. Leave the `True` part as is.

5. Modify the False part as shown in the next step below.
6. Finally, wrap the whole expression with one IIF() statement that checks whether the original measure is empty. The definition of that measure should look like this:

```
MEMBER [Measures].[Proof] AS
    iif( IsEmpty( [Measures].[Order Count] ),
        null,
        iif( IsLeaf( [Date].[Fiscal Weeks].CurrentMember ),
            [Measures].[Order Count],
            '( ' +
            Generate( [Date].[Fiscal Weeks]
                .CurrentMember.Children,
            iif( IsEmpty( [Measures]
                .[Average of an average] ),
                '(null)',
                CStr(
                    Round( [Measures]
                        .[Average of an average],
                        0 ) )
                ),
                ' + ' ) +
            ' ) / ' +
            CStr( NonEmpty( [Date].[Fiscal Weeks]
                .CurrentMember.Children,
            [Measures].[Order Count]
            ).Count )
        )
    )
)
```

7. Add that measure on columns and execute the query. The result will look like this:

	Proof
All Periods	( 56 + 84 + 435 + 195 + (null) ) / 4
FY 2006	+ 46 + 47 + 33 + 98 + 59 + 50 + 57 + 50 + 14 + 65 + 43 + 46 + 42 + 123 + 43 + 49 + 33 + 111 +
FY 2007	- 49 + 39 + 47 + 41 + 189 + 87 + 68 + 72 + 28 + 100 + 55 + 60 + 54 + 182 + 77 + 65 + 73 + 162
FY 2008	15 + 396 + 424 + 649 + 495 + 470 + 442 + 278 + 253 + 428 + 429 + 410 + 424 + 637 + 421 + 49
FY 2009	( 105 + 217 + 220 + 223 + 211 + (null) + (null) + (null) + (null) + (null) ) / 5

## How it works...

The general principle of debugging MDX queries is to show some text, that is, current member names, their properties, position in a set, their descendants, and ancestors, whatever helps. Other times we'll convert the complete sets that we're operating with into a string, just to see the members inside and their order. For numeric values, if they are formed using several sub-calculations like in this example, we try to compose that evaluation as a string too. In short, we're displaying textual values of items we are interested in.

In our example, the main part where something interesting is going on is the `False` parameter of the inner `IIF()` function. Therefore, that's the place we're building a string in. Since it's relatively hard to explain what exactly we are doing there, let's take one more look at the previous image before we continue any further.

`Measure Proof` is a string representation of all individual values used to calculate each row. It is represented as a sum of `N` values, where `N` is number of children of the current member, divided by their count. Additionally, null values are preserved and displayed as well in the string, but the count omits them.

Now, the calculation itself. First, there's an open bracket in the form of a string. Then the `Generate()` function is applied, only this time it's the second version of that function, the one that returns not a set but a string. More information about it can be found here:

<http://tinyurl.com/MDXGenerate>

Partial strings generated during iteration need to be concatenated. For that reason the third argument of the `Generate()` function was used with the value "+".

The `Generate()` function, as explained in the recipe *Iterating on a set in order to create a new one* is a type of loop. In this case, it takes each child of a current member of the **Fiscal Weeks** user hierarchy and tests whether it is empty or not. If it is, a constant string is used ('(null)'), if not, the value of the measure is rounded to zero decimals.

Which measure? That same measure we're calculating the result for. Hence, it's again a call for iteration, this time using each child, one by one, because they are in the context at the time of call.

In the new pass, those members will be leaf members. They'll collect the value of measure `Order Count` and get out of that pass.

Once all the children are evaluated, the individual values will be concatenated using a " + " sign with a space on each side for better readability.

But the process is not over, only recursion is.

Next, we have to close the bracket which we opened in the beginning of the process and we have to calculate the denominator. Notice the measure inside the denominator is not calling for recursion. In order to get the count of members, we used the `NonEmpty()` function over the original measure. That returns the members which have values.

Finally, we haven't mentioned this specifically so far, but the outer `IIF()` statement checks if we're on a member that has no result. If so, we can skip that member. Remember, we had to do that because the inner part of the `Proof` measure is a string which is never null.

### There's more...

In the process of dissecting, evaluating and debugging calculations and queries, various MDX functions can be used. Some of them are mentioned below. However it is advised that you look for additional information on MSDN and other sources.

- ▶ String functions, namely **MembertToStr** and **SetToStr**, for converting members and sets into strings.
- ▶ Set functions, namely the **Generate()** function and especially its string variant, which is a very powerful method for iterating on a set and collecting partial calculations in form of strings.
- ▶ Metadata functions (also known as hierarchy and level functions), for collecting information about members and their hierarchies.
- ▶ Logical functions, for testing on leaf level and emptiness.
- ▶ VBA functions, for handling errors (**IsError()**) and string manipulations.



Don't forget to use the **AddCalculatedMembers()** function if you need to include calculated members.



### Useful string functions

Here's a list of VBA functions that can be used in MDX:

<http://tinyurl.com/MDXVBA>

Here's a list of MDX functions grouped by types:

<http://tinyurl.com/MDXfunctions>

### See also

The recipe *Optimizing MDX queries using NonEmpty() function* shows how to keep only relevant members for debugging purposes and prevent all members of a hierarchy from being returned as the result.

## Using NON\_EMPTY\_BEHAVIOR

In SSAS 2005, the **NON\_EMPTY\_BEHAVIOR** property (NEB for short) was used for optimization of calculated members and cells as well as MDX script assignments. The optimization was done by indicating under which circumstances the expression can be ignored because the dependent expression returns null and vice versa. The rules for proper usage were either unknown or a bit too complex for many, so there was a lot of a misuse and incorrect results.

The situation changed with SSAS 2008. The new engine handled many scenarios automatically, without the NEB specified. It has been said that NEB is required only in a small number of situations where the expression is very complex or dense. In all others it should be left out.

The recipe in front of you shows how to specify NEB for calculated measure.

### Getting ready

We're going to recycle the query in the recipe *Dissecting and debugging MDX queries*, the calculated member named `Proof`, to be precise, but we'll comment the NON EMPTY line of the query in order to see the effect of our action. Start a new query in SSMS and check that you're working on the right database, then write this query and execute it:

```
WITH
MEMBER [Measures].[Average of an average] AS
    iif( IsLeaf( [Date].[Fiscal Weeks].CurrentMember ),
        [Measures].[Order Count],
        Avg( [Date].[Fiscal Weeks].CurrentMember.Children,
            [Measures].[Average of an average] )
    )
, FORMAT_STRING = '#,#'
MEMBER [Measures].[Proof] AS
    iif( IsEmpty( [Measures].[Order Count] ),
        null,
        iif( IsLeaf( [Date].[Fiscal Weeks].CurrentMember ),
            [Measures].[Order Count],
            '(' +
            Generate( [Date].[Fiscal Weeks]
                .CurrentMember.Children,
            iif( IsEmpty( [Measures]
                .[Average of an average]),
                '(null)',
                CStr(
                    Round( [Measures]
                        .[Average of an average],

```

```

          0 ) )
      ),
      ' + ' ) +
      ' ) / ' +
      CStr( NonEmpty( [Date].[Fiscal Weeks]
                      .CurrentMember.Children,
                      [Measures].[Order Count]
                  ).Count )
      )
  )
SELECT
  { [Measures].[Order Count],
    [Measures].[Proof],
    [Measures].[Average of an average] } ON 0,
  // NON EMPTY
  { Descendants( [Date].[Fiscal Weeks].[All Periods],
                 1 , SELF_AND_BEFORE) } ON 1
FROM
  [Adventure Works]

```

The idea is to provide the NEB expression for the **Proof** measure.

### How to do it...

Follow these steps to add the NON\_EMPTY\_BEHAVIOR to a calculation:

1. Provide the NEB expression for the **Proof** measure and specify `[Measures].[Order Count]` for it. This is what the new part should look like:  
`, NON_EMPTY_BEHAVIOR = [Measures].[Order Count]`
2. Run the query again. Nothing has changed; the result is still the same and it was obtained as fast as the initial query.

	Order Count	Proof	Average of an average
All Periods	31,455	( 56 + 84 + 435 + 195 + (null) ) ...	193
FY 2006	2,945	( 72 + 29 + 30 + 44 + 107 + 37 ...	56
FY 2007	4,477	( 143 + 59 + 56 + 54 + 192 + 5...	84
FY 2008	23,057	( 461 + 112 + 104 + 126 + 360 ...	435
FY 2009	976	( 105 + 217 + 220 + 223 + 211 ...	195
FY 2011	(null)	(null)	(null)

## How it works...

As we said in the introduction to this recipe, SSAS engine handles most of the non-empty cells automatically and we don't have to provide NEB. In this experiment we noticed no visible improvement in query speed. Careful measurements using a tool called MDX Studio (<http://tinyurl.com/MDXStudioFolder>) revealed that NEB did two things: it caused fewer cells to be calculated (366 instead of 540), but it increased the duration of the query, although very, very little. In short, no benefit was gained from providing the NEB property.

## There's more...

The advantage of NEB manifests in complex calculations where many cells have to be calculated in order to get the result. NEB reduces that number of cells and hence makes such calculations faster.

In spite of its name, the true nature of the NON\_EMPTY\_BEHAVIOR property is that it is a performance hint only, not a function for removing empty cells. It's should be seen as an "empty values hint", not a "behavior". In other words, the NON\_EMPTY\_BEHAVIOR should not be used for removing empty cells instead of the IIF() function, NonEmpty() function or other similar functions just because that seems easier or more readable to do. True, in most cases it will work the same, but not always which makes it unreliable. Since it's only a hint and not a command like the previously mentioned functions, the engine might occasionally reject that suggestion and calculate on its own. Surely you don't want that to happen in your project. Therefore, experiment with NEB on complex calculations, having a hint might help the engine in that case. But if you need to remove empty cells, make sure there's also an adequate function for that inside the calculation, in addition to that hint.

### Did you know?

NEB can be used in an MDX script as well as in a query.

NEB can contain not only one or more measures, but a tuple as well. More information about the NEB property and when and how it can be applied can be found in SSAS 2008 Performance Guide:

<http://tinyurl.com/PerfGuide2008>

## See also

*The Optimizing MDX queries using the NonEmpty() function* and *Dissecting and debugging MDX queries* recipes deal with reducing the number of rows returned from a query by identifying rows with null values.

## Optimizing MDX queries using the NonEmpty() function

**NonEmpty()** is a very powerful MDX function. It is primarily used to reduce sets in a very fast manner.

In the previous recipe we used it to get the count of children of the current member that are not empty for the measure **Order Count**. In this recipe, we'll show how it can be used again, this time using the **Customer** and **Date** dimensions.

### Getting ready

Start a new query in SSMS and check that you're working on the right database. Then write the following query and execute it:

```
SELECT
    { [Measures].[Internet Sales Amount] } ON 0,
    NON EMPTY
    Filter(
        { [Customer].[Customer].[Customer].MEMBERS } *
        { [Date].[Date].[Date].MEMBERS },
        [Measures].[Internet Sales Amount] > 1000
    ) ON 1
FROM
    [Adventure Works]
```

The query shows the sales per customer and dates of their purchases and isolates only those combinations where the purchase was over 1000 USD.

After some time, maybe a minute or so, the query will return the results.

A screenshot of the SSMS Results tab. The table has three columns: Customer Name, Purchase Date, and Internet Sales Amount. The data is as follows:

Internet Sales Amount		
Aaron A. Allen	June 4, 2006	\$3,399.99
Aaron A. Hayes	March 31, 2008	\$2,329.98
Aaron C. Campbell	March 13, 2008	\$1,155.48
Aaron C. Diaz	December 2, 2005	\$3,578.27
Aaron C. Diaz	September 26, 2007	\$2,451.30
Aaron C. Scott	January 21, 2008	\$2,492.32
Aaron E. Baker	March 12, 2008	\$1,750.98
Aaron E. Evans	December 13, 2007	\$2,433.04
Aaron Foster	January 8, 2008	\$2,430.44
Aaron Foster	June 9, 2008	\$2,482.03
Aaron Gonzales	June 18, 2008	\$1,810.46
Aaron J. Hughes	January 17, 2007	\$2,049.10

Now let's see how to improve the execution time.

## How to do it...

Follow these steps to improve the query performance by adding the `NonEmpty()` function:

1. Wrap `NonEmpty()` around the cross-join of customers and dates so that it becomes the first argument of that function.
2. Use the measure on columns as the second argument of that function.
3. This is how the MDX query should look like:

```
SELECT
    { [Measures].[Internet Sales Amount] } ON 0,
NON EMPTY
    Filter(
        NonEmpty(
            { [Customer].[Customer].[Customer].MEMBERS } *
            { [Date].[Date].[Date].MEMBERS },
            { [Measures].[Internet Sales Amount] }
        ),
        [Measures].[Internet Sales Amount] > 1000
    ) ON 1
FROM
    [Adventure Works]
```

4. Execute that query and observe the results as well the time required for execution.  
The query returned the same results, only much faster, right?

## How it works...

Both the **Customer** and **Date** dimensions are medium-sized dimensions. The cross-join they make contains several million combinations. The `Filter()` operation is not optimized to work in block mode, which means a lot of calculations will have to be performed by the engine to evaluate the set on rows.

Fortunately, the `NonEmpty()` function exists. This function can be used to reduce any set, especially multidimensional sets that are result of a cross-join operation. A reduced set has fewer cells to be calculated and therefore the query runs much faster.

## There's more...

Regardless of the benefits that were shown in this recipe, `NonEmpty()` should be used with caution. The best practice says we should use it with sets (named sets and axes) as well as in functions which are not optimized to work in block mode, such as with the `Filter()` function. On the other hand, we should avoid using `NonEmpty()` in aggregate functions such as `Sum()` and other MDX set functions that are optimized to work in block mode. The use of `NonEmpty()` inside optimized functions will prevent them from evaluating the set in block mode. This is because the set will not be compact once it passes the `NonEmpty()` function. The function will break it into many small non-empty chunks and each of these chunks will have to be evaluated separately. This will inevitably increase the duration of the query. In such cases, it is better to leave the original set intact, no matter its size. The engine will know how to run over it in optimized mode.

### Common mistakes and useful tips

If a second set in the `NonEmpty()` function is not provided, the expression is evaluated in the context of the current measure in the moment of evaluation and current members of attribute hierarchies, also in the time of evaluation. In other words, if you're defining a calculated measure and you forget to include a measure in the second set, the expression is evaluated for that same measure which leads to null, a default initial value of every measure. If you're simply evaluating the set on axis, it will be evaluated in the context of the current measure, the default measure in the cube or the one provided in the slicer. Again, perhaps not something you expected. In order to prevent these problems, **ALWAYS** include a measure in the second set.

`NonEmpty()` reduces sets, just like a few other functions, namely `Filter()` and `Existing()` do. But what's special about `NonEmpty()` is that it reduces sets extremely efficiently and quickly. Because of that, there are some rules about where to position `NonEmpty()` in calculations made by the composition of MDX functions (one function wrapping the other). If we're trying to detect multi-select, that is, multiple members in slicer, `NonEmpty()` should go inside with the `EXISTING` function/keyword outside. The reason is that although they both shrink sets efficiently, `NonEmpty()` works great if the set is intact. `EXISTING` is not affected by the order of members or compactness of the set. Therefore, `NonEmpty()` should be applied earlier.

You may get **System.OutOfMemory** errors if you use the `CrossJoin()` operation on many large hierarchies because the cross-join generates a cartesian product of those hierarchies. In that case, consider using `NonEmpty()` to reduce the space to a smaller subcube. Also, don't forget to group the hierarchies by their dimension inside the cross-join.

### NonEmpty() versus NON EMPTY

Both the `NonEmpty()` function and the `NON EMPTY` keyword reduce sets, but they do it in a different way.

The `NON EMPTY` keyword removes empty rows, columns, or both, depending on the axis on which that keyword is used in the query. Therefore, the `NON EMPTY` operator tries to push the evaluation of cells to an early stage whenever possible. This way the set on axis comes already reduced and the final result is faster.

Take a look at the initial query in this recipe, remove the `Filter( )` function, run the query, and notice how fast the results come although the multidimensional set again counts millions of tuples. The trick is that the `NON EMPTY` operator uses the set on the opposite axis, the columns, to reduce the set on rows. Therefore, it can be said that `NON EMPTY` is highly dependent on members on axes and their values in columns and rows.

Contrary to the `NON EMPTY` operator found only on axes, the `NonEmpty( )` function can be used anywhere in the query.

The `NonEmpty( )` function removes all the members from its first set where the value of one or more measures in the second set is empty. If no measure is specified, the function is evaluated in the context of the current member.

In other words, the `NonEmpty( )` function is highly dependent on members in the second set, the slicer, or the current coordinate in general.

## See also

*Using NON\_EMPTY\_BEHAVIOR.*

## Implementing logical AND on members from the same hierarchy

This recipe shows how to implement AND logic using members from the same hierarchy. It is recommended that you read the recipe *Implementing logical OR on members from different hierarchies* before continuing with this recipe because you'll have a much better understanding of the problem and solution.

## Getting ready

Start a new query in SSMS and check that you're working on the right database. Then type in the following query and execute it:

```
SELECT
    { [Measures].[Reseller Order Quantity],
      [Measures].[Reseller Order Count] } ON 0,
    NON EMPTY
    { [Date].[Month of Year].MEMBERS } ON 1
FROM
```

*Elementary MDX Techniques* —————

```
[Adventure Works]
WHERE
( [Promotion].[Promotion Type].&[New Product] )
```

The query displays all months in the year with the **New Product** promotion type.

	Reseller Order Quantity	Reseller Order Count
All Periods	2,323	116
July	513	24
August	811	39
September	999	53

If we replace the promotion in a slicer with another one, let's say **Excess Inventory** promotion type, we'll get one month less.

	Reseller Order Quantity	Reseller Order Count
All Periods	304	61
July	135	24
August	169	37

The idea is to have a single query which displays the result where both of these promotion types occur in the same month. In other words, we want to show the values for July and August.

We have several ways of doing it, but this recipe will focus on the slicer-subselect solution. Other solutions will be mentioned in further sections of this recipe.

Let's take one closer look at possible combinations and their values in BIDS before we start.

Drop Filter Fields Here		Drop Column Fields Here	
Month of Year ▾	Promotion Type ▾	Reseller Order Count	Reseller Order Quantity
⊖ June	Discontinued Product	53	635
	No Discount	347	15,650
	Volume Discount	53	2,302
	Total	349	18,587
⊖ July	Excess Inventory	24	135
	No Discount	201	12,318
	Seasonal Discount	66	1,172
	Volume Discount	65	4,055
	New Product	24	513
	Total	204	18,193
⊖ August	Excess Inventory	37	169
	No Discount	395	22,778
	Volume Discount	93	4,478
	New Product	39	811
	Total	399	28,236
⊖ September	No Discount	343	19,294
	Volume Discount	91	4,861
	New Product	53	999
	Total	347	25,154
⊖ October	No Discount	212	11,547
	Volume Discount	67	2,507
	Total	212	14,054
⊖ November	No Discount	401	19,555
	Volume Discount	62	2,067
	Total	401	21,622
⊖ December	No Discount	353	16,818
	Volume Discount	68	2,343
	Total	353	19,161
Grand Total		3,796	214,378

## How to do it...

Follow these steps to implement the AND logic between members of the same hierarchy:

1. Rearrange the query so that it includes a subselect part with those two members inside (see the code that follows).
2. Put the **Promotion Type** hierarchy on rows instead of the **Month of Year** hierarchy.
3. Wrap the member in slicer (member of **Promotion Type** hierarchy) with the `EXISTS( )` function so that it becomes its second argument.
4. Put the **Month of Year** level (not hierarchy!) as the first argument. Level has 3-part syntax; hierarchy has 2-part syntax. Don't omit the third part because it won't work.
5. Provide the measure group of interest, in this case it is "Reseller Sales".

6. Wrap another `EXISTS()` function around the previous one so that the previous becomes its first argument.
7. Add the second member of the **Promotion Type** hierarchy as the second argument of that function. Use the same measure group as before.
8. Run the following query:

```
SELECT
    { [Measures].[Reseller Order Quantity],
      [Measures].[Reseller Order Count] } ON 0,
    NON EMPTY
    { [Promotion].[Promotion Type].MEMBERS } ON 1
FROM
(
    SELECT
        { [Promotion].[Promotion Type].&[Excess Inventory],
          [Promotion].[Promotion Type].&[New Product] } ON 0
    FROM
        [Adventure Works]
)
WHERE
(
    Exists(
        Exists( { [Date].[Month of Year].[Month of Year].MEMBERS },
            { [Promotion].[Promotion Type].&[Excess Inventory] },
            "Reseller Sales"
        ),
        { [Promotion].[Promotion Type].&[New Product] },
        "Reseller Sales"
    )
)
```

9. The result of the query shows the aggregate for July and August, months where both promotion types occur:

	Reseller Order Quantity	Reseller Order Count
All Promotions	1,628	124
Excess Inventory	304	61
New Product	1,324	63

10. Compare these results with the ones in the previous image (showing combination of promotion types and months) and you'll notice the aggregates match the sum of individual values.

## How it works...

In order to perform AND logic on the same hierarchy, we must cascade the conditions. The inner set returns all the months that have the **Excess Inventory** promotion type (that's 3 of them as seen on the initial screenshot). The outer set restricts the inner set even more by filtering out all months which don't have the other promotion type as well. That leaves only two months, **July** and **August**.

Since there is no MDX expression that would work and return the AND result using those two members only, we must use another granularity as the base of the report. In this case, months that survive the combined restriction are months for which the result will be returned.

It is totally up to us to decide which hierarchy and which level we want to use as a new granularity as long as we adhere to some common sense rules.

First, the relation between new hierarchy's level members and the members for slicing should be many-to-many. This is always so in case of different dimensions (**Promotion** and **Date**), a case covered in this example. In case of the same dimension, the solution will work only for a single member that is related to both members in the AND operation. Whether that will be something other than the **All** member depends on the hierarchy and members selected for the AND operation. For example, two promotion types used in this example share only one ancestor – the **All** member, which can be verified in the **Promotions** user hierarchy of that dimension.

Second, the whole idea has to be valid. If we have two promotions and they occur on order items, then it's fine to use orders for granularity in slicer. However, if we have two resellers and they occur on orders, then this time it is not OK to use orders because there will never be an intersection of two resellers on the same order. We must use something else, a higher granularity like a day, promotion, or similar where two resellers can exist together. That's the basic idea.

Which hierarchy to use? That usually becomes quite obvious once we ask ourselves the question behind the report. For example, the last query, as seen in the previous screenshot, returned the two promotion types we started with this recipe, promotion types that come **together on a monthly basis**. Two things are important here: **together** and **basis**. The term "together" represents the AND logic. The term "monthly basis" is in fact the new granularity for the report (that which goes in slicer).

That explains the slicer part of the solution. What about the subselect part? Why is it there?

The subselect part serves the purpose of adjusting the results. Without it, we would get the wrong total. Let me explain this in more detail.

If you remove the subselect part of the query and execute it again, it will return the result displayed in the following image.

	Reseller Order Quantity	Reseller Order Count
All Promotions	46,429	603
Excess Inventory	304	61
New Product	1,324	63
No Discount	35,096	596
Seasonal Discount	1,172	66
Volume Discount	8,533	158

The numbers on this image match the aggregated values displayed in one of the previous images that show the combination of promotion types and months, the highlighted part of it. Those months were now in the slicer; hence the result is aggregated by them (603 orders = 204 in July + 399 in August and so on for every cell).

The query returned all promotion types because nothing limited them in the query. The slicer effectively has months, not promotion types, right?

There are two things we can do to correct it, that is, to display the result for those two hierarchies only. One is to put them in slicer so that they cross-join with the existing slicer. The other is to put them in subselect. I prefer the second option because that way we can still have them on an axis of the query. Otherwise we have a conflict with the slicer (a hierarchy cannot appear in slicer and on an axis, but it can appear in subselect and on an axis). That's why the subselect was used.

The subselect, as seen before, limits the promotion types that appear on an axis and adjusts their total so that it becomes the visual total for those two members. This is exactly what we need, the value for individual promotion types and their correct total.

To conclude, in order to implement AND logic, we have done two things. First, we have established a new granularity in slicer. Second, we used the subselect to adjust the total.

### There's more...

This is not the only way to implement AND logic. We can do it on axis as well. In that case all we have to do is put a construct from the slicer on rows and leave the subselect as is.

## Where to put what?

Based on a request, the AND logic can be implemented on rows or in slicer. If there's a request to hide the hierarchy for which we're applying the AND logic, we should put the corresponding MDX expression in slicer. On the other hand, if there's an explicit request to show members on rows, we must put the construct on rows. There we can cross-join it with additional hierarchies if required.

## A very complex scenario

In case of a more complex scenario where 3 different hierarchies need to be combined, we can apply the same solution, which in a general case should have N cascades in slicer and N members in subselect. The N is the number of members from the same hierarchy.

In case we need to combine many members using AND logic, some of them originating from different hierarchies and some from the same, the solution becomes very complex.

You are advised to watch for the order in cascades and dimensionality of the potential tuples.

## See also

A recipe with similar theme is *Implementing logical OR on members from different hierarchies*.



# 2

## Working with Time

In this chapter, we will cover:

- ▶ Calculating the YTD (Year-To-Date) value
- ▶ Calculating the YoY (Year-over-Year) growth (parallel periods)
- ▶ Calculating moving averages
- ▶ Finding the last date with data
- ▶ Getting values on the last date with data
- ▶ Hiding calculation values on future dates
- ▶ Calculating today's date using the string functions
- ▶ Calculating today's date using the MemberValue function
- ▶ Calculating today's date using an attribute hierarchy
- ▶ Calculating the difference between two dates
- ▶ Calculating the difference between two times
- ▶ Calculating parallel periods for multiple dates in a set
- ▶ Calculating parallel periods for multiple dates in slicer

### Introduction

Time handling features are an important part of every BI system. Programming languages, database systems, they all incorporate various time-related functions and Microsoft SQL Server Analysis Services (SSAS) is no exception there. In fact, that's one of its main strengths.

The MDX language has various time-related functions designed to work with a special type of dimension called the **Time** and its typed attributes. While it's true that some of those functions work with any type of dimension, their usefulness is most obvious when applied to time-type dimensions. An additional prerequisite is the existence of multi-level hierarchies, also known as user hierarchies, in which types of levels must be set correctly or some of the time-related functions will either give false results or will not work at all.

Because of the reasons described above and the fact that almost every cube will have one or more time dimensions, we've decided to dedicate a whole chapter to this topic, that is, for time calculations. In this chapter we're dealing with typical operations such as year-to-date calculations, running totals and jumping from one period to another. We go in details with each operation, explaining known and less known variants and pitfalls.

We then transition to two recipes covering the thin line that separates dates with data from those that are empty. Further on, we explain how to prevent calculations from having values after a certain point in time. In most BI projects, that often means yesterday's or today's date, so we're continuing with a set of recipes showing various ways of calculating that often-needed date.

The chapter ends with a two recipes explaining how to calculate the parallel period for a set of members and how to measure date and time spans.

## Calculating the YTD (Year-To-Date) value

In this recipe we will look at how to calculate the YTD value of another measure, that is, the accumulated value of all dates in a year up to the current member of that date dimension. In addition to that, we'll cover common exceptions and problems.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In order for this type of calculation to work, we need a dimension marked as **Time** in the **Type** property in **Dimension Structure** tab of **BIDS**. That should not be a problem because almost every database contains at least one such dimension and **Adventure Works** is no exception here. In this example, we're going to use the **Date** dimension.

Here's the query we'll start from:

```
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Date].[Calendar Weeks].[Calendar Week].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Once executed, the above query returns reseller sales values for every week in the database.

## How to do it...

Follow these steps to create a calculated measure with YTD calculation:

1. Add the WITH block of the query.
2. Create a new calculated measure within the WITH block and name it **Reseller Sales YTD**.
3. The new measure should return the sum of the measure Reseller Sales Amount using the YTD( ) function and the current date member of the hierarchy of interest.
4. Add the new measure on axis 0 and execute the complete query:

```
WITH
MEMBER [Measures].[Reseller Sales YTD] AS
    Sum( YTD( [Date].[Calendar Weeks].CurrentMember ),
        [Measures].[Reseller Sales Amount] )
SELECT
    { [Measures].[Reseller Sales Amount],
      [Measures].[Reseller Sales YTD] } ON 0,
    { [Date].[Calendar Weeks].[Calendar Week].MEMBERS } ON 1
FROM
    [Adventure Works]
```

5. The result will include the second column, the one with YTD values. Notice how the values in the second column increase over time:

	Reseller Sales Amount	Reseller Sales YTD
Week 27 CY 2005	\$489,328.58	\$489,328.58
Week 28 CY 2005	(null)	\$489,328.58
Week 29 CY 2005	(null)	\$489,328.58
Week 30 CY 2005	(null)	\$489,328.58
Week 31 CY 2005	\$1,538,408.31	\$2,027,736.89
Week 32 CY 2005	(null)	\$2,027,736.89
Week 33 CY 2005	(null)	\$2,027,736.89
Week 34 CY 2005	(null)	\$2,027,736.89
Week 35 CY 2005	\$1,165,897.08	\$3,193,633.97
Week 36 CY 2005	(null)	\$3,193,633.97
Week 37 CY 2005	(null)	\$3,193,633.97
Week 38 CY 2005	(null)	\$3,193,633.97
Week 39 CY 2005	(null)	\$3,193,633.97
Week 40 CY 2005	\$844,721.00	\$4,038,354.97

## How it works...

The `YTD()` function returns the set of members from the specified date hierarchy starting from the first date of the year and ending with the specified member. The first date of the year is calculated according to the level marked as **Year** type in that hierarchy. For example, the `YTD()` value for the member **Week 35 CY 2005** is a set of members starting from **Week 27 CY 2005** (normally it would have been Week 1 CY 2005 but the dimension starts with Week 27 CY 2005, that's its first member) up to that member because the upper level containing years is of the **Year** type.

The set is then summed up using the `SUM()` function and the **Reseller Sales Amount** measure. If we scroll down, we'll see that the cumulative sum resets every year which means that `YTD()` works as expected.

In this example we used the most common aggregation function, `SUM()`, in order to aggregate the values of the measure throughout the calculated set. `SUM()` was used because the aggregation type of the **Reseller Sales Amount** measure is **Sum**. Alternatively, we could have used the `Aggregate()` function instead. More information about that function can be found later in this recipe.

## There's more...

Sometimes it is necessary to create a single calculation that will work for any user hierarchy of the date dimension. In that case, the solution is to prepare several `YTD()` functions, each using a different hierarchy, cross-join them, and then aggregate that set using a proper aggregation function (Sum, Aggregate, and so on). However, bear in mind that this will only work if all user hierarchies used in the expression share the same year level. In other words, that there is no offset in years among them (like it exists between fiscal and calendar hierarchies in Adventure Works cube).

Why does it have to be so? Because the crossjoin is producing the set intersection of members on those hierarchies. Sets are generated relative to the position when the year starts. If there is offset in years, it is possible that sets won't have an intersection. In that case, the result will be an empty space. This was just a helpful tip, now let's continue with working examples.

Here's an example that works for both monthly and weekly hierarchies:

```

WITH
MEMBER [Measures].[Reseller Sales YTD] AS
    Sum( YTD( [Date].[Calendar Weeks].CurrentMember ) *
        YTD( [Date].[Calendar].CurrentMember ),
        [Measures].[Reseller Sales Amount] )
SELECT
    { [Measures].[Reseller Sales Amount],
      [Measures].[Reseller Sales YTD] } ON 0,
    { [Date].[Calendar Weeks].[Calendar Week].MEMBERS } ON 1
FROM
    [Adventure Works]

```

If we replace `[Date].[Calendar Weeks].[Calendar Week].MEMBERS` with `[Date].[Calendar].[Month].MEMBERS`, the calculation will continue to work. Without the cross-join part, that wouldn't be the case. Try it in order to see for yourself! Just be aware that if you slice by additional attribute hierarchies, the calculation might become wrong.

In short, there are many obstacles in getting the time-based calculation right. Partially it depends on the design of the time dimension (which attributes exists, which are made hidden, how are the relations defined, and so on), partially on the complexity of the calculations provided and their ability to handle various scenarios. A better place to define time-based calculation is the MDX script. There, we can define scoped assignments, but that's a separate topic which will be covered later, in *Chapter 6*. The title of the recipe is *Using utility dimension to implement time-based calculations*.

In the meantime, here are some articles related to that topic:

<http://tinyurl.com/MoshaDateCalcs>

<http://tinyurl.com/DateToolDim>

### Inception-To-Date calculation

A similar calculation is the *Inception-To-Date* calculation in which we're calculating the sum of all dates up to the current member, that is, we do not perform a reset in the beginning of every year. In that case, the `YTD()` part of the expression should be replaced with this:

Null : [Date].[Calendar Weeks].CurrentMember

### What to be careful about

The argument of the `YTD()` function is optional. When not specified, the first dimension of the **Time** type in the measure group is used. More precisely, the current member of the first user hierarchy with a level of type **Years**.

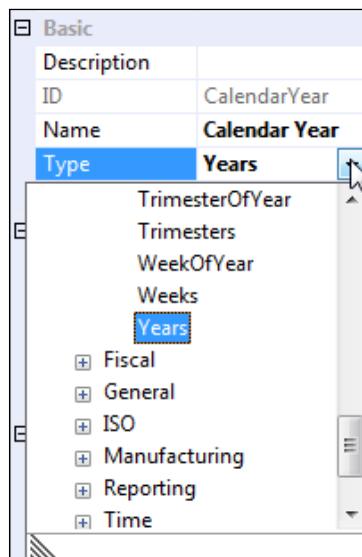
This is quite convenient in case of a simple **Date** dimension; a dimension with a single user hierarchy. In case of multiple hierarchies or a role-playing dimension, the `YTD()` function might not work if we forget to specify the hierarchy for which we expect it to work.

This can be easily verified. Omit the `[Date].[Calendar Weeks].CurrentMember` part in the initial query and see that both columns return the same values. The `YTD()` function is not working anymore.

Therefore, it is best to always use the argument in the `YTD()` function.

### Common problems and how to avoid them

In our example we used the `[Date].[Calendar Weeks]` user hierarchy. That hierarchy has the level **Calendar Year** created from the same attribute. The type of attribute is **Years**, which can be verified in the **Properties** pane of **BIDS**.



However, the **Date** dimension in the **Adventure Works** cube has fiscal attributes and user hierarchies built from them as well. The fiscal hierarchy equivalent to `[Date].[Calendar Weeks]` hierarchy is the `[Date].[Fiscal Weeks]` hierarchy. There, the top level is named **Fiscal Year**, created from the same attribute. This time, the type of the attribute is **FiscalYear**, not **Year**. If we exchange those two hierarchies in our example query, the `YTD()` function will not work on the new hierarchy, it will return an error:

	Reseller Sales Amount	Reseller Sales YTD
Week 1 FY 2006	\$489,328.58	#Error
Week 2 FY 2006	(null)	#Error
Week 3 FY 2006	(null)	#Error
Week 4 FY 2006	(null)	#Error
Week 5 FY 2006	(\$1,000,000.01)	#Error
Week 6 FY 2006	(null)	#Error
Week 7 FY 2006	(null)	#Error
Week 8 FY 2006	(null)	#Error

The name of the solution is – the `PeriodsToDate()` function!

`YTD()` is in fact a short version of the `PeriodsToDate()` function which works only if the **Year** type level is specified in a user hierarchy. When it is not so (that is some BI developers tend to forget to set it up correctly or in case the level is defined as, let's say, **Fiscal Year** like in this test), we can use the `PeriodsToDate()` function as follows:

```
MEMBER [Measures].[Reseller Sales YTD] AS
    Sum( PeriodsToDate( [Date].[Fiscal Weeks].[Fiscal Year],
                        [Date].[Fiscal Weeks].CurrentMember ),
        [Measures].[Reseller Sales Amount] )
```

`PeriodsToDate()` might therefore be used as a safer variant of the `YTD()` function.

### YTD() and future dates

It's worth noting that the value returned by a SUM-YTD combination is never empty once a value is encountered in a particular year. Only the years with no values at all will remain completely blank for all their descendants.

This can cause problems for the descendants of the member that represents the current year (and future years as well). The **NON EMPTY** keyword will not be able to remove empty rows, meaning, we'll get YTD values in the future.

We might be tempted to use the **NON\_EMPTY\_BEHAVIOR** operator to solve this problem but it wouldn't help. Moreover, it would be completely wrong to use it, because it is only a hint to the engine which may or may not be used. It is not a mechanism for removing empty values, as explained in the previous chapter.

In short, we need to set some rows to null, those positioned after the member representing today's date. We'll cover the proper approach to this challenge in the recipe *Finding the last date with data* later in this chapter.

### The Aggregate() function

The `Aggregate()` function is a more general function than `Sum()` or other explicit aggregation functions like `Count()`, `Max()`, and `Min()`. The good thing about it is that you can use it on all aggregation measure types, **DistinctCount** included. Additionally, you should use it on calculated members created on non-measure dimensions, like in the **utility** type dimensions.

The bad thing about it is that you cannot use it on calculated measures. In other words, you cannot write `Aggregate()`, put a set inside as the first argument, a calculated measure as the second argument, and expect it to work. The reason for it is that calculated measures don't have an inherited aggregation type like regular measures do. Meaning that there is no property which says how the values in the calculated measure should be aggregated on higher dimension levels. Regular measures, on the other hand, do. There, you can choose among the **Sum**, **Max**, **Min**, and other aggregation types. Therefore, the generic `Aggregate()` function cannot work in the case of calculated measures. When you need to aggregate values on a calculated measure, you must be explicit about which aggregation you want to use (**Sum**, **Max**, **Min**, and so on) by wrapping the set and the calculated measure with one of those explicit aggregation functions.

### See also

For the reasons explained in the last section of this recipe, you should take a look at the recipe *Finding the last date with data*, also in this chapter.

## Calculating the YoY (Year-over-Year) growth (parallel periods)

This recipe explains how to calculate the value in a parallel period, the value for the same period in a previous year, previous quarter, or some other level in the date dimension. We're going to cover the most common scenario – calculating the value for the same period in the previous year, because most businesses have yearly cycles. Once we have that value, we can calculate how much the value in the current period has increased or decreased with respect to the parallel period's value.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Date** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Date].[Fiscal].[Month].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Once executed, the previous query returns the value of **Reseller Sales Amount** for all fiscal months.

## How to do it...

Follow these steps to create a calculated measure with YoY calculation:

1. Add the **WITH** block of the query.
2. Create a new calculated measure there and name it **Reseller Sales PP**.
3. The new measure should return the value of the measure **Reseller Sales Amount** measure using the `ParallelPeriod( )` function. In other words, the definition of the new measure should be as follows:

```
MEMBER [Measures].[Reseller Sales PP] As
  ( [Measures].[Reseller Sales Amount],
    ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
      [Date].[Fiscal].CurrentMember ) )
```

4. Specify the format string property of the new measure to match the format of the original measure. In this case that should be the currency format.
5. Create the second calculated measure and name it **Reseller Sales YoY %**.
6. The definition of that measure should be the ratio of the current member's value against the parallel period member's value. Be sure to handle potential division by zero errors (see the recipe *Handling division by zero errors* in Chapter 1).
7. Include both calculated measures on axis 0 and execute the query which should look like this:

```
WITH
MEMBER [Measures].[Reseller Sales PP] As
  ( [Measures].[Reseller Sales Amount],
    ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
      [Date].[Fiscal].CurrentMember ) )
  , FORMAT_STRING = 'Currency'
MEMBER [Measures].[Reseller Sales YoY %] As
  iif( [Measures].[Reseller Sales PP] = 0, null,
    ( [Measures].[Reseller Sales Amount] /
      [Measures].[Reseller Sales PP] ) )
  , FORMAT_STRING = 'Percent'
SELECT
  { [Measures].[Reseller Sales Amount],
    [Measures].[Reseller Sales PP],
    [Measures].[Reseller Sales YoY %] } ON 0,
  { [Date].[Fiscal].[Month].MEMBERS } ON 1
FROM
  [Adventure Works]
```

*Working with Time* —————

8. The result will include two additional columns, one with the PP values and the other with the YoY change. Notice how the values in the second column repeat over time and that YoY % ratio shows the growth over time.

	Reseller Sales Amount	Reseller Sales PP	Reseller Sales YoY %
July 2005	\$489,328.58	(null)	(null)
August 2005	\$1,538,408.31	(null)	(null)
September 2005	\$1,165,897.08	(null)	(null)
October 2005	\$844,721.00	(null)	(null)
November 2005	\$2,324,135.80	(null)	(null)
December 2005	\$1,702,944.54	(null)	(null)
January 2006	\$713,116.69	(null)	(null)
February 2006	\$1,900,788.93	(null)	(null)
March 2006	\$1,455,280.41	(null)	(null)
April 2006	\$882,899.94	(null)	(null)
May 2006	\$2,269,116.71	(null)	(null)
June 2006	\$1,001,803.77	(null)	(null)
July 2006	\$2,393,689.53	\$489,328.58	489.18%
August 2006	\$3,601,190.71	\$1,538,408.31	234.09%
September 2006	\$2,885,359.20	\$1,165,897.08	247.48%
October 2006	\$1,802,154.21	\$844,721.00	213.34%
November 2006	\$3,053,816.33	\$2,324,135.80	131.40%
December 2006	\$2,185,213.21	\$1,702,944.54	128.32%

**How it works...**

The `ParallelPeriod()` function has three arguments, all of them optional. The first argument indicates the level on which to look for that member's ancestor, typically the year level like in this example. The second argument indicates how many members to go back on the ancestor's level, typically one, as in this example. The last argument indicates the member for which the function is to be applied.

Given the right combination of arguments, the function returns a member that is in the same relative position as a specified member, under a new ancestor.

The value for the parallel period's member is obtained using a tuple which is formed with a measure and the new member. In our example, this represents the definition of the PP measure.

The growth is calculated as the ratio of the current member's value over the parallel period member's value, in other words, as a ratio of two measures. In our example, that was YoY % measure.

A small detail we had to take care of was using the correct format string for each measure in our example, otherwise we would get unformatted, hard-to-analyze values.

### There's more...

The `ParallelPeriod()` function is almost always used on date dimensions although it can be used on any type of dimension. For example, this query is perfectly valid:

```
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { ParallelPeriod( [Geography].[Geography].[Country],
                      2,
                      [Geography].[Geography].[State-Province]
                      .&[CA]&[US] ) } ON 1
FROM
    [Adventure Works]
```

The query returns **Hamburg** on rows, which is the 3rd state-province in the alphabetical list of states-provinces under **Germany**. **Germany** is two countries back from the **USA**, whose member **California**, used in this query, is the 3<sup>rd</sup> state-province underneath that country in the **Geography.Geography** user hierarchy. **UK**, one member back from the **USA**, has only one state-province: **England**. If we change the second argument to 1 instead, we'll get nothing on rows because there's no 3rd state-province under **UK**. Feel free to try it.

All arguments of the `ParallelPeriod()` function are optional. When not specified, the first dimension of type **Time** in the measure group is used, more precisely, the previous member of the current member's parent. This can lead to unexpected results as discussed in the previous recipe. Therefore, it is recommended that you use all arguments of the `ParallelPeriod()` function.

## Working with Time

---

In our example we used the [Date].[Fiscal] user hierarchy. That hierarchy has all 12 months in every year which is not the case with the [Date].[Calendar] user hierarchy where there's only 6 months in the first year. This can lead to strange results. For example, if you search-replace the word "Fiscal" with the word "Calendar" in the query we used in this recipe, you'll get this as the result:

	Internet Sales Amount	Internet Sales PP	Internet Sales YoY %
July 2005	\$473,388.16	(null)	(null)
August 2005	\$506,191.69	(null)	(null)
September 2005	\$473,943.03	(null)	(null)
October 2005	\$513,329.47	(null)	(null)
November 2005	\$543,993.41	(null)	(null)
December 2005	\$755,527.89	(null)	(null)
January 2006	\$596,746.56	\$473,388.16	126.06%
February 2006	\$550,816.69	\$506,191.69	108.82%
March 2006	\$644,135.20	\$473,943.03	135.91%
April 2006	\$663,692.29	\$513,329.47	129.29%
May 2006	\$673,556.20	\$543,993.41	123.82%
June 2006	\$676,763.65	\$755,527.89	89.57%
July 2006	\$500,365.16	(null)	(null)
August 2006	\$546,001.47	(null)	(null)
September 2006	\$350,466.99	(null)	(null)
October 2006	\$415,390.23	(null)	(null)
November 2006	\$335,095.09	(null)	(null)
December 2006	\$577,314.00	(null)	(null)

Notice how the values are incorrect for the year 2006. That's because `ParallelPeriod()` function is not a time-aware function, it merely does what it's designed for – taking the member that is in the same relative position. Gaps in your time dimension are another potential problem. Therefore, always make the complete date dimensions, with all 12 months in every year and all dates in them, not just working days or similar shortcuts. Remember, Analysis Services isn't doing date math. It's just navigating using the member's relative position. Therefore, make sure you have laid a good foundation for that.

However, that's not always possible. There's an offset of 6 months between fiscal and calendar years, meaning if you want both of them as date hierarchies, you have a problem – one of them will not have all of the months in the first year.

The solution is to test the current member in the calculation and to provide a special logic for the first year, fiscal or calendar; the one that doesn't have all months in it. This is most efficiently done with a scope statement in the MDX script.

Another problem in calculating the YoY value is leap years. One possible solution for that is presented in this blog article:

<http://tinyurl.com/LeapYears>

## See also

The `ParallelPeriod()` function operates on a single member. However, there are times when we will need to calculate the parallel period for a set of members. The recipes *Calculating parallel periods for multiple members in a set* and *Calculating parallel periods for multiple members in a slicer* deal with this more complex request.

## Calculating moving averages

The moving average, also known as the rolling average, is a statistical technique often used in events with unpredictable short-term fluctuations in order to smooth their curve and to visualize the pattern of behavior. In this recipe, we're going to look at how to calculate moving averages in MDX.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Date** hierarchy of the **Date** dimension. Here's the query we'll start from:

```
SELECT  
    { [Measures].[Internet Order Count] } ON 0,  
    { [Date].[Date].[Date].MEMBERS } ON 1  
FROM  
    [Adventure Works]
```

Execute it. The result shows the count of Internet orders for each date in the `Date.Date` attribute hierarchy. Our task is to calculate the simple moving average (SMA) for dates in the year 2006 based on the count of orders in the previous 30 days.

## How to do it...

Follow these steps to calculate moving averages:

1. Add the WHERE part of the query and put the year 2006 inside using any available hierarchy.
2. Add the WITH part and define a new calculated measure. Name it **SMA 30**.
3. Define that measure using the AVG() and LastPeriods() functions.
4. Test to see if you get a managed query similar to this. If so, execute it.

```
WITH
MEMBER [Measures].[SMA 30] AS
    Avg( LastPeriods( 30, [Date].[Date].CurrentMember ),
        [Measures].[Internet Order Count] )
SELECT
    { [Measures].[Internet Order Count],
        [Measures].[SMA 30] } ON 0,
    { [Date].[Date].[Date].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Calendar Year].&[2006] )
```

5. The second column in the result set will represent the simple moving average based on the last 30 days.

	Internet Order Count	SMA 30
January 1, 2006	5	8
January 2, 2006	4	7
January 3, 2006	8	7
January 4, 2006	5	7
January 5, 2006	3	7
January 6, 2006	6	7
January 7, 2006	4	7
January 8, 2006	8	7
January 9, 2006	5	7
January 10, 2006	4	7
January 11, 2006	9	7
January 12, 2006	7	7
January 13, 2006	3	7
January 14, 2006	9	7
January 15, 2006	7	7

## How it works...

The moving average is a calculation that uses the window of N items for which it calculates the statistical mean; the average value. The window starts with the first item and then progressively shifts to the next one until the whole set of items is passed.

The function that acts as the window is the `LastPeriods()` function. It takes N items, in this example, 30 dates. That set is then used to calculate the average orders using the `Avg()` function.

## There's more...

Another way of specifying what the `LastPeriods()` function does is to use a range of members. The last member of the range is usually the current member of the hierarchy on an axis. The first member is the member that is N-1 members before on the same level in that hierarchy. In other words, this is the equivalent syntax: `[Date].[Date].CurrentMember.Lag(11) : [Date].[Date].CurrentMember`

The modification of the scope of the window is very easy. For example, in case we need to calculate the moving average up to the previous member, we can use this syntax: `[Date].[Date].CurrentMember.Lag(12) : [Date].[Date].PrevMember`



The `LastPeriods()` function is not on the list of optimized functions on this web page: <http://tinyurl.com/Improved2008R2>. However, tests show no difference in duration with respect to its range alternative. Still, if you come across a situation where the `LastPeriods()` function performs slowly, try its range alternative.

Finally, in case we want to parameterize the expression (for example, to be used in SQL Server Reporting Services), these would be generic forms of the previous expressions:

```
[Date].[Date].CurrentMember.Lag( @span - @offset ) : [Date].[Date].CurrentMember.Lag( @offset )
```

and

```
LastPeriods( @span, [Date].[Date].CurrentMember.Lag( @offset ) ).
```

The `@span` parameter is a positive value which determines the size of the window. The `@offset` parameter determines how much the right side of the window is moved from the current member's position. This shift can be either a positive or negative value. The value of zero means there is no shift at all, the most common scenario.

Other ways to calculate the moving averages

The simple moving average is just one of many variants of calculating the moving averages. A good overview of a possible variant can be found in Wikipedia:

<http://tinyurl.com/WikiMovingAvg>

MDX examples of other variants of moving averages can be found in Moshav Pasumansky's blog article:

<http://tinyurl.com/MoshavMovingAvg>

### Moving averages and the future dates

It's worth noting that the value returned by the moving average calculation is not empty for dates in future because the window is looking backwards, so that there will be N values in the future.

This can cause problems manifested in the fact that the `NON EMPTY` keyword will not be able to remove empty rows.

We might be tempted to use `NON_EMPTY_BEHAVIOR` to solve this problem but it wouldn't help. Moreover, it would be completely wrong as explained in the previous chapter. We don't want to set all the empty rows to null, but only those positioned after the member representing today's date. We'll cover the proper approach to this challenge in the following recipes.

### See also

The recipe *Hiding calculation values on future dates* solves a problem with future dates and is therefore relevant to this recipe.

## Finding the last date with data

In this recipe we're going to learn how to find the last date with data for a particular combination of members in the cube. We'll start with a general calculation, not dependent on the time context and later show how to make it time-sensitive, if required so.

### Getting ready

Open Business Intelligence Development Studio (BIDS) and then open **Adventure Works DW 2008** solution. Double-click the **Date** dimension found in the **Solution Explorer**. Select the **Date** attribute and locate the property **ValueColumn** in the bottom of the **Properties** pane.

KeyColumns	Date.DateKey (Integer)
NameColumn	Date.SimpleDate (WChar)
ValueColumn	Date.Date (Date)
Source	Date.Date
TableID	Date
ColumnID	FullDateAlternateKey
DataType	Date
DataSize	0
NullProcessing	Automatic

There's a value in that property. Column **FullDateAlternateKey**, of the **Date** type, is specified as the **ValueColumn** of the key attribute property - the **Date** attribute. This check is important because without that property filled correctly, this recipe won't work.

Next, start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Date** hierarchy of the **Date** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Internet Order Count] } ON 0,
    { [Date].[Date].[Date].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Execute it, and then scroll down to the end. By scrolling up again, try to identify the last date with data. It should be the **July 31, 2008** date, as highlighted in the following image.

Internet Order Count
July 27, 2008
29
July 28, 2008
32
July 29, 2008
31
July 30, 2008
23
July 31, 2008
40
August 1, 2008
(null)
August 2, 2008
(null)
August 3, 2008
(null)
August 4, 2008
(null)
August 5, 2008
(null)
August 6, 2008
(null)
August 7, 2008
(null)

Now let's see how we can get this automatically, as the result of a calculation.

## How to do it...

Follow these steps to create a calculated measure that returns the last date with data:

1. Add the WITH part of the query.
2. Create a new calculated measure there and name it **Last date**.
3. Use the Max() function with Date attribute members as its first argument.
4. The second argument should be the MemberValue() function applied on the current member of **Date.Date.Date** hierarchy, but only if the value of the **Internet Order Count** measure is not empty or 0.
5. Add the Last Date measure on columns axis.
6. Put the **Promotion.Promotion** hierarchy on rows instead.
7. Run the query which, should look like this:

```
WITH
MEMBER [Measures].[Last date] AS
    Max( [Date].[Date].[Date].MEMBERS,
        iif( [Measures].[Internet Order Count] = 0,
            null,
            [Date].[Date].CurrentMember.MemberValue
        )
    )
SELECT
    { [Measures].[Internet Order Count],
        [Measures].[Last date] } ON 0,
    { [Promotion].[Promotion].MEMBERS } ON 1
FROM
    [Adventure Works]
```

8. The result will show the last date of Internet orders for each promotion as shown in the following image:

The screenshot shows a results grid from SSMS with three columns: 'Internet Order Count' and 'Last date' are visible, while 'Promotion Name' is the primary key. The data includes various promotional offers like 'All Promotions', 'No Discount', and several 'Volume Discount' types, along with specific events like 'Sport Helmet Discou...', 'Road-650 Overstock', and 'Mountain Tire Sale'. Some entries have null values for the last date.

	Internet Order Count	Last date
All Promotions	27,659	7/31/2008
No Discount	27,119	7/31/2008
Volume Discount 11t...	2,075	6/30/2008
Volume Discount 15t...	(null)	(null)
Volume Discount 25 t...	(null)	(null)
Volume Discount 41t...	(null)	(null)
Volume Discount ove...	(null)	(null)
Mountain-100 Cleara...	(null)	(null)
Sport Helmet Discou...	(null)	(null)
Road-650 Overstock	(null)	(null)
Mountain Tire Sale	(null)	(null)
Sport Helmet Discou...	(null)	(null)
LL Road Frame Sale	(null)	(null)
Touring-3000 Promoti...	20	9/28/2007
Touring-1000 Promoti...	13	9/23/2007
Half-Price Pedal Sale	(null)	(null)
Mountain-500 Silver ...	(null)	(null)

## How it works...

The **Date** dimension in the **Adventure Works DW 2008R2** database is designed in such way that we can conveniently use the `MemberValue()` function on the **Date** attribute in order to get the date value for each member in that hierarchy. This best practice act allows us to use that value inside the `Max()` function and hence, get the member with the highest value, the last date.

The other thing we must do is to limit the search only to dates with orders. The inner `iif()` statement which provides null for dates with no orders not only takes care of that, but it also makes the set sparse and therefore allows for block mode evaluation of the outer `Max()` function.

Finally, the result of the `MemberValue()` function applied on the highest date with orders, which is that same date only in the form of a typed value, is what gets returned in the end. The type of the value it returns in this case is the date type, that's why the calculated measure is nicely formatted without any intervention.

In case there was no **ValueColumn** property defined on the **Date.Date** attribute, we could use the `Name`, `Caption` or some other property to identify the last date or to use it in further calculations.

## There's more...

In the previous example, the `Max()` function was used on all dates in the **Date.Date** hierarchy. As such, it is relatively inflexible. This means that it won't react to other hierarchies of the same dimension on axes which would normally reduce that set of dates. In other words, there are situations when the expression has to be made **context-sensitive** so that it changes in respect to other hierarchies. How do we achieve that? Using the **EXISTING** operator in front of the set!

For example, let's run the following query:

```
WITH
MEMBER [Measures].[Last date] AS
    Max( [Date].[Date].[Date].MEMBERS,
        iif( [Measures].[Internet Order Count] = 0,
            null,
            [Date].[Date].CurrentMember.MemberValue
        )
    )
MEMBER [Measures].[Last existing date] AS
    Max( EXISTING [Date].[Date].[Date].MEMBERS,
        iif( [Measures].[Internet Order Count] = 0,
            null,
            [Date].[Date].CurrentMember.MemberValue
        )
    )
SELECT
    { [Measures].[Internet Order Count],
        [Measures].[Last date],
        [Measures].[Last existing date] } ON 0,
    { [Date].[Calendar Year].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Sales Territory].[Sales Territory Country].&[France] )
```

Now, values in the second column (the **Last date** measure) will all be the same – showing **07/29/2008**. On the other hand, values in the third column will differ, as seen in this image:

	Internet Order Count	Last date	Last existing date
All Periods	2,484	7/29/2008	7/29/2008
CY 2005	59	7/29/2008	12/28/2005
CY 2006	233	7/29/2008	12/31/2006
CY 2007	1,034	7/29/2008	12/31/2007
CY 2008	1,158	7/29/2008	7/29/2008
CY 2010	(null)	7/29/2008	(null)

Those two types of calculation represent different things and should be used in the right context. In other words, if there is a need to get the last date no matter what, then that's a variant without the EXISTING part. In all other cases the EXISTING keywords should be used.

One thing is important to remember: the usage of the EXISTING keyword slows down the performance of the query. That's the cost we have to pay for having flexible calculations.

## See also

The next two recipes, *Getting values on the last date with data* and *Hiding calculation values on future dates*, are relevant for this recipe because they show how to return the value of measures on the last date with data and how to fix calculations that return values on dates after the last date with the data in a cube.

## Getting values on the last date with data

In this recipe we're going to learn how to get the value of a measure on the last date with data. If you haven't read the previous recipe, do it before reading this one as this recipe continues where the previous recipe had stopped.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the simplified version of the query from the previous chapter, simplified in a sense that it has only one measure, the time-sensitive measure:

```

WITH
MEMBER [Measures].[Last existing date] AS
    Max( EXISTING [Date].[Date].[Date].MEMBERS,
        iif( [Measures].[Internet Order Count] = 0,
            null,
            [Date].[Date].CurrentMember.MemberValue
        )
    )
SELECT
    { [Measures].[Internet Order Count],
        [Measures].[Last existing date] } ON 0,
    { [Date].[Calendar Year].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Sales Territory].[Sales Territory Country].&[France]      )

```

The following image shows the result of query execution:

All Periods	Internet Order Count	Last existing date
All Periods	2,484	7/29/2008
CY 2005	59	12/28/2005
CY 2006	233	12/31/2006
CY 2007	1,034	12/31/2007
CY 2008	1,158	7/29/2008
CY 2010	(null)	(null)

Now, let's see how to get the values on those last dates with data.

## How to do it...

Follow these steps to get a measure's value on the last date with data:

1. Remove the `Last existing date` calculated measure from the `WITH` part of the query and from columns axis.
2. Define a new calculated measure and name it **Value N**.
3. The definition of this new measure should be a tuple with two members, members we will identify or build in the following steps.
4. Use the `Internet Order Count` measure as one of the members in the tuple.
5. The other part of the tuple should be an expression which in its inner part has the `NonEmpty()` function applied over members of the **Date.Date** hierarchy and the measure `Internet Order Count`.
6. Use the `EXISTING` operator in front of the `NonEmpty()` function.
7. Extract the last member of that set using the `Tail()` function.
8. Convert the resulting set into a member using the `Item()` function.
9. The final query should look like this:

```
WITH
MEMBER [Measures].[Value N] AS
  ( Tail( EXISTING
    NonEmpty( [Date].[Date].[Date].MEMBERS,
              [Measures].[Internet Order Count] ),
    1
  ).Item(0),
  [Measures].[Internet Order Count] )
SELECT
```

```

{ [Measures].[Internet Order Count],
  [Measures].[Value N] } ON 0,
{ [Date].[Calendar Year].MEMBERS } ON 1
FROM
  [Adventure Works]
WHERE
  ( [Sales Territory].[Sales Territory Country].&[France] )

```

10. Once executed, the result will show values of the measure Internet Order Count on the last dates with data, as visible in the following image:

	Internet Order Count	Value N
All Periods	2,484	1
CY 2005	59	1
CY 2006	233	1
CY 2007	1,034	11
CY 2008	1,158	1
CY 2010	(null)	(null)

### How it works...

The value of the last date with data is calculated from the scratch. First, we have isolated dates with orders using the `NonEmpty()` function. Then we have applied the `EXISTING` operator in order to get dates relevant to the existing context. The `Tail()` function was used to isolate the last date in that set, while the `Item()` function converts that one-member set into a member.

Once we have the last date with data, we can use it inside the tuple with the measure of our interest, in this case, the Internet Order Count measure, to get the value in that coordinate.

### There's more...

The non-empty variant of calculating the last date with data is not the only one. We have several more option here, all of which make use of the `Last existing date` calculated measure. Let's see which ones they are.

One approach might have been to use the `Filter()` function on all dates in order to find the one that has the same `MemberValue` as calculated in the measure `Last existing date`. However, that is the worst approach, the slowest one, and it shouldn't be used. The reason why it's slow is because the `Filter()` function needs to iterate over the complete set of dates in every cell in order to isolate a single date, the last one with data.

Another problem with that approach is that we already know which date we need, the measure `Last existing date` returns that, so there's really no need to iterate, right?

What it actually returns is a value, not a member, and that's the problem. We cannot simply put that value in the tuple; we need a member because tuples are formed from members, not values.

The other approach is based on the idea that we might be able to convert that value into a member. Conversion can be done using the `StrToMember()` function, with the **CONSTRAINED** flag provided in it to enable faster execution of that function. Here are two expressions that work for the **Date** dimension in the **Adventure Works** - they return the same result visible in the previous image:

```
MEMBER [Measures].[Value SN] AS
    iif( IsEmpty([Measures].[Last existing date]), null,
        ( StrToMember( '[Date].[Date].[' +
            Format( [Measures].[Last existing date],
                "MMMM dd, yyyy" ) + ']', CONSTRAINED ),
            [Measures].[Internet Order Count] ) )

MEMBER [Measures].[Value SK] AS
    iif( IsEmpty([Measures].[Last existing date]), null,
        ( StrToMember( '[Date].[Date].&[' +
            Format( [Measures].[Last existing date],
                "yyyyMMdd" ) + ']', CONSTRAINED ),
            [Measures].[Internet Order Count] ) )
```

The first calculated measure (with the `SN` suffix) builds the member using its name, the second one (with the `SK` suffix) using its key. The "`S`" stands for string-based solution; the "`N`" in the first solution in this recipe stands for nonempty-based solution, in case you wondered.

Both expressions make use of the `Format()` function and apply the appropriate format for the date returned by the `Last existing date` calculated measure.

Since there might not be any date in a particular context, the `iif()` function is used to return null in those situations, otherwise the appropriate tuple is formed and its value is returned as the result of the expression.

### How to know which format to use?

If you drag and drop any member from the **Date.Date.Date** level in the **Query Editor**, you will see its unique name. In case of **Adventure Works** cube, the name will look like this:

```
[Date].[Date].&[20050701]
```

Now, to build a string, all you have to do is replace the part with the day, month, and year with the appropriate tokens in the format string. This web page might help in that:

<http://tinyurl.com/FormatDate>

For the name variant you have to analyze the name of the member in the cube structure and match it with appropriate tokens in the format string. Look how we made it in this example and try to do the same in your projects.

Here's another idea. You can define a separate measure which builds the unique name and use it in the query for debugging purposes. In other words, break the expression into smaller chunks and test each separately. Once you see that it's fine, use it in the tuple.

### Optimizing time-non-sensitive calculation

Remember that in case you don't need a time-sensitive calculation, a calculation which evaluates the dates in each context (and hence is naturally slower because of that), you can use the same expressions provided in this recipe, just remove the EXISTING operator in them.

#### See also

The next recipe, *Hiding calculation values on future dates*, is relevant for this recipe because it shows how to fix calculations that return values on dates after the last date with the data in a cube.

## Hiding calculation values on future dates

Time-based calculations are often written either relative to a single member in the Date dimension or relative to a range of members in it. The first scenario includes calculations that return the value of today, yesterday, this month, this year, and so on. The second scenario includes calculations we've already covered in this chapter such as year-to-date calculation and moving averages calculation.

The problem with the latter type of calculations is that they are not empty (the proper term is *not null*) where they should be - in the *future*. The term *future* should be interpreted here as a set of consecutive date members in the end of the **Date** hierarchy for which no regular cube measure has data.

The problem manifests in unnecessary dates being present in the result in spite of **NON EMPTY** keyword being used in the query. Naturally, the **NON EMPTY** keyword can't help because the values of such calculated measures are not empty.

This recipe shows how to solve that problem.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on **Script View** and position the cursor immediately after the **CALCULATE** command.

In this example, we're going to continue the first recipe in this chapter. We're going to use the YTD version of the **Reseller Sales Amount** measure and we're going to fix the problem that occurs with it – it shows values for future dates. Let's repeat its definition one more time here, this time by defining it in the MDX script:

```
CREATE MEMBER CurrentCube.[Measures].[Reseller Sales YTD] AS  
    Sum( YTD( [Date].[Calendar].CurrentMember ) *  
        YTD( [Date].[Calendar Weeks].CurrentMember ),  
        [Measures].[Reseller Sales Amount] );
```

Deploy the solution and then go to the **Cube Browser** tab to verify the measure's behavior. Create a report with two measures, years and months, from the **Calendar** user hierarchy. The following image shows how the report should look like.

Drop Filter Fields Here		Drop Column Fields Here	
Calendar Year ▾	Month	Reseller Sales Amount	Reseller Sales YTD
[+] CY 2005		\$8,065,435.31	\$8,065,435.31
[+] CY 2006		\$24,144,429.65	\$24,144,429.65
[+] CY 2007		\$32,202,669.43	\$32,202,669.43
[+] CY 2008	[+] January 2008	\$1,662,547.32	\$1,662,547.32
	[+] February 2008	\$2,700,766.80	\$4,363,314.13
	[+] March 2008	\$2,739,370.98	\$7,102,685.11
	[+] April 2008	\$2,204,623.41	\$9,307,308.52
	[+] May 2008	\$3,315,275.00	\$12,622,583.53
	[+] June 2008	\$3,415,479.07	\$16,038,062.60
	[+] July 2008		\$16,038,062.60
	[+] August 2008		\$16,038,062.60
	Total	\$16,038,062.60	\$16,038,062.60
[+] CY 2010			
Grand Total		\$80,450,596.98	

The previous image shows that the YTD version of the measure displays values for months after July 2008, the last month with the data (we just learned that about July 2008 in two previous recipes in this chapter). And we have decided to correct that.

## How to do it...

Follow these steps to implement the scope for hiding calculation values on future dates:

1. Go to the **Calculations** tab and position the cursor at the end of the MDX script.
2. Add several `Scope( )` statements in your MDX script using all **attribute hierarchies** that have chronological character (hierarchies in which the number of members is not fixed, but is expected to grow in time):

```

Scope( { Tail( NonEmpty( [Date].[Date].[Date].MEMBERS,
                           [Measures].[Reseller Sales Amount] ),
              1 ).Item(0).NextMember : null } );
This = null;
End Scope;

Scope( { Tail( NonEmpty( [Date].[Month Name]
                           .[Month Name].MEMBERS,
                           [Measures].[Reseller Sales Amount] ),
              1 ).Item(0).NextMember : null } );
This = null;
End Scope;

Scope( { Tail( NonEmpty( [Date].[Calendar Quarter]
                           .[Calendar Quarter].MEMBERS,
                           [Measures].[Reseller Sales Amount] ),
              1 ).Item(0).NextMember : null } );
This = null;
End Scope;

Scope( { Tail( NonEmpty( [Date].[Calendar Semester]
                           .[Calendar Semester].MEMBERS,
                           [Measures].[Reseller Sales Amount] ),
              1 ).Item(0).NextMember : null } );
This = null;
End Scope;

Scope( { Tail( NonEmpty( [Date].[Calendar Year]
                           .[Calendar Year].MEMBERS,
                           [Measures].[Reseller Sales Amount] ),
              1 ).Item(0).NextMember : null } );
This = null;
End Scope;

Scope( { Tail( NonEmpty( [Date].[Calendar Week]
                           .[Calendar Week].MEMBERS,
                           [Measures].[Reseller Sales Amount] ),
              1 ).Item(0).NextMember : null } );
This = null;
End Scope;

```

3. Deploy the solution.
4. Return to the **Cube Browser** tab and reconnect to the cube. The result will be refreshed and no values will be in future dates, as highlighted in the following image:

Drop Filter Fields Here		Drop Column Fields Here	
<b>Calendar Year ▾   Month</b>		Reseller Sales Amount	Reseller Sales YTD
[+]	CY 2005	\$8,065,435.31	\$8,065,435.31
[+]	CY 2006	\$24,144,429.65	\$24,144,429.65
[+]	CY 2007	\$32,202,669.43	\$32,202,669.43
[+]	CY 2008	\$1,662,547.32	\$1,662,547.32
[+]	January 2008	\$2,700,766.80	\$4,363,314.13
[+]	February 2008	\$2,739,370.98	\$7,102,685.11
[+]	March 2008	\$2,204,623.41	\$9,307,308.52
[+]	April 2008	\$3,315,275.00	\$12,622,583.53
[+]	May 2008	\$3,415,479.07	\$16,038,062.60
[+]	June 2008		
[+]	July 2008		
[+]	August 2008		
	Total	\$16,038,062.60	\$16,038,062.60
[+]	CY 2010		
	Grand Total	\$80,450,596.98	

## How it works...

The best way of hiding values for future dates is to use the scope assignments in MDX script. There, you can define a scope for the date attribute of your time dimension and later multiply that scope by the number of chronological attribute hierarchies in your time dimension. Let's explain one more time what a chronological hierarchy is, this time with an example.

The `[Data].[Date]` attribute hierarchy is a chronological hierarchy, `[Date].[Day in Week]` is not. The difference between the two is that the first hierarchy grows in time and new members will appear. Each member represents a unique point in time, which never repeats.

Everything's different for the second hierarchy. Days in the week will not grow in time, they are fixed. They don't represent unique points in time; they tend to repeat every 7 days. Finally, parallel periods, year-to-date, and similar calculations don't make sense with such hierarchies.

Now, back to the explanation.

There are 6 attribute hierarchies that make sense to be used in these scopes. Some of them are hidden, but their names can be seen in the **Dimension Editor**, when you open the **Date** dimension. One more thing about chronological hierarchies and how you can recognize them – they form user hierarchies, at least here in Adventure Works cube (but often in other projects).

Each scope has a range of members starting from the member immediately next to the last date with data all the way up to the very last member in each hierarchy. That's exactly what is needed – when the last member with data is found, everything after it can be considered the future and therefore overwritten with null. Ways of calculating the last date with data are explained in the previous two recipes. Here we're using the time-independent variant of the last date with data calculation. We don't want the last date to be dependent on the query context; we simply need the last date.

### There's more...

Returning anything else than null would be inefficient performance-wise. The ability that SSAS engine runs calculation in block mode largely depends on their sparsity. Nulls are what makes the calculation sparse; any other value makes it dense. That's the reason why we always have to use nulls for the part of the cube we don't care.

## Calculating today's date using the string functions

Calculating today's date is one of those problems every BI developer encounters sooner or later. It is also a repeatedly asked question on Internet forums, probably because books don't cover this topic at all or at least not with concrete examples.

The reason behind this might be in the very nature of the calculation – the result of the query executed by a reader on a particular date must inevitably differ from the result shown in the book. An additional reason might be that the implementation of the Date dimension in Adventure Works surely differs in some cases from the Date dimension in a concrete project that the reader is working on. There are just too many options with the formatting of dates.

This, however, should not prevent us from explaining **the concept** using the appropriate examples which, in turn, should be generic enough so that you can apply them in any SSAS database. In fact, we're dealing with this intriguing topic in three recipes!

This recipe demonstrates the most intuitive technique of doing it, that is, generating today's date as a string and then converting that string into a dimension member. The other two recipes, following immediately after this one, show some not-so-intuitive but nevertheless perfectly valid and often better ways of performing the same thing. You are advised to read them all in order to get an overview of possibilities.

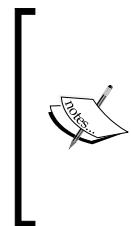
### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Execute this query:

```
SELECT  
    { } ON 0,  
    { [Date].[Calendar].[Date].MEMBERS } ON 1  
FROM  
    [Adventure Works]
```

In this example we're using the **Calendar** hierarchy of the **Date** dimension. The result contains all dates on rows and nothing on columns. We've explained this type of query in the first recipe of the first chapter.



Building the correct string for Adventure Works cube is not a problem. Building the correct string for any database is almost impossible. The string has to be precise or it won't work. That's why we'll use the step-by-step approach here. We'll also highlight the important points for each step. Additionally, a step-by-step way is much easier to debug by visualizing which step is not calculating correctly. Hence achieving the correct string for today's date becomes faster.

If you scroll the results down, you'll notice several things. First, there's no today's date there. The Adventure Works cube contains only the dates up to and including November 2010. Moreover, they are not consecutive. There are many gaps in the last few years which is something you should definitely avoid in your projects. As the last complete year is 2007, we're going to build the current date for that year, here as well as in the next two recipes. We'll also include a switch for shifting years which will allow you to apply the same recipe in any database.

## How to do it...

Follow these steps to calculate today's date using the string functions:

1. Add the **WITH** part in the query.
2. Define a new calculated measure using the VBA function `Now()` and try to match the name of date members. In case of Adventure Works cube, the required definition is this: `Format(Now(), 'MMMM dd, yyyy')`
3. Name the measure **Caption for Today** and include it in the query.
4. Execute the query and see how you matched the measure's value with the name of each member on rows. If it doesn't, try to fix the format to fit your regional settings. This link provides detailed information about what each token represents:  
<http://tinyurl.com/FormatDate>
5. Add the second calculated measure with the following definition:  
`[Date].[Calendar].CurrentMember.UniqueName`

6. Name it **Member's Unique Name** and include it in the query.
7. Execute the query and notice the part of the measure's value that is not constant and that is changed in each row. Try to detect what part is built from in terms of years, months, and dates or how it relates in general to dates on rows.
8. Add the third calculated measure by formatting the result of the Now( ) function based on the discoveries you made in the previous step. In other words, this: Format(Now(), 'yyyyMMdd'), because unique names are built using the "yyyyMMdd" sequence.
9. Name it **Key for Today** and include it in the query.
10. Execute the query. The value should repeat in every row giving you the current date formatted as "yyyyMMdd".
11. In case of a database with no today's date in the **Date** dimension, such is the case here, add the fourth calculated measure. This measure should replace the year part with another year, already present in the **Date** dimension. That should be the year with all dates; otherwise you'll get an error in subsequent steps of this recipe. The definition of that measure in case of the **Adventure Works** cube is this: '2007' + Right([Measures].[Key for Today], 4).
12. Name it **Key for Today (AW)** and include it in the query.
13. Execute the query and find the row where that same key can be found as a part of the measure **Member's Unique Name**. If you can, that means you're doing everything fine so far.
14. Add the fifth calculated measure, name it **Today (string)** and define it by concatenating the fixed part of the unique name with the variable part defined as the measure **Key for Today** or **Key for Today (AW)**, depending on whether you have or don't have today's date in your Date dimension. In this example we'll use the latter because Adventure Works doesn't have it. This is the definition of the measure: '[Date].[Calendar].[Date].&[ ' + [Measures].[Key for Today (AW)] + ' ]' .
15. Include that fifth measure as well in the query and execute it. In that particular row mentioned above the value of this measure should match completely to the value of the measure **Member's Unique Name**.
16. Add a calculated set. Name it **Today** and define it using the StrToMember( ) function with the CONSTRAINED flag.
17. Execute the query. The set you just made represents today's date and can be used in all further calculations you make.

## How it works...

The result of your query should look like this. It may be slightly different, but should follow the same form and function. The match between the **Today (string)** measure and the **Member's Unique Name** measure for a single row is visible in the following image. That row will differ from your result but you should nevertheless have such a row if you've followed the instructions carefully.

	Caption for Today	Member's Unique Name	Key for Today (AW)	Today (string)
September 16, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070916]	20070921	[Date].[Calendar].[Date].&[20070921]
September 17, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070917]	20070921	[Date].[Calendar].[Date].&[20070921]
September 18, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070918]	20070921	[Date].[Calendar].[Date].&[20070921]
September 19, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070919]	20070921	[Date].[Calendar].[Date].&[20070921]
September 20, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070920]	20070921	[Date].[Calendar].[Date].&[20070921]
September 21, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070921]	20070921	[Date].[Calendar].[Date].&[20070921]
September 22, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070922]	20070921	[Date].[Calendar].[Date].&[20070921]
September 23, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070923]	20070921	[Date].[Calendar].[Date].&[20070921]
September 24, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070924]	20070921	[Date].[Calendar].[Date].&[20070921]
September 25, 2007	September 21, 2010	[Date].[Calendar].[Date].&[20070925]	20070921	[Date].[Calendar].[Date].&[20070921]

The query behind the result is this one:

```
WITH
MEMBER [Measures].[Caption for Today] AS
    Format(Now(), 'MMMM dd, yyyy')
MEMBER [Measures].[Member's Unique Name] AS
    [Date].[Calendar].CurrentMember.UniqueName
MEMBER [Measures].[Key for Today] AS
    Format(Now(), 'yyyyMmdd')
MEMBER [Measures].[Key for Today (AW)] AS
    '2007' + Right([Measures].[Key for Today], 4)
MEMBER [Measures].[Today (string)] AS
    '[Date].[Calendar].[Date].&[' +
    [Measures].[Key for Today (AW)] + ']'
SET [Today] AS
    StrToMember( [Measures].[Today (string)], CONSTRAINED )
SELECT
    { [Measures].[Caption for Today],
        [Measures].[Member's Unique Name],
        --[Measures].[Key for Today],
        [Measures].[Key for Today (AW)],
        [Measures].[Today (string)] } ON 0,
    { [Date].[Calendar].[Date].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Basically, that's the query you get by following step by step instructions from above. With one difference, the measure **Key for Today** is intentionally left out of the query so that the result can fit the book size. You can leave it as is in your query, uncommented, to see its values.

The first measure is there only to make you practice building the format string for dates; it has no significance for the final calculation. "MMMM" displays the full month name (that is **September**), "dd" displays the date using two digits (that is **21**), "yyyy" displays the 4-digit year (that is **2010**). As mentioned earlier, there's a link explaining various tokens. Be sure to learn what's available because a Date dimension built using your regional settings might need a different date format.

The second measure is here to show how the unique member's name is built, so that we can build the exact string using today's date as a variable part of that name. Again, it is not used in the final calculation; it's here just to help achieve the correct string.

The third measure is the one that's important. Here its definition is relatively simple, but in your real projects you might have a more complex definition. The crucial part is that there's no way to know in advance what the correct format for a particular Date dimension would be. Therefore, the second measure, the one explained a moment ago, is here to enable us to identify the variable part of the unique name and to conclude how to build that part using date parts such as year, month, and date. With a bit of imagination you should be able to create the correct format just by looking at unique names of a few members. In case your date dimension's key is built using the best practice (an integer in form of this: yyyyMMdd) the definition provided in this example should fit perfectly.

The fourth measure is needed only when there is no today's date in the **Date** dimension. This is sort of a correction. The corrective factor can be applied again in various ways. Here we've used the logic to keep the date and month part of the key and replace the year with another. Why 2007? Because it's one of the years that has all dates in it. All of them? Almost. If you're running this example on that particular date of every leap year, sleep it over and give it another shot tomorrow. It's as good as it gets with **Adventure Works**. Hopefully you won't have these kinds of problems in your database. Again, in the case of a standard **Date** dimension build on best practices, the calculation should work as-is.

The fifth measure is the main part. Here we're actually building the final string. We're concatenating the fixed part of the unique name with one of the measures, third or fourth. In case of **Adventure Works**, that's measure number four. In your projects, that's measure three. You don't actually need the fourth measure, so feel free to remove it.

Finally, we built a named set from that final string using the `StrToMember()` function.

The `CONSTRAINED` flag is here because of two things. It automatically tells us if we've made a mistake; if there's an error in the string building process (although the error will say something else instead). It also ensures that the evaluation is faster and hence the query performance will be faster as well.

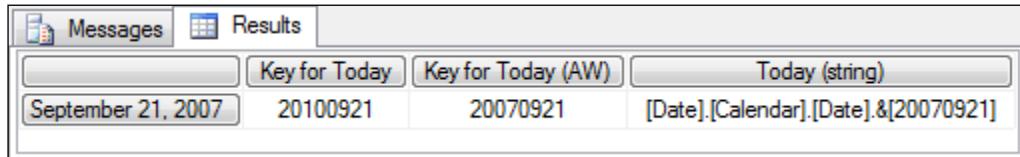
## Working with Time

---

That set can be used relatively easily. For example, here's the core part of the previous query where we've put that set on rows instead of the members on Date level of the Date.Calendar hierarchy.

```
WITH
MEMBER [Measures].[Key for Today] AS
    Format(Now(), 'yyyyMMdd')
MEMBER [Measures].[Key for Today (AW)] AS
    '2007' + Right([Measures].[Key for Today], 4)
MEMBER [Measures].[Today (string)] AS
    '[Date].[Calendar].[Date].&[' +
    [Measures].[Key for Today (AW)] + ']'
SET [Today] AS
    StrToMember( [Measures].[Today (string)], CONSTRAINED )
SELECT
    { [Measures].[Key for Today],
        [Measures].[Key for Today (AW)],
        [Measures].[Today (string)] } ON 0,
    { [Today] } ON 1
FROM
    [Adventure Works]
```

The result will have only one row. The member on that row will be today's date (or its parallel member in one of the previous years if arranged so) in the calculations.



	Key for Today	Key for Today (AW)	Today (string)
	September 21, 2007	20100921	[Date].[Calendar].[Date].&[20070921]

### There's more...

A named set, contrary to calculated member, preserves the original regular member. Regular members have a position in their level (also known as their ordinal), they have (potentially) descendants and ancestors, related and relating members on other hierarchies of the same dimension, and many other features. Calculated members don't have those things. They are placed as the last child of a regular member they've defined on and are not related to any other member except the root member of other hierarchies of the same dimension. That's why the idea is to make a set, not a calculated member.

Another thing worth pointing out here is that using Now() in calculated member stops the use of formula engine cache. See here for more info: <http://tinyurl.com/FormulaCacheChris>

Conversion, or shall we say, extraction of a single member in that set is relatively easy: `[Today].Item(0)`. Actually, we should specify `.Item(0).Item(0)`, but since there's only one hierarchy in the tuple forming that set, one `.Item(0)` is enough.

Defining new calculated measures is also easy. Today's sales measure would be defined like this: `([Today].Item(0), [Measures].[Internet Sales Amount])`.

### Relative periods

The solution doesn't stop there. Once we've made the definition for today's date, all related periods follow easily, more or less. Here are some examples. Again, we're defining sets, not calculated members:

```
SET [Yesterday] AS [Today].Item(0).PrevMember  
SET [This Month] AS [Today].Item(0).Parent  
SET [Prev Month] AS [This Month].Item(0).PrevMember  
SET [This Year] AS [Today].Item(0).Parent.Parent.parent.Parent  
SET [Prev Year] AS [This Year].Item(0).PrevMember  
SET [This Month Prev Year] AS [This Month].Item(0).Lag(12)
```

Moreover, you can anticipate the need for past or future relative periods and implement sets like [Next month], [1-30 days ahead], [31-60 days ahead], and so on.

In the case of role-playing dimensions, you can build independent sets for each role-playing dimension, that is, [Today] and [Due Today] sets, each pointing to today's system date in its own dimension (and hierarchy).

### Today or Yesterday?

SSAS cubes are often processed during the night. In that case, the last date with any value will be yesterday, not today. Nevertheless, you can apply the same principle explained here and hide the today's set. Hiding a set is done by adding the word **HIDDEN** before its name. Alternatively, you can leave it unhidden, it's totally up to you.

Also realize that if the concept of "Today" is based upon the `Now()` function in MDX, then your queries will not know when the nightly ETL and cube processing has completed, but will rather flip to "Tomorrow" at midnight.

### Alternative to shifting years

As the **Adventure Works** cube is static, it does not have the current year. We had to hardcode 2007 in the calculations above. The alternative is to change your system clock back to 2007. However, that may not always be possible. For example, if you're working on a server and other people are using it. That's why we've implemented the year-shifting logic in our example which should be OK in any situation.

## Potential problems

Sets are perfectly valid objects in MDX queries which means you should be able to bring them in the slicer or put them on columns or rows where they would interact with other dimensions found there. The ideal solution would be to construct the Today set, define a lot of relative periods as sets and the end users can use them as usual. However, that's not always the case. Some SSAS front-ends have problems working with sets and treat them as third-class citizens. For example, you cannot put a named set on filters in an Excel 2007 or Excel 2010 PivotTable. If that's the case, you must be ready to make compromises when defining today's date instead as a calculated member, as explained earlier. You may also read following recipes to find alternative solutions.

## See also

The recipes *Calculating today's date using the MemberValue function* and *Calculating today's date using an attribute hierarchy* provide an alternative solution to calculating today's date. You should read all of them in order to understand the pros and cons of each approach.

## Calculating today's date using the **MemberValue** function

The second way to calculate today's date is using the `MemberValue()` function. This is something we've already used in *Finding the last date with data* recipe. In case you haven't read it yet, do it before continuing with this recipe, at least the part that shows the **ValueColumn** property.

## Getting ready

Open Business Intelligence Development Studio (BIDS) and then open **Adventure Works DW 2008** solution. Double-click the **Date** dimension found in the **Solution Explorer**. Select the **Date** attribute and locate the **ValueColumn** property. It should not be empty, otherwise this recipe won't work. It should have the **date** type column from the underlying time dimension table.

## How to do it...

Follow these steps to calculate today's date using the `MemberValue` function:

1. Write and execute the following query in SQL Server Management Studio connected to the same cube mentioned above:

```
WITH  
MEMBER [Measures].[Caption for Today] AS
```

```

Format(Now(), 'MMMM dd, yyyy')
MEMBER [Measures].[Member Value] AS
    [Date].[Calendar].CurrentMember.MemberValue
MEMBER [Measures].[MV for Today] AS
    Format(Now(), 'M/d/yyyy')
MEMBER [Measures].[MV for Today (AW)] AS
    CDate( Left( [Measures].[MV for Today],
        Len([Measures].[MV for Today]) - 4) + '2007'
    )
SET [Today] AS
    Filter( [Date].[Calendar].[Date].MEMBERS,
        [Measures].[Member Value] =
        [Measures].[MV for Today (AW)] )
SELECT
    { [Measures].[Caption for Today],
        [Measures].[Member Value],
        [Measures].[MV for Today],
        [Measures].[MV for Today (AW)] } ON 0,
    { [Date].[Calendar].[Date].MEMBERS } ON 1
FROM
    [Adventure Works]

```

2. Then use this query to test the set:

```

WITH
MEMBER [Measures].[Member Value] AS
    [Date].[Calendar].CurrentMember.MemberValue
MEMBER [Measures].[MV for Today] AS
    Format(Now(), 'M/d/yyyy')
MEMBER [Measures].[MV for Today (AW)] AS
    CDate( Left( [Measures].[MV for Today],
        Len([Measures].[MV for Today]) - 4) + '2007'
    )
SET [Today] AS
    Filter( [Date].[Calendar].[Date].MEMBERS,
        [Measures].[Member Value] =
        [Measures].[MV for Today (AW)] )
SELECT
    { [Measures].[Member Value],
        [Measures].[MV for Today],
        [Measures].[MV for Today (AW)] } ON 0,
    { [Today] } ON 1
FROM
    [Adventure Works]

```

3. The result should contain only a single row with today's date.

## How it works...

The first query is used to build the Today set in a step-by-step fashion. The first measure is used for testing behavior of various date part tokens. The next measure is used to extract the MemberValue from each date found on the rows and is later used in the set's definition, on one side of criteria. The third measure, **MV for Today**, is the main measure. Its definition is obtained by deducing the correct format for the MemberValue from the observations made by analyzing the values in the previous measure. The step-by-step process is explained in the previous recipe.

As the **Adventure Works** solution doesn't have the current date, we're forced to shift it to the year **2007**. The Day and Month stay the same. This is implemented in the fourth measure: **MV for Today (AW)**. Finally, the set is defined using a `Filter( )` function which returns only one member, the one where the MemberValue is equal to the **MV for Today (AW)** measure's value.

The second query is here to verify the result. It consists of three measures relevant to the set's condition and the set itself.

## There's more...

The recipe *Calculating today's date using the string functions* explains how to enhance the cube design by adding relative periods in the form of additional sets. Look for the *Related periods* section of that recipe, it's also applicable to this recipe.

## Using the **ValueColumn** property in Date dimension

Many SSAS front-ends use the so-called *Time Intelligence* implementation. That means they enable the usage of special MDX functions such as `YTD()`, `ParallelPeriod()`, and others in their GUI. The availability of those functions is often determined by dimension type (has to be of type **Date**) and by the existence of the **ValueColumn** property typed as **Date** on the key attribute of the dimension, or both. Specifically, Excel 2007 and Excel 2010 look for the latter. Be sure to check those things when you're working on your date dimension.

Here's a link to the document which explains how to design cubes for Excel:

<http://tinyurl.com/DesignCubesForExcel>

## See also

*Calculating today's date using the string functions* and *Calculating today's date using an attribute hierarchy* are the recipes which provide an alternative solution to calculating today's date. You should read both of them in order to understand the pros and cons of each approach.

## Calculating today's date using an attribute hierarchy

The third way to calculate today's date is by using an attribute hierarchy. This is potentially the best way.

Instead of all the complexity with sets, strings, and other things in the previous two recipes, here we simply add a new column to the **Date** table and have the ETL maintain a flag for today's date. Then we slice by that attribute instead of using the `Now()` function in MDX. Plus, we don't have to wait to switch to "tomorrow" in MDX queries until the ETL completes and the cube is processed.

### Getting ready

Open the **Adventure Works DW 2008** solution in BIDS. Double-click the **Adventure Works DW** data source view. Locate the **Date** dimension in the left **Tables** pane and click on it.

### How to do it...

Follow these steps to calculate today's date using an attribute hierarchy:

1. Right-click the **Date** table and select **New Named Calculation**.
2. Enter **Today** for **Column Name** and this for the **Expression**:

```
case when convert(varchar(8), FullDateAlternateKey, 112) =
      convert(varchar(8), GetDate(), 112)
      then 'Yes'
      else 'No'
end
```

3. Close the dialog and explore that table. No record will have **Yes** in the last column.
4. Since we're on **Adventure Works** which doesn't have the current date, we have to adjust the calculation by shifting it to the year 2007. Define the new **Named Calculation** using this formula and name it **Today AW**:

```
case when convert(varchar(8), FullDateAlternateKey, 112) =
      '2007' +
      right(convert(varchar(8), GetDate(), 112), 4)
      then 'Yes'
      else 'No'
end
```

5. Close the dialog and explore that table. You should notice that this time, one row in the year 2007 has **'Yes'** in the last column. It will be the row with the day and month the same as your system's day and month.

6. Save and close data source view.
7. Double-click the **Date** dimension in the **Solution Explorer**.
8. Drag the **Today DW** column from the **Date** dimension table to the list of attributes on the left side and rename it to **Today**. Leave the relation to the key attribute as is – flexible. This will be the only attribute of that kind in this dimension. All others should be rigid.
9. Save, deploy, and process the full **Date** dimension.
10. Double-click the **Adventure Works** cube in the **Solution Explorer**, navigate to the **Browser** tab, and click on a button to process the cube.
11. Once it is processed, click the **Reconnect** button and drag the **Internet Sales Amount** measure in data part of the browser.
12. Add the **[Date].[Calendar]** hierarchy on rows and expand it few times. Notice that all the dates are here.
13. Add the **[Date].[Today DW]** hierarchy in slicer and uncheck the '**No**' member in it. Notice that the result contains only the current date. You can do the same using any other hierarchy of the **Date** dimension; all will be sliced by our new attribute.

### How it works...

This recipe depends on the fact that SSAS implements the so-called *auto-exists* algorithm. The main characteristic of it is that when two hierarchies of the same dimension are found in the query, the one in the slicer automatically reduces the other on axis so that only some of the members remain there, those for which an intersection exists.

In other words, if we put the **Yes** member in slicer as we did a moment ago, only those years, months, quarters, days in week, and so on that are valid for today's date remain. Meaning the current year, month, day in week, and so on. Only one member from each hierarchy.

The same query will give different results each day. That is exactly what we wanted to achieve. The usage is fairly simple – dragging the **Yes** member into slicer, which should be possible in any SSAS front-end.

The beauty of this solution is not only in the elegance of creating queries, but in the fact that it is the fastest method to implement the logic for today's date. Attribute relations offer better performance than string-handling functions and filtering.

### There's more...

The solution doesn't have to stop with the **[Date].[Today DW]** hierarchy. We can add **Today** as a set in the MDX script. This time, however, we'll have to use the **Exists()** function in order to get the related members of other hierarchies. Later on we can use navigational functions to take the right part of the hierarchy.

For example, Today should be defined like this:

```
SET [Today] AS  
Exists( [Date].[Calendar].[Date].MEMBERS,  
[Date].[Today AW].&[Yes] )
```

Once we have the Today named set, other variants are easy to derive from it. We've covered some of them in the recipe *Calculating today's date using the string functions*.

However, be aware that named sets, when used inside aggregating functions like `Sum()` and others, will prevent the use of block evaluation. Here's a link to the page that talks about which things are improved and which aren't in SQL Server 2008 R2 Analysis Services:

<http://tinyurl.com/Improved2008R2>

### Member "Yes" as a default member?

Short and simple – DONT! This might cause problems when other hierarchies of the Date dimension are used in slicer. That is, users of your cube might accidentally force a coordinate which does not exist.

For example, if they decide to put January in the slicer when the default Yes member implies the current quarter is not **Q1** but let's say **Q4**, they'll get an empty result without understanding what happened and why.

A solution exists, though. In such cases they should add the All member of the **[Date].[Today DW]** hierarchy in the slicer as well, to remove restrictions imposed by the default member. The question is – will you be able to explain that to your users?

A better way is to instruct them to explicitly put the Yes member in the slicer whenever required. Yes, that's extra work for them, but this way they will have control over the context and not be surprised by it.

### Other approaches

There's another method and that is using many-to-many relationships. The advantage of doing this over creating new attributes on the same dimension is that we only expose a single new hierarchy, even if it is on a new dimension.

### See also

Recipes *Calculating today's date using the string functions* and *Calculating today's date using the MemberValue function* are the recipes which provide an alternative solution to calculating today's date. You should read both of them in order to understand the pros and cons of each approach.

## Calculating the difference between two dates

This recipe shows how to calculate the difference between two dates. We're going to use an example with promotions, where we'll calculate the time span of a promotion, how many days it lasted.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**. Then execute this query:

```
SELECT
    { [Measures].[Reseller Order Count] } ON 0,
    { [Promotion].[Start Date].[Start Date].MEMBERS *
      [Promotion].[End Date].[End Date].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Promotion].[Promotion Type].&[Discontinued Product] )
```

The query shows that the **Discontinued Product** promotion appeared twice with various time spans. Our task is to calculate how many days it lasted each time.

### How to do it...

Follow these steps to calculate the difference between two dates:

1. Add the **WITH** part of the query.
2. Define two calculated measures that are going to collect the **ValueColumn** property of the **Start Date** and **End Date** hierarchies of the **Promotion** dimension.
3. Define the third calculated measure as `Number of days` using the VBA function `DateDiff()` and the two helper calculated measures defined a moment ago. Be sure to increase the second date by one in order to calculate the duration, meaning that both start and end dates will count.
4. Include all three calculated measures on columns and run the query which should look like the following:

```
WITH
    MEMBER [Measures].[Start Date] AS
        [Promotion].[Start Date].CurrentMember.MemberValue
    MEMBER [Measures].[End Date] AS
```

```

[Promotion].[End Date].CurrentMember.MemberValue
MEMBER [Measures].[Number of days] AS
    DateDiff('d', [Measures].[Start Date],
        [Measures].[End Date] + 1)
SELECT
    { [Measures].[Reseller Order Count],
        [Measures].[Start Date],
        [Measures].[End Date],
        [Measures].[Number of days] } ON 0,
    { [Promotion].[Start Date].[Start Date].MEMBERS *
        [Promotion].[End Date].[End Date].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Promotion].[Promotion Type].&[Discontinued Product] )

```

5. Check your result. It should look like this:

		Reseller Order Count	Start Date	End Date	Number of days
May 15, 2006	June 30, 2006	24	5/15/2006	6/30/2006	47
May 1, 2008	June 30, 2008	62	5/1/2008	6/30/2008	61

### How it works...

`DateDiff()` is a VBA function. It can also be found in T-SQL. What we have to do is specify the part of the dates in which we'd like the difference to be expressed. In our case, we used the `d` token which corresponds to the day level.

The duration is calculated as a difference plus one because both boundaries must be included. In the second row, it's easy to see that the 31 days in May and 30 days in June must equal 61.

### There's more...

`DateDiff()` expects the **date** type items as its second and third arguments. Luckily, we had exactly the required type in the **ValueColumn** property. This can be checked by opening BIDS and analyzing the Promotion dimension. If it weren't the case, we would have to construct expressions of **date** type and use them inside. Here are few working examples:

```

CDate( '2010-09-21' )

CDate( [Promotion].[Start Date].CurrentMember.Name )

```

## Dates in other scenarios

The example in this recipe highlighted a case where dates were found on two different hierarchies. That will not always be so. There will be situations when you'll only have a single date on a hierarchy or no dates at all.

In short, there are two ways how you can get those dates. You can calculate them in the form of two measures or locate them on the Date hierarchy of your Date dimension.

The example illustrated in this recipe used the `DateDiff()` function, a better fit for a scenario with measures. The *There's more* section warned that we should convert the value of measures (or expressions) to an appropriate date type (if it isn't already so), because the `DateDiff()` function requires dates.

The other scenario is to locate the dates on the Date hierarchy. For example, when you're calculating the number of consecutive days with no change in quantity of products in the warehouse, or similar.

In that case, there's no need for `DateDiff()`. Simply form a range of members by specifying the first date followed by a colon and then the second date. Finally, count the members in that set.

Actually, the usage of the `Count()` function might turn off block computation, so use its `Sum()` alternative:

```
Sum( {<member1> : <member2>} , 1 )
```

Where `<member1>` and `<member2>` are placeholders for the range.

This will give you the same count of members in that range, but the `Sum()` function is optimized to work in block mode while `Count()` over a range is not.

When the members in that range are dates (which will typically be so), counting them will return the duration in days. If you need a different granularity (let's say the number of weeks, hours or minutes), simply multiply the duration in days with the appropriate factor (1/7, 24 or 24\*60, respectively). Additionally, for the `DateDiff()` function, you can provide the appropriate first argument. See here for options:

<http://tinyurl.com/DateDiffExcel>

## The problem of non-consecutive dates

A problem will arise if dates are not consecutive, that is, if some of them are missing in your date dimension. Here we are referring to weekends, holidays, and others which are sometimes left out of the date dimension guided by the thinking that there's no data in them so why include them in the dimension? You should know that such design is not recommended and the solution provided in this recipe will not work. Moreover, this will not be the only problem you'll encounter by this bad design. Therefore, consider redesigning your date dimension or look for alternative solutions listed in the *See also* section of this recipe.

## See also

When the dates are close to one another, you might want to calculate the time difference instead. This is described in the following recipe, *Calculating the difference between two times*.

## Calculating the difference between two times

This recipe is similar to the previous one, but here we'll show how to calculate the difference in time and present the result appropriately.

What's specific about time, and by time we mean everything on and below the day granularity, is that periods are proportionally divided. A day has 24 hours, an hour has 60 minutes and a minute has 60 seconds. On the other hand, the above-day granularity is irregular - days in a month vary throughout the year, and days in the year vary on leap years.

The nice thing about having proportional periods is that we can present the result in various units. For example, we can say that an event lasted for 48 hours but we can also say 2 days. On the other hand, we can say 2 days, but we cannot say 0.06 months because a month is not a constant unit of time.

This ability will be demonstrated in the following example as well.

## Getting ready

The **Adventure Works** database doesn't contain any attribute or measure that has hours, minutes, or seconds. Hence we will create two calculated measures, one representing the start and the other representing the end of an event. Here are those measures:

```
WITH
MEMBER [Measures].[Start Time] AS
    CDate('2010-09-18 00:40:00')
MEMBER [Measures].[End Time] AS
    CDate('2010-09-21 10:27:00')
SELECT
    { } ON 0
FROM
    [Adventure Works]
```

## How to do it...

Follow these steps to calculate the difference between two times:

1. Define a new calculated measure as a difference of two initial measures introduced above and name it **Duration in days**.
2. Put that new measure on axis 0 as a single measure and run the query which should look like this:

```
WITH
MEMBER [Measures].[Start Time] AS
    CDate('2010-09-18 00:40:00')
MEMBER [Measures].[End Time] AS
    CDate('2010-09-21 10:27:00')
MEMBER [Measures].[Duration in days] AS
    [Measures].[End Time] - [Measures].[Start Time]
SELECT
    { [Measures].[Duration in days] } ON 0
FROM
    [Adventure Works]
```

3. The result represents number of days between those two events.

## How it works...

Each event has a starting point and an ending point. If those points in time are represented as dates, being the **date** type, then we can apply the simple operation of subtraction in order to get the duration of that event. In case those were not date type points, we should convert them into the date format, as shown in this example (string to date conversion using the `CDate()` VBA function).

## There's more...

It is possible to shift the result into another time unit. For example, we can calculate the duration in hours by multiplying the initial expression with the number of hours in a day.

```
MEMBER [Measures].[Duration in hours] AS
    ([Measures].[End Time] - [Measures].[Start Time]) * 24
    , FORMAT_STRING = '#,##0.0'
```

Add this member into the initial query and observe the results.

Likewise, we can get the duration in minutes and seconds if required. Multiplications are by 60 and 3600, respectively, in addition to 24 already there for the number of hours.

## Formatting the duration

Duration values can be formatted. Here's an example that shows how the original **Duration in days** calculation can be formatted so that the decimal part becomes displayed in a well-understood hh:mm:ss format, where hh stands for hours, mm for minutes, and ss for seconds:

```
MEMBER [Measures].[My Format] AS
    iif([Measures].[Duration in days] > 1,
        CStr(Int([Measures].[Duration in days])) +
        " ", "0 ") +
    'hh:mm:ss'

MEMBER [Measures].[Duration d hh:mm:ss] AS
    ([Measures].[End Time] - [Measures].[Start Time])
    , FORMAT_STRING = [Measures].[My Format]
```

Add this member into the initial query and observe the results.

Notice that we've wrapped the expression for `FORMAT_STRING` in a separate calculated measure. The reason for it is to improve the performance through caching. Only cell values are cached; expressions on `FORMAT_STRING` are not cached. That's why it pays off to define them in separate measures.

## Examples of formatting the duration on the web

Here are a few links with good examples of formatting the duration on the web:

- ▶ <http://tinyurl.com/FormatDurationMosha>
- ▶ <http://tinyurl.com/FormatDurationVidas>

## Counting working days only

In case you're interested in counting the working days only, Marco Russo, one of the reviewers of this book, presented his approach to this problem in his blog post:

<http://tinyurl.com/WorkingDaysMarco>

### See also

When the dates are far from each other, you might want to calculate the date difference instead. This is described in the previous recipe, *Calculating the difference between two dates*.

## Calculating parallel periods for multiple dates in a set

In the recipe *Calculating the YoY (Year-over-Year) growth (parallel periods)* we've shown how the `ParallelPeriod()` function works and how it can be used to calculate the YoY growth. All we had to do is specify a member, ancestor's level, and an offset, and the *parallel* member was returned as a result.

OLAP works in discrete space and therefore many functions, `ParallelPeriod()` included, expect a single member as their argument. On the other hand, relational reports are almost always designed using a date range, with **Date1** and **Date2** parameters for that report. As the relational reporting has a longer tradition than multidimensional, people are used to thinking that way. They expect many multidimensional reports to follow the same logic; operating on a range which is neither easy nor efficient. A proper cube design should help a lot, therefore eliminating the need for ranges and increasing the performance of the cube. There should be various attributes of time dimension in the cube: months, weeks, quarters, and so on. They comprise common ranges and should be used instead of range of dates.

However, there are times when the cube design cannot cover all the combinations and in that case, the request that reports should operate on a range are quite legitimate. The question arises - can we do the same in OLAP as in relational reporting if requested from us? Can we calculate the growth based on a range of members and not just a single member?

Yes, we can! The solution in MDX exists. This recipe shows how to solve the problem with multiple dates defined as a set. The next recipe, also the last in this chapter, shows how to deal with a more complex case – when dates are present in the slicer.

### Getting ready

We're going to make a simple query. We will analyze sales by colors for a date range that starts in December and ends just before Christmas. We'd like to analyze how we are doing in respect to the previous year, for the same period.

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**. Then execute this query:

```
WITH
MEMBER [Internet Sales CP] AS
Sum( { [Date].[Calendar].[Date].&[20071201] :
      [Date].[Calendar].[Date].&[20071224] },
     [Measures].[Internet Sales Amount] )
SELECT
{ [Internet Sales CP] } ON 0,
```

---

```
{ [Product].[Color].MEMBERS } ON 1
FROM
[Adventure Works]
```

The query has a date range. The aggregate of that range in form of a sum is calculated in a calculated measure which is then displayed for each color and total included as the first row.

The absolute values aren't telling us much and therefore we want to calculate YoY % values using the same date range, but in the previous year.

## How to do it...

Follow these steps to calculate parallel periods for multiple dates in a set:

1. Define a new calculated measure which returns the value for the same period but in the previous year. Name it **Internet Sales PP**, where PP stands for parallel period. The expression should look like this:

```
MEMBER [Internet Sales PP] As
Sum( { [Date].[Calendar].[Date].&[20071201] :
      [Date].[Calendar].[Date].&[20071224] },
     ( ParallelPeriod( [Date].[Calendar].[Calendar Year],
                       1,
                       [Date].[Calendar].CurrentMember ),
       [Measures].[Internet Sales Amount] )
   )
, FORMAT_STRING = 'Currency'
```

2. Define another measure, **Internet Sales YoY %** as a ratio of the PP measure over CP measure. The expression should be as follows:

```
MEMBER [Internet Sales YoY %] As
iif( [Internet Sales PP] = 0, null,
     ( [Internet Sales CP] / [Internet Sales PP] ) )
, FORMAT_STRING = 'Percent'
```

3. Add both calculated measures to the query and execute it. The query should look like this:

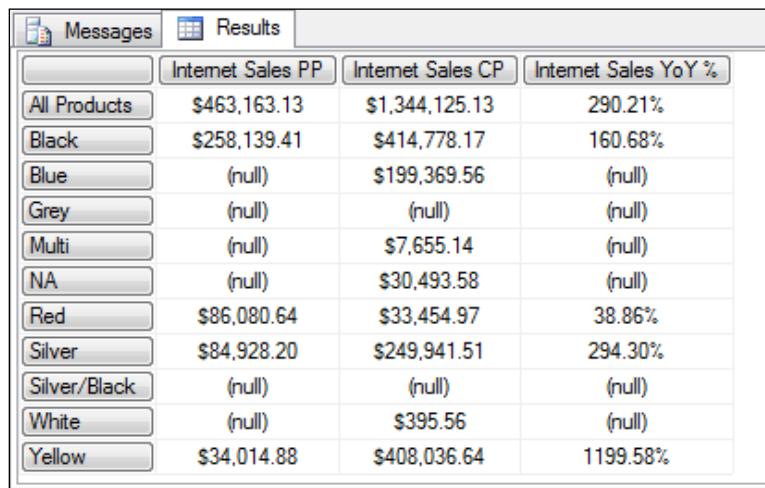
```
WITH
MEMBER [Internet Sales CP] AS
Sum( { [Date].[Calendar].[Date].&[20071201] :
      [Date].[Calendar].[Date].&[20071224] },
     [Measures].[Internet Sales Amount] )
MEMBER [Internet Sales PP] As
Sum( { [Date].[Calendar].[Date].&[20071201] :
      [Date].[Calendar].[Date].&[20071224] },
     ( ParallelPeriod( [Date].[Calendar].[Calendar Year],
```

```

        1,
        [Date].[Calendar].CurrentMember ),
[Measures].[Internet Sales Amount] )
)
, FORMAT_STRING = 'Currency'
MEMBER [Internet Sales YoY %] As
    iif( [Internet Sales PP] = 0, null,
        ( [Internet Sales CP] / [Internet Sales PP] ) )
    , FORMAT_STRING = 'Percent'
SELECT
    { [Internet Sales PP],
    [Internet Sales CP],
    [Internet Sales YoY %] } ON 0,
    { [Product].[Color].MEMBERS } ON 1
FROM
    [Adventure Works]

```

The results show that three colors had better results than before, one had worse (**Red**), but the overall result is better almost three times. There were also four new colors in the current season:



The screenshot shows the SSMS Results tab displaying a table of sales data. The table has four columns: Internet Sales PP, Internet Sales CP, and Internet Sales YoY %. The rows list various product colors and their sales figures. The table includes All Products as a summary row and several new colors (Silver, Silver/Black, Yellow) in addition to the previous ones.

	Internet Sales PP	Internet Sales CP	Internet Sales YoY %
All Products	\$463,163.13	\$1,344,125.13	290.21%
Black	\$258,139.41	\$414,778.17	160.68%
Blue	(null)	\$199,369.56	(null)
Grey	(null)	(null)	(null)
Multi	(null)	\$7,655.14	(null)
NA	(null)	\$30,493.58	(null)
Red	\$86,080.64	\$33,454.97	38.86%
Silver	\$84,928.20	\$249,941.51	294.30%
Silver/Black	(null)	(null)	(null)
White	(null)	\$395.56	(null)
Yellow	\$34,014.88	\$408,036.64	1199.58%

## How it works...

In order to calculate the parallel period's value for a set of members (a range in this example), we apply the same principle we use when calculating the value for the current season - summarizing the value of a measure on that range.

The calculation for the previous season differs in the expression part only. There we no longer use a measure, but instead a tuple. That tuple is formed using the original measure combined with the parallel period's member. In other words, we're reaching for the value in another coordinate and summing those values.

The calculation for the YoY % ratio is pretty straightforward. We check division by zero and specify the appropriate format string.

### There's more...

The other approach is to calculate another set, the previous season's range, and then apply the sum. Here's the required expression:

```
MEMBER [Internet Sales PP] AS  
  
    Sum(  
        Generate(  
            { [Date].[Calendar].[Date].&[20071201] :  
              [Date].[Calendar].[Date].&[20071224] },  
            { ParallelPeriod( [Date].[Calendar]  
                            .[Calendar Year],  
                            1,  
                            [Date].[Calendar]  
                            .CurrentMember.Item(0) )  
            } ),  
        [Measures].[Internet Sales Amount] )
```

Here, we're iterating on the old set using the `Generate()` function in order to shift each member of that set to its parallel member.

Note that we deliberately skipped defining any named sets for this scenario because they, when used inside aggregating functions like `Sum()`, prevent the block evaluation. We should put the sets inside those functions.

One more thing – for measures with non-linear aggregation functions (that is, the `DistinctCount`), the `Aggregate()` function should be used instead of `Sum()`.

### Parameters

The set of dates defined as a range can often be parameterized like this:

```
{ StrToMember( @Date1, CONSTRAINED ) :  
  StrToMember( @Date2, CONSTRAINED ) }
```

This way, the query (or SSRS report) becomes equivalent to its relational reporting counterpart.

## Reporting covered by design

We mentioned in the introduction of this recipe that it is often possible to improve the performance of the queries operating on a range of members by modifying the cube design. How it's done is explained in the first recipe of Chapter 6 in more detail.

### See also

Recipes *Calculating the YoY (Year-over-Year) growth (parallel periods)* and *Calculating parallel periods for multiple dates in slicer* deal with a similar topic. The `Generate()` function, very useful here, is also covered in the recipe *Iterating on a set in order to create a new one* in *Chapter 1, Elementary MDX Techniques*.

## Calculating parallel periods for multiple dates in a slicer

In the recipe *Calculating the YoY (Year-over-Year) growth (parallel periods)*, we've shown how the `ParallelPeriod()` function works when there's a single member involved. In *Calculating parallel periods for multiple dates in a set*, we've showed how to do the same, but on a range of members defined in a set. This recipe presents the solution to a special case when the set of members is found in slicer.

### Getting ready

We'll use the same case as in the previous recipe; we'll calculate the growth in the pre-Christmas season for each color of our products.

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**. Then execute this query:

```
SELECT
    { [Internet Sales Amount] } ON 0,
    { [Product].[Color].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( { [Date].[Calendar].[Date].&[20071201] :
        [Date].[Calendar].[Date].&[20071224] } )
```

The previous query returns the values of the **Internet Sales Amount** for each color. Notice that when the range is provided in slicer, there's no need to define new calculated measures as in the previous recipe; the SSAS engine automatically aggregates each measure using its aggregation function. Because of that, this is the preferred approach of implementing multi-select, although rarely found in SSAS front-ends.

## How to do it...

Follow these steps to calculate parallel periods for multiple dates in slicer:

1. Define a new calculated member. Name it **Internet Sales PP**. The definition for it should be the sum of parallel period's values on existing dates of the **Date.Calendar** hierarchy.

```
MEMBER [Internet Sales PP] As
    Sum( EXISTING [Date].[Calendar].[Date].MEMBERS,
        ( ParallelPeriod( [Date].[Calendar].[Calendar Year],
            1,
            [Date Range].Current.Item(0) ),
            [Measures].[Internet Sales Amount] )
        )
    , FORMAT_STRING = 'Currency'
```

2. Add another calculated measure. Name it **Internet Sales YoY %** and define it as a ratio of the original **Internet Sales Amount** measure over the **Internet Sales PP** measure. Be sure to implement a test for division by zero.
3. Add both calculated measures on the columns axis.
4. The final query should look like this.

```
WITH
MEMBER [Internet Sales PP] As
    Sum( EXISTING [Date].[Calendar].[Date].MEMBERS,
        ( ParallelPeriod( [Date].[Calendar].[Calendar Year],
            1,
            [Date].[Calendar].CurrentMember ),
            [Measures].[Internet Sales Amount] )
        )
    , FORMAT_STRING = 'Currency'
MEMBER [Internet Sales YoY %] As
    iif( [Internet Sales PP] = 0, null,
        ( [Measures].[Internet Sales Amount] /
        [Internet Sales PP] ) )
    , FORMAT_STRING = 'Percent'
SELECT
    { [Internet Sales PP],
```

## Working with Time

---

```
[Internet Sales Amount],  
[Internet Sales YoY %] } ON 0,  
{ [Product].[Color].MEMBERS } ON 1  
FROM  
[Adventure Works]  
WHERE  
( { [Date].[Calendar].[Date].&[20071201] :  
[Date].[Calendar].[Date].&[20071224] } )
```

5. Execute it and observe the results. They should match the results in the previous recipe because the same date range was used in both recipes.

	Internet Sales PP	Internet Sales Amount	Internet Sales YoY %
All Products	\$463,163.13	\$1,344,125.13	290.21%
Black	\$258,139.41	\$414,778.17	160.68%
Blue	(null)	\$199,369.56	(null)
Grey	(null)	(null)	(null)
Multi	(null)	\$7,655.14	(null)
NA	(null)	\$30,493.58	(null)
Red	\$86,080.64	\$33,454.97	38.86%
Silver	\$84,928.20	\$249,941.51	294.30%
Silver/Black	(null)	(null)	(null)
White	(null)	\$395.56	(null)
Yellow	\$34,014.88	\$408,036.64	1199.58%

## How it works...

The range defined in the `Sum( )` function serves the purpose of collecting all the members on the leaf level, the **Date** level in this case, that are valid for the current context. Since the slicer is the part of a query which establishes the context, this is a way we can detect a currently selected range of members (or any current member on axes in general). Once we know which dates are there, we can use that range to sum the values of the measure in the parallel period.

Finally, the ratio is calculated using both measures.

## There's more...

We can never know **exactly** what was in the slicer, we can only "see the shadow" of it. Let's see why.

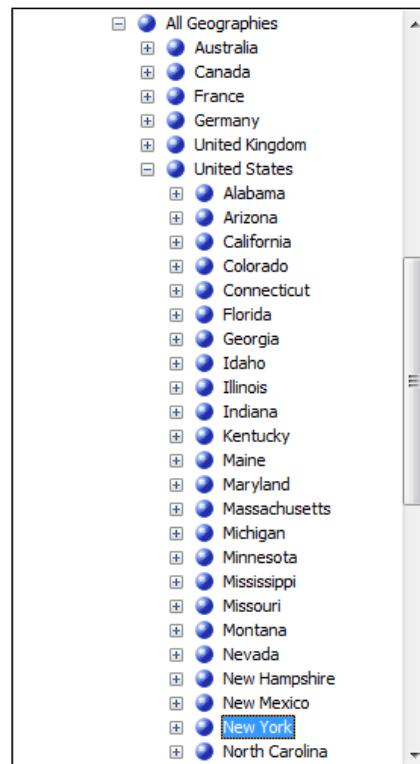
There are two MDX functions which serve the purpose of identifying members in the context. The first is the `CurrentMember` function. The function undoubtedly identifies single members, but it cannot be used for detecting multiple members in context. That's what the `Existing` function does. However, that one is not so precise. In other words, each of them have their purpose, advantages and disadvantages.

Suppose the slicer contains the city **New York**, a member of the **Geography.Geography** user hierarchy. Using the `CurrentMember` function, we can immediately identify the exact member of that hierarchy. We know that the slicer contains **New York** city, not anything above, below, left or right.

However, if there's also the **UK** country member, the usage of `CurrentMember` is inappropriate; it will result in an error.

In that case, we must use the `Existing` function. That function detects members of a level, not the hierarchy, which makes it less precise. If used on the **Country** level, it will return **USA** and **UK** although **USA** wasn't in the slicer, but one of its descendants, **New York** city. If used on a **State-Province** level, it will return **New York** and all the children of the **UK** member.

The following image can shed more light on it. It shows members **New York**, **UK**, **USA** and their relative positions; the different levels they are on.



The problem with this is that we never know exactly which members were there. The only way to calculate this correctly is to use the leaf level of a hierarchy because only then can we be sure that our calculation is correct. This can have a serious impact on the performance; leaf level calculations are slow in OLAP.

The other problem with detecting the context is that neither the `Existing` function nor the `CurrentMember` function detects what's in the subselect part of the query. That is not a problem per se, because both the slicer and the subselect have their purpose. The subselect doesn't set the context and so there's no need to know what was in there. However, Excel 2007 and 2010 use subselect in many situations where the slicer should be used instead and that makes many calculations useless because they can't detect the context and adjust to it. Make sure to test calculations in your client tool of choice to see whether it also uses unnecessary subselects.

Here's a blog post by Mosha Pasumansky that shows how to use dynamically-named sets to detect the contents of subselect:

<http://tinyurl.com/MoshaSubselect>

## See also

Recipes *Calculating the YoY (Year-over-Year) growth (parallel periods)* and *Calculating parallel periods for multiple dates in a set* deal with a similar topic, how to calculate the set of dates in a parallel period. The difference among them comes from the fact that the original period can be present in various places of an MDX query or that there are one or more dates the parallel period should be calculated for. Based on that, an appropriate recipe should be applied. Therefore, in order to understand and memorize the differences among them, it is suggested that you read all of the recipes dealing with parallel periods.

# 3

## Concise Reporting

In this chapter, we will cover:

- ▶ Isolating the best N members in a set
- ▶ Isolating the worst N members in a set
- ▶ Identifying the best/worst members for each member of another hierarchy
- ▶ Displaying a few important members, others as a single row, and the total at the end
- ▶ Combining two hierarchies into one
- ▶ Finding the name of a child with the best/worst value
- ▶ Highlighting siblings with the best/worst values
- ▶ Implementing bubble-up exceptions

### Introduction

In this chapter we're going to focus on how to make analytical reports more compact and more concise.

When we say "reports", we're assuming the result will be a pivot table, the analytical component found in any SSAS front-end in one form or another. We're not talking about SQL Server Reporting Services (SSRS) reports here, although these methods can be implemented there as well.

The problem with pivots is that they tend to grow a lot and very easily. All it takes is to include several hierarchies on rows, some of them perhaps on columns, and we've got ourselves a very large table to be analyzed.

The analysis of a large table is quite difficult. Also, a chart might not look pretty when the number of items crosses a certain threshold.

The solution is to make compact reports; to focus on what's important to us in a particular case. This chapter offers several techniques for reducing the amount of data and not losing the information about it.

We start with several recipes dealing with isolation of important members and their combination in case of multiple hierarchies. We also present a way of including the rest of the members in the report and still containing everything on one screen.

In addition to that, we cover techniques for including additional information on a reduced set of members (from upper levels), color-coding the foreground and/or background of cells, and extracting information in a single measure.

The chapter also includes a trick to reduce the number of metadata columns in case you need to.

Many of the recipes rely on some of the elementary MDX techniques explained in the first chapter. Therefore, make yourself familiar with that chapter first if you haven't already done so.

## Isolating the best N members in a set

Hierarchies can contain a lot of members. In this recipe we are going to show how to extract only the significant ones; members with the highest value for a certain measure.

This requirement is often necessary because not only does it allow end users to focus their efforts on a smaller set of members, the queries are much faster as well.

We'll base our example on the `TopCount()` function, a function that returns the exact number of members as specified. In addition to that function, MDX has two more similar functions, namely `TopPercent()` and `TopSum()`. Contrary to `TopCount()` function, these functions return an unknown number of members. In other words, they are designed to return a set of members based on their contribution, in percentage or in absolute value, respectively.

Further similarities and differences between `TopCount()`, `TopSum()`, and `TopPercent()` functions will be covered in later sections of this recipe.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Reseller** dimension. Here's the query we'll start from:

```
WITH
    SET [Ordered Resellers] AS
        Order( [Reseller].[Reseller].[Reseller].MEMBERS,
               [Measures].[Reseller Sales Amount],
               BDESC )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Ordered Resellers] } ON 1
FROM
    [Adventure Works]
```

Once executed, this query returns reseller sales values for every individual reseller, where the resellers themselves are sorted in descending order. Our task is to extract only five of them, those with the highest sales amount.

	Reseller Sales Amount
Brakes and Gears	\$877,107.19
Excellent Riding Supplies	\$853,849.18
Vigorous Exercise Company	\$841,908.77
Totes & Baskets Company	\$816,755.58
Retail Mall	\$799,277.90
Corner Bicycle Supply	\$787,773.04
Outdoor Equipment Store	\$746,317.53
Thorough Parts and Repair Servi...	\$740,985.83
Health Spa, Limited	\$730,798.71
Fitness Toy Store	\$727,272.65
Latest Sports Equipment	\$724,299.64
First Bike Store	\$711,864.76
Great Bikes	\$700,803.79

### How to do it...

1. Create a new calculated set and name it **Top 5 Resellers**.
2. Define it using the `TopCount()` function where the first argument is the set of reseller members, the second is the number 5, and the third is the measure Reseller Sales Amount.
3. Remove the **Ordered Resellers** set from the query and put the **Top 5 Resellers** on rows instead.

4. The query should look like the following. Execute it.

```
WITH
SET [Top 5 Resellers] AS
    TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
        5,
        [Measures].[Reseller Sales Amount] )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Top 5 Resellers] } ON 1
FROM
    [Adventure Works]
```

5. Only the five rows with the highest values should remain, as displayed in the screenshot below:

The screenshot shows a results grid from SSMS. The title bar says 'Messages' and 'Results'. The results grid has two columns: 'Reseller' and 'Reseller Sales Amount'. The data is as follows:

	Reseller Sales Amount
Brakes and Gears	\$877,107.19
Excellent Riding Supplies	\$853,849.18
Vigorous Exercise Company	\$841,908.77
Totes & Baskets Company	\$816,755.58
Retail Mall	\$799,277.90

6. Compare the rows returned with the ones from the initial query in this recipe. They should be the same - in the exact same order and with the exact same values.

### How it works...

The `TopCount()` function takes three arguments. The first one is a set of members that is to be limited. The second argument is the number of members to be returned. The third argument is an expression to be used for determining the order of members.

In this example we asked for the five resellers with the highest value of the measure **Reseller Sales Amount**. Using the `TopCount()` function we got exactly that.

It's worth mentioning that the `TopCount()` function always sorts the rows in descending order.

## There's more...

As seen in this example, returning the top N members is quite easy. Returning the proper ones is quite different entirely. In MDX, it is relatively easy to make a mistake and return other members, not the ones we intended to. Why's that?

The most important argument of the `TopCount( )` function is the third argument. That is what determines how the members will be sorted internally so that only N of them remain afterwards. As mentioned earlier, the argument is an expression. Meaning it can be a single measure, an expression including several measures, a single tuple, multiple tuples, or anything else that evaluates to a scalar value.

Two kinds of mistakes can happen here.

One is to position a member of another hierarchy (the year 2007) on the opposite axis of the query (on columns), and to expect the `TopCount( )` function to include that in its third argument. In other words, expecting that `TopCount( )` function will return top N members in the year 2007. It will not! The result will be there, see the upper query in the image below. The question is – what does it represent.

```

Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query Adventure Works DW 2008 Execute
Year 2007 on the opposite axis.mdx
Cube: A
Me
SELECT
{ [Measures].[Reseller Sales Amount] } *
{ [Date].[Calendar Year].&[2007] } ON 0,
{ [Top 5 Resellers] } ON 1
FROM
!!!
```

	Reseller Sales Amount
Brakes and Gears	\$361,627.85
Excellent Riding Supplies	\$323,324.43
Vigorous Exercise Company	\$252,613.29
Totes & Baskets Company	\$328,615.66
Retail Mall	\$291,364.58

Query executed successfully. Adventure Works DW 2008R2 | 00:00:01

```

All member in slicer.mdx
Cube: A
Me
WHERE
( [Date].[Calendar].[All Periods] )
!!!
```

	Reseller Sales Amount				
All Periods	CY 2005	CY 2006	CY 2007	CY 2008	
Brakes and Gears	\$877,107.19	\$88,003.66	\$247,559.40	\$361,627.85	\$179,916.29
Excellent Riding Supplies	\$853,849.18	\$73,842.73	\$317,507.39	\$323,324.43	\$139,174.63
Vigorous Exercise Company	\$841,908.77	\$162,738.91	\$316,681.80	\$252,613.29	\$109,874.78
Totes & Baskets Company	\$816,755.58	\$56,811.47	\$296,913.35	\$328,615.66	\$134,415.10
Retail Mall	\$799,277.90	\$79,747.30	\$336,746.33	\$291,364.58	\$91,419.69

Query executed successfully. Adventure Works DW 2008R2 | 00:00:01

The result contains the values of some members in the year 2007, but the members themselves are obtained in the context of all years. In other words, we got the best N members in all years and then displayed their values for a single year, 2007.

Take a look at the previous image one more time, especially the lower query and notice that the result of the upper query matches with the column containing the year 2007. Notice, however, that the members are ordered in the context of all years, highlighted in the first column. That's how the problem occurs.

Sometimes this makes sense, but oftentimes it will be nothing other than a mistake.

Can it be recognized? Can we see that we've made such mistake? Yes, although not necessarily always.

When the results are not ordered in descending order or when the number of rows is less than specified, those are good signs that something went wrong with the calculation. Be sure to watch for those signs. This checkpoint is deliberately made in the last step of this recipe's *How to do it ...* section.

So, what's the reason for this behavior?

Axes are independent. Only the members in slicer are implicitly included in the third argument of the `TopCount()` function (which is a mechanism known as *Deep Autoexists*: <http://tinyurl.com/AutoExists>). To be precise, any outer MDX construct also sets the context, but here we didn't have such a case, so we can focus on the slicer and axes relation only. To conclude, only when the year 2007 is found in the slicer, the third argument will be expanded into a tuple and the result will be evaluated as the top N member in year 2007.

Execute the following query and compare its result with the query that had the year 2007 on the opposite axis (visible in the previous image).

```
WITH
SET [Top 5 Resellers] AS
    TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
        5,
        [Measures].[Reseller Sales Amount] )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Top 5 Resellers] } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Calendar Year].&[2007] )
```

Notice that members on rows changed. These resellers are the top 5 resellers in the year 2007. Also notice that their values are in descending order.

Reseller Sales Amount	
Field Trip Store	\$368,440.99
Brakes and Gears	\$361,627.85
Outdoor Equipment Store	\$360,839.91
Totes & Baskets Compa...	\$328,615.66
Westside Plaza	\$324,308.79

Now, let's see the other mistake – the opposite one. That is, when we want to override the context but forgot to do so in the third argument of the `TopCount()` function, or we make it wrong. For example, when the year 2007 is in the slicer, and we want to get the top N members from the previous year.

Why would we want to do such a thing? Because we want to analyze how our last year's best resellers are doing this year.

If that's the case, we must provide a tuple as the third argument. The idea of the tuple is to overwrite the context set by the slicer with the member from the same hierarchy used inside the tuple. Remember, it has to be the same hierarchy or it won't work.

Following the previous examples, we must define the set like this:

```
SET [Top 5 Resellers in 2006] AS
    TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
        5,
        ( [Measures].[Reseller Sales Amount],
            [Date].[Calendar Year].&[2006] )
    )
```

Develop a habit of naming the top N sets that use tuples appropriately (see above). And remember that although the slicer or the outer MDX construct determines the context for the values to be displayed and for the functions to be evaluated, we can always override that in order to adjust the set we're after.

### Testing the correctness of the result

The equivalent syntax of the `TopCount()` function is this:

```
Head( Order( [Reseller].[Reseller].[Reseller].MEMBERS,
    [Measures].[Reseller Sales Amount],
    BDESC ), 5 )
```

Although this construct can be useful for testing the correctness of the result, `TopCount()` is the preferred way of implementing the requirement of isolating the best N members. This is because the `order()` function is a relatively slow MDX function because it materializes the set and the query optimizer may not be successful in optimizing the query by recognizing the Head-Order construct as a `TopCount()` function.

### Multidimensional sets

In the case of multidimensional sets, the second argument determines the number of tuples to be returned. Rows are sorted in the descending order again.

### TopPercent() and TopSum() functions

As we said in the introduction, `TopPercent()` and `TopSum()` are two functions similar to the `TopCount()` function. The first one returns an unknown number of members. In `TopPercent()`, the second argument determines the percentage of them to be returned, starting from the ones with the highest values and ending when the total value of the members included compared to the total value of all members reaches the percentage specified in that function. The second one works on the same principle except that the absolute value and not the percentage is what is specified and compared. For example, `TopPercent()` with 80 means we want top members who form 80% percent of the total result. `TopSum()` with 1,000,000 means we want members whose total forms that value, looking from the member with the highest value and adding all of them below until that value is reached.

The same principles, ideas, and warnings apply to all Top-Something functions.

### See also

*Isolating the worst N members in a set* and *Identifying the best/worst members for each member of another hierarchy*.

## Isolating the worst N members in a set

In the previous recipe we've shown how to identify members with the highest result. In this recipe we'll do the opposite and return those with the lowest result.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Reseller** dimension. Here's the query we'll start from:

```
WITH
    SET [Ordered Resellers] AS
        Order( [Reseller].[Reseller].[Reseller].MEMBERS,
               [Measures].[Reseller Sales Amount],
               BASC )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Ordered Resellers] } ON 1
FROM
    [Adventure Works]
```

Once executed, that query returns resellers' sales values for every individual reseller, where the resellers themselves are sorted in the ascending order. Our task is to extract only the five with the worst sales amount.

## How to do it...

1. Create a new calculated set and name it **Bottom 5 Resellers**.
2. Define it using the `BottomCount()` function where the first argument is the set of reseller members, the second is the number 5, and the third is the measure `Reseller Sales Amount`.
3. Apply the `NonEmpty()` function over the set specified as the first argument using the same measure as in the third argument of the `BottomCount()` function.

```
SET [Bottom 5 Resellers] AS
    BottomCount(
        NonEmpty( [Reseller].[Reseller].[Reseller].MEMBERS,
                  { [Measures].[Reseller Sales Amount] } ),
                  5,
                  [Measures].[Reseller Sales Amount]
    )
```

4. Remove the **Ordered Resellers** set from the query and put the **Bottom 5 Resellers** on rows instead.
5. The query should look like the following. Execute it.

```
WITH
    SET [Bottom 5 Resellers] AS
        BottomCount(
            NonEmpty( [Reseller].[Reseller].[Reseller].MEMBERS,
                      { [Measures].[Reseller Sales Amount] } ),
                      5,
                      [Measures].[Reseller Sales Amount]
```

```
)  
SELECT  
    { [Measures].[Reseller Sales Amount] } ON 0,  
    { [Bottom 5 Resellers] } ON 1  
FROM  
    [Adventure Works]
```

6. Only the five rows with the lowest values should remain, as displayed in the screenshot below:

	Reseller Sales Amount
Mobile Outlet	\$1.37
Parts Shop	\$24.29
Eleventh Bike Store	\$57.68
Large Bike Shop	\$58.07
Essential Bike Works	\$59.33

7. Compare the rows returned with the ones from the initial query in this recipe. They should be the same, in the exact same order, and with the exact same values, once the initial empty rows are ignored.

### How it works...

The `BottomCount()` function takes three arguments. The first one is a set of members that is going to be limited. The second argument is the number of members to be returned. The third argument is the expression for determining the order of members.

In this example we asked for the five resellers with the lowest value of the measure **Reseller Sales Amount**. Using the `BottomCount()` function only, we won't get exactly what we want. It can be rectified relatively easy.

If we repeat the same process again, this time by skipping the third step in the *How to do it* list, we would get five rows with empty values. How come?

Values, once sorted, can be separated into four groups. The first is the one with positive values. The second is the one with zero values. The third is the one with null values, and the fourth is the one with negative values. Those groups appear in that particular order once sorted as descending, and in the reverse order once sorted ascending.

In the `TopCount()` function covered in the previous recipe, we didn't experience the effect of null values because the results were all positive. In the `BottomCount()` function, this is something that needs to be taken care of, particularly if there are no negative values. The reason why we want to get rid of null values is because from a business perspective, those values represent no activity. What we're interested in is to identify members with activity, but whose activity was the poorest.

That's the reason we applied the `NonEmpty()` function in the third step of this recipe. That action removed all members with a null value in that particular context, leaving only members with activity to the outer `BottomCount()` function.

It's worth mentioning that the `BottomCount()` function always sorts the rows in ascending order.

### There's more...

The same thing that's been said for context and the `TopCount()` function in the previous recipe is valid in this case too. Therefore you are advised to read that recipe after finishing this one.

### **BottomPercent() and BottomSum() functions**

`BottomPercent()` and `BottomSum()` are two functions similar to the `BottomCount()` function. They are the opposite functions of the `TopPercent()` and `TopSum()` functions explained in the last section of the previous recipe. The same principles, ideas, and warnings also apply here.

### See also

*Isolating the best N members in a set and Identifying the best/worst members for each member of another hierarchy.*

## Identifying the best/worst members for each member of another hierarchy

Quite often it is necessary to analyze the combination of hierarchies in a way that the top or bottom N members of the inner hierarchy are displayed for each member of the outer hierarchy. Again, it's sort of a report reduction where the important combinations of members are preserved and the rest of the cross-join is left out.

This recipe shows how to create a `TopCount()` calculation relative to another hierarchy.

## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Sales Territory** dimension and the **Reseller** dimension. Here's the query we'll start from.

```
WITH
    SET [Ordered Resellers] AS
        Order( [Reseller].[Reseller].[Reseller].MEMBERS,
               [Measures].[Reseller Sales Amount] ,
               BDESC )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    NON EMPTY
    { [Sales Territory].[Sales Territory Country].MEMBERS *
      [Ordered Resellers] } ON 1
FROM
    [Adventure Works]
```

Once executed, that query returns reseller sales values for every individual reseller and country, where the resellers themselves are sorted in ascending order for each country:

The screenshot shows the SQL Server Management Studio interface with the 'Results' tab selected. The results grid displays a list of resellers and their sales amounts, ordered by sales amount for each territory. The columns are labeled 'All Sales Territories' (containing the territory names), 'Reseller' (containing the reseller names), and 'Reseller Sales Amount' (containing the sales values). The sales values are listed in descending order for each territory.

All Sales Territories	Reseller	Reseller Sales Amount
All Sales Territories	Brakes and Gears	\$877,107.19
All Sales Territories	Excellent Riding Supplies	\$853,849.18
All Sales Territories	Vigorous Exercise Company	\$841,908.77
All Sales Territories	Totes & Baskets Company	\$816,755.58
All Sales Territories	Retail Mall	\$799,277.90
All Sales Territories	Comer Bicycle Supply	\$787,773.04
All Sales Territories	Outdoor Equipment Store	\$746,317.53
All Sales Territories	Thorough Parts and Repair Servi...	\$740,985.83
All Sales Territories	Health Spa, Limited	\$730,798.71
All Sales Territories	Fitness Toy Store	\$727,272.65
All Sales Territories	Latest Sports Equipment	\$724,299.64
All Sales Territories	First Bike Store	\$711,864.76
All Sales Territories	Great Bikes	\$700,803.79

Our task is to extract the five resellers with the best sales for each country. In other words, we expect different resellers in each country.

## How to do it...

1. Define a new calculated set; name it **Top 5 Resellers per Country**.
2. Use the `Generate()` function as a way of performing the iteration.
3. Provide the set of countries found on rows as the first argument of that `Generate()` function.
4. Provide the second argument of that function in the form of a cross-join of the current member of countries hierarchy and the `TopCount()` function applied to the set of resellers.
5. Put that new calculated set on rows, instead of everything there.
6. Verify that the query looks like this, and then execute it.

```

WITH
SET [Top 5 Resellers per Country] AS
Generate(
    [Sales Territory].[Sales Territory Country].MEMBERS,
    { [Sales Territory].[Sales Territory Country]
        .CurrentMember } *
    TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
        5,
        [Measures].[Reseller Sales Amount] )
)
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    NON EMPTY
    { [Top 5 Resellers per Country] } ON 1
FROM
    [Adventure Works]

```

7. Once returned, the result will include different resellers for each country, the top 5 of them to be precise. Verify that.

		Reseller Sales Amount
All Sales Territories	Brakes and Gears	\$877,107.19
All Sales Territories	Excellent Riding Supplies	\$853,849.18
All Sales Territories	Vigorous Exercise Compa...	\$841,908.77
All Sales Territories	Totes & Baskets Company	\$816,755.58
All Sales Territories	Retail Mail	\$799,277.90
Australia	Nationwide Supply	\$221,169.78
Australia	Rich Department Store	\$148,996.51
Australia	Gears and Parts Company	\$145,407.74
Australia	Budget Toy Store	\$145,380.13
Australia	Helmets and Cycles	\$116,300.95
Canada	Vigorous Exercise Compa...	\$841,908.77
Canada	Retail Mall	\$799,277.90
Canada	Corner Bicycle Supply	\$787,773.04
Canada	Health Spa, Limited	\$730,798.71
Canada	Top Sports Supply	\$602,559.89

8. Notice that the results are ordered as descending inside each particular country.

### How it works...

`Generate()` is a loop. Using that function, we can iterate over a set of members and create another set of members, as explained in the Chapter 1 recipe, *Iterating on a set in order to create a new one*.

In this recipe, we used it to iterate over a set of countries and to create another set, a set that is formed as a combination of the current country in the loop and the top 5 resellers in the context of that country. The loop sets the context, that's why we don't need to use a tuple as the third argument of the `TopCount()` function:

```
( [Sales Territory].[Sales Territory Country].CurrentMember,  
  [Measures].[Reseller Sales Amount] )
```

That current member is already implicitly there, set by the outer loop, the `Generate()` function.

In short, we had to build a multidimensional set in advance and we displayed it on rows. That set is obtained in iteration, where the top 5 resellers are identified for each country.

### There's more...

Was it really necessary to define such a complex syntax? Couldn't we have done something simpler instead? Let's see.

One idea would be to define **Top 5 Resellers** as we did in the first recipe of this chapter and use it on rows.

```
WITH  
SET [Top 5 Resellers] AS  
  TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,  
            5,  
            [Measures].[Reseller Sales Amount] )  
SELECT  
  { [Measures].[Reseller Sales Amount] } ON 0,  
  NON EMPTY  
  { [Sales Territory].[Sales Territory Country]  
    .[Sales Territory Country].MEMBERS *  
    [Top 5 Resellers] } ON 1  
FROM  
  [Adventure Works]
```

When such a query is run, it returns the total of 5 rows as seen on the following screenshot.

The screenshot shows a software interface with two tabs at the top: 'Messages' and 'Results'. The 'Results' tab is selected, displaying a table with three columns. The columns are labeled 'Country', 'Reseller Name', and 'Reseller Sales Amount'. The data is as follows:

Country	Reseller Name	Reseller Sales Amount
Canada	Vigorous Exercise Company	\$841,908.77
Canada	Retail Mall	\$799,277.90
United States	Brakes and Gears	\$877,107.19
United States	Excellent Riding Supplies	\$853,849.18
United States	Totes & Baskets Company	\$816,755.58

The results are not sorted, which is the first indicator that something is wrong. The number of returned items is less than expected. That's the second indicator.

In order to see what went wrong, we need to comment the NON EMPTY keyword and execute the same query again. This time it is more understandable what's going on.

The screenshot shows a software interface with two tabs at the top: 'Messages' and 'Results'. The 'Results' tab is selected, displaying a table with three columns. The columns are labeled 'Country', 'Reseller Name', and 'Reseller Sales Amount'. The data is as follows:

Country	Reseller Name	Reseller Sales Amount
Australia	Brakes and Gears	(null)
Australia	Excellent Riding Sup...	(null)
Australia	Vigorous Exercise Co...	(null)
Australia	Totes & Baskets Com...	(null)
Australia	Retail Mall	(null)
Canada	Brakes and Gears	(null)
Canada	Excellent Riding Sup...	(null)
Canada	Vigorous Exercise Co...	\$841,908.77
Canada	Totes & Baskets Com...	(null)
Canada	Retail Mall	\$799,277.90
France	Brakes and Gears	(null)
France	Excellent Riding Sup...	(null)
France	Vigorous Exercise Co...	(null)
France	Totes & Baskets Com...	(null)
France	Retail Mall	(null)

Resellers repeat in each country. The group of resellers marked in the previous screenshot can be found above and below that country.

It wouldn't help to expand the third argument of the `TopCount()` function either:

```
( [Sales Territory].[Sales Territory Country].CurrentMember,  
[Measures].[Reseller Sales Amount] )
```

The reason why this all doesn't work is because calculated sets are evaluated once – after the slicer and before the iteration on cells. Therefore, the `TopCount()` function made with or without the current country is evaluated in the context of the default country, the root member. That's why the countries repeat each other in the query.

The object that evaluates per cell is a calculated member, but we need a set of members; we need 5 members and therefore we cannot use that in this case. The only thing that's left us is to push countries in the set as well. By having everything in advance before the iteration on cells begins, we can prepare the required multidimensional set of countries and their best resellers.

### Support for the relative context and multidimensional sets in SSAS front-ends

Front-ends perform cross-join operations while allowing end users to filter and isolate members of hierarchies in order to limit the size of the report. Some front-ends allow even functions like `TopCount()` to be applied visually, without editing the actual MDX. The thing they rarely do is allow the `TopCount()` function to be applied relatively to another hierarchy.

As we saw, without the relative component, top N members are calculated in the context of other hierarchy's root member, not individual members in the query. The only solution is to define a multidimensional set using the `Generate()` function. Then comes another problem – multidimensional sets (sets that contain members from more than one attribute) are not supported in many front-ends, for example, in Excel 2007 and 2010.

Test the limitations of your tool in order to know whether you can use it as a front-end's feature; implement it in a cube as a multidimensional set or write an MDX query.

### See also

*Isolating the best N members in a set* and *Isolating the worst N members in a set*.

## Displaying few important members, others as a single row, and the total at the end

There are times when isolating the best or worst members is not enough. End users often want to see the total of all members as well as the difference, a single row representing all other members not included in the report. This recipe shows how to fulfill this requirement.

## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Reseller** dimension. Here's the query we'll start from:

```
WITH
SET [Ordered Resellers] AS
    Order( [Reseller].[Reseller].[Reseller].MEMBERS,
           [Measures].[Reseller Sales Amount],
           BDESC )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Ordered Resellers] } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns values of the **Reseller Sales Amount** measure for every single reseller where the resellers are sorted in descending order.

## How to do it...

1. Create a new calculated set and name it **Top 5 Resellers**.
2. Define it using the `TopCount()` function where the first argument is the set of reseller members, the second is the number 5, and the third is the measure `Reseller Sales Amount`. In short, the definition should be this:

```
SET [Top 5 Resellers] AS
    TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
              5,
              [Measures].[Reseller Sales Amount] )
```

3. Remove the **Ordered Resellers** set from the query and put the **Top 5 Resellers** on rows instead.
4. Execute the query. Only the five rows with the highest values should remain.
5. Next, create a new calculated member and name it **Other Resellers**. Use the following definition:

```
MEMBER [Reseller].[Reseller].[All].[Other Resellers] AS
    Aggregate( - [Top 5 Resellers] )
```

6. Include that member on rows, next to the set.

7. Finally, include the root member of the **Reseller.Reseller** hierarchy in a more generic way by adding its [All] level as the last set on rows and run the query:

```
WITH
SET [Top 5 Resellers] AS
    TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
              5,
              [Measures].[Reseller Sales Amount] )
MEMBER [Reseller].[Reseller].[All].[Other Resellers] AS
    Aggregate( - [Top 5 Resellers] )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    { [Top 5 Resellers],
      [Reseller].[Reseller].[All].[Other Resellers],
      [Reseller].[Reseller].[All] } ON 1
FROM
    [Adventure Works]
```

8. The result will display 7 rows: top 5 resellers, other resellers in a single row, and the total in form of the root member:

Results	
	Reseller Sales Amount
Brakes and Gears	\$877,107.19
Excellent Riding Supplies	\$853,849.18
Vigorous Exercise Company	\$841,908.77
Totes & Baskets Company	\$816,755.58
Retail Mall	\$799,277.90
Other Resellers	\$76,261,698.37
All Resellers	\$80,450,596.98

### How it works...

The first part of the solution is to isolate top N members. This is done relatively easily, using the `TopCount()` function. The more detailed explanation of this part is covered in the first recipe of this chapter.

The second part is what is special about this type of report. All other members are obtained using the negation of a set, which is written as a minus followed by that set. This is explained in chapter one's recipe, *Implementing NOT IN set logic*.

Next, that negative set expression is wrapped inside the `Aggregate()` function which compacts that set in a single calculated member. That calculated member is the row that follows top N members in the result.

Finally, the root member of that hierarchy comes as the last row which acts as a total of rows. Here it's worth noting that we didn't use the root member's name; we've used its level name. Look in the code and see what came as the result in the previous screenshot. The reason for this is explained in the next section.

### There's more...

The beauty of this solution is that the initial set, in this case **Top 5 Resellers**, can be any set. It can even be a parameter of the report. The rest of the query will work for the **Reseller**.  
**Reseller** hierarchy without the need to change anything. In other words, the initial set can be the bottom N members, members that are a result of a `Filter()` function, existing members for the current context; pretty much anything. It doesn't matter how many members there are in the initial set. All the rest will be aggregated in a single row.

Another thing that makes this query generic is the way we referred to the root member. We didn't use its unique name (which would be different for different hierarchies). We used its level name instead. As that's the only non-calculated member on that level, we achieved the same effect. How to continue on this path to make this query even more generic is explained in the following sections.

There's another thing we can do to make this report a bit more user-friendly. We can put our own calculated member (the root member) on rows instead of the last set.

The root member, found in the **[All]** level, is usually named as **All** or **All Something**. In our example it's - **All Resellers**. A "total" might fit better occasionally. In other words, we could define a new calculated member and name it **Total**. Here's its definition:

```
MEMBER [Reseller].[Reseller].[All].[Total] AS  
    [Reseller].[Reseller].[All]
```

Include that member on rows instead; it will be an alias for the **All Resellers** member with a friendly name.

### Tips and tricks

If the requirement is such, calculated members can be included by wrapping the set of members with the `AddCalculatedMembers()` function or by using an appropriate function argument such as `ALLMEMBERS`.

### Making the query even more generic

The query in this recipe referred to the **Reseller.Reseller** hierarchy. By removing any reference to that hierarchy, we can make the query more generic. That means we should omit the *Reseller* word from the name of the top N set and the name of the calculated member that represents all other members.

We could then parameterize the whole query and insert any hierarchy and its level in it, in their corresponding places. In other words, any reference to the **Reseller.Reseller** hierarchy in front of the **[All]** level could be parameterized as the unique name of a hierarchy while the reference to the **Reseller.Reseller.Reseller** level inside the set could be parameterized as the unique name of a level on that hierarchy. Once such a query is built from the template, it can be executed as a regular query.

Of course, the measure might be parameterized as well, to provide meaningful results for the chosen hierarchy and its level.

## See also

The *Implementing NOT IN set logic*, *Isolating the best N members in a set*, and *Isolating the worst N members in a set* recipes.

# Combining two hierarchies into one

The result of a query contains as many metadata columns as there were hierarchies on rows. For example, if we put two hierarchies on rows, color and size of products, there will be two columns of metadata information, one for each hierarchy. In the first, we will have all colors and all sizes in the second. Depending on the relation between those hierarchies, we will get either a full cross-join for unrelated hierarchies (different dimensions) or a reduced set of valid combinations (in the case of the same dimension). In any case, there will be two columns.

End users don't look at the data that closely. Reports are sometimes very asymmetric, and further, they don't follow the logic of a consistent table. In other words, things get mixed up in the same row or column, although it's clear that the entities don't originate from the same object.

The question is – can this be achieved in MDX? I know you've already guessed it - yes it can. This recipe shows the trick of how to make a report compact by combining two hierarchies in a single metadata column.

## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Date** dimension and its two incompatible hierarchies, months and weeks. We're going to show how to create a report that contains all months up to the "current" month and then how to add the last week of sales in the same column.

Here's the query we'll start from:

```
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    NON EMPTY
    { [Date].[Calendar].[Month].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Calendar Year].&[2008] )
```

Once executed, the query returns values of the **Reseller Sales Amount** measure for six months of the year **2008**:

The screenshot shows the SSMS Results tab with a table titled "Reseller Sales Amount". The table contains six rows, each representing a month from January 2008 to June 2008, along with their corresponding sales amounts.

	Reseller Sales Amount
January 2008	\$1,662,547.32
February 2008	\$2,700,766.80
March 2008	\$2,739,370.98
April 2008	\$2,204,623.41
May 2008	\$3,315,275.00
June 2008	\$3,415,479.07

The data in the **Adventure Works** cube ends with **June 2008** which is the reason why we picked this example to simulate the "current" month in a year situation.

### How to do it...

1. Replace the user hierarchy on rows with the appropriate attribute hierarchy to avoid problems with attribute relations. In this case that would be the **[Date].[Month of Year]** hierarchy.
2. Create a new calculated member in the **[Date].[Month of Year]** hierarchy and name it **Last week**. The definition of that member should include the 23<sup>rd</sup> week, the last week with the data, like this:

```
MEMBER [Date].[Month of Year].[All Periods].[Last week] AS
    ( [Date].[Calendar Week of Year].&[23],
        [Date].[Month of Year].[All Periods] )
```

3. Include that member in rows as well and run the query which should look like this:

```
WITH
MEMBER [Date].[Month of Year].[All Periods].[Last week] AS
    ( [Date].[Calendar Week of Year].&[23],
      [Date].[Month of Year].[All Periods] )
SELECT
    { [Measures].[Reseller Sales Amount] } ON 0,
    NON EMPTY
    { [Date].[Month of Year].[Month of Year].MEMBERS,
      [Date].[Month of Year].[All Periods].[Last week] } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Calendar Year].&[2008] )
```

4. Verify that the result includes the new calculated member in the last row, as in the following screenshot:

	Reseller Sales Amount
January	\$1,662,547.32
February	\$2,700,766.80
March	\$2,739,370.98
April	\$2,204,623.41
May	\$3,315,275.00
June	\$3,415,479.07
Last week	\$3,416,234.85

### How it works...

Hierarchies don't add up in columns, they cross-join and form multiple columns. The trick is to create a calculated member in the initial hierarchy, the one that has more members to be displayed in the result that will point to a member in another hierarchy. That way everything is fine with the dimensionality of the result and we can have a single column for the result because we are using only one hierarchy.

What's important is to provide a tuple, not just a plain reference to the other hierarchy's member. That tuple, visible in the final query above, is formed using the root member of the "host" hierarchy. The reason for this is not very obvious.

Every expression is evaluated in its context. The current member of the **[Date].[Month of Year]** hierarchy is that member itself. As is a calculated member, it has no intersection with the 23<sup>rd</sup> week (or any other week) and hence, the result of the expression without the root member in that tuple would return null. By forcing the root member, we're actually saying we don't want our new member to interfere in the context. We're overriding the calculated member being implicitly there in that expression with the root member of the "host" hierarchy.

One more thing related to the result of the query: the result for the **Last week** member is slightly higher than the result of the **June** member. That might look wrong, but if we analyze what's going on in another query or in the pivot table of the **Cube Browser**, we will notice that the reason for this is that the 23<sup>rd</sup> week spans from **May** to **June**. Since that is also the last week with data, it's no wonder why the value for the week is higher than the value for the month – the month has just started!

### There's more...

The reason why we used an attribute hierarchy and not the initial user hierarchy is because of the attribute relations. The **Date** dimension is a very complex dimension with many related and unrelated attributes. It might be challenging to provide the proper tuple in a case with the original user hierarchy on rows because of all the relations between members in slicer, other axes, and a calculated member being defined. An hierarchy, on the other hand, requires that we use its root member only inside the tuple.

You should always look for a way to combine attribute hierarchies whenever you're in a situation which requires that you combine two of them in one column. The color and the size of products is another example.

### **Use it, but don't abuse it**

If it has been explicitly stated that the report should combine different hierarchies in a single column, we can reach for this solution. It will be rare, but a possible case. In all other cases a much better solution is to have each hierarchy in its own column.

### Limitations

Besides the already-mentioned limitation with user hierarchies, there is another one. When more than one member should be "hosted" in the other hierarchy, all of them should be defined as calculated members, one by one. This can be an administrative burden and so it is advised that this solution is used only in cases with few members to be projected on the other hierarchy. Naturally, the hierarchy with more elements to be shown in report should be the "hosting" one.

## Finding the name of a child with the best/worst value

Sometimes there is a need to perform a for-each loop in order to get the top or bottom members in the inner hierarchy for each member in the outer hierarchy. The recipe *Identifying the best/worst members for each member of another hierarchy* deals with exactly that kind of a topic.

In this recipe, we'll show another possibility. We'll demonstrate how to identify the member with the best/worst value, only this time we are not going to return the member itself. This time, we are going to return only its name in a calculated measure.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension. Here's the query we'll start from:

```
WITH
MEMBER [Measures].[Subcategory] AS
    iif( IsEmpty([Measures].[Internet Sales Amount]) ,
        null,
        [Product].[Product Categories]
            .CurrentMember.Parent.Name )
SELECT
    { [Measures].[Subcategory],
        [Measures].[Internet Sales Amount] } ON 0,
    NON EMPTY
    { Descendants( [Product].[Product Categories].[Category],
                    2, SELF_AND_BEFORE ) } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns the values of the **Internet Sales Amount** measure for every product and subcategory. The additional measure serves as an indicator in which part of the hierarchy of each member can be located. That can be used later for verification purposes.

## How to do it...

1. Modify the query so that it returns only subcategories on rows. In other words, specify 1 and SELF as arguments of the Descendants( ) function.
  2. Remove the first calculated measure from the query.
  3. Define a new calculated member and name it **Best child**. Its definition should test whether we're on a leaf member or not. If so, we should provide null.
- If not, we should again test whether the **Internet Sales Amount** is null and make sure the result of the calculated member matches that. Otherwise, we should calculate the top 1 child based on the measure **Internet Sales Amount**, and return the name of that child.
4. Include that calculated measure on columns as the second measure.
  5. The final query should look like this:

```

WITH
MEMBER [Measures].[Best child] AS
    iif( IsLeaf( [Product].[Product Categories]
        .CurrentMember ),
        null,
        iif( IsEmpty([Measures].[Internet Sales Amount] ),
            null,
            TopCount( [Product].[Product Categories]
                .CurrentMember.Children,
                1, [Measures].[Internet Sales Amount]
            ).Item(0).Name
        )
    )
SELECT
{ [Measures].[Internet Sales Amount],
[Measures].[Best child] } ON 0,
NON EMPTY
{ Descendants( [Product].[Product Categories].[Category],
1, SELF ) } ON 1
FROM
[Adventure Works]

```

## Concise Reporting

---

6. The results will look like this:

	Internet Sales Amount	Best child
Bike Racks	\$39,360.00	Hitch Rack - 4-Bike
Bike Stands	\$39,591.00	All-Purpose Bike Stand
Bottles and Cages	\$56,798.19	Water Bottle - 30 oz.
Cleaners	\$7,218.60	Bike Wash - Dissolver
Fenders	\$46,619.58	Fender Set - Mountain
Helmets	\$225,335.60	Sport-100 Helmet, Red
Hydration Packs	\$40,307.67	Hydration Pack - 70 oz.
Tires and Tubes	\$245,529.32	HL Mountain Tire
Mountain Bikes	\$9,952,759.56	Mountain-200 Black, 42
Road Bikes	\$14,520,584.04	Road-150 Red, 48
Touring Bikes	\$3,844,801.05	Touring-1000 Blue, 46
Caps	\$19,688.10	AWC Logo Cap
Gloves	\$35,020.70	Half-Finger Gloves, M
Jerseys	\$172,950.68	Long-Sleeve Logo Jersey, L
Shorts	\$71,319.81	Women's Mountain Shorts, L
Socks	\$5,106.32	Racing Socks, M
Vests	\$35,687.00	Classic Vest, M

## How it works...

Leaf members don't have children. That's why we provided a branch in the definition of the calculated member and eliminated them from the start. In the case of a non-leaf member, a single child with the highest value is returned using the TopCount( ) function. Actually, the child's name, to be precise.

The inner iif( ) function took care of empty values and preserved them as empty whenever the initial measure, the **Internet Sales Amount** measure, was null. This way the NON EMPTY operator was able to exclude the same number of empty rows as in the initial query.

## There's more...

Now that we have the name of the best child, we can include additional information.

For example, the following query shows how to display the child's value as well as its percentage:

```
WITH  
MEMBER [Measures].[Best child] AS  
    iif( IsLeaf( [Product].[Product Categories]  
        .CurrentMember ),
```

```

        null,
        iif( IsEmpty([Measures].[Internet Sales Amount] ),
            null,
            TopCount( [Product].[Product Categories]
                .CurrentMember.Children,
                1, [Measures].[Internet Sales Amount]
            ).Item(0).Name
        )
    )
)
MEMBER [Measures].[Best child value] AS
    iif( IsLeaf( [Product].[Product Categories]
        .CurrentMember ),
        null,
        iif( IsEmpty([Measures].[Internet Sales Amount] ),
            null,
            ( TopCount( [Product].[Product Categories]
                .CurrentMember.Children,
                1, [Measures].[Internet Sales Amount]
            ).Item(0),
            [Measures].[Internet Sales Amount] )
        )
    )
    ,
    FORMAT_STRING = 'Currency'
)
MEMBER [Measures].[Best child %] AS
    [Measures].[Best child value]
    /
    [Measures].[Internet Sales Amount]
    ,
    FORMAT_STRING = 'Percent'
)
SELECT
    { [Measures].[Internet Sales Amount],
        [Measures].[Best child],
        [Measures].[Best child value],
        [Measures].[Best child %] } ON 0,
    NON EMPTY
    { Descendants( [Product].[Product Categories].[Category],
        1, SELF ) } ON 1
FROM
    [Adventure Works]

```

A child's value is obtained on the same principle as the child's name, except this time the tuple was used in order to get the value in that particular coordinate.

The percentage is calculated in a standard way.

## Concise Reporting

---

Once executed, the query from above returns this result:

	Internet Sales Amount	Best child	Best child value	Best child %
Bike Racks	\$39,360.00	Hitch Rack - 4-Bike	\$39,360.00	100.00%
Bike Stands	\$39,591.00	All-Purpose Bike Stand	\$39,591.00	100.00%
Bottles and Cages	\$56,798.19	Water Bottle - 30 oz.	\$21,177.56	37.29%
Cleaners	\$7,218.60	Bike Wash - Dissolver	\$7,218.60	100.00%
Fenders	\$46,619.58	Fender Set - Mountain	\$46,619.58	100.00%
Helmets	\$225,335.60	Sport-100 Helmet, Red	\$78,027.70	34.63%
Hydration Packs	\$40,307.67	Hydration Pack - 70 oz.	\$40,307.67	100.00%
Tires and Tubes	\$245,529.32	HL Mountain Tire	\$48,860.00	19.90%
Mountain Bikes	\$9,952,759.56	Mountain-200 Black, 42	\$979,960.73	9.85%
Road Bikes	\$14,520,584.04	Road-150 Red, 48	\$1,205,876.99	8.30%
Touring Bikes	\$3,844,801.05	Touring-1000 Blue, 46	\$421,980.39	10.98%
Caps	\$19,688.10	AWC Logo Cap	\$19,688.10	100.00%
Gloves	\$35,020.70	Half-Finger Gloves, M	\$12,220.51	34.90%
Jerseys	\$172,950.68	Long-Sleeve Logo Jersey, L	\$22,595.48	13.06%
Shorts	\$71,319.81	Women's Mountain Shorts, L	\$25,406.37	35.62%
Socks	\$5,106.32	Racing Socks, M	\$2,679.02	52.46%
Vests	\$35,687.00	Classic Vest, M	\$12,636.50	35.41%

## Variations on a theme

Using the same principle, it is possible to get the member with the worst value. We have to be careful and apply `NonEmpty()` first in order to ignore empty values, as explained in the recipe *Isolating the worst N members in a set*.

### Displaying more than one member's caption

It is possible to display several names inside the same cell. If that is the case, we must use the `Generate()` function in conjunction with its third syntax, to start the iteration. Once the iteration is in place, everything else remains the same.

### See also

- ▶ *Isolating the best N members in a set*
- ▶ *Isolating the worst N members in a set*
- ▶ *Identifying the best/worst members for each member of another hierarchy*
- ▶ *Displaying a few important members, others as a single row, and the total at the end*

## Highlighting siblings with the best/worst values

Data analysis becomes easier once we provide more information than a simple black and white grid allows us. One way of doing this is to color-code some cells. In this recipe, we'll show how to highlight the cells with the minimum and the maximum values among siblings and how to color-code them based on their values relative to their siblings' values.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Internet Sales Amount] } ON 0,
    NON EMPTY
    { Descendants( [Product].[Product Categories].[Category],
        1, SELF ) } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns values of the **Internet Sales Amount** measure for every single product subcategory.

### How to do it...

1. Define a calculated measure that will show the name of the parent for each subcategory. Name it **Category**.
2. Be sure to provide the `null` value whenever the initial measure is `null` and then include that measure on columns as well, as the first of the two.
3. Define a cell calculation for the **Internet Sales Amount** measure and name it **Highlighted Amount**.
4. Define the `BACK_COLOR` property for the **Highlighted Amount** cell calculation. Use an expression that tests whether the current value is a max/min value.
5. Provide the adequate RGB values: green for max, red for min, and null for everything else.
6. Include the `CELL PROPERTIES` required to display color information of a cell in the end of the query.

7. When everything is done, the query should look like this. Verify!

```
WITH
MEMBER [Measures].[Category] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ), 
        null,
        [Product].[Product Categories]
        .CurrentMember.Parent.Name )
CELL CALCULATION [Highlighted Amount]
FOR '{ [Measures].[Internet Sales Amount] }' AS
    [Measures].[Internet Sales Amount]
    , BACK_COLOR =
        iif( [Measures].CurrentMember =
            Max( [Product].[Product Categories]
            .CurrentMember.Siblings,
            [Measures].CurrentMember ),
            RGB(128,242,128), // green
        iif( [Measures].CurrentMember =
            Min( [Product].[Product Categories]
            .CurrentMember.Siblings,
            [Measures].CurrentMember ),
            RGB(242,128,128), // red
            null )
        )
SELECT
    { [Measures].[Category],
        [Measures].[Colored Amount] } ON 0,
    NON EMPTY
    { Descendants( [Product].[Product Categories].[Category],
        1, SELF ) } ON 1
FROM
    [Adventure Works]
CELL PROPERTIES
    VALUE,
    FORMATTED_VALUE,
    FORE_COLOR,
    BACK_COLOR
```

8. Once executed, the query returns the result presented on the following screenshot:

	Category	Internet Sales Amount
Bike Racks	Accessories	\$39,360.00
Bike Stands	Accessories	\$39,591.00
Bottles and Cages	Accessories	\$56,798.19
Cleaners	Accessories	\$7,218.60
Fenders	Accessories	\$46,619.58
Helmets	Accessories	\$225,335.60
Hydration Packs	Accessories	\$40,307.67
Tires and Tubes	Accessories	\$245,529.32
Mountain Bikes	Bikes	\$9,952,759.56
Road Bikes	Bikes	\$14,520,584.04
Touring Bikes	Bikes	\$3,844,801.05
Caps	Clothing	\$19,688.10
Gloves	Clothing	\$35,020.70
Jerseys	Clothing	\$172,950.68
Shorts	Clothing	\$71,319.81
Socks	Clothing	\$5,106.32
Vests	Clothing	\$35,687.00

9. Notice that the highest value in a category is highlighted with light green and the lowest value per category is highlighted with light red. That's the visual cue we're after.

### How it works...

As explained in the recipe *Applying conditional formatting on calculations*, a recipe which can be found in the first chapter, the `BACK_COLOR` property controls the color of the background of a cell. It can be defined as a constant value, but it can also be an expression. In this case, we used a combination – conditional formatting using fixed values.

In order to help ourselves, we've included additional information in the form of another measure that will mark siblings and therefore enable us to verify if the calculation works correctly.

We had to use the `iif( )` function in the definition of that measure because it evaluates as a string, a string which is never null. Therefore we've bound it to the original measure so that it returns values only for rows with data.

The `Max()` and `Min()` functions return the highest and the lowest value in the specified set of sibling members. The current measure's value is compared to the maximum and minimum values respectively. In case of a match, the appropriate color is chosen (green for the max and red for the min values).

Finally, because the measure already existed, we used the `CELL CALCULATION` syntax to define (or overwrite) additional properties for it, such as the `BACK_COLOR` property in this example. The `CELL CALCULATION` is one of the three elements that can be defined in the `WITH` part of the query. It is analogous to the `SCOPE` statement inside the MDX script. In other words, it can be used to define or overwrite a value inside a particular subcube or to define or overwrite properties such as `BACK_COLOR`.

### There's more...

It is also possible to color-code any measure in a different way. For example, it is possible to provide a range of colors, so that the cell with the highest value has one color, the cell with the lowest value another color, and all in between a gradient of those colors. See for yourself by running this query and observing the result afterwards:

```
WITH
MEMBER [Measures].[Category] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null,
        [Product].[Product Categories]
            .CurrentMember.Parent.Name )
MEMBER [Measures].[Rank in siblings] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null,
        Rank( [Product].[Product Categories].CurrentMember,
            NonEmpty( [Product].[Product Categories]
                .CurrentMember.Siblings,
                [Measures].[Internet Sales Amount] ),
            [Measures].[Internet Sales Amount] )
        )
MEMBER [Measures].[Count of siblings] AS
    Sum( [Product].[Product Categories]
        .CurrentMember.Siblings,
        iif( IsEmpty( [Measures].[Internet Sales Amount] ),
            null, 1 )
    )
MEMBER [Measures].[R] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null,
        255 / ( [Measures].[Count of siblings] - 1 ) *
        ( [Measures].[Count of siblings] -
```

```
[Measures].[Rank in siblings] ) ) -- all shades
, FORMAT_STRING = '#,#'
, VISIBLE = 1
MEMBER [Measures].[G] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null, 0 ) -- fixed dark green
    , VISIBLE = 1
MEMBER [Measures].[B] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null,
        100 / [Measures].[Count of siblings] *
        [Measures].[Rank in siblings] ) -- dark shades
    , FORMAT_STRING = '#,#'
    , VISIBLE = 1
CELL CALCULATION [Highlighted Amount]
FOR '{ [Measures].[Internet Sales Amount] }' AS
    [Measures].[Internet Sales Amount]
    , BACK_COLOR =
        RGB( [Measures].[R],
            [Measures].[G],
            [Measures].[B] )
    , FORE_COLOR = RGB( 255, 255, 255 ) -- white
SELECT
    { [Measures].[Category],
        [Measures].[Rank in siblings],
        [Measures].[Internet Sales Amount],
        [Measures].[R],
        [Measures].[G],
        [Measures].[B] } ON 0,
NON EMPTY
    { Descendants( [Product].[Product Categories].[Category],
        1, SELF ) } ON 1
FROM
    [Adventure Works]
CELL PROPERTIES
    VALUE,
    FORMATTED_VALUE,
    FORE_COLOR,
    BACK_COLOR
```

## Concise Reporting

---

The previous query, once executed, returns the result presented in the following screenshot:

The screenshot shows a reporting application window with two tabs at the top: 'Messages' and 'Results'. The 'Results' tab is selected and displays a table of data. The table has columns: Product Name, Category, Rank in siblings, Internet Sales Amount, and three calculated measures R, G, and B. The data is sorted by Internet Sales Amount in descending order. The background color of each row is determined by the R, G, and B values, creating a gradient effect.

	Category	Rank in siblings	Internet Sales Amount	R	G	B
Bike Racks	Accessories	7	\$39,360.00	36	0	88
Bike Stands	Accessories	6	\$39,591.00	73	0	75
Bottles and Cages	Accessories	3	\$56,798.19	182	0	38
Cleaners	Accessories	8	\$7,218.60	0	0	100
Fenders	Accessories	4	\$46,619.58	146	0	50
Helmets	Accessories	2	\$225,335.60	219	0	25
Hydration Packs	Accessories	5	\$40,307.67	109	0	63
Tires and Tubes	Accessories	1	\$245,529.32	255	0	13
Mountain Bikes	Bikes	2	\$9,952,759.56	128	0	67
Road Bikes	Bikes	1	\$14,520,584.04	255	0	33
Touring Bikes	Bikes	3	\$3,844,801.05	0	0	100
Caps	Clothing	5	\$19,688.10	51	0	83
Gloves	Clothing	4	\$35,020.70	102	0	67
Jerseys	Clothing	1	\$172,950.68	255	0	17
Shorts	Clothing	2	\$71,319.81	204	0	33
Socks	Clothing	6	\$5,106.32	0	0	100
Vests	Clothing	3	\$35,687.00	153	0	50

The **Category** calculated measure is the same as in the previous query.

**Rank in siblings** is a measure that returns the rank of an individual subcategory when compared to the results of its siblings. This measure is included in the columns so that the verification becomes relatively easy.

**Count of siblings** is a measure which returns the number of sibling members and is used to establish the boundaries as well as the increment in the color gradient.

The R, G, and B values are calculated measures used to define the background color for the **Internet Sales Amount** measure. The gradient is calculated using the combination of rank and count measures with additional offsets so that the colors look better (keeping them relatively dark). In addition to that, the `FORE_COLOR` property is set to white so that proper contrast is preserved. The three calculated measures are displayed last in the result just to show how the values change. You can freely hide those measures and remove them from columns axis. The `VISIBLE` property is there to remind you about that.

It's worth mentioning that one component of the color should decrease as the other increases. Here it is implemented with the rank and count-rank expressions. As one increases, the other decreases.

For a more complex color-coding, an appropriate stored procedure installed on the SSAS server is another solution unless the front-end and its pivot table grid already supports these features.

### Troubleshooting

Don't forget to include the required cell properties, namely BACK\_COLOR and FORE\_COLOR, or the recipe solution won't work. The same applies for the front-end you're using. If you don't see the effect, check whether there's an option to turn those cell properties on.

### See also

*Applying conditional formatting on calculations*, a recipe in the first chapter, and the next recipe, *Implementing bubble-up exceptions*.

## Implementing bubble-up exceptions

In the previous recipe we dealt with highlighting the cells based on their results in comparison with sibling members which can be visualized in a horizontal direction. In this recipe we'll take a look at how to do the same but in vertical direction using the descendants of members in the report.

**Bubble-up** exceptions are a nice way of visualizing the information about descendants of a member without making reports huge. By coding the information about the result of descendants, we can have compact reports on a higher level while still having some kind of information about what's going on below.

The information we are going to "bubble-up" will be presented in the form of color-coding the cells.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Reseller Sales Amount],
      [Measures].[Reseller Gross Profit Margin] } ON 0,
    NON EMPTY
```

```
{ Descendants( [Product].[Product Categories].[Category],  
    1, BEFORE ) } ON 1  
FROM  
    [Adventure Works]  
WHERE  
    ( [Date].[Fiscal].[Fiscal Year].&[2007] )
```

Once executed, the query returns values of the **Reseller Sales Amount** and **Reseller Gross Profit Margin** measures in the fiscal year **2007** for each product category.

## How to do it...

1. Create a new calculated measure that will serve as an alias for the **Reseller Gross Profit Margin** measure. Name it **Margin with Bubble-up**.
2. Define the **FORE\_COLOR** property of that measure so that it turns red when at least one descendant on the lower level has negative values for the margin.
3. Include that new measure on columns, as the third measure there.
4. Include the **CELL PROPERTIES** part of the query and enlist the **FORE\_COLOR** property as one of them to be returned by the query.
5. Execute the query, which should look like this:

```
WITH  
MEMBER [Measures].[Margin with Bubble-up] AS  
    [Measures].[Reseller Gross Profit Margin]  
    , FORE_COLOR =  
        iif(  
            Min( Descendants( [Product].[Product Categories]  
                .CurrentMember,  
                1, SELF ),  
            [Measures].CurrentMember ) < 0,  
            RGB( 255, 0, 0 ), -- red  
            null  
        )  
SELECT  
    { [Measures].[Reseller Sales Amount],  
        [Measures].[Reseller Gross Profit Margin],  
        [Measures].[Margin with Bubble-up] } ON 0,  
NON EMPTY  
    { Descendants( [Product].[Product Categories].[Category],  
        1, BEFORE ) } ON 1  
FROM  
    [Adventure Works]  
WHERE  
    ( [Date].[Fiscal].[Fiscal Year].&[2007] )
```

## CELL PROPERTIES

```
VALUE,
FORMATTED_VALUE,
FORE_COLOR,
BACK_COLOR
```

6. Verify that the new measure is red in two rows, **Bikes** and **Clothing**:

	Reseller Sales Amount	Reseller Gross Profit Margin	Margin with Bubble-up
Accessories	\$124,433.35	29.23%	29.23%
Bikes	\$22,417,419.69	1.64%	1.64%
Clothing	\$750,716.33	23.46%	23.46%
Components	\$4,629,101.14	12.40%	12.40%

## How it works...

As mentioned earlier, the new measure is just an alias for the real one; there's nothing special in its definition. Its real value is in the additional expressions, namely, the `FORE_COLOR` property used in this example. Speaking of which, it could have been the `BACK_COLOR` property as well, that's totally up to us to choose which looks better in any particular case.

In that expression, we're analyzing descendants of the current member and extracting only the ones with a negative result for the current measure, the **Margin with Bubble-up** measure. As that measure is an alias for the **Reseller Gross Profit Margin** measure, we are actually testing the latter measure.

Once we've isolated descendants with negative values, we look for the minimal value. If that's a negative number, we turn the color to red. If not, we leave it as is.

## There's more...

The construct on rows is just another way of specifying that we want to see product categories there. That construct allows us to make a small change in the query, which can be a parameter of the report, and to drill one level below.

The part that needs a change is the second argument of the `Descendants( )` function on rows. The value of that argument was **1**, signaling that we want only product categories and nothing else.

## Concise Reporting

---

If we change that argument to 2, we will get the result as displayed in the following screenshot:

The screenshot shows a reporting interface with a 'Messages' tab and a 'Results' tab. The 'Results' tab is active and displays a table with four columns: 'Reseller Sales Amount', 'Reseller Gross Profit Margin', and 'Margin with Bubble-up'. The table lists various product categories and subcategories, including Accessories, Helmets, Locks, Pumps, Bikes, Mountain Bikes, Road Bikes, Clothing, Bib-Shorts, Caps, Gloves, Jerseys, Shorts, Tights, Components, Forks, Handlebars, Headsets, and Mountain Frames. The 'Margin with Bubble-up' column uses color-coding to highlight negative values in red, such as '1.64%' for Bikes and several other categories.

	Reseller Sales Amount	Reseller Gross Profit Margin	Margin with Bubble-up
Accessories	\$124,433.35	29.23%	29.23%
Helmets	\$94,693.44	28.67%	28.67%
Locks	\$16,225.22	30.98%	30.98%
Pumps	\$13,514.69	31.05%	31.05%
Bikes	\$22,417,419.69	1.64%	1.64%
Mountain Bikes	\$9,184,858.57	9.40%	9.40%
Road Bikes	\$13,232,561.12	-3.75%	-3.75%
Clothing	\$750,716.33	23.46%	23.46%
Bib-Shorts	\$166,739.71	30.74%	30.74%
Caps	\$11,698.80	-3.00%	-3.00%
Gloves	\$155,267.18	27.15%	27.15%
Jerseys	\$133,279.01	-2.22%	-2.22%
Shorts	\$81,898.62	30.93%	30.93%
Tights	\$201,833.01	30.08%	30.08%
Components	\$4,629,101.14	12.40%	12.40%
Forks	\$77,931.69	25.90%	25.90%
Handlebars	\$77,104.06	25.96%	25.96%
Headsets	\$60,942.20	25.35%	25.35%
Mountain Frames	\$1,728,121.66	11.18%	11.18%

In other words, now the query includes both product categories and product subcategories on rows.

What's good about it is that our measure reacts in this scenario as well. Several subcategories are in red. At the same time, they have negative values which signal two things: not only that some of the descendants are negative, but that the member itself is negative.

The reason this worked is because our color-coding expression used a relative depth for descendants, which can be seen in the second argument of the `Descendants( )` function, only this time it is the function in the `FORE_COLOR` property, not the one on rows. The value of that argument was **1**.

If it is required, we can specify an absolute level, not a relative one. We would do this by specifying the name of a particular level instead of the number. For example, if we want our expression to detect negative values only in the subcategories, we should specify it like this:

```
Descendants( [Product].[Product Categories].CurrentMember,  
              [Product].[Product Categories].[Subcategory],  
              SELF )
```

If we want the expression to work on the lowest level, we should use this construct:

```
Descendants( [Product].[Product Categories].CurrentMember, ,  
              LEAVES )
```

In short, there are a lot of possibilities and you're free to experiment.

### Practical value of bubble-up exceptions

Members colored in red signal that some of the members on the lower level have negative margins. This way, end users can save a lot of their precious time because they are guided where to drill next instead of descending on a whole level below. Or worse, not descending at all, believing everything is fine, because it might look that way when we are on higher levels.

### Potential problems

On very large hierarchies, there can be problems with performance if the bubble-up exceptions are set too low.

### See also

A recipe from the first chapter, *Applying conditional formatting on calculations*, and the preceding recipe, *Highlighting siblings with the best/worst values*.



# 4

# Navigation

In this chapter, we will cover:

- ▶ Detecting a particular member in a hierarchy
- ▶ Detecting the root member
- ▶ Detecting members on the same branch
- ▶ Finding related members in the same dimension
- ▶ Finding related members in another dimension
- ▶ Calculating various percentages
- ▶ Calculating various averages
- ▶ Calculating various ranks

## Introduction

One of the advantages of multidimensional cubes is their rich metadata model backed up with a significant number of MDX functions that enable easy navigation and data retrieval from any part of the cube. We can navigate on levels, hierarchies, dimensions, and cubes. The idea of this chapter is to show common tasks related to navigation and data retrieval.

Some of the examples illustrate how to test whether the current context is the one we're expecting or not. Others build on that and continue by providing a relative calculation that takes the current context and compares its value to some other context, usually a related one. Examples of such calculations are percentage of parent, percentage of total, rank among siblings, rank on a level, and so on.

Multidimensional cubes are conceptually enormous structures filled with empty space, with no data at all in almost all but a few combinations. There are times when the Analysis Services engine takes care of that by utilizing various algorithms that compact the cube space, but there are times when we have to do that by ourselves.

The MDX language incorporates functions that enable fast retrieval of related members, to move from one hierarchy to another, from one dimension to another, and to get only members valid in the current context. This technique improves query performance because the engine is not forced to calculate on non-relevant space. There are several recipes in this chapter covering that topic.

Let's start!

## Detecting a particular member in a hierarchy

The instructions that follow show how to test if the current context is pointing to a member of our interest. This situation is frequently encountered. Whether there's a need to include or exclude a certain member in the calculation, to perform a calculation for that member only or for the rest of them, this recipe should provide a good enough guideline on how to start.

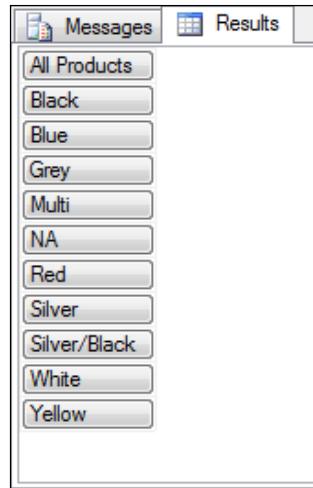
### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension. Here's the query we'll start from:

```
SELECT
    { } ON 0,
    { [Product].[Color].AllMembers } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns all product colors including the root member. The type of query that has nothing on columns is explained in Chapter 1, in the recipe *Skipping axis*.



Our task is to detect the **NA** member. Once we detect it, we can provide various calculations, as explained in the introduction, but that's outside of the scope of this recipe. Here we're focusing only on how to perform the detection.

### How to do it...

Follow these steps to detect the **NA** member:

1. Add the **WITH** block of the query.
2. Create a new calculated measure and name it **Member is detected**.
3. Define it as **True** for the **NA** member and **null**, **not False**, in all other cases.
4. Add this calculated measure on axis 0.
5. Execute the query which should look like this:

```

WITH
MEMBER [Measures].[Member is detected] AS
    iif( [Product].[Color].CurrentMember Is
        [Product].[Color].&[NA],
        True,
        null
    )
SELECT
    { [Measures].[Member is detected] } ON 0,
    { [Product].[Color].AllMembers } ON 1
FROM
    [Adventure Works]
  
```

## Navigation

---

6. Verify that the result matches the following image:

	Member is detected
All Products	(null)
Black	(null)
Blue	(null)
Grey	(null)
Multi	(null)
NA	True
Red	(null)
Silver	(null)
Silver/Bla...	(null)
White	(null)
Yellow	(null)

## How it works...

The standard part of query execution is the iteration on axes. In that phase we can detect the current context on an axis in the query using the `CurrentMember` function. That function returns the member we're currently on in a particular hierarchy. By testing the current member with a particular member of the same hierarchy, we can know when we've hit the row or column with that member in it.

## There's more...

The solution presented here is good in situations when there is a **temporary** need to isolate a particular member in the query. The emphasis is on the word "temporary." Typical examples include highlighting the rows or columns of a report or providing two calculations for a single measure based on the context.

In case the required behavior has a more **permanent** character, defining the member in MDX script or using the `Scope( )` statement are better approaches.

Take a look at the following script which can be added in the MDX script of the Adventure Works cube:

```
Create Member CurrentCube.[Measures].[Member is detected]
As null;

Scope( [Product].[Color].&[NA],
```

---

```
[Measures].[Member is detected] ) );
This = True;
End Scope;
```

First, the measure is defined as `null`. Then, the scope is applied to it in the context of **NA** color. When the scope becomes active, it will provide `True` as the result. In all other cases, the initial definition of the measure will prevail.

The `Scope()` statement is basically doing the same thing as the `iif()` statement in a calculated member, with one important difference – the scope statement itself is evaluated (resolved) only once, when the first user connects to the cube. That goes for named sets and left side of assignments too, which can easily be verified by observing the *Execute MDX Script Begin/End* events in SQL Server Profiler. More about this is in chapter 9, in the recipe *Capturing MDX queries generated by SSAS front-ends*.

The right side of assignments (here, the value `True`) is evaluated at query time and hence it makes no difference if the expression on that side was used in an MDX script or in a query. However, as the scope statement gets evaluated only once, it may be reasonable to put a complex expression in scope instead of `iif()` function.

### Important remarks

The calculation didn't use the boolean value `False` as the result of the negative branch. Instead, the value of `null` was provided in order to keep the calculation sparse and hence preserve performance. The `iif()` function is optimized to work in block mode if one of the branches is `null`.

### An indicator versus the final calculation

We could have used a numerical or textual value for the `True` part of the branch. However, we used the expression which evaluates to a boolean value. This way we don't have to compare the value of the indicator to that numerical or textual value in the subsequent calculations. We can simply refer to the indicator measure itself since its value will be of a boolean type.

Is the indicator measure really necessary? No, we can apply the required calculations for both branches of the `iif()` statement directly in the initial calculation. The criteria should be the reusability of that indicator. If it is going to be used in more than one of the following calculations, it makes sense to create one. Otherwise you can skip creating the indicator and go directly with calculations for both branches.

### Comparing members versus comparing values

It is important to differentiate the comparison of members from the comparison of values. The first is performed using the `Is` keyword, the second using the `=` sign. It is essential that you learn and understand that difference. Otherwise, you'll experience problems with your queries.

## Navigation

---

Additionally, compare members directly using their unique names whenever you can. Don't compare their properties like name or similar. In other words, the following condition will work, but it will **not** give you the best possible performance:

```
[Product].[Color].CurrentMember.Name = 'NA'
```

The other important difference is that names can repeat, especially in multi-level user hierarchies. For example, the **Geography**.**Geography** hierarchy has **New York** the state and **New York** the city. Obviously, comparing that member by its name would be a bad choice.

## Detecting complex combination of members

When the business logic is complex it might be required to detect several members, not just one of them. In that case apply the same principles described in this recipe. Your only concern in that case is to handle the logic correctly. The MDX language offers various logical functions for that scenario.

In the case of OR logic, here are few additional hints:

- ▶ You can define a set of members and use the `Intersect()` function to test whether the set formed from the current member has intersection with the predefined set.
- ▶ In case of poor performance, you might want to consider creating a new attribute hierarchy based on a Yes/No value in the field derived using the case statement in your DW/DSV, as explained in chapter 6, recipe Using a new attribute to separate members on a level. This way you can have pre-aggregated values for those two members.
- ▶ In case of a very complex logic you'd be better off by defining a new column in your fact table and creating a dimension from it. That way you are pushing the logic in DW and using SSAS cubes for what they do best – slicing and aggregation of the data.

## See also

The recipe *Detecting the root member* covers a similar topic, but in a much more narrow case. It is worth reading right after this recipe because of the additional insights it provides.

## Detecting the root member

There are times when we need to know whether the current context is pointing to the root member of a hierarchy or not.

Root member is the topmost member of a hierarchy. It is present in all hierarchies (in user hierarchies as well as in attributes hierarchies) as long as the `IsAggregatable` property is in its default state, enabled.

This recipe shows how to detect the root member.

## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the same **Color** hierarchy of the **Product** dimension like in the previous recipe, *Detecting a particular member of a hierarchy*. Here's that query:

```
SELECT
    { } ON 0,
    { [Product].[Color].AllMembers } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns all product colors including the root member. This special type of query is explained in *Chapter 1*, the recipe *Skipping axis*.

Our task is to detect the root member got repeated from the previous chapter, makes no sense to have it here.

## How to do it...

Follow these steps to create a calculated member that detects the root member:

1. Add the **WITH** block of the query.
2. Create a new calculated measure and name it **Root member detected**.
3. Define it as **True** for the branch where the detection occurs and **null** for the other part. The condition should look like this:
4. Add this measure on axis 0.
5. Execute the query which should look like this:

```
WITH
MEMBER [Measures].[Root member detected] AS
    iif( [Product].[Color].CurrentMember Is
        [Product].[Color].[All Products],
        True,
        null
    )
SELECT
    { [Measures].[Root member detected] } ON 0,
    { [Product].[Color].AllMembers } ON 1
FROM
    [Adventure Works]
```

## Navigation

---

6. Verify that the result matches the following image:

Messages	Results
	Root member detected
All Products	True
Black	(null)
Blue	(null)
Grey	(null)
Multi	(null)
NA	(null)
Red	(null)
Silver	(null)
Silver/Black	(null)
White	(null)
Yellow	(null)

## How it works...

The `CurrentMember` function detects the member we're currently on in a particular hierarchy, so that it can also be used to detect the root member.

The condition in our expression evaluates as the boolean value `True` when it is met. In all other cases, the returned value is `null`.

## There's more...

The name of the root member can change in time. On the other hand, we might forget to adjust that in our calculations. But the solution exists.

Instead of using the reference to the root member, we can use the reference to its internal alias. Here's the syntax:

```
MEMBER [Measures].[Root member detected] AS
    iif( [Product].[Color].CurrentMember Is
        [Product].[Color].[All],
        True,
        null
    )
```

It may not be so obvious but we changed the name of the root member and our calculation still works. That's because we used the alias for the root member which is the name `All`. Anything other than the correct name of the root member and its alias will result in an error. Depending on a particular configuration of SSAS cube, the error will be reported or bypassed. Mostly it's the second case, because it's a default option.

This trick will work on any hierarchy, but may not work in future SSAS versions based on the fact that it isn't documented anywhere.

There are, of course, other ways to detect the root member. One is to test the level ordinal of the current member, like this:

```
MEMBER [Measures].[Root member detected] AS
    iif( [Product].[Color].CurrentMember.Level.Ordinal = 0,
        True,
        null
    )
```

If the level ordinal (internal position starting from 0 to  $n$ , where  $n$  is the number of user-defined levels in that hierarchy) is zero, then the current member is located on the topmost level.

A potential problem with this calculation is that any calculated member defined on the hierarchy itself (not on another regular member) will also be positioned on the topmost level. Its level ordinal will be zero too. However if you exclude calculated members, the expression will work.

And here's yet another expression for detecting the topmost regular member:

```
MEMBER [Measures].[Root member detected] AS
    iif( [Product].[Color].CurrentMember Is
        Extract(Root([Product]),
        [Product].[Color]).item(0),
        True,
        null
    )
```

The idea with the last expression is that it uses the `Root()` function which manifests in a tuple formed from regular top-level members of all hierarchies of the dimension used as the argument of that function. The `Extract()` function isolates a single hierarchy and its members in that tuple. The `Item()` function converts the set of members into a single member. That's the topmost regular member of that hierarchy extracted from the tuple.

### The scope-based solution

In case the required behavior has a permanent character, defining the calculation in MDX script, either using the `Scope()` statement or using the **CREATE MEMBER** statement only, is a better approach than the query-based `iif()` statement.

The following script is equivalent to the initial example. Define it in the MDX script of the Adventure Works cube using BIDS, deploy, and then verify its result in the cube browser using the same hierarchies as in the initial example:

```
Create Member CurrentCube.[Measures].[Root member detected]
As null;

Scope( ([Product].[Color].[All],
        [Measures].[Root member detected] ) );
    This = True;
End Scope;
```

One more thing: detection of this kind is usually done so that another calculation can exploit it. It is possible to directly specify the scope for an existing measure and to provide the calculation for it. Here's an example that returns the average value for all colors as the value of the root member:

```
Scope( ([Product].[Color].[All],
        [Measures].[Internet Sales Amount] ) );
    This = Avg( [Product].[Color].[All].Children,
                [Measures].CurrentMember );
End Scope;
```

### See also

The recipe *Detecting a particular member of a hierarchy*.

## Detecting members on the same branch

So far we've covered cases when there is a need to isolate a single member in the hierarchy, be it a root member or any other member in the hierarchy. This recipe deals with detecting whether the ascendants or descendants of the current member are on a particular cascade or branch in that hierarchy.

Let's illustrate this with a few examples.

Suppose we want to analyze dates. There's a concept of the current date, but since that date is usually found in a multilevel user hierarchy, we can also talk about the current month, current quarter, current year, and so forth.

Another example is the regional customer hierarchy. There we can detect whether a particular customer is living in a particular city, or if that city is in the state or country in the context. In other words, we can test above and below, as long as there are levels in each direction.

In this recipe we're going to demonstrate how to check if a particular member in a middle level is on the selected drill-up/down part in the hierarchy.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Double-click the **Adventure Works** cube and go to **Calculations** tab. Choose **Script View**. Position the cursor at the end of the script.

We'll base the example for this recipe on providing a special calculation for touring bikes.

## How to do it...

Follow these steps to detect members on the same branch:

1. Add the `Scope()` statement.
2. Specify that you want **Touring Bikes** and all of their descendants included in this scope, like this:

```
Scope( Descendants(
    [Product].[Product Categories].[Subcategory].&[3], ,
    SELF_AND_AFTER
)
);
[Measures].[Internet Sales Amount] = 1;
End Scope;
```

3. Save and deploy (or just press the **Deploy MDX Script** icon if you're using **BIDS Helper**).
4. Go to the **Cube Browser** tab and optionally re-connect.
5. Put the measure **Internet Sales Amount** on rows.

## Navigation

---

6. Bring the [Product].[Product Categories] user hierarchy on rows and expand it until you see all the touring bikes, as displayed in the following image:

Drop Filter Fields Here		Drop Column Fields Here	
Category ▾	Subcategory	Product	Internet Sales Amount
[+]	Accessories		\$700,759.96
[+]	Bikes	[+]	\$9,952,759.56
[+]		[+]	\$14,520,584.04
		Touring-1000 Blue, 46	1
		Touring-1000 Blue, 50	1
		Touring-1000 Blue, 54	1
		Touring-1000 Blue, 60	1
		Touring-1000 Yellow, 46	1
		Touring-1000 Yellow, 50	1
		Touring-1000 Yellow, 54	1
		Touring-1000 Yellow, 60	1
		Touring-2000 Blue, 46	1
		Touring-2000 Blue, 50	1
		Touring-2000 Blue, 54	1
		Touring-2000 Blue, 60	1
		Touring-3000 Blue, 44	1
		Touring-3000 Blue, 50	1
		Touring-3000 Blue, 54	1
		Touring-3000 Blue, 58	1
		Touring-3000 Blue, 62	1
		Touring-3000 Yellow, 44	1
		Touring-3000 Yellow, 50	1
		Touring-3000 Yellow, 54	1
		Touring-3000 Yellow, 58	1
		Touring-3000 Yellow, 62	1
		Total	1
		Total	\$24,473,344.60
[+]	Clothing		\$339,772.61
		Grand Total	\$25,513,877.17

7. Verify that the result for all touring bikes, including their total, is 1, as defined in the scope statement.
8. Bring another measure, for example, **Reseller Sales Amount** and verify that this measure remains intact; the scope is not affecting it.

## How it works...

The **Scope( )** statement is used to isolate a particular part of the cube, also called the **subcube**, in order to provide a special calculation for that space. It starts with the **Scope** keyword followed by the subcube and ends with the **End Scope** phrase. Everything inside is valid only for that particular scope.

In this case, we've used the descendants of the member **Touring Bikes** in order to form the subcube with all touring bikes including their total, that is, that same member. There are various ways that we can collect members on a particular branch using the `Ascendants()` and `Descendants()` function. These ways will be covered in the later sections of this recipe.

Once we've set our subcube correctly, we can provide various assignments inside that scope. Here we've specified that the value for the **Internet Sales Amount** measure is to be 1 for every cell in that subcube.

It is worth noticing that the **Touring Bikes Total** is not calculated as an aggregate of its children because the member representing that total was included in the scope. Therefore, a constant value of 1 has been assigned to it, the same way it was assigned to its children. Consequently, the **Touring Bikes** member contributes with its own value, 1, in its parent's result.

If the `AFTER` flag had been used in the `Descendants()` function, the **Touring Bikes** member would have been left out of that scope. In that case, its value would be equal to the number of its children, that is, 22. This time that value would be used as its contribution to its parent's result.

It might be worth noting that regular measures roll up their children's values using aggregate functions (Sum, Max, Min, and others) specified for those measures. Calculated measures don't roll up. They are calculated by evaluating their expression for every single member, be it parent or child.

### There's more...

The example above showed how to isolate a lower part of a hierarchy, a part below a particular member. In this section we're going to show that the opposite is not so easy, but still possible.

In case we need to scope a part of the hierarchy above the current member, we might be tempted to do the obvious, to use the opposite MDX function, the `Ascendants()` function. However, that would result in an error because the subcube wouldn't be compact any more. The term **arbitrary shape** represents a subcube formed by two or more smaller subcubes of different granularity, something that is not allowed in the scopes. The solution is to break the bigger subcube into many smaller ones, so that each can be considered compact, with data of consistent granularity. More about which shape is arbitrary and which is not can be found in Appendix and on this site:

<http://tinyurl.com/ArbitraryShapes>

## *Navigation*

---

Here's an example for the **Mountain Bikes** member in which we show how to set the value to 2 for all of its ancestors.

```
Scope( Ancestors(
    [Product].[Product Categories].[Subcategory].&[1],
    1
)
);
[Measures].[Internet Sales Amount] = 2;
End Scope;

Scope( Ancestors(
    [Product].[Product Categories].[Subcategory].&[1],
    2
)
);
[Measures].[Internet Sales Amount] = 2;
End Scope;
```

This is the result:

Drop Filter Fields Here			Drop Column Fields Here
Category	Subcategory	Product	Internet Sales Amount
Accessories			\$700,759.96
Bikes	Mountain Bikes		\$9,952,759.56
	Road Bikes		\$14,520,584.04
	Touring Bikes	Touring-1000 Blue, 46	1
		Touring-1000 Blue, 50	1
		Touring-1000 Blue, 54	1
		Touring-1000 Blue, 60	1
		Touring-1000 Yellow, 46	1
		Touring-1000 Yellow, 50	1
		Touring-1000 Yellow, 54	1
		Touring-1000 Yellow, 60	1
		Touring-2000 Blue, 46	1
		Touring-2000 Blue, 50	1
		Touring-2000 Blue, 54	1
		Touring-2000 Blue, 60	1
		Touring-3000 Blue, 44	1
		Touring-3000 Blue, 50	1
		Touring-3000 Blue, 54	1
		Touring-3000 Blue, 58	1
		Touring-3000 Blue, 62	1
		Touring-3000 Yellow, 44	1
		Touring-3000 Yellow, 50	1
		Touring-3000 Yellow, 54	1
		Touring-3000 Yellow, 58	1
		Touring-3000 Yellow, 62	1
		Total	1
		Total	2
Clothing			\$339,772.61
		Grand Total	2

The value 1 is from the previous scope, the one which sets the value to 1 for **Touring Bikes**. But, notice the value 2 in the **Bikes Total** row. The same value can be seen in the **Grand Total** row, highlighted in the bottom of the image. The calculation worked as expected.

### The query-based alternative

This recipe can easily be turned into a query-based recipe in case it is not required that the behavior of the scope is persisted in all queries.

Here's the query that returns exactly the same thing as those scope statements we've covered so far, one in the initial example and two in the *There's more* section:

```

WITH
MEMBER [Measures].[Branch detected on member or below] AS
    iif( IsAncestor(
        [Product].[Product Categories].[Subcategory].&[3],
        [Product].[Product Categories].CurrentMember
    ))
OR
    [Product].[Product Categories].[Subcategory].&[3] Is
    [Product].[Product Categories].CurrentMember,
    True,
    null
)
MEMBER [Measures].[Branch detected on member or above] AS
    iif( Intersect(
        Ascendants(
            [Product].[Product Categories].[Subcategory].&[3]
        ),
        [Product].[Product Categories].CurrentMember
    ).Count > 0,
    True,
    null
)
SELECT
    { [Measures].[Branch detected on member or above],
      [Measures].[Branch detected on member or below] } ON 0,
    NON EMPTY
    { [Product].[Product Categories].AllMembers } ON 1
FROM
    [Adventure Works]
```

## Navigation

---

The first calculated measure uses the `IsAncestor()` function and detects whether the current member is below the selected member in the hierarchy. That function returns `False` for the member itself. Therefore, we have to incorporate additional logic in the form of testing for the presence of a particular member, which is explained in the first recipe of this chapter, *Detecting a particular member in a hierarchy*.

The second calculation uses the `Ascendants()` function to get all the members above and including the selected one. When that set is obtained, we test if the current member is in the set of ascendants. The test is performed using the `Intersect()` and `Count` functions.

Here's another query-based alternative, this time using a CELL CALCULATION:

```
WITH
CELL CALCULATION [TouringBikes]
FOR '( [Measures].[Internet Sales Amount],
      Descendants( [Product].[Product Categories]
                    .[Subcategory].&[3], ,
                    SELF_AND_AFTER
                  )
                )'
AS 1
SELECT
{ [Measures].[Internet Sales Amount] } ON 0,
Descendants(
  [Product].[Product Categories].[Subcategory].&[3], ,
  SELF_AND_AFTER
) ON 1
FROM
[Adventure Works]
```

As you can see, a cell calculation does the same thing a `Scope()` statement does in MDX script – it provides an expression for a particular subcube. A CELL CALCULATION is one of the three elements that can be defined using the `WITH` keyword in MDX query. Here's more information about it: <http://tinyurl.com/CellCalculations>

Now, what happens if we want to exclude the selected member in those calculations?

The first calculation in the query we started this section with is easy. We simply have to omit the part next to the OR statement, keeping only the `IsAncestor()` part of the expression.

The second calculation is a bit more complex but it can also be done. All we have to do is extract the selected member from the set of descendants. This can be done relatively easily using the `Except()` function:

```
Except(
    Ascendants(
        [Product].[Product Categories].[Subcategory].&[3]
    ),
    [Product].[Product Categories].CurrentMember )
```

Other parts of the calculation remain the same.

In the query with the `CELL CALCULATION`, we have to change the `SELF_AND_AFTER` flag into the `AFTER` flag. We don't have to do the same for the set on rows, only in cell calculation, where this behavior is defined.

### What to look for

In certain conditions, some MDX functions generate empty sets as their result; others always return a non-empty set. It is therefore good to know which one is preferred in which scenario, because an empty set will result in an empty value and this may cause problems in some calculations.

The `Descendants()` function will almost always return a result. If nothing else, then the member or set defined as its first argument. Contrary to that function, the `Children` function will return an empty set when applied to members on the leaf level.

The `Ascendants()` function behaves pretty much the same as the `Descendants()` function. If nothing else, it will at least return the member in its argument. Contrary to that, the `Parent` function applied to the root member or a top level calculated member returns an empty set. The same is true for the `Ancestors()` and the `Ancestor()` functions. When out of boundaries, they return an empty set as well.

Based on how you want your calculation to act, you should use the function that is more appropriate in a particular case.

### Various options of the `Descendants()` function

The following link provides more information about the `Descendants()` function and how to use its arguments: <http://tinyurl.com/MDXDescendants>

### See also

The Recipes *Detecting a particular member of a hierarchy* and *Detecting the root member* cover a similar topic and are worth reading because of the additional insights they provide.

## Finding related members in the same dimension

The dimensionality of a cube equals the number of hierarchies used in it. This encompasses all the user and attribute hierarchies, visible or not, as long as they are enabled, and one special dimension/hierarchy called *Measures*. A cube with 10 dimensions, each having 10 attribute hierarchies is a 101D object! Compare that to any 3D object in your environment, take a Rubik's cube for example, and you'll immediately be amazed by the space a typical SSAS cube forms. The number of coordinates, or shall we say cells, in that space is simply beyond our imagination.

Fortunately, a great deal of that space is empty and the SSAS engine has ways to optimize that. It even exposes some of the optimization features to us through several MDX functions we can use when needed.

The need to combine various dimensions and their hierarchies is obvious. It is the very nature of data analysis. We regularly want to know which set of members is related to another set of members. Sometimes it is a by-design feature of SSAS. For example, when we put a member in slicer and observe what happens to related sets on axes. Other times, we need to explicitly specify what we want. The reason behind this might be that the member or set in slicer would interfere with other calculations which we would like to prevent. All in all, a great flexibility is given to us.

One type of optimization is related to hierarchies of a dimension and this is the type we're covering in this recipe. The other, related to optimizing the combination of dimensions, will be explained in the recipe that follows immediately after this one.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension and we're going to find the number of available colors in each of the product subcategories.

Here's the query we'll start from:

```
SELECT
    { [Measures].[Internet Order Count] } ON 0,
    { [Product].[Subcategory].[Subcategory].Members *
      [Product].[Color].[Color].Members } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns all product subcategories cross joined with product colors. Scroll down your result and compare it with the following image:

The screenshot shows a software interface with a title bar 'Messages' and 'Results'. The results grid has three columns: 'Subcategory' (e.g., Road Bikes, Road Frames, Saddles, Shorts, Socks, Tights, Tires and Tubes, Touring Bikes, Touring Frames, Vests, Wheels), 'Color' (e.g., Black, Red, Yellow, NA), and 'Internet Order Count' (e.g., 3,030, 2,719, 2,319, etc.). Many rows have '(null)' in the 'Color' column.

		Internet Order Count
Road Bikes	Black	3,030
Road Bikes	Red	2,719
Road Bikes	Yellow	2,319
Road Frames	Black	(null)
Road Frames	Red	(null)
Road Frames	Yellow	(null)
Saddles	NA	(null)
Shorts	Black	1,019
Socks	White	568
Tights	Black	(null)
Tires and Tubes	NA	9,867
Touring Bikes	Blue	1,283
Touring Bikes	Yellow	884
Touring Frames	Blue	(null)
Touring Frames	Yellow	(null)
Vests	Blue	562
Wheels	Black	(null)

It is worth noticing that the result is not a Cartesian product of those two hierarchies. The engine has automatically reduced the two-dimensional set on rows to a set of existing combinations. The reason why this was possible follows.

Those two hierarchies belong to the same dimension which is visible from the query. A dimension originates from the underlying table (or a set of them in a snowflake model). The table has columns which in turn become attributes. There are a finite number of various combinations of attributes, and that number is almost always less than a Cartesian product of those attributes. The engine merely has to read that table and return the set of distinct combinations for a particular case.

Of course, that's not the exact way that it's being done, but you get a good idea of how the multidimensional space is automatically shrunk whenever possible.

Notice also that this is not a result of the `NON EMPTY` keyword on rows because we didn't put it there. That keyword does something else, it removes empty fact rows. As seen in the image above, we have many rows with the value of null in them. We deliberately didn't use that keyword to show the difference between what is known as the **auto-exists algorithm** and what `NON EMPTY` does.

Now, let's get back to the solution and see how to get the number of colors per subcategory.

## How to do it...

Follow these steps to find related members in the same dimension:

1. Add the WITH part of the query.
2. Create a new calculated measure and name it **Number of colors**.
3. Remove the set with [Product].[Color] hierarchy from rows and move it inside the definition of the new measure. Only product subcategories should remain on rows.
4. Use the EXISTING function before the set of color members and wrap everything in the Count( ) function.
5. Add this calculated measure on axis 0, next to the existing measure.
6. Execute the query which should look like this:

```
WITH
MEMBER [Measures].[Number of colors] AS
    Count( EXISTING [Product].[Color].[Color].Members )
SELECT
    { [Measures].[Internet Order Count],
      [Measures].[Number of colors] } ON 0,
    { [Product].[Subcategory].[Subcategory].Members
        } ON 1
FROM
    [Adventure Works]
```

7. Scroll down to the end and verify that the result matches the following image:

	Internet Order Count	Number of colors
Road Bikes	8,068	3
Road Frames	(null)	3
Saddles	(null)	1
Shorts	1,019	1
Socks	568	1
Tights	(null)	1
Tires and Tubes	9,867	1
Touring Bikes	2,167	2
Touring Frames	(null)	2
Vests	562	1
Wheels	(null)	1

## How it works...

The EXISTING keyword forces the succeeding set to be evaluated in the current context. Without it, the current context would be ignored and we would get all colors.

After that, we apply the Count( ) function in order to get the dynamic count of colors, a value calculated for each row separately.

## There's more...

There might be situations when you'll have multiple hierarchies of the same dimension in the context, but you'll only want some of them to have an impact on the selected set. In other words, there could have been sizes next to product subcategories on rows. If you use the EXISTING keyword on colors, you'll get the number of colors for each combination of the subcategory and the size. In case you need your calculation to ignore the current size member and get the number of colors per subcategory only, you will have to take another approach. If you're wondering why you would do such a thing, just imagine you need an indicator which gives you a percentage of the color per size and subcategory. That indicator would have an unusual expression in its denominator and the usual expression in its numerator.

OK, so what's the solution in this case and how do we make such a calculation?

The Exists( ) function comes to the rescue. In fact, that function does the same thing as the EXISTING keyword, but it requires a second argument in which we need to specify the context for the evaluation.

Here's an example query:

```
WITH
  MEMBER [Measures].[Number of colors] AS
    Count( EXISTING [Product].[Color].[Color].Members )
  MEMBER [Measures].[Number of colors per subcategory] AS
    Count( Exists( [Product].[Color].[Color].Members,
                  { [Product].[Subcategory].CurrentMember } ) )
SELECT
  { [Measures].[Internet Order Count],
    [Measures].[Number of colors],
    [Measures].[Number of colors per subcategory] } ON 0,
  { [Product].[Subcategory].[Subcategory].Members *
    [Product].[Size Range].[Size Range].Members } ON 1
FROM
  [Adventure Works]
```

## Navigation

---

Once run, this query returns two different color-count measures. The first is unique to each row, the second changes by subcategory only. Their ratio, not present but easily obtainable in the query, would return the percentage of color coverage. For example, in the following image it is obvious that there's only 33% color coverage for 38-40 CM Road Bikes, which may or may not be a signal to fill the store with additional colors for that subcategory. The important thing is that we were able to control the context, fine-tune it.

		Internet Order Count	Number of colors	Number of colors per subcategory
Pedals	NA	(null)	1	1
Pumps	NA	(null)	1	1
Road Bikes	38-40 CM	782	1	3
Road Bikes	42-46 CM	2,235	3	3
Road Bikes	48-52 CM	3,091	3	3
Road Bikes	54-58 CM	1,355	2	3
Road Bikes	60-62 CM	605	2	3
Road Frames	38-40 CM	(null)	1	3
Road Frames	42-46 CM	(null)	3	3
Road Frames	48-52 CM	(null)	3	3
Road Frames	54-58 CM	(null)	2	3
Road Frames	60-62 CM	(null)	2	3
Saddles	NA	(null)	1	1
Shorts	L	363	1	1

We can also turn it the other way around. The EXISTING keyword is in fact a shortcut, a shorter version of the `Exists()` function which says, "take everything available as the second argument, don't force me to specify everything." The EXISTING keyword is therefore a more flexible, generic variant which handles any context. When we want to take control over the context, we can step back to the `Exists()` function.

### Tips and trick related to the EXISTING Keyword

Another way of specifying the EXISTING keyword is by using the MDX function with the same name. In other words, this:

```
Existing( [Product].[Color].[Color].Members )
```

This may come in handy with cross-joins because the cross-join operator \* has precedence over the EXISTING keyword. Meaning that the EXISTING keyword will be applied to a cross-joined set, and not the first set in the cross-join. In order to prevent this ambiguity, wrap the set including the EXISTING keyword in curly brackets, like this:

```
{ EXISTING [Product].[Color].[Color].Members }
```

## Filter() vs. Exists(), Existing(), and EXISTING

Never iterate on a set unless you really have to because iteration is slow. Look for utilizing specialized functions which operate on sets whenever possible. They are designed to leverage the internal structures and therefore operate much faster.

This recipe hopefully showed there's no need to filter a set of members if that set is related to the current context, in other words, that it belongs to the same dimension as explained in the introduction.

### A friendly warning

After reading the subsequent recipe about finding related members on a different dimension, you might be tempted to use the technique described in that recipe here as well. The idea of not having to memorize each approach separately is an attractive one. A unique, all-purpose way of finding related a member no matter where is.

You should know that, although it would work, it would be an inefficient solution. The performance could suffer greatly.

The reason why lies in the fact that if you stick with the solution presented in this recipe, the SSAS engine will be able to perform a fast auto-exist operation on a single dimension table. The solution presented in the subsequent chapter relies on a join between the dimension table and the fact table. Now, all of a sudden, the engine has N times larger table to scan. That could be a very costly operation if the engine is not optimized to reduce the unnecessary complexity involved here. What's best is that complexity is not needed at all in this scenario! Therefore, try to make a difference in your mind between finding related members in the same dimension and finding related members in another dimension, and approach each case using a different, but appropriate technique.

The procedure mentioned above serves the purpose of illustrating the concept; it doesn't necessarily represent the actual implementation in the engine.

### See also

Other aspects of the EXISTING keyword are covered in chapter 1, in the recipe *Optimizing MDX query using the NonEmpty() function*. You may gain a better understanding of that keyword by reading that recipe.

Also, read the recipe *Finding related members in another dimension* in order to understand the difference between finding related members in the same dimension and in different dimensions.

## Finding related members in another dimension

As mentioned in the introduction of the previous recipe, *Finding related members in the same dimension*, this recipe deals with a slightly different scenario. It explains how to find the related members using two or more dimensions.

Before we start, let's get one thing straight - when we say a **dimension**, we mean any hierarchy in that dimension from now on.

Dimensions, unlike hierarchies of the same dimension, are independent objects. Without a third table in form of the fact table, they are unrelated, at least in the dimensional modeling sense. When a fact table is inserted among them, the many-to-many relation comes into existence.

In other words, there are two different types of combination we can make with dimensions. One, naturally, is the **Cartesian product**. It is obtained by cross-joining members in both dimensions. In relational terms, that would represent the CROSS JOIN of two tables. Since those tables are two independent objects, we get a real Cartesian product.

The other combination is a combination over a fact table, the intermediate table of two or more dimensions. That table can serve as a filter. We can use it to get members in the second dimension, members which have records, or non-empty values for the selected members in the first dimension. A typical example would be "find me products that are available on a particular territory, are bought by a particular customer, or are shipped on a particular date." The first dimension in this case is territory, customers and dates respectively; the second one is the product dimension.

The above example is covered in SSAS by design. All we have to do is position the selected member or more of them in slicer, turn the NON EMPTY keyword on rows, provide a set of members from the other dimension on rows and a measure of our interest on columns. The result matches our expectations. But what's more important, this operation is natural for any SSAS client, which means it's available in ad-hoc analysis.

However, as we've learned quite a few times throughout this book, there are always situations when we need to have a control over certain calculations, and those calculations might need to get the related members in another dimension. Is there a way to support this? Yes, this recipe shows how.

## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Reseller** dimension and we're going to find how many subcategories each of the top 100 customers is ordering.

Here's the query we'll start from:

```
SELECT
    { [Measures].[Reseller Order Count] } ON 0,
    { TopCount( [Reseller].[Reseller].[Reseller].Members,
        100,
        [Measures].[Reseller Order Count] ) *
    [Product].[Subcategory].[Subcategory].Members } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns the top 100 resellers based on their ordering frequency combined with product subcategories. The **NON EMPTY** keyword is omitted intentionally in order to show a Cartesian product in action.

100 customers combined with 37 subcategories makes 3700 rows, that's a Cartesian product.

Scroll down to the last row, highlight it, and check the status bar of SSMS. That extra row, 3701th, is the column header row.

		Reseller Order Count
Big-Time Bike Store	Socks	1
Big-Time Bike Store	Tights	(null)
Big-Time Bike Store	Tires and Tubes	(null)
Big-Time Bike Store	Touring Bikes	(null)
Big-Time Bike Store	Touring Frames	(null)
Big-Time Bike Store	Vests	(null)
Big-Time Bike Store	Wheels	3

Ln 3701      Col 1

Adventure Works DW 2008R2 | 00:00:02

## How to do it...

Follow these steps to find related members in another dimension:

1. Add the **WITH** part of the query.
2. Create a new calculated measure and name it **Count of SC - E**.
3. Remove the set with the **[Product].[Subcategory]** hierarchy from rows and move it inside the definition of the new measure. Only the resellers should remain on rows.
4. Use the variant of the **Exists()** function which has the third argument, the measure group name. In this case, you should use the measure group containing the measure Reseller Order Count. The name of that measure group is Reseller Orders.
5. Finally, wrap everything with the **Count()** function.
6. Add this calculated measure on axis 0, next to the existing measure.
7. Execute the query which should look like this:

```
WITH
MEMBER [Measures].[Count of SC - E] AS
    Count( Exists(
        [Product].[Subcategory].[Subcategory].Members,
        'Reseller Orders' ) )
SELECT
    { [Measures].[Reseller Order Count],
      [Measures].[Count of SC - E] } ON 0,
    { TopCount( [Reseller].[Reseller].[Reseller].Members,
      100,
      [Measures].[Reseller Order Count] ) *
      [Product].[Subcategory].[Subcategory].Members } ON 1
FROM
    [Adventure Works]
```

8. Verify that the result matches the following image:

	Reseller Order Count	Count of SC - E
Advanced Bike Components	12	22
Area Bike Accessories	12	19
Basic Sports Equipment	12	18
Better Bike Shop	12	21
Bike Dealers Association	12	22
Bike Goods	12	8
Brakes and Gears	12	13
Brightwork Company	12	17

## How it works...

The `Exists()` function has two variants. The variant with two sets is useful for intersecting related attributes as shown in the previous recipe. The variant with the measure group name is ideal for combining dimensions across a fact table. In fact, the measure group is a fact table itself.

What this other variant does is instructs the engine to return distinct members of a hierarchy who have combinations with either the current member in context or the set of members specified in the second argument. In our example, we've omitted that second argument. We're calculating the result for every reseller on rows which means the required context will be established in the evaluation phase of the query. There's no need to use the current member as the second set; that member will be there implicitly.

Once we get a set of distinct members, all we have to do is count them.

## There's more...

The alternative, although not exactly the same solution, would be to use the `NonEmpty()` function. Here's the query which, when run, shows that both count measures return the same results for each reseller.

```
WITH
MEMBER [Measures].[Count of SC - E] AS
    Count( Exists(
        [Product].[Subcategory].[Subcategory].Members,
        'Reseller Orders' ) )
MEMBER [Measures].[Count of SC - NE] AS
    Count( NonEmpty(
        [Product].[Subcategory].[Subcategory].Members,
        { [Measures].[Reseller Order Count] } ) )
SELECT
    { [Measures].[Reseller Order Count],
      [Measures].[Count of SC - E],
      [Measures].[Count of SC - NE] } ON 0,
    { TopCount( [Reseller].[Reseller].Members,
      100,
      [Measures].[Reseller Order Count]
    ) } ON 1
FROM
    [Adventure Works]
```

Since we've given a hint that this alternative is not the exact one, now's the time to shed more light upon that.

## Navigation

---

The difference between the second variant of the `Exists()` function and the `NonEmpty()` function is subtle. It manifests only on measures for which the `NullProcessing` property is set to `Preserve`. `NonEmpty()` is a more destructive function in that case, it ignores fact records with nulls while `Exists()` preserves them. This way we can have two counts and use the one that suits us in a particular case, which is good to know. The other subtle difference is that the `Exists()` function ignores the MDX script and simply does a storage engine query. For example, if the MDX script nulls out a measure, the `Exists()` function will still return values.

The queries in this recipe so far illustrated the concept behind the `Exists()` and `NonEmpty()` functions. These functions can be used to isolate related members on other dimensions. However, from the performance perspective, they are not great when you need to count members on other dimensions because the `count-exists` and the `count-nonempty` combinations are not optimized to run in block mode. The `sum-iif` combination, on the other hand, is optimized to run in block mode. Therefore, whenever you need to do something more than simply isolating related members on other dimensions (such as counting, and so on), consider using a combination of functions that you know run in block mode.

Here's the query that outperforms the two queries shown so far in this recipe:

```
WITH
MEMBER [Measures].[Count of SC - SI] AS
    Sum( [Product].[Subcategory].[Subcategory].MEMBERS,
        iif( IsEmpty( [Measures].[Reseller Order Count] ),
            null,
            1
        )
    )
SELECT
    { [Measures].[Count of SC - SI] } ON 0,
    { TopCount( [Reseller].[Reseller].[Reseller].MEMBERS,
        100,
        [Measures].[Reseller Order Count] ) *
        [Product].[Subcategory].[Subcategory].MEMBERS } ON 1
FROM
    [Adventure Works]
```

## Leaf and non-leaf calculations

The examples in this and the previous recipe were designed somewhat complex from a technical perspective, but perfectly valid and as expected from an analyst's perspective. It was required to get the count of existing members on a non-leaf level.

When it's required to get the count of members on a leaf level, designing a distinct count measure using the dimension key in the fact table might be a better option. It works by design; there is no code maintenance and it's much faster than its potential MDX counterpart. Therefore, look for a by-design solution whenever possible; don't assume that things should be handled in MDX just because this recipe indicated such. Chapter 6 deals with that in more detail.

When it is required to get the count on a non-leaf attribute, that's the time when MDX and relations between hierarchies and dimensions come in play as equally valid solutions. Because either you are going to include that higher granularity attribute in your fact table (hardly likely, especially on large fact tables) and then build a distinct count measure from it, or you can build a new measure group at the non-leaf grain, or you will look for an MDX alternative like we did in this example and the one in the previous chapter. Additionally, the non-leaf levels will typically, although not always have much lower cardinality than the leaf level, which means that MDX calculations will perform significantly better than they would on a leaf level.

This section is here to make you think about those things so that you're aware of the choices you have and so that you can make the right decision.

## See also

Other aspects of the `NonEmpty()` function are covered in *Chapter 1*, in the recipe *Optimizing MDX queries using the NonEmpty() function*. You may gain a better understanding of that function by reading that recipe.

Also, read the recipe *Finding related members in the same dimension* in order to understand the difference between finding related members in the same dimension and in the different dimensions.

## Calculating various percentages

The next couple of recipes show how to calculate relative percentages, averages, and ranks. We're starting with percentages.

Having a ratio of current member's value over its parent's value is an often-required calculation. It's a form of normalizing the hard-to-grasp values in the table. When the individual amounts become percentages, it immediately becomes clear where the good values are.

There are many kinds of percentages or shares, but we'll take the typical three and present them in this recipe. These are: percentage of parent's value, percentage of level's total, and the percentage of the hierarchy's total.

This recipe will show how to calculate them using the ragged **Sales Territory** hierarchy of the **Sales Territory** dimension. A **ragged hierarchy** is a hierarchy whose lower level members are hidden in some positions. For example, the USA member is the only country in that hierarchy which has children, the regions. No other country in that hierarchy has them, which will be clearly visible in the example below. So let's start!

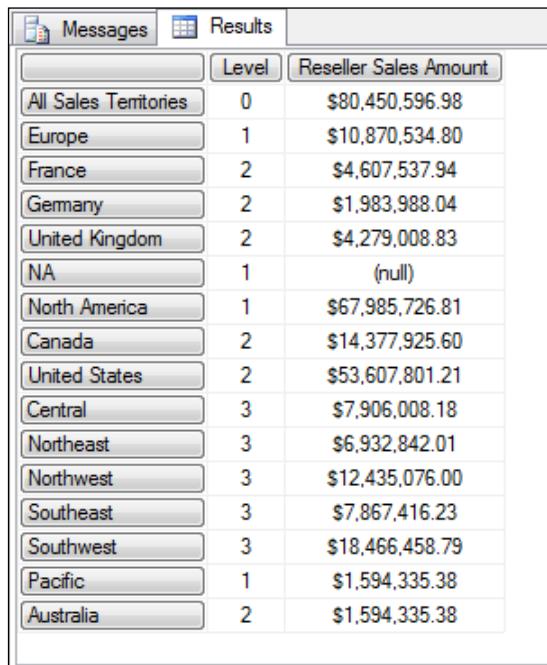
## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Here's the query we'll start from:

```
WITH
MEMBER [Measures].[Level] AS
    [Sales Territory].[Sales Territory]
    .CurrentMember.Level.Ordinal
SELECT
    { [Measures].[Level],
        [Measures].[Reseller Sales Amount] } ON 0,
    { [Sales Territory].[Sales Territory].AllMembers } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns hierarchized territories on rows.



The screenshot shows a SQL Server Management Studio window with the 'Results' tab selected. The results grid displays a hierarchy of sales territories with their respective levels and reseller sales amounts. The columns are labeled 'Level' and 'Reseller Sales Amount'. The data is as follows:

	Level	Reseller Sales Amount
All Sales Territories	0	\$80,450,596.98
Europe	1	\$10,870,534.80
France	2	\$4,607,537.94
Germany	2	\$1,983,988.04
United Kingdom	2	\$4,279,008.83
NA	1	(null)
North America	1	\$67,985,726.81
Canada	2	\$14,377,925.60
United States	2	\$53,607,801.21
Central	3	\$7,906,008.18
Northeast	3	\$6,932,842.01
Northwest	3	\$12,435,076.00
Southeast	3	\$7,867,416.23
Southwest	3	\$18,466,458.79
Pacific	1	\$1,594,335.38
Australia	2	\$1,594,335.38

On columns we have a little helper, the level ordinal displayed in a form of a measure. Also, we'll need a measure to calculate the percentage from. In this case, we'll use the Reseller Sales Amount measure.

## How to do it...

Follow these steps to calculate various percentages:

1. Create the first calculated measure for the percentage of parent's value, name it **Parent %** and provide the following definition for it:

```
MEMBER [Measures].[Parent %] AS
    iif( [Sales Territory].[Sales Territory].CurrentMember Is
        [Sales Territory].[Sales Territory].[All],
        1,
        [Measures].[Reseller Sales Amount] /
        (
            [Measures].[Reseller Sales Amount],
            [Sales Territory].[Sales Territory]
            .CurrentMember.Parent
        )
    ), FORMAT_STRING = 'Percent'
```

2. Create the second calculated measure for the percentage of level's total value, name it **Level %** and provide the following definition for it:

```
MEMBER [Measures].[Level %] AS
    [Measures].[Reseller Sales Amount] /
    Aggregate( [Sales Territory].[Sales Territory]
    .CurrentMember.Level.Members,
        [Measures].[Reseller Sales Amount] )
    , FORMAT_STRING = 'Percent'
```

3. Create the third calculated measure for the percentage of hierarchy's total value, name it **Hierarchy %** and provide the following definition for it:

```
MEMBER [Measures].[Hierarchy %] AS
    [Measures].[Reseller Sales Amount] /
    ( [Sales Territory].[Sales Territory].[All],
        [Measures].[Reseller Sales Amount] )
    , FORMAT_STRING = 'Percent'
```

## Navigation

---

4. Include all three calculated measures in columns, execute the query, and verify that the result matches the following image:

The screenshot shows a results grid from SSMS with the following data:

	Level	Reseller Sales Amount	Parent %	Level %	Hierarchy %
All Sales Territories	0	\$80,450,596.98	100.00%	100.00%	100.00%
Europe	1	\$10,870,534.80	13.51%	13.51%	13.51%
France	2	\$4,607,537.94	42.39%	5.73%	5.73%
Germany	2	\$1,983,988.04	18.25%	2.47%	2.47%
United Kingdom	2	\$4,279,008.83	39.36%	5.32%	5.32%
NA	1	(null)	(null)	(null)	(null)
North America	1	\$67,985,726.81	84.51%	84.51%	84.51%
Canada	2	\$14,377,925.60	21.15%	17.87%	17.87%
United States	2	\$53,607,801.21	78.85%	66.63%	66.63%
Central	3	\$7,906,008.18	14.75%	14.75%	9.83%
Northeast	3	\$6,932,842.01	12.93%	12.93%	8.62%
Northwest	3	\$12,435,076.00	23.20%	23.20%	15.46%
Southeast	3	\$7,867,416.23	14.68%	14.68%	9.78%
Southwest	3	\$18,466,458.79	34.45%	34.45%	22.95%
Pacific	1	\$1,594,335.38	1.98%	1.98%	1.98%
Australia	2	\$1,594,335.38	100.00%	1.98%	1.98%

## How it works...

The **Parent %** measure returns the ratio of the current member's value over its parent's value. The parent's value is calculated relative to the current member. In other words, the ratio returns a different value for each territory.

One additional thing we have to take care of is handling the problem of the non-existing parent of the root member. There's no such thing as the parent of the root member, meaning, the calculation would result in an error for that cell. In order to take care of that, we've wrapped the calculation in an additional `iif()` statement. In it, we've provided the value of 1 (100% later) when the root member becomes the current member during the iteration phase on rows.

Similarly, we've defined the other two calculated measures.

The **Level %** and the **Hierarchy %** measures return the same thing unless applied to a ragged hierarchy like in this case because the aggregate of a level is practically the same as the value of the root member. Check this in the preceding image. Only the members on the third level have different values because other level 2 members don't have children.

The first of those measures, the **Level %** measure, calculates an aggregate of the members on the same level and uses it in the denominator of the ratio. The second measure, **Hierarchy %**, directly picks the coordinate with the root member because that tuple represents the total value of the hierarchy.

### There's more...

If there is a request to make another type of percentage calculation, remember that several techniques might come in handy to you.

First, you can use a scope if you need to calculate the percentage only for a part of your hierarchy.

Next, if there is a single member you need to use in a denominator's tuple, use the `Ancestor()` function and provide the appropriate level. If there isn't a single member, you will have to aggregate the set of members using the `Aggregate()` function, as shown in our example for the **Level %** calculation.

Finally, remember to take care of division by zero problems and problems related to non-existing members in the hierarchy. One such example is the **Parent %** calculation where we were detecting the root member because that's the only member in that hierarchy without the parent.

### Use cases

The **Parent %** measure is the most requested and useful one. It is applicable in any hierarchy and gives an easy way to comprehend information about members in hierarchy.

**Hierarchy %** is useful in the parent-child hierarchy to calculate the individual percentages of members scattered in various positions and levels of a parent-child hierarchy. In addition to that, this calculation is useful in user hierarchies when there is a need to calculate the percentage of members in lower levels with respect to hierarchy's total, because the immediate parents and ancestors break the result into a series of 100% contributions.

Finally, the **Level %** measure is also useful in parent-child hierarchies.

Besides ragged hierarchies, the difference between the **Level %** and the **Hierarchy %** will manifest in other non-symmetrical structures, that is hierarchies with custom roll-ups and special scopes applied to them. Financial structures (dimensions) like P&L and Balance sheet are examples of these special types of hierarchies. In those scenarios you might want to consider having both percentages.

The **Hierarchy %** calculated measure is performance-wise a better measure because it picks a coordinate in the cube directly. Moreover, that coordinate is the coordinate with the root member, something we can expect to have an aggregate for. Unless the situation really requires both of these measures, use the **Hierarchy %** measure only.

### The alternative syntax for the parent member

Just for the record, there is an alternative syntax for the parent, the one that is safe enough so that we don't have to detect the root member as we did in our example. That definition is this:

```
Tail(  
    Ascendants(  
        [Sales Territory].[Sales Territory].CurrentMember  
    )  
).Item(0)
```

The expression basically says, "take the last of the ascendants of the current member and extract it from that set as a member."

The ascendants are always hierarchized which means that the last member in the set of ascendants is the parent we're after. Once we have that set, we can use the `Tail()` function to get the last member of that set. If we omit the second argument of that function, the value of 1 is implicitly applied.

The result of that function is still a set so we have to use the `Item()` function in order to convert that single-member set into as a member object. Only then can we use it in our tuple.

### The case of non-existing root member

When the `IsAggregatable` property of a hierarchy is set to `False`, both the previous and the initial expressions for detecting the parent member would fail. In that case, you should find the minimal level ordinal first and then compare each member's level ordinal with that value:

```
MEMBER [Measures].[MinLevel] AS  
    Min( [Sales Territory].[Sales Territory].ALLMEMBERS,  
        [Measures].[Level] )  
  
MEMBER [Measures].[Parent %] AS  
    iif( [Sales Territory].[Sales Territory]  
        .CurrentMember.Level.Ordinal =  
        [Measures].[MinLevel],  
        1,  
        [Measures].[Reseller Sales Amount] /  
        (  
            [Measures].[Reseller Sales Amount],  
            [Sales Territory].[Sales Territory]  
            .CurrentMember.Parent  
        )  
    ), FORMAT_STRING = 'Percent'
```

Here, the `MinLevel` measure will return 0, but when used on another hierarchy with the `IsAggregatable` property set to false (i.e. **Organization.Organizations**), the same expression will return 1.

Remember, the measure `Level` used in the definition of the `MinLevel` calculated measure is defined in the initial query of this recipe.

### The percentage of leaf member values

The percentage on leaves is the fourth type of the percentage we can calculate. The reason we didn't is that an aggregate on leaves is almost always the same value as the value in the root member. If you have a different situation, use the following calculation:

```
MEMBER [Measures].[Leaves %] AS  
    [Measures].[Reseller Sales Amount] /  
    Aggregate(  
        Descendants( [Sales Territory].[Sales Territory]  
            .[All], ,  
            leaves ),  
        [Measures].[Reseller Sales Amount] )  
    , FORMAT_STRING = 'Percent'
```

This calculation takes all leaf members and then applies an aggregate of them in respect to the measure provided. Because of this, it might be significantly slower.

### See also

The following recipes are part of the 3-part series of recipes in this chapter dealing with the topic of relative calculations: *Calculating various averages* and *Calculating various ranks*. It is recommended that you read all of them in order to get more thorough picture.

## Calculating various averages

This is the next recipe in our series of relative calculations. In this recipe, we're going to show how to calculate the average among siblings, the average in the complete hierarchy and the average on leaves. A good introduction is given in the previous recipe, the one about calculating various percentages. It is recommended that you read this series of recipes exactly as laid out in the book.

### Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Here's the query we'll start from:

```
WITH
MEMBER [Measures].[Level] AS
    [Sales Territory].[Sales Territory]
    .CurrentMember.Level.Ordinal
SELECT
    { [Measures].[Level],
        [Measures].[Reseller Sales Amount] } ON 0,
    { [Sales Territory].[Sales Territory].AllMembers } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns hierarchized territories on rows.

On columns we have a little helper, the level ordinal displayed in a form of a measure. In addition to that, we have the main measure, the Reseller Sales Amount measure. That's the measure we'll use to calculate the average.

## How to do it...

Follow these steps to calculate various averages:

1. Create the first calculated measure for the average among the siblings, name it **Siblings AVG**, and provide the definition for it using the Avg( ) function and the Siblings function:

```
MEMBER [Measures].[Siblings AVG] AS
    Avg( [Sales Territory].[Sales Territory]
        .CurrentMember.Siblings,
        [Measures].[Reseller Sales Amount] )
```

2. Create the second calculated measure for the average on level, name it **Level AVG**, and provide the definition for it using the Avg( ) function and the Level function:

```
MEMBER [Measures].[Level AVG] AS
    Avg( [Sales Territory].[Sales Territory]
        .CurrentMember.Level.Members,
        [Measures].[Reseller Sales Amount] )
```

3. Create the third calculated measure for the average on hierarchy, name it **Hierarchy AVG**, and provide the definition for it using the Avg( ) function.

```
MEMBER [Measures].[Hierarchy AVG] AS
    Avg( [Sales Territory].[Sales Territory].Members,
        [Measures].[Reseller Sales Amount] )
```

4. Create the fourth calculated measure for the average on leaves, name it **Leaves AVG**, and provide the definition for it using the `Avg( )` function and the version of the `Descendants( )` function which returns leaves:

```
MEMBER [Measures].[Leaves AVG] AS
    Avg( Descendants( [Sales Territory].[Sales Territory]
        .[All], ,
        Leaves ),
    [Measures].[Reseller Sales Amount] )
```

5. Include all four calculated measures on columns, execute the query, and verify that the result matches the following image:

The screenshot shows a results grid from SSMS with the following columns: Level, Reseller Sales Amount, Siblings AVG, Level AVG, Hierarchy AVG, and Leaves AVG. The data rows represent various geographical levels from All Sales Territories down to individual countries like Australia.

	Level	Reseller Sales Amount	Siblings AVG	Level AVG	Hierarchy AVG	Leaves AVG
All Sales Territories	0	\$80,450,596.98	\$80,450,596.98	\$80,450,596.98	\$19,663,972.81	\$8,045,059.70
Europe	1	\$10,870,534.80	\$26,816,865.66	\$26,816,865.66	\$19,663,972.81	\$8,045,059.70
France	2	\$4,607,537.94	\$3,623,511.60	\$13,408,432.83	\$19,663,972.81	\$8,045,059.70
Germany	2	\$1,983,988.04	\$3,623,511.60	\$13,408,432.83	\$19,663,972.81	\$8,045,059.70
United Kingdom	2	\$4,279,008.83	\$3,623,511.60	\$13,408,432.83	\$19,663,972.81	\$8,045,059.70
NA	1	(null)	\$26,816,865.66	\$26,816,865.66	\$19,663,972.81	\$8,045,059.70
North America	1	\$67,985,726.81	\$26,816,865.66	\$26,816,865.66	\$19,663,972.81	\$8,045,059.70
Canada	2	\$14,377,925.60	\$33,992,863.40	\$13,408,432.83	\$19,663,972.81	\$8,045,059.70
United States	2	\$53,607,801.21	\$33,992,863.40	\$13,408,432.83	\$19,663,972.81	\$8,045,059.70
Central	3	\$7,906,008.18	\$10,721,560.24	\$10,721,560.24	\$19,663,972.81	\$8,045,059.70
Northeast	3	\$6,932,842.01	\$10,721,560.24	\$10,721,560.24	\$19,663,972.81	\$8,045,059.70
Northwest	3	\$12,435,076.00	\$10,721,560.24	\$10,721,560.24	\$19,663,972.81	\$8,045,059.70
Southeast	3	\$7,867,416.23	\$10,721,560.24	\$10,721,560.24	\$19,663,972.81	\$8,045,059.70
Southwest	3	\$18,466,458.79	\$10,721,560.24	\$10,721,560.24	\$19,663,972.81	\$8,045,059.70
Pacific	1	\$1,594,335.38	\$26,816,865.66	\$26,816,865.66	\$19,663,972.81	\$8,045,059.70
Australia	2	\$1,594,335.38	\$1,594,335.38	\$13,408,432.83	\$19,663,972.81	\$8,045,059.70

### How it works...

The averages are calculated using the standard MDX function called `Avg( )`. That function takes a set of members and calculates the average value of the measure provided as the second argument of that function throughout that set of members. The only thing we have to take care of is to provide a good set of members, the one we need the average to be calculated on.

The set of members for the average among siblings calculation is obtained using the `Siblings` function. The set of members for the average on level calculation is obtained using the `Level` function. The set of members for the average on hierarchy calculation is obtained using the `Members` function only, applied directly to the hierarchy, not an individual level. That construct returns all members in that hierarchy, starting from the root member and ending with all leaf members. Finally, the set of members for the average on leaves calculation is obtained by isolating the leaf members using the `Descendants( )` function.

## There's more...

Most of these averages measures are not intended to be displayed in a grid as we've done here. They are more frequently used as a denominator in various ratios or other types of calculations. For example, it might be interesting to see the discrepancy of each value against one or more average values. That way we can separate members performing below the average from those above the average, particularly if we apply additional conditional formatting of the foreground or the background colors of the cells.

The first two average calculations are applicable to any hierarchy. The Hierarchy AVG and the Leaves AVG measures are more applicable in parent-child hierarchies and other types of non-symmetrical hierarchies. Here are few examples to support that.

The **Employee.Employees** hierarchy is a parent-child hierarchy. Employees are found on all levels in that hierarchy. It makes sense to calculate the average value for all employees, no matter which level they are found at, so that we can compare each employee's individual value against the average value of all employees.

The same process can be restricted to employees that have no one underneath them in the corporate hierarchy. That is, employees which are located as leaves in that hierarchy. It makes sense to calculate their individual contribution against the average value of all such leaf employees. This way we could notice and promote those above average employees by hiring new ones as their subordinates.

### Preserving empty rows

The average is rarely an empty value. In other words, it's a dense type of measure or a calculation in general.

The consequence of that is that we need to handle when it should default to the `null` value.

Take a look at the image with the results again and you'll notice there's a row with the `null` value for the Reseller Sales Amount, but the averages are not `null` themselves. That's what a dense measure is.

Oftentimes we don't want all those empty rows, that is, rows where the original measure is `null`, but a calculated one is not. The reason is because we generally tend to use the `NON EMPTY` keyword on axes in order to get only a subset of members with results. In this example, we've deliberately omitted that keyword in order to demonstrate this issue, but in a real situation we would have applied it on rows.

The thing is that `NON EMPTY` works only when the complete row is empty. Any calculated measure we introduce can spoil that.

In order to deal with this issue, we can wrap all calculations for averages in an outer `iif()` statement and detect empty value for the original measure there. This way we will be able to lose empty rows because our calculation will return `null` when the original measure is empty. As mentioned in many examples in this book, the `iif()` function is optimized to perform well when one of the branches is null.

Here's an example how to correct previous calculations:

```
MEMBER [Measures].[Siblings AVG] AS
    iif( IsEmpty( [Measures].[Reseller Sales Amount] ),
        null,
        Avg( [Sales Territory].[Sales Territory]
            .CurrentMember.Siblings,
            [Measures].[Reseller Sales Amount]
        )
    )
```

We could do the same for the rest of the measures.

One more thing: although it's not the focus of this recipe, the same tip applies to the `Level` measure. If you intend to preserve it in the query, adjust it so that it can be empty when required.

### **Other specifics of average calculations**

The `Avg()` function returns the average value of non-empty cells only. If you want to create something like a rolling 3-month average where some of the months may be empty, you will need to modify or adjust your calculation.

One way of doing this is to break the calculation in two so that the numerator calculates the sum of a set and denominator calculates the count of members (or tuples in general) in that set. The `Count()` function has the optional second argument which enables us to count all members in a set, not just the empty ones which is a default option. The required flag to be used in this case is called `INCLUDEEMPTY`.

The other solution is to modify the second argument in the `Avg()` function so that the expression is never `null`, either by using the `CoalesceEmpty()` function or by using a dense calculated measure, which is never empty. The sum over count approach is the preferred approach because it preserves the format of measures.

Here you can find more information about MDX functions mentioned in this section and their specifics, including examples:

- ▶ <http://tinyurl.com/AvgInMDX>
- ▶ <http://tinyurl.com/CountInMDX>
- ▶ <http://tinyurl.com/CoalesceEmptyInMDX>

## Navigation

---

The Sum-Count approach is also the best approach when you need the average on fact granularity, when you need an aggregation function to calculate the average sales amount or similar. Notice there's no Avg aggregation function, only Sum, Count, Max, Min, DistinctCount, and those semi-aggregatable functions available only in the Enterprise version of SQL Server Analysis Services. In other words, the average aggregation is done by creating two regular measures, one that sums the value and another one that counts the rows, and then a calculated measure as their ratio.



You can always create various scope statements inside the MDX script, thereby breaking the complexity of the requested functionality into a subset of smaller, more compact calculations.

## See also

The following recipe is part of the 3-part series of recipes in this chapter dealing with the topic of relative calculations: *Calculating various percentages* and *Calculating various ranks*. It is recommended that you read all of them in order to get more thorough picture.

## Calculating various ranks

This is the final recipe in our series. In this recipe, we're going to show how to calculate the rank among siblings, the rank on a level, and the rank in the complete hierarchy. A good introduction is given in the initial recipe of this series, the one about calculating various percentages. Therefore it is recommended that you read this series of recipes exactly as laid out in the book in order to get a better picture of the possibilities and ways we do things here.

## Getting ready

Start SQL Server Management Studio and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Here's the query we'll start from:

```
WITH
MEMBER [Measures].[Level] AS
    [Sales Territory].[Sales Territory]
    .CurrentMember.Level.Ordinal
SELECT
    { [Measures].[Level],
        [Measures].[Reseller Sales Amount] } ON 0,
    { [Sales Territory].[Sales Territory].AllMembers } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns the complete [Sales Territory].[Sales Territory] user hierarchy on rows, hierarchized.

On columns we have a little helper, the level ordinal, displayed in the form of a measure. In addition to that, we have the main measure, the Reseller Sales Amount measure. That's the measure we're going to use to calculate the ranks.

## How to do it...

Follow these steps to create various rank calculations:

1. Create the first calculated measure, name it **Siblings Rank**, and provide the definition for it using the Rank( ) function and the Siblings function.

```
MEMBER [Measures].[Siblings Rank] AS  
    Rank( [Sales Territory].[Sales Territory].CurrentMember,  
          [Sales Territory].[Sales Territory]  
          .CurrentMember.Siblings,  
          [Measures].[Reseller Sales Amount] )
```

2. Create the second calculated measure, name it **Level Rank**, and provide the definition for it using the Rank( ) function and the Level function.

```
MEMBER [Measures].[Level Rank] AS  
    Rank( [Sales Territory].[Sales Territory].CurrentMember,  
          [Sales Territory].[Sales Territory]  
          .CurrentMember.Level.Members,  
          [Measures].[Reseller Sales Amount] )
```

3. Create the third calculated measure, name it **Hierarchy Rank**, and provide the definition for it using the Rank( ) function. This time, use a named set for the second argument. Name that set **Hierarchy Set**.

```
MEMBER [Measures].[Hierarchy Rank] AS  
    Rank( [Sales Territory].[Sales Territory].CurrentMember,  
          [Hierarchy Set],  
          [Measures].[Reseller Sales Amount] )
```

4. Define that set above all calculated measures as a set of all hierarchy members:

```
SET [Hierarchy Set] AS  
    [Sales Territory].[Sales Territory].Members
```

5. Include all three calculated measures on columns and execute the query.

6. Verify that the result matches the following image:

	Level	Reseller Sales Amount	Siblings Rank	Level Rank	Hierarchy Rank
All Sales Territories	0	\$80,450,596.98	1	1	1
Europe	1	\$10,870,534.80	2	2	7
France	2	\$4,607,537.94	1	3	11
Germany	2	\$1,983,988.04	3	5	13
United Kingdom	2	\$4,279,008.83	2	4	12
NA	1	(null)	4	4	16
North America	1	\$67,985,726.81	1	1	2
Canada	2	\$14,377,925.60	2	2	5
United States	2	\$53,607,801.21	1	1	3
Central	3	\$7,906,008.18	3	3	8
Northeast	3	\$6,932,842.01	5	5	10
Northwest	3	\$12,435,076.00	2	2	6
Southeast	3	\$7,867,416.23	4	4	9
Southwest	3	\$18,466,458.79	1	1	4
Pacific	1	\$1,594,335.38	3	3	14
Australia	2	\$1,594,335.38	1	6	14

### How it works...

There are two variants of the `Rank()` function, the one with three arguments and the one with two arguments. In this example, we've used the first variant. The following sections of this recipe deal with specifics of the latter variant.

When the third argument is supplied to the `Rank()` function and that argument is supposed to be a numeric expression, the `Rank()` function evaluates the numeric expression for the member specified as the first argument and compares it to the members of the set specified as the second argument. The function then determines the ordinal, the 1-based position of the member in that set according to the results.

`Siblings Rank` is calculated against the current member's siblings, that is, children of the same parent. `Level Rank` is calculated against all members in the level of the current member's level. Finally, the third rank, `Hierarchy Rank`, is calculated against all members in that hierarchy. That is also the only set not dependent on the current context and therefore we've moved it outside the calculation for performance reasons. Notice that we didn't do the same for the previous two rank calculations because we couldn't.

The rules are relatively simple. Whenever there's a set that doesn't depend on the current context, it is better to extract it from the calculation and define it as a named set. That way it will be evaluated only once, after the evaluation of the subselect and slicer, and before the evaluation of axes. During the process of cell evaluation, which is the next phase in the query execution, such set acts like a constant which makes the calculations run faster.

On the other hand, when the set's definition references the current context, that is, current members of the hierarchies on columns or rows, we must leave the set inside the calculation, or else our calculation will not be dynamic. This will manifest in all values being the same.

### There's more...

The other variant of the `Rank()` function, the one without the third argument, is a variant in which the order of members in the set supplied as the second argument plays a crucial role. It's important to remember that the `Rank()` function will not do the sorting for us; it is us who will have to supply the pre-sorted set of members and use it as the second argument of that function. Don't forget that.

### Tie in ranks

Contrary to tied ranks which might happen in the 3-argument version of the `Rank()` function, the 2-argument version will never have tied ranks because the 2-argument version determines the order of members in a set and there's never a tie in the order; all rank positions are unique. However, in the former version, the rank position depends on the value of the numeric expression. Occasionally, that value can repeat which is why the ties exist.

For example, the last two rows of the **Hierarchy Rank** column in the previous image both have the value 14 as their rank. Consequently, there's no rank 15 in that column; the sequence continues with 16.

### Preserving empty rows

Ranks are never null. They are a dense type of calculation. The consequence of that is that we need to handle rows where the original measure is empty.

Take a look at the image with the results again and you'll notice that there is a row with the null value for the `Reseller Sales Amount`, but the ranks themselves are not null. This is something we need to take care of.

## *Navigation*

---

First, we can use the `iif()` statement to handle empty rows, just like we did in the previous recipe. Here's an example of how we can correct rank calculations so that they skip rows with empty values:

```
MEMBER [Measures].[Siblings Rank] AS
    iif( IsEmpty( [Measures].[Reseller Sales Amount] ),
        null,
        Rank( [Sales Territory].[Sales Territory]
            .CurrentMember,
            [Sales Territory].[Sales Territory]
            .CurrentMember.Siblings,
            [Measures].[Reseller Sales Amount]
        )
    )
```

This extra layer in our calculation does indeed eliminate the rows where the original measure is empty, making it a good approach. The problem is that the solution is still not a good enough one. Not yet, and we're about to find out why.

The problem lies in the fact that ranks can have holes; they might not be consecutive. A subtle surprise we may not notice at first, but a perfectly understandable behavior once we realize that ranks are actually calculated on a whole set of members, not just the non-empty ones.

While calculating the rank on all members may or may not be what we want, it is more often that we **do not** want that. In that case, there's still something we can do.

We can make the set in the second argument more compact. We can eliminate all extra members so that: a) our ranks are consecutive and b) our calculation is faster, because it will operate on a smaller set.

The magic is, as many times so far, the `NonEmpty()` function. Here's how the modified query could look:

```
WITH
SET [Hierarchy Set] AS
    NonEmpty( [Sales Territory].[Sales Territory].Members,
        { [Measures].[Reseller Sales Amount] } )
MEMBER [Measures].[Level] AS
    iif( IsEmpty( [Measures].[Reseller Sales Amount] ),
        null,
        [Sales Territory].[Sales Territory]
        .CurrentMember.Level.Ordinal
    )
MEMBER [Measures].[Siblings Rank] AS
    iif( IsEmpty( [Measures].[Reseller Sales Amount] ),
        null,
        Rank( [Sales Territory].[Sales Territory]
```

```

        .CurrentMember,
        NonEmpty( [Sales Territory].[Sales Territory]
            .CurrentMember.Siblings,
            { [Measures].[Reseller Sales Amount] }
        ),
        [Measures].[Reseller Sales Amount] )
    )
MEMBER [Measures].[Level Rank] AS
    iif( IsEmpty( [Measures].[Reseller Sales Amount] ),
        null,
        Rank( [Sales Territory].[Sales Territory]
            .CurrentMember,
            NonEmpty( [Sales Territory].[Sales Territory]
                .CurrentMember.Level.Members,
                { [Measures].[Reseller Sales Amount] }
            ),
            [Measures].[Reseller Sales Amount] )
    )
MEMBER [Measures].[Hierarchy Rank] AS
    iif( IsEmpty( [Measures].[Reseller Sales Amount] ),
        null,
        Rank( [Sales Territory].[Sales Territory]
            .CurrentMember,
            [Hierarchy Set],
            [Measures].[Reseller Sales Amount] )
    )
SELECT
    { [Measures].[Level],
        [Measures].[Reseller Sales Amount],
        [Measures].[Siblings Rank],
        [Measures].[Level Rank],
        [Measures].[Hierarchy Rank] } ON 0,
    { [Sales Territory].[Sales Territory].AllMembers } ON 1
FROM
    [Adventure Works]

```

From the query above, it's evident that we've added the outer `iif()` clause in each calculated member. That's the part that handles the detection of empty rows.

Additionally we wrapped all sets in a `NonEmpty()` function, specifying that the second argument of that function should be the `Reseller Sales Amount` measure. Notice that we've done it in the named set too, not just for inner sets in `Rank()` functions, but in that case we applied the `NonEmpty()` function in the set, not inside the `Rank()` function. This is important because as we said before, the set is invariant to the context and it makes sense to prepare it in full in advance.

## *Navigation*

---

Now, the effect of the `NonEmpty()` function may not be visible in 3-part variants of the `Rank()` function, especially if there are no negative values, because the negative values come after the zeros and nulls. It is immediately clear when we have discontinuous ranks. Therefore, we're going to make another example, a really simple example using the product colors.

Write the following query:

```
WITH
    SET [Color Set] AS
        [Product].[Color].[Color].Members
    SET [Color Set NonEmpty] AS
        NonEmpty( [Product].[Color].[Color].Members,
            { [Measures].[Internet Sales Amount] } )
MEMBER [Measures].[Level Rank] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null,
        Rank( [Product].[Color].CurrentMember,
            [Color Set] )
    )
MEMBER [Measures].[Level Rank NonEmpty] AS
    iif( IsEmpty( [Measures].[Internet Sales Amount] ),
        null,
        Rank( [Product].[Color].CurrentMember,
            [Color Set NonEmpty] )
    )
SELECT
    { [Measures].[Internet Sales Amount],
        [Measures].[Level Rank],
        [Measures].[Level Rank NonEmpty] } ON 0,
    { [Product].[Color].[Color].Members } ON 1
FROM
    [Adventure Works]
```

This query displays all product colors and their corresponding sales values. The query also contains two named sets, one having all product colors, the other having only the colors with sales. The two calculated measures return ranks, one using the first set and the other using the second named set.

Now execute that query and observe the results:

The screenshot shows a Results grid with four columns: Internet Sales Amount, Level Rank, and two named sets named 'Level Rank' and 'NonEmpty'. The 'Level Rank' column contains integer values from 1 to 10. The 'NonEmpty' column contains values corresponding to the 'Level Rank' column but excludes rows where the 'Internet Sales Amount' is null or zero. The 'NonEmpty' column has values 1 through 8.

	Internet Sales Amount	Level Rank	Level Rank NonEmpty
Black	\$8,838,411.96	1	1
Blue	\$2,279,096.28	2	2
Grey	(null)	(null)	(null)
Multi	\$106,470.74	4	3
NA	\$435,116.69	5	4
Red	\$7,724,330.52	6	5
Silver	\$5,113,389.08	7	6
Silver/Black	(null)	(null)	(null)
White	\$5,106.32	9	7
Yellow	\$4,856,755.63	10	8

In the above image the highlighted section shows the adjusted ranks.

The left rank runs from 1 to 10 and occasionally doesn't display the result. The right rank runs from 1 to 8 because the empty rows are excluded in the named set used as its second argument.

Now, imagine we apply the NON EMPTY operator on rows. That action would eliminate the grey and silver/black colors. In this situation, only the right rank would return the results as expected in most cases. Now you know how to make the rank work this way, so use it.

One more thing: in the previous query, there was a set of members on rows, including all product colors. Notice that the same set appears in both named sets.

The first named set is defined exactly as the set on rows. The second named set contains non-empty colors only. That definition is actually the equivalent of what we would get on rows if the NON EMPTY operator was applied. In short, we can put one of the named sets on rows instead and have a more manageable code.

### Ranks in multidimensional sets

The first argument of the Rank( ) function can in fact be a tuple, not just a member, which enables calculation of ranks for multidimensional sets.

What we need to be extra careful about in this case is the dimensionality of first two arguments. **The dimensionality must match.** In other words, the order of hierarchies in a tuple must match the order of hierarchies in a set.

Everything that's been said for one-dimensional sets applies here too.

## The pluses and minuses of named sets

When named sets are used in calculations, calculations can either profit from it or perform worse. The latter is more often the case, but it certainly doesn't classify sets in MDX as bad per se. Quite the opposite – sets are a very valuable mechanism in MDX, just like the fire in our ordinary life. If you learn to tell the difference when to use them and when not, you're safe with named sets and sets in general.

Currently, the SSAS engine is not optimized to run in block computation mode when sets (named sets or set aliases, that is, sets defined inline in MDX expressions) are used as an argument of functions that perform the aggregation of values. Typical examples are the `Sum()` and the `Count()` functions. Those functions are optimized to run in block mode but if you provide a named set or set alias as their argument, you will turn this optimization off and those functions will run in a much slower cell-by-cell mode. This lack of support may change in future releases of SSAS but for the time being you should avoid using named sets and set aliases in those functions unless the situation really requires so. This requirement will rarely be mentioned for named sets, but it may be a valid scenario for set aliases if you need to achieve very flexible and dynamic calculations and you're ready to make a compromise in performance as a tradeoff.

On the other hand, other functions like the `Rank()` function can profit from using named sets. This is because a different execution pattern is applied in their case. Let's illustrate this with an example.

Let's suppose that the `Sum()` and `Count()` functions iterate along a vertical line in cell-by-cell mode and compress that line in a dot when operating in the block mode. Naturally, returning a value in a single point is a much faster operation than iterating along the line.

The rank has to be evaluated for each cell; it does not run in block mode. In other words, it has to go along that vertical line anyway. Unfortunately, that's not the entire issue. Along the way, the `Rank()` function has to evaluate the set on which to perform the rank operation, a set which can be visualized as yet another, this time a horizontal line. The problem is that this new line can change per each dot on the vertical line which means the engine can anticipate nothing but an irregular surface on which to calculate rank values. This is slow because all of a sudden we have two loops, one for the vertical path and the other for the horizontal. Could this be optimized? Yes, by removing the unnecessary loop again which in this case is the inner loop. Here's how.

When we define a named set and use it inside the `Rank()` function, we're actually telling the SSAS engine that this set is invariant to the current context and as such can be treated as constant. That is, we declare there's a rectangle, not an irregular surface. This way the engine can traverse the vertical line and not bother evaluating the horizontal line which is a kind of block optimization again. In other words, it can perform much better.

Other situations when you might want to use named sets is when you're not aggregating values but instead applying set operations like the difference or intersection of sets. In those situations sets also perform well, much better than using the `Filter()` function to do the same thing, for example.

To conclude, named sets are a mixed bag, use them but with caution. Don't forget you can always test which version of your calculation performs better, the one with named sets or the one without them. The recipe *Capturing MDX queries generated by SSAS front-ends* in chapter 9 will show you how to test that.

## See also

The two recipes prior to this are part of the 3-part series of recipes in this chapter dealing with the topic of relative calculations: *Calculating various percentages* and *Calculating various averages*. It is recommended that you read all of them in order to get more thorough picture.

Defining a generic rank measure, the one which calculates the rank for any hierarchy placed on rows or columns, is a possible but more difficult scenario. *Chapter 7, Context-aware calculations*, deals with this topic. If you're interested in a generic solution, read that chapter from the start to the end. A single recipe, the last recipe in this case, will not be sufficient to explain everything you need to take care about when using or creating generic calculations.



# 5

## Business Analytics

In this chapter, we will cover:

- ▶ Forecasting using linear regression
- ▶ Forecasting using periodic cycles
- ▶ Allocating non-allocated company expenses to departments
- ▶ Calculating number of days from the last sales to identify the slow-moving goods
- ▶ Analyzing fluctuation of customers
- ▶ Implementing ABC analysis

### Introduction

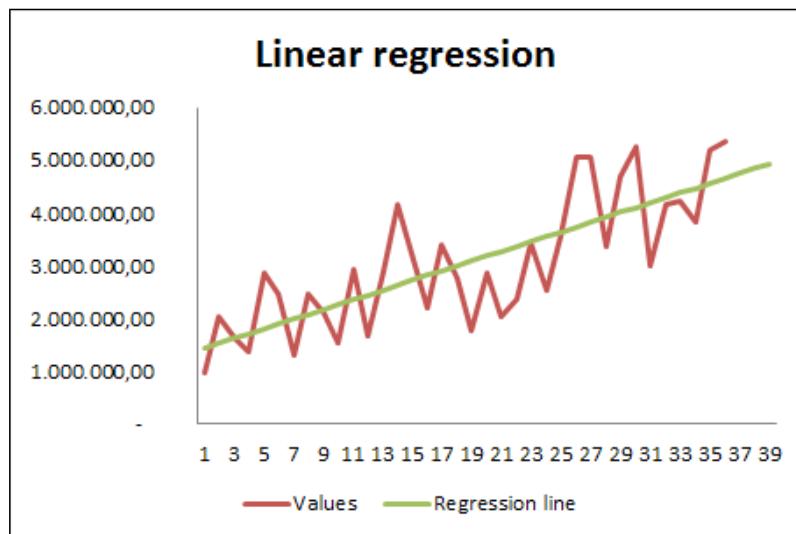
In this chapter we'll focus on how to perform some of the typical business analyses such as forecasting, allocating values, and calculating the number of days from an event. We'll learn how to use linear regression to calculate the trend line which tells us how the business is doing, but also how to predict future results using something better than the trend line. Following that, we'll repeat what we learned about calculating the difference between two dates in the second chapter. We'll need it here to perform an analysis of our products and determine if there are "slow-movers" or "non-movers". We'd like to identify them in order to either free the space in stores for something better or to sell them at a discounted value, because they are only generating expense. Finally, we'll show how to allocate common expenses on the individual departments in order to have a better picture of how each of our profit centers is performing.

The second half of this chapter shows how to determine the behavior of individual members. In this part we have two recipes. One is related to something we can manage, like the products we keep on stock, have on shelves, and sell to the customers. It teaches us how to classify items and separate them into the A, B, or C group, all based on the contribution of each item. The other recipe deals with the new, returning, and lost customer. It illustrates an approach which becomes useful when we need to track changes in periods. That again allows us to have a different perspective on our business.

Let's start!

## Forecasting using the linear regression

This recipe illustrates the method of estimating the future values based on the linear trend. The linear trend is calculated using the least-square distance, a statistical method of determining the line that fits the best through the set of values in the chart, as displayed on the image below:



There are two characteristics that describe the trend line: the slope and the intercept. They are the necessary ingredients in the regression line equation:  $y = a \cdot x + b$ . The "a" in the equation is the slope, growth rate or the gradient. When positive, the line progresses with values with every increase of the "x" in it. We say that the trend is positive, that we're witnessing the growth.

The "b" is the value that "y" has before we apply any "x" to it, the initial elevation from the x-axis in the chart. Apart from that, it has no influence on the change of "x" in the equation. In the previous image, the "b" is value where the line starts (where "x" is zero). The "a" can be understood as the angle that the line forms with the axis at the bottom. The greater the angle, the faster the values of "y" increase.

Linear regression is often used in a way that "x" is a period and "y" is the value in a particular time period, for example, a month. Once determined, the regression line can be used to calculate the value of "y" for any given "x", including, but not limited to the initial set of members. The previous image clearly shows that.

The regression line is often used to forecast values outside the initial period. For example, we can use a period of 60 months and calculate the value for each of the following 12 months.

This recipe shows how to estimate the values of the two months subsequent to the period of 36 months that were used in the calculation. The image you saw in the introduction of this recipe shows the regression line and forecasted values for those two months.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Date** dimension. Here's the query we'll start from:

```
SELECT  
    { [Measures].[Sales Amount] } ON 0,  
    { [Date].[Fiscal].[Month].MEMBERS } ON 1  
FROM  
    [Adventure Works]
```

Once executed, the query returns 39 months, from July 2005 to August 2008 and one extra month in the year 2010, November.



The **Date** dimension in the Adventure Works DW 2008R2 database does not follow the best practice guide which says that all months should be consecutive with no holes inside the year. Because of that, we'll focus our attention on up to August 2008 (including) and disregard the following month, November 2010, which although is the next member in the hierarchy, is in fact more than two years apart.

The last month with data is July 2008. Based on the value shown in that month, which is several times smaller than the value of the previous month, we can conclude that July 2008 is only partially full. Therefore, our task will be to forecast the value July will have when it ends and to forecast the value for August, all based on the sales of 3 previous years. As explained in the introduction, we'll use the linear regression method in this recipe while the next recipe will show how to do the same using another approach – the method of periodic cycles.

## How to do it...

Follow these steps to add the linear regression code to your MDX:

1. Add the `WITH` part of the query.
2. Create two named sets. One named `Full Period` and the other named `Input Period`. The definition for the first set should be the same set that is on the axis 1. The definition of the second set should be a range of months up to and including **June 2008**. Don't forget to use the `Fiscal` hierarchy.
3. Create several calculated members. The first, `Input X`, is a measure that determines the rank of the current fiscal member in the named set `Input Period`.
4. The second, `Input Y`, is a measure that is an alias for the `Sales Amount` measure.
5. The third, `Output X`, is a measure similar to `Input X`, only this time we're determining the rank in the first set, `Full Period`.
6. Finally, the fourth calculated member, `Forecast`, will be a measure that contains the formula which combines all the previous calculated members and sets and calculates the regression line using the `LinRegPoint()` function:

```
LinRegPoint(  
    [Measures].[Output X],  
    [Input Period],  
    [Measures].[Input Y],  
    [Measures].[Input X]  
)
```

7. Specify `$#,##0.00` as a format string of the `Forecast` measure.
8. Add the `Forecast` measure on axis 0, next to the `Sales Amount` measure.
9. Verify that your query looks like the following one. Execute it.

```
WITH  
SET [Full Period] AS  
    [Date].[Fiscal].[Month].MEMBERS  
SET [Input Period] AS  
    { null : [Date].[Fiscal].[Month].&[2008]&[6] }  
MEMBER [Measures].[Input X] AS  
    Rank( [Date].[Fiscal].CurrentMember, [Input Period] )  
MEMBER [Measures].[Input Y] AS
```

```

[Measures].[Sales Amount]
MEMBER [Measures].[Output X] AS
    Rank( [Date].[Fiscal].CurrentMember, [Full Period] )
MEMBER [Measures].[Forecast] AS
    // = Output Y
    LinRegPoint(
        [Measures].[Output X],
        [Input Period],
        [Measures].[Input Y],
        [Measures].[Input X]
    )
    , FORMAT_STRING = '$#,##0.00'
SELECT
    { [Measures].[Sales Amount],
    [Measures].[Forecast] } ON 0,
    { [Date].[Fiscal].[Month].MEMBERS } ON 1
FROM
    [Adventure Works]

```

10. Verify that the result matches the following image. The highlighted cells contain the forecasted values:

	Sales Amount	Forecast
November 2007	\$4,680,142.51	\$4,014,228.77
December 2007	\$5,242,736.50	\$4,106,170.04
January 2008	\$3,002,792.27	\$4,198,111.32
February 2008	\$4,163,246.63	\$4,290,052.59
March 2008	\$4,220,276.16	\$4,381,993.87
April 2008	\$3,813,373.94	\$4,473,935.14
May 2008	\$5,193,592.51	\$4,565,876.42
June 2008	\$5,364,840.18	\$4,657,817.69
July 2008	\$50,840.63	\$4,749,758.97
August 2008	(null)	\$4,841,700.24
November 2010	(null)	\$4,933,641.52

## How it works...

In this example we've used a bit more calculated members and sets than necessary. We've defined two named sets and three calculated members before we came to the main part of the query: the fourth calculated member, which contains the formula for forecasting values based on the linear trend. We did this extra bit of work in order to have a simple and easy-to-memorize expression for the otherwise complicated `LinRegPoint()` function.

The `LinRegPoint()` function has four parameters. Names used in this example suggest what these parameters represent. Take a closer look at it one more time, notice their names, and then come back.

The last three parameters (`Input Period`, `Input X`, and `Input Y`) are used to determine the regression line  $y = a \cdot x + b$ . Here we've specified a set of members (July 2005 – June 2008 period), their ordinal positions in that set (values 1 – 36 obtained using the `Rank()` function on that period), and their corresponding values (measures `Sales Amount`). We've prefixed all of them with "Input" in order to emphasize that these parameters are used as the input for the calculation of the regression line.

Once the regression line is calculated internally, it is used in combination with the first parameter, `Output X`. This two-step process is a composition of functions where the result of the inner one is used as the argument in the outer one. This is what makes the `LinRegPoint()` function difficult to understand and remember. Hopefully, by making a distinction between parameters, that is, prefixing them as input and output parameters, the function becomes less difficult.

Now starts the second phase, the invisible outer function.

The first parameter (`Output X`) is used to specify the ordinal position of the member which we want to calculate the value for. Notice we've used a different set here because our primary need was to get the value for members whose position is 37 and 38, July and August 2008.

Notice also that the measure `Forecast` represents the `Output Y` value which was hinted at in the code. In other words, for a given regression line (based on the "Input" parameters), we have calculated the "y" value for any given "x", including, but not limited to the initial set of members. That was step two in the evaluation of the `LinRegPoint()` function.

## There's more...

The `LinRegPoint()` is not the only MDX function which can be used to calculate the values based on the regression line. Remember we said this function evaluates in a two-step process. The result of the inner step is not one but actually two numbers. One is the slope, the other intercept. In other words, the "a" and "b" in the  $y = a \cdot x + b$  equation.

You might feel more comfortable using the slope ("a") and the intercept ("b") in order to forecast a value. If that's the case then here's the add-on to the previous example.

Add this part of the code in the previous query, add those three calculated members on axis 0 and run the query:

```

MEMBER [Measures].[Slope] AS // = a
    LinRegSlope(
        [Input Period],
        [Measures].[Input Y],
        [Measures].[Input X]
    )
    , FORMAT_STRING = '#,##'
MEMBER [Measures].[Intercept] AS // = b
    LinRegIntercept(
        [Input Period],
        [Measures].[Input Y],
        [Measures].[Input X]
    )
    , FORMAT_STRING = '#,##'
MEMBER [Measures].[Verify y = a * x + b] AS
    [Measures].[Slope] * [Measures].[Output X] +
    [Measures].[Intercept]
    // y = a * x + b
    , FORMAT_STRING = '$#,##0.00'

```

The result will look like in the following image:

	Sales Amount	Forecast	Slope	Intercept	Verify y = a * x + b
November 2007	\$4,680,142.51	\$4,014,228.77	91,941	1,347,932	\$4,014,228.77
December 2007	\$5,242,736.50	\$4,106,170.04	91,941	1,347,932	\$4,106,170.04
January 2008	\$3,002,792.27	\$4,198,111.32	91,941	1,347,932	\$4,198,111.32
February 2008	\$4,163,246.63	\$4,290,052.59	91,941	1,347,932	\$4,290,052.59
March 2008	\$4,220,276.16	\$4,381,993.87	91,941	1,347,932	\$4,381,993.87
April 2008	\$3,813,373.94	\$4,473,935.14	91,941	1,347,932	\$4,473,935.14
May 2008	\$5,193,592.51	\$4,565,876.42	91,941	1,347,932	\$4,565,876.42
June 2008	\$5,364,840.18	\$4,657,817.69	91,941	1,347,932	\$4,657,817.69
July 2008	\$50,840.63	\$4,749,758.97	91,941	1,347,932	\$4,749,758.97
August 2008	(null)	\$4,841,700.24	91,941	1,347,932	\$4,841,700.24
November 2010	(null)	\$4,933,641.52	91,941	1,347,932	\$4,933,641.52

Three extra measures were the `Slope`, `Intercept`, and `Verify y = a * x + b` measures. The first two were defined using the `LinRegSlope()` and `LinRegIntercept()` functions respectively. The same last three parameters were used here as in the `LinRegPoint()` function before. What `LinRegPoint()` does extra is that it takes one step further and combines those two values with an additional parameter in order to calculate the final "y".

From the previous image it is clear that the `LinRegSlope()` and `LinRegIntercept()` can be used instead of the `LinRegPoint()` function – the values are exactly the same. All it takes is to combine them in the equation  $y = a \cdot x + b$ . Therefore, it is up to you to choose the preferred way for you to forecast the value based on the regression line.

### Tips and tricks

The `Rank()` function is used to determine the ordinal position of a member in a set. Here we've used it twice: to calculate the position of members in the `Input Period` set and to calculate the position of members in the `Full Period` set.

`Rank()` performs better if we extract the set used as the second argument and define it as a separate named set, which we did. This is because in that case the set is evaluated only once and not repeatedly for each cell.

### Where to find more information?

Here are the MSDN links for the definition of the functions mentioned in this recipe:

<http://tinyurl.com/LinRegPoint>

<http://tinyurl.com/LinRegSlope>

<http://tinyurl.com/LinRegIntercept>

Here's a link to the Moshav Pasumansky's coverage of the linear regression example:

<http://tinyurl.com/MoshavLinReg>

### See also

The recipe *Forecasting using the periodic cycles* covers a similar topic.

## Forecasting using the periodic cycles

Linear trend, covered in the recipe *Forecasting using the linear regression*, is just an overview of how the business is going. The regression line serves us merely as an indicator of where we will be in the future if everything continues the same. What it doesn't tell us, at least not well enough, is what the individual values on that path will be. The values that method returns are too simplified, too imprecise.

Of course, there are other ways of forecasting the future values, i.e. using polynomial curves or similar, which, generally speaking, will fit better to the shape of achieved business values and therefore represent future values more precisely. However, we won't make an example using them; we will use something simpler but effective as well.

The characteristic of many businesses is that events tend to repeat themselves after some time. In other words, many of them have a cyclic behavior where that period is a year, a month, a week, a day, or whatever.

The simplification can be summarized like this: there's an envelope of events taking place during a cycle and there is a linear trend which says how much better or worse the next period is going to be compared to the previous one. When we say "the envelope", we mean a shape of events and their values. For example, there might be a spike in the curve for the spring/autumn parts when people buy shoes, there might be a spike in the activity in the end of the month when people need to finish their reports, there might be a spike in the middle of the week because people are more business-focused in that part of the week, or there might even be a spike in the early morning or late afternoon because that's when people travel to and from the work. All in all, things are not flat and moreover, the shape of their non-flatness more or less looks the same in every cycle.

If we calculate the values for the next cycle using the trend line but keep the shape of the events in the cycle, we will have a better approximation and estimation of the future results than by simply using a linear trend. It's not that we'll replace it with something else. No, we're just going to improve it by adjusting the stiffness of the regression line.

## Getting ready

Open **Business Intelligence Development Studio** (BIDS) and then open **Adventure Works DW 2008** solution. Double-click **Adventure Works** cube and go to **Calculations** tab. Choose **Script View**. Position the cursor at the end of the script where you'll create a calculated measure Sales, null by default, and then scope that measure so that only the values up to the **December 2007** are not null.

```
Create Member CurrentCube.[Measures].[Sales] As null;

Scope( ( { null : [Date].[Fiscal].[Month].&[2007]&[12] } ,
         [Measures].[Sales] ) );
      This = [Measures].[Sales Amount];
End Scope;
```

We are going to use this measure to forecast its values for the next half of the year. We've deliberately used a calculated measure being equal to the original measure, Sales Amount, because the latter one already has values in that period. In other words, we'll be able to test how good our forecasting expression is.

Once you're done, deploy the changes, preferably using the BIDS Helper, if not BIDS itself.

Then start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Write this query:

```
SELECT
    { [Measures].[Sales] } ON 0,
    { Descendants( [Date].[Fiscal].[Fiscal Year].&[2008],
                    [Date].[Fiscal].[Month] ) } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns 12 months of the fiscal year 2008. Half of them have values, the other half are blank:

	Sales
July 2007	\$3,552,319.38
August 2007	\$5,060,385.02
September 2007	\$5,057,832.17
October 2007	\$3,362,565.46
November 2007	\$4,680,142.51
December 2007	\$5,242,736.50
January 2008	(null)
February 2008	(null)
March 2008	(null)
April 2008	(null)
May 2008	(null)
June 2008	(null)

Now let's see if we can predict the values for those empty six months based on the previous years and the shape of the curve in those previous fiscal years on a monthly granularity.

### How to do it...

We're going to define six calculated measures and add them to the columns axis.

1. Add the WITH part of the query.

2. Create the calculated measure Sales PP and define it as the value of the measure Sales in the parallel period.
3. Create the calculated measure Sales YTD and define it as a year-to-date value of the Sales measure.
4. Create the calculated measure Sales PP YTD and define it as a year-to-date value of the Sales measure in the parallel period. The YTD value should only be calculated up to the parallel period of the last month with the data. For this you might need some help, so here's the syntax:

```
Sum(
    PeriodsToDate( [Date].[Fiscal].[Fiscal Year],
    ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        Tail( NonEmpty( { null :
            [Date].[Fiscal].CurrentMember },
            [Measures].[Sales] ),
        1 ).Item(0) ) ),
    [Measures].[Sales] )
```

5. Define a calculated measure Ratio being equal to the ratio of YTD value vs. PP YTD value. Don't forget to take care of division by zero.
6. Define a calculated measure Forecast being equal to the parallel period's value of the measure Sales corrected (multiplied) by the measure Ratio.
7. Define a calculated measure Forecast YTD being equal to the sum of year-to-date values of the measures Sales and Forecast, but only for months when Sales is null.
8. Add all those measures on columns of the query and verify that your query looks like the following one:

```
WITH
MEMBER [Measures].[Sales PP] AS
    ( ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
        [Date].[Fiscal].CurrentMember ),
    [Measures].[Sales] )
    , FORMAT_STRING = '$#,##0.00'
MEMBER [Measures].[Sales YTD] AS
    Sum(
        PeriodsToDate( [Date].[Fiscal].[Fiscal Year],
        [Date].[Fiscal].CurrentMember ),
        [Measures].[Sales] )
    , FORMAT_STRING = '$#,##0.00'
MEMBER [Measures].[Sales PP YTD] AS
    Sum(
        PeriodsToDate( [Date].[Fiscal].[Fiscal Year],
        ParallelPeriod( [Date].[Fiscal].[Fiscal Year], 1,
```

```
Tail( NonEmpty( { null :
                  [Date].[Fiscal].CurrentMember },
                  [Measures].[Sales] ),
      1 ).Item(0) ) ),
[Measures].[Sales] )
, FORMAT_STRING = '$#,##0.00'
MEMBER [Measures].[Ratio] AS
    iif( [Measures].[Sales PP YTD] = 0,
         null,
         [Measures].[Sales YTD] / [Measures].[Sales PP YTD] )
, FORMAT_STRING = '#,##0.00'
MEMBER [Measures].[Forecast] AS
    iif( IsEmpty( [Measures].[Sales] ),
         [Measures].[Ratio] * [Measures].[Sales PP],
         null )
, FORMAT_STRING = '$#,##0.00'
MEMBER [Measures].[Forecast YTD] AS
    iif( IsEmpty( [Measures].[Sales] ),
         Sum(
             PeriodsToDate( [Date].[Fiscal].[Fiscal Year],
                           [Date].[Fiscal].CurrentMember ),
             [Measures].[Sales] +
             [Measures].[Forecast] ),
         null )
, FORMAT_STRING = '$#,##0.00'
SELECT
{ [Measures].[Sales],
  [Measures].[Sales PP],
  [Measures].[Sales YTD],
  [Measures].[Sales PP YTD],
  [Measures].[Ratio],
  [Measures].[Forecast],
  [Measures].[Forecast YTD]
} ON 0,
{ Descendants( [Date].[Fiscal].[Fiscal Year].&[2008],
               [Date].[Fiscal].[Month] ) } ON 1
FROM
[Adventure Works]
```

9. Execute the query.

10. Verify that the result matches the following image. The highlighted cells represent the forecasted values:

	Sales	Sales PP	Sales YTD	Sales PP YTD	Ratio	Forecast	Forecast YTD
July 2007	\$3,552,319.38	\$2,894,054.68	\$3,552,319.38	\$2,894,054.68	1.23	(null)	(null)
August 2007	\$5,060,385.02	\$4,147,192.18	\$8,612,704.40	\$7,041,246.87	1.22	(null)	(null)
September 2007	\$5,057,832.17	\$3,235,826.19	\$13,670,536.57	\$10,277,073.06	1.33	(null)	(null)
October 2007	\$3,362,565.46	\$2,217,544.45	\$17,033,102.02	\$12,494,617.50	1.36	(null)	(null)
November 2007	\$4,680,142.51	\$3,388,911.41	\$21,713,244.54	\$15,883,528.92	1.37	(null)	(null)
December 2007	\$5,242,736.50	\$2,762,527.22	\$26,955,981.04	\$18,646,056.13	1.45	(null)	(null)
January 2008	(null)	\$1,756,407.01	\$26,955,981.04	\$18,646,056.13	1.45	\$2,539,178.99	\$29,495,160.03
February 2008	(null)	\$2,873,936.93	\$26,955,981.04	\$18,646,056.13	1.45	\$4,154,754.70	\$33,649,914.72
March 2008	(null)	\$2,049,529.87	\$26,955,981.04	\$18,646,056.13	1.45	\$2,962,936.93	\$36,612,851.66
April 2008	(null)	\$2,371,677.70	\$26,955,981.04	\$18,646,056.13	1.45	\$3,428,655.30	\$40,041,506.95
May 2008	(null)	\$3,443,525.25	\$26,955,981.04	\$18,646,056.13	1.45	\$4,978,189.52	\$45,019,696.47
June 2008	(null)	\$2,542,671.93	\$26,955,981.04	\$18,646,056.13	1.45	\$3,675,855.95	\$48,695,552.43

### How it works...

The first calculated measure, Sales PP, represents a value of the measure Sales in the parallel period, the previous year in this case. This value will serve as a basis for calculating the value of the same period in the current year later on. Notice that the measure Sales PP has values in each month.

The second calculated measure, Sales YTD, is used as one of the values in the Ratio measure, the measure which determines the growth rate and which is subsequently used to correct the parallel period value by the growth ratio. Notice that the **Sales YTD** values remain constant after **December 2007** because the values of the measure it depends on, Sales, are empty in the second half of the fiscal year.

The third calculated measure, Sales PP YTD, is a bit more complex. It depends on the Sales PP measure and should serve as the denominator in the growth rate calculation. However, values of the Sales PP measure are not empty in the second half of the fiscal year which means that the cumulative sum would continue to grow. This would corrupt the ratio because the ratio represents the growth rate of the current period versus the previous period, the year-to date of each period. If one stops, the other should stop too. That's why there's an additional part in the definition which takes care of finding the last period with data and limiting the parallel period to the exact same period. This was achieved using the *Tail-NonEmpty-Item* combination where the NonEmpty( ) function looks for non-empty members up to the current period, Tail( ) takes the last one, and the Item( ) function converts that into a single member. Notice in the previous image that it works exactly as planned. Sales PP YTD stops growing when Sales YTD stops increasing, all because Sales is empty.

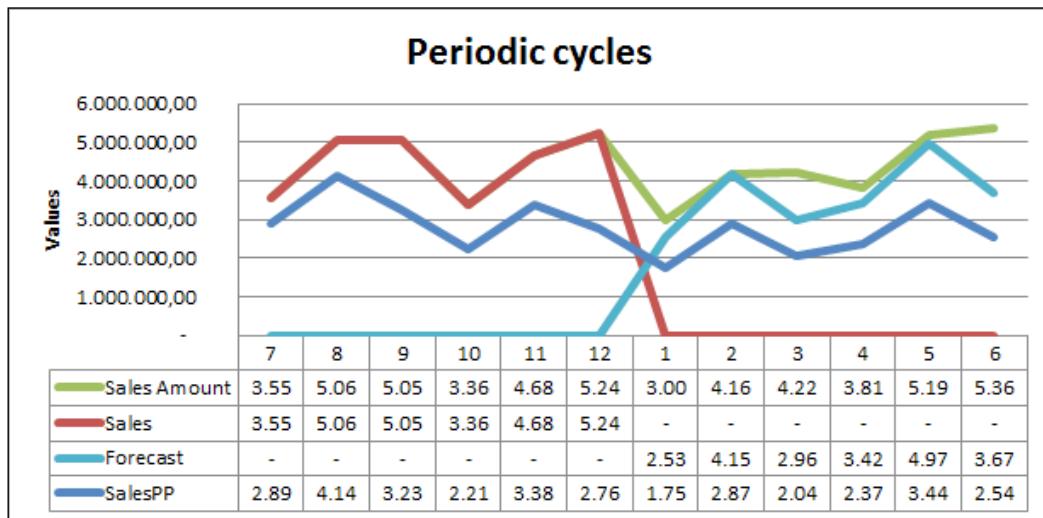
Once we have that fixed, we can continue and build a ratio measure which, as stated earlier, represents the growth rate between all the periods in the current fiscal year so far versus the same periods in the previous year. As both of the year-to-date measures become constant at some point in time, the ratio also becomes constant as visible in the previous image.

The Forecast measure is where we take the value of the same period in the previous year and multiply it by the growth rate in form of the Ratio measure. That means we get to preserve the shape of the curve from the previous year while we increase or decrease the value of it by an appropriate factor. It is important to notice that the ratio is not constant, it changes in time, and it adjusts itself based on all the year-to-date values, as long as they exist. When there aren't any, it becomes a constant.

Finally, the last calculated measure, Forecast YTD, is here to show what would have happened to the Sales YTD measure if there were values in the last six months of that fiscal year. Of course, it does that by summing the year-to-date values of both the Sales and Forecast measures.

### There's more...

The following image illustrates the principle behind the periodic cycles' calculation quite easily:



The **Sales Amount** series represents values of the original measure, the one we used in the scope. The **Sales** series represents values of the calculated measure which has the same value as the original Sales Amount measure, which stops in December, as visible in the previous image. The **Forecast** series continues where the **Sales** series stops and takes the shape of the last series, the **SalesPP** series, which represents the values of the Sales Amount measure in the previous year's months.

The **Sales** series versus the **Sales PP** series determines the ratio. Once there is no data, the **Forecast** series takes the shape of the **Sales PP** series, but multiplied by the ratio.

The previous image illustrates that the forecasted values are not exact with the actual **Sales Amount** data, the measure that contains real values, but they are much better than they would be using the linear regression only. There's certainly a possibility to improve the formula for the ratio calculation in a way that it incorporates many previous periods and not just the previous year as in this example, in addition to adding more weight to the recent years, perhaps. You are encouraged to experiment with this recipe in order to get more accurate forecasts of the data.

### Other approaches

The technique described in this recipe will fail to achieve good values when applied to non-periodic business processes or processes with periodic cycles but irregular events inside the cycles. The recipe *Calculating moving averages* in chapter 2 might be a good starting point in those situations.

If you find this recipe interesting, here's another variant in a blog article by Chris Webb, one of the reviewers of this book:

<http://tinyurl.com/ChrisSeasonCalcs>

Finally, Data Mining and its Time Series algorithm can be a more accurate way of forecasting future values, but that's outside the scope of this book. If you're interested in learning more about it, here are two links to get you started in that direction. One explains what the Time Series algorithm is and the second is Chris Webb's application of it:

<http://tinyurl.com/DataMiningTimeSeries>

<http://tinyurl.com/ChrisDataMiningForecast>

### See also

The recipe *Forecasting using the linear regression* deals with a similar topic.

## Allocating the non-allocated company expenses to departments

There are two types of expenses, direct and indirect. It is relatively easy for any company to allocate a direct expense to a corresponding department because it actually happens in that department and is in no way associated with other departments. This type of expense is usually allocated in real-time, at the moment it enters the system. An example of such an expense is salaries.

The other type of expense is an indirect expense, for example, an electricity or heating bill. These expenses are usually entered in the system using a special department like the corporation itself, a common expense department, or simply nothing (a null value). The company, in order to have a clear picture of how each department is doing, usually wants to allocate these expenses to all departments. Of course, there are many ways how it can be done and none of them are perfect. Because of that, the allocation is usually done at the later stage, not in the real-time.

For example, the allocation can be implemented in the ETL (Extract, Transform, Load) phase. In that case, the modified data enters the Data Warehouse (DW) and later the cube. The other approach is to leave the data as-is in the DW and use the MDX script in the cube to modify it. The first scenario will have better performance, the second more flexibility.

Flexibility means that the company can apply several calculations and choose which set of allocation keys fits the best in a particular case, a sort of what-if analysis. It also means that the company may decide to use multiple allocation schemes at the same time, one for each business process. However, the allocation keys can change in time. Maintaining complex scope statements can become difficult. The performance of such scopes will degrade too. In short, each approach has its advantages and disadvantages.

In this recipe, we'll focus on the second scenario, the one where we apply MDX. After all, this is a book about MDX and its applications. Here, you'll learn how to perform the scope type allocation, how to allocate the values from one member in the hierarchy on its siblings so that the total remains the same. Once you learn the principle, you will be able to choose any set of allocation keys and apply it whenever and wherever required.

## Getting ready

In this example, we're going to use the measure **Amount** and two related dimensions, **Account** and **Department**.

Open **Business Intelligence Development Studio** (BIDS) and then open **Adventure Works DW 2008** solution. Double-click **Adventure Works** cube. Locate the measure **Amount** in the **Financial Reporting** measure group. Go to the next tab, **Dimension Usage**, and check that the two dimensions mentioned previously are related to this measure group.

Now, go to the **Cube Browser** tab and create a pivot which shows accounts and departments on rows and the value of the measure **Amount** in the data part. Expand the accounts until the member **Operating Expenses** located under **Net Income | Operating Profit** is visible. Expand the departments until all of them are visible. Once you're done, you'll see the following image:

Drop Filter Fields Here					Drop Column Fields
Account Level 01 ▾   Account Level 02		Account Level 03	Department Level 01 ▾   Department Level 02		Amount
<input checked="" type="checkbox"/> Balance Sheet					\$0.00
<input type="checkbox"/> Net Income	<input type="checkbox"/> Operating Profit	<input type="checkbox"/> Operating Expenses	<input type="checkbox"/> Corporate	Executive General and Administration	\$397,061.00
				Inventory Management	\$1,316,774.00
				Manufacturing	\$827,480.00
				Quality Assurance	\$575,197.00
				Research and Development	\$11,647,823.00
				Sales and Marketing	\$660,885.50
				Total	\$27,661,868.50
			<input checked="" type="checkbox"/> Gross Margin		\$44,390,103.00
					\$50,017.50
			<input checked="" type="checkbox"/> Other Income and Expense		
			<input checked="" type="checkbox"/> Taxes		\$4,168,749.00
					NA
<input checked="" type="checkbox"/> Statistical Accounts					

Notice a strange thing in the result. The total for the **Corporate** department in the **Operating Expenses** account (the highlighted 27+ million) is not equal to the sum of the individual departmental values. Roughly 12 million is missing. Why's that?

Double-click the **Department** dimension in the **Solution Browser** pane. Select the attribute that represents the parent key and check its property **MembersWithData** found under the **Parent-Child** group of properties. Notice that it's bold, which means the property has a non-default value. The option that was selected is **NonLeafDataHidden**. That means this parent-child hierarchy hides members which represent the data members of the non-leaf members. In this case there's only one, the **Corporate** member, found on the first level in that hierarchy.

Change the value of this property to **NonLeafDataVisible** and deploy the solution, preferably using the **Deploy Changes Only** deployment mode (configurable in the **Project Properties** form) in order to speed up the process.

Now, return to the **Cube Browser** and reconnect to the cube.

A new member will appear on the second level, **Corporate**. This member is not the same as the member on the first level. It is a special member representing its parent's individual value, that, together with the value of other siblings, goes into the final value of their parent, the **Corporate** member on the first level. This hidden **Corporate** member, sometimes called the **datamember**, has a value of 12 million which is now clearly visible in the pivot. The exact amount, highlighted in the following image, was missing in our equation a moment ago:

Drop Filter Fields Here					Drop Column Fields
Account Level 01 ▾   Account Level 02		Account Level 03	Department Level 01 ▾   Department Level 02		Amount
<input checked="" type="checkbox"/> Balance Sheet					\$0.00
<input type="checkbox"/> Net Income	<input type="checkbox"/> Operating Profit	<input type="checkbox"/> Operating Expenses	<input type="checkbox"/> Corporate	Corporate	\$12,236,648.00
				Executive General and Administration	\$397,061.00
				Inventory Management	\$1,316,774.00
				Manufacturing	\$827,480.00
				Quality Assurance	\$575,197.00
				Research and Development	\$11,647,823.00
				Sales and Marketing	\$660,885.50
				Total	\$27,661,868.50
			<input checked="" type="checkbox"/> Gross Margin		\$44,390,103.00
					\$50,017.50
			<input checked="" type="checkbox"/> Other Income and Expense		
			<input checked="" type="checkbox"/> Taxes		\$4,168,749.00
					NA
<input checked="" type="checkbox"/> Statistical Accounts					

This new member will represent the unallocated expenses we will try to spread to other departments.

OK, let's start.

Double-click the **Adventure Works** cube and go to the **Calculations** tab. Choose **Script View**. Position the cursor at the end of the script and follow the steps in the next section.

## How to do it...

The solution consists of one calculated measure, two scopes one inside the other, and modifications of the previously defined calculated measure inside each scope:

1. Create a new calculated measure in MDX script named `Amount alloc` and set it equal to the measure `Amount`. Be sure to specify `$#,##0.00` as the format of that measure and place it in the same measure group as the original measure:

```
Create Member CurrentCube.[Measures].[Amount alloc]
As [Measures].[Amount]
, Format_String = '$#,##0.00'
, Associated_Measure_Group = 'Financial Reporting'
```

2. Create a scope statement in which you'll specify this new measure, the level 2 department members, the `Operating Expenses` account, and all of its descendants.

```
Scope( ( [Measures].[Amount alloc],
[Department].[Departments]
.[Department Level 02].MEMBERS,
Descendants( [Account].[Accounts].&[58] ) ) );
```

3. The value in this subcube should be increased by a percentage of the value of the **Corporate** datamember. The percentage of the allocation key being used here is going to be the percentage of the individual department in respect to its parents' value. Specify this using the expression below:

```
This = [Measures].[Amount] +
( [Department].[Departments].&[1].DATAMEMBER,
[Measures].[Amount] ) *
( [Department].[Departments].CurrentMember,
[Measures].[Amount] ) /
Aggregate(
Except( [Department].[Departments]
.[Department Level 02].MEMBERS,
[Department].[Departments]
.&[1].DATAMEMBER ),
[Measures].[Amount] );
```

4. Create another scope statement in which you'll specify that the value of the **Corporate** datamember should be null once all allocation is done:

```
Scope( [Department].[Departments].&[1].DATAMEMBER );
      This = null;
```

5. Provide two End Scope statements to close the scopes.

6. The complete code should look like this:

```
Create Member CurrentCube.[Measures].[Amount alloc]
As [Measures].[Amount]
, Format_String = '$#,##0.00'
, Associated_Measure_Group = 'Financial Reporting' ;

Scope( ([Measures].[Amount alloc],
        [Department].[Departments]
        .[Department Level 02].MEMBERS,
        Descendants([Account].[Accounts].&[58] ) ) );
This = [Measures].[Amount] +
      ( [Department].[Departments].&[1].DATAMEMBER,
        [Measures].[Amount] ) *
      ( [Department].[Departments].CurrentMember,
        [Measures].[Amount] ) /
      Aggregate(
        Except( [Department].[Departments]
        .[Department Level 02]
        .MEMBERS,
        [Department].[Departments]
        .&[1].DATAMEMBER ),
        [Measures].[Amount] );
Scope( [Department].[Departments].&[1].DATAMEMBER );
      This = null;
End Scope;
End Scope;
```

7. Deploy the changes using the BIDS Helper or BIDS itself.  
 8. Go to the **Cube Browser**, reconnect, and add the new measure in the pivot.

9. Verify that the result matches the following image. The highlighted cells are the cells for which the value is changed. The **Corporate** member has no value while the individual members are increased in proportion to their initial value. Below, the total remained the same:

			Drop Column Fields Here	
Account Level 03	Department Level 01 ▾	Department Level 02	Amount	Amount alloc
		Corporate	\$12,236,648.00	\$0.00
Operating Expenses	Corporate	Executive General and Administration	\$397,061.00	\$712,044.87
		Inventory Management	\$1,316,774.00	\$2,361,355.50
		Manufacturing	\$827,480.00	\$1,483,910.26
		Quality Assurance	\$575,197.00	\$1,031,494.09
		Research and Development	\$11,647,823.00	\$20,887,905.50
		Sales and Marketing	\$660,885.50	\$1,185,158.28
		Total	\$27,661,868.50	\$27,661,868.50
Gross Margin			\$44,390,103.00	\$44,390,103.00
			\$50,017.50	\$50,017.50
			\$4,168,749.00	\$4,168,749.00
			NA	NA

## How it works...

The new calculated measure is used for allocating the values; the original measure preserves its values. This is not the only way how to perform the allocation, but it's the one that suits us at the moment because it allows us to easily compare the original and the new values. If you're interested in the other approach, skip to the next section which illustrates how to allocate values directly in the original measure.

The new calculated measure is defined to be an alias for the `Amount` measure, meaning it will return the exact same values.

The scope statement is used for specifying the subcube for which we want to apply a different evaluation of the cells. This subcube is formed using the new measure, the **Operating Expenses** account and all of its descendants, and finally, all the departments in the level two of the `Department`.`Departments` hierarchy. Once we have established this scope, we can apply the new calculation.

The new calculation basically says this: take the original `Amount` measure's value for the current context of that subcube and increase it by a percentage of the `Amount` measure's value of the `Corporate` datamember. The sum of all the percentages should naturally be 1 in order for the total to remain the same.

The percentage is a ratio of the current member's value versus the aggregated value of all its siblings except the *Corporate datamember*. We're skipping that member because that's the member whose value we're dividing among its siblings. We don't want some of it to return to that member again. Actually, we're not deducing its value by this expression; we're merely evaluating a proper percentage of it which we're using later on to increase the value of each sibling. That's why there was a need for the last statement in that scope. That statement resets the value of the *Corporate datamember*.

### There's more...

The other way around is to use an existing measure while keeping the original value in the separate calculated measure. Here's how the MDX script would look like in that case:

```
Create Member CurrentCube.[Measures].[Amount preserve]
As [Measures].[Amount]
, Format_String = '$#,##0.00'
, Associated_Measure_Group = 'Financial Reporting' ;

Freeze( ( [Measures].[Amount preserve] ) );

Scope( ( [Measures].[Amount],
[Department].[Departments]
.[Department Level 02].MEMBERS,
Descendants( [Account].[Accounts].&[58] ) ) );
This = [Measures].[Amount preserve] +
( [Department].[Departments].&[1].DATAMEMBER,
[Measures].[Amount preserve] ) *
( [Department].[Departments].CurrentMember,
[Measures].[Amount preserve] ) /
Aggregate(
Except( [Department].[Departments]
.[Department Level 02].MEMBERS,
[Department].[Departments]
.&[1].DATAMEMBER ),
[Measures].[Amount preserve] );
Scope( [Department].[Departments].&[1].DATAMEMBER );
This = null;
End Scope;
End Scope;
```

At first, it looks like the regular and calculated measures have just switched their positions inside the code. However, there's more to it; it's not that simple.

A calculated measure referring to another measure is in fact a **pointer** to that measure's value, not a constant. In order to evaluate its expression, the referred measure has to be evaluated first.

The assignment inside the `Scope()` statement (the first `This` part) uses the calculated measure which requires the value of the original `Amount` measure to be evaluated. But, the `Amount` measure is used in the very same scope definition, so in order to evaluate the `Amount` measure, the engine has to enter the scope and evaluate the calculated measure specified in the assignment in that scope. Now we're in the same position we started from, which means we've run into an infinite recursion. Seen from that perspective, it becomes obvious we have to change something in the script.

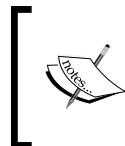
Just before the `Scope()` statement, there's a `Freeze()` statement. This statement takes a snapshot of the subcube provided as its argument and it does so at that particular position in the MDX script. Using `Freeze()`, we have prevented the reevaluation of the calculated measure in all subsequent expressions. The `Scope()` statement doesn't end up in infinite recursion this time. The assignment inside the scope takes the snapshot value of the calculated measure; it doesn't trigger its reevaluation. In other words, there's no infinite recursion this time.

Here's the screenshot of the **Cube Browser** in this case:

			Drop Column Fields Here	
Account Level 03	Department Level 01 ▾	Department Level 02	Amount	Amount preserve
Operating Expenses	Corporate	Corporate	\$0.00	\$0.00
		Executive General and Administration	\$712,044.87	\$397,061.00
		Inventory Management	\$2,361,355.50	\$1,316,774.00
		Manufacturing	\$1,483,910.26	\$827,480.00
		Quality Assurance	\$1,031,494.09	\$575,197.00
		Research and Development	\$20,887,905.50	\$11,647,823.00
		Sales and Marketing	\$1,185,158.28	\$660,885.50
		Total	\$27,661,868.50	\$27,661,868.50
+ Gross Margin			\$44,390,103.00	\$44,390,103.00
			\$50,017.50	\$50,017.50
			\$4,168,749.00	\$4,168,749.00
			NA	NA

By looking at highlighted rows, it looks like the measures have switched places, but now we know the required expressions in MDX script are not done that way. We had to use the `Freeze()` statement because of the difference between regular and calculated measures and how they get evaluated.

If you want to learn more about the `Freeze()` statement, here's a link to the MSDN site:  
<http://tinyurl.com/MDXFreeze>



Although this approach allows for drillthrough (because we're not using a calculated measure), the values returned will not match the cube's data. This is because allocations have been implemented in the MDX script, not in the original DW data.

### How to choose a proper allocation scheme?

This recipe showed how to allocate the values based on the key which is calculated as a percentage of the value of each member against the aggregate of all the other siblings to be increased. Two things are important here: we've used the same measure and we've used the same coordinate in the cube. However, this doesn't have to be so.

We can choose any measure we want, any that we find appropriate. For example, we might have allocated the values based on the **Sales Amount**, **Total Product Cost**, **Order Count**, or anything similar. We might have also taken another coordinate for the allocation. For example, there is a **Headcount** member in the **Account.Accounts** hierarchy. We could have allocated **Operating Expense** according to the number of headcounts.

To conclude, it is totally up to you to choose your allocation scheme as long as the sum of the allocation percentages (ratios) remains 1.

## Calculating the number of days from the last sales to identify the slow-moving goods

Slow-moving or non-moving goods are the inventory which had no sales in a given time frame from today. It is of crucial importance for every company to maintain its inventory levels and to try to sell those two types of inventory even if it means reducing the price because that inventory represents money locked in the system. Timely handling of non-moving or slowly-moving goods releases that money which can in turn be invested in goods with a better turnover rate.

While it's possible to store the pre-calculated number of days from the last sales (or the date of the last sales) directly in dimension table, this is not the best solution. For example, if we use a member property, or better, an attribute of that dimension the way product dimension and its attributes **Start Date**, **End Date**, and **Days to Manufacture** are created, that property or attribute is in fact a static value. It doesn't change with the context of the query. In other words, it is the same for all countries, stores, and any other dimension in the cube. It is valid only for the **All** members of other dimensions.

If we want to make that value dynamic, we have to create a new fact table in DW. This table should have the number of days from the last sales calculation for every combination of dimensions. Once we have done that, we should use the *Max* type of aggregation on that column. No other calculations would be necessary; the design of the cube would work for us.

For any subcube, that is, context of the query, the max value of that measure would be returned. Enhancements in the form of additional measures using the *Min*, *Sum*, and *Count* aggregations would offer even better statistics over that subcube.

The other option, of course, is to use MDX calculations. It may be more convenient to calculate the number of days from the last sales on the fly. If it works, we can continue to use it. If it appears to be too slow, we can begin working on the fact table approach.

In this recipe, we'll show the MDX approach.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Date** and the **Product** dimensions. Here's the query we'll start from:

```
SELECT
    { [Measures].[Order Count] } ON 0,
    NON EMPTY
    { [Product].[Product].[Product].MEMBERS } *
    { [Date].[Date].[Date].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Status].&[Current] )
```

Once executed, the query returns 20857 rows; a set of dates for each active product where the sales occurred:

		Order Count
Women's Mountain Shorts, M	July 8, 2008	1
Women's Mountain Shorts, M	July 13, 2008	1
Women's Mountain Shorts, M	July 15, 2008	1
Women's Mountain Shorts, M	July 16, 2008	4
Women's Mountain Shorts, M	July 17, 2008	1
Women's Mountain Shorts, M	July 24, 2008	2
Women's Mountain Shorts, M	July 27, 2008	1
Women's Mountain Shorts, M	July 28, 2008	1
Women's Mountain Shorts, M	July 29, 2008	2
Women's Mountain Shorts, S	July 1, 2007	15
Women's Mountain Shorts, S	July 21, 2007	1
Women's Mountain Shorts, S	July 24, 2007	1
Women's Mountain Shorts, S	August 1, 2007	40
Women's Mountain Shorts, S	August 3, 2007	1
Women's Mountain Shorts, S	August 5, 2007	1
Women's Mountain Shorts, S	August 7, 2007	1
Women's Mountain Shorts, S	August 11, 2007	1

Scroll through the result in order to see that products don't have the same last date of sale.

In this recipe we'll show how to extract the last date of sales and convert it to the number. Finally, we'll sort the products in descending order based on that number of days from the last sales.

### How to do it...

We are going to define two calculated measures and add them to the columns axis. The last of those measures will return the number of days from the last sale:

1. Add the `WITH` part of the query.
2. Create the first calculated measure, `Last Date`, by taking the `MemberValue` property of the last date with sales for a particular product.
3. Create the second calculated measure, `Number of Days`, and define it as a difference in days between **7/31/2008** and the `Last Date` measure. Here we're simulating that the end of July 2008 in Adventure Works cube represents what could otherwise be calculated as today.

4. Since the Number of Days measure will never be null, be sure to provide handling for that using the iif() function in the definition.
5. Include both measures on the columns axis of the query.
6. Arrange the products on the rows axis in descending order based on the Number of Days measures.
7. Verify that the query looks like this:

```
WITH
MEMBER [Measures].[Last Date] AS
    Max( [Date].[Date].[Date].MEMBERS,
        iif( [Measures].[Order Count] = 0,
            null,
            [Date].[Date].CurrentMember.MemberValue
        )
    )
MEMBER [Measures].[Number of Days] AS
    iif( IsEmpty( [Measures].[Order Count] ),
        null,
        DateDiff( 'd',
            [Measures].[Last Date],
            CDate('2008-07-31')
        )
    )
SELECT
    { [Measures].[Order Count],
        [Measures].[Last Date],
        [Measures].[Number of Days] } ON 0,
    NON EMPTY
    { Order( [Product].[Product].[Product].MEMBERS,
        [Measures].[Number of days],
        BDESC ) } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Status].&[Current] )
```

8. Run the query and observe the results. There are 183 rows, but only some of them show no sales in the last 90 days, a value we might consider being the threshold for the slow-moving products in our business:

The screenshot shows a results grid from SSMS with the following columns: Order Count, Last Date, and Number of Days. The data includes various bicycle components and frames, such as LL Mountain Frame - Black, HL Road Frame - Black, and ML Road Frame-W.

	Order Count	Last Date	Number of Days
LL Mountain Frame - Black...	6	8/1/2007	365
LL Mountain Frame - Black...	9	8/1/2007	365
LL Touring Frame - Blue, 58	2	8/1/2007	365
ML Mountain Frame-W - Si...	5	8/1/2007	365
HL Road Frame - Black, 48	26	9/1/2007	334
HL Road Frame - Red, 48	23	9/1/2007	334
LL Mountain Frame - Silver...	16	9/1/2007	334
LL Road Frame - Black, 60	26	9/1/2007	334
LL Road Handlebars	53	9/1/2007	334
LL Road Seat/Saddle	6	9/1/2007	334
LL Touring Frame - Blue, 44	14	9/1/2007	334
LL Touring Frame - Blue, 62	9	9/1/2007	334
LL Touring Frame - Yellow,...	20	9/1/2007	334
LL Touring Handlebars	30	9/1/2007	334
LL Touring Seat/Saddle	43	9/1/2007	334
ML Crankset	28	9/1/2007	334
ML Road Frame-W - Yello...	31	12/1/2007	243
Chain	250	6/1/2008	60
Front Brakes	266	6/1/2008	60
Front Derailleur	257	6/1/2008	60
HL Bottom Bracket	218	6/1/2008	60

## How it works...

The measure `Last Date` is calculated using the `Max()` function on all dates. For each product on rows, there will be a set of dates just like in the initial query, and this function returns the highest value of the expression provided as its second argument on that set of dates. The value which gets returned is in fact the **ValueColumn** property of the **Date.Date** attribute read using the `MemberValue` function. What's great about that property in the **Adventure Works DW 2008R2** database is that it was built using the `datetime` data type column. This allowed us to map each date with its typed property and to return the value in the form of a date. That date represents the last date with orders for each row.

In order to gain on performance, we're maintaining sparsity of the calculation by eliminating dates without orders using the `iif()` function inside the `Max()` function. This is equivalent to filtering the dates without orders, only better in terms of performance because, as mentioned throughout this book, the aggregation function over the `iif()` function works very fast when one of the branches of the `iif()` function is null.

The next calculated measure is the `Number of Days` measure. This measure takes the result of the previous calculated measure and applies the `DateDiff( )` function to calculate the difference in days (that's what the "d" stands for). In this example, we've used the fixed date because the **Adventure Works** database doesn't get new value. For a real-world situation you should use the `Date( )` or `Date( ) - 1` expressions for today or yesterday, respectively. If you need greater precision than a day, use the `Now( )` function and the appropriate period token instead.

For the rows where the `Order Count` measure is null, the `Last Date` measure will be null also. However, that cannot be said for the second calculated measure. The `Number of Days` measure will be evaluated as the difference from the first date possible instead. That's the reason why the `iif( )` function needs to be introduced. It serves the purpose of eliminating rows without orders.

Finally, our wish was to arrange the products in descending order according to the value of the `Number of Days` measure. That was achieved using the `Order( )` function applied on the set of members on rows axis.

### There's more...

In case you have a similar example but need to calculate the first date and not the last date, simply replace the `Max( )` function with the `Min( )` function.

It is also possible to modify the `iif( )` function, for example, to get only products whose inventory quantity is above a certain value.

Also, consider inserting the `EXISTING` operator in front of the set of dates if you want your calculation to be sensitive to related hierarchies (of the Date dimension in this case) on columns and/or rows.

### What's missing here?

The Adventure Works cube doesn't contain the `Inventory` measure group, that is, a measure group which would represent the current stock: how many products we have in a particular territory where the Adventure Works Cycle organization operates.

In real-world cubes, you'll have the inventory. What's important in that situation is to eliminate all the products which are not available in a particular warehouse. You only need to calculate the `Number of Days` measure for products with quantity greater than zero.

This condition can be specified in the slicer or in the subselect, the same way the condition to include only the current products was specified in our example. If there is no such attribute and there are reasons why it wouldn't be, you can use the dynamic set in the MDX script for that.

## Don't mix TopCount and Max functions

The `TopCount( )` function returns top members and the `Max( )` function returns the top value. Think carefully what exactly you need in a particular scenario, the best member in the set or the best value in the set. While it's possible to get the best value by placing the best member in a tuple with a measure, it may be better to use the `Max( )` function directly. Look for ways to bind members in a set with their typed properties when the measure depends on dimension properties rather than values in the fact table.

### See also

The recipe *Calculating the difference between two dates* covered in the second chapter uses the `DateDiff( )` function. The recipe *Finding the last date with data*, also in chapter 2, explains how to get the last date with data, which is relevant to this recipe. Chapter 2 as a whole will be helpful as it deals with time-based calculations.

## Analyzing fluctuation of customers

The analysis of the customers is very important tool. Every company takes care of its customers, or at least it should, because it is relatively easy to lose one while it's much harder to acquire a new one.

In this recipe, we're going to see how to perform the analysis of customers, to get indicators of our success with the customers. Indicators such as the number of loyal customers, the number of new customers, and the number of lost customers. Additional sections of the recipe illustrate how to identify which customers are in a particular group.

The idea is pretty simple to understand, so let's start.

### Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Customer** and **Date** dimensions and the measure **Customer Count** which can be found in the **Internet Customers** measure group. Here's the query we'll start from:

```
SELECT
    { [Measures].[Customer Count] } ON 0,
    { [Date].[Fiscal].[Fiscal Quarter].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns 14 rows with quarters, two of which are special in a way that they represent the near future for the Adventure Works Cycle Company. The reason is simple - data is entered up to this point in time only.

Our task in this recipe is to make the necessary calculation in order to have indicators of customer flow. In other words, our task is to get the count of new, loyal, and lost customers as an expression that works in any period or context in general.

## How to do it...

The solution consists of five calculated measures that we need to define in the query and then use them on columns.

1. Add the `WITH` part of the query.
2. Create the first calculated measure, name it `Agg TD` and define it as an inception-to-date calculation of the measure `Customer Count`.
3. Create the second calculated measure, name it `Agg TD prev` and define it similarly to the previous measure, only this time, limit the time range to the member that is previous to the current member.
4. Create the `Lost Customers` calculated measure and define it as the difference between the `Agg TD` measure and the `Customer Count` measure.
5. Create the `New Customers` calculated measure and define it as the difference between the `Agg TD` measure and the `Agg TD prev` measure.
6. Finally, create the `Loyal Customers` calculated measure and define it as the difference between the `Customer Count` measure and the `New Customers` measure.
7. Include all the calculated measures on the columns axis of the query.
8. Verify that the query looks like this:

```
WITH
MEMBER [Measures].[Agg TD] AS
    Aggregate( null : [Date].[Fiscal].CurrentMember,
               [Measures].[Customer Count] )
MEMBER [Measures].[Agg TD prev] AS
    Aggregate( null : [Date].[Fiscal].PrevMember,
               [Measures].[Customer Count] )
MEMBER [Measures].[Lost Customers] AS
    [Measures].[Agg TD] - [Measures].[Customer Count]
MEMBER [Measures].[New Customers] AS
    [Measures].[Agg TD] - [Measures].[Agg TD prev]
MEMBER [Measures].[Loyal Customers] AS
    [Measures].[Customer Count] - [Measures].[New Customers]
SELECT
    { [Measures].[Customer Count],
```

```
[Measures].[Agg TD],
[Measures].[Agg TD prev],
[Measures].[Lost Customers],
[Measures].[New Customers],
[Measures].[Loyal Customers] } ON 0,
{ [Date].[Fiscal].[Fiscal Quarter].MEMBERS } ON 1
FROM
[Adventure Works]
```

9. Run the query and observe the results, especially the last three calculated measures. It is obvious that in the initial phase of this company there was a problem with retaining the existing customers.

	Customer Count	Agg TD	Agg TD prev	Lost Customers	New Customers	Loyal Customers
Q1 FY 2006	448	448	(null)	0	448	0
Q2 FY 2006	565	1,013	448	448	565	0
Q3 FY 2006	558	1,571	1,013	1,013	558	0
Q4 FY 2006	635	2,206	1,571	1,571	635	0
Q1 FY 2007	732	2,938	2,206	2,206	732	0
Q2 FY 2007	752	3,690	2,938	2,938	752	0
Q3 FY 2007	788	4,478	3,690	3,690	788	0
Q4 FY 2007	950	5,428	4,478	4,478	950	0
Q1 FY 2008	3,486	7,952	5,428	4,466	2,524	962
Q2 FY 2008	5,090	11,388	7,952	6,298	3,436	1,654
Q3 FY 2008	5,237	14,496	11,388	9,259	3,108	2,129
Q4 FY 2008	6,052	17,918	14,496	11,866	3,422	2,630
Q1 FY 2009	931	18,484	17,918	17,553	566	365
Q2 FY 2011	(null)	18,484	18,484	18,484	0	0

### How it works...

The expressions used in this example are relatively simple expressions, but that doesn't necessarily mean they are easy to comprehend. The key to understand how these calculations work is in realizing that a distinct count type of measure was used. What's special about that type of aggregation is that it is not an additive aggregation.

In this recipe we've taken advantage of their non-additive behavior and managed to get really easy-to-memorize formulas.

We're starting with the two special calculated measures. The first one calculates the inception-to-date value of the Customer Count measure. For an ordinary measure with **Sum** as an aggregated function, this would mean we want to sum all the values up to the current point in time. For the extraordinary **Distinct Count** type of measures, this means we want to get the count of distinct customers up to the current point in time. One more time, this is not a sum of all customers, this is a value that is greater or equal to the count of customers in any period and less than or equal to the total number of customers.

The second calculated measure does the same thing except it stops in the previous time period.

Combining those two measures with the original Customer Count measure leads to the solution for the other three calculated measures.

The Lost Customers measure gets the value obtained by subtracting the count of customers in the current period from the count of customers so far. The New Customers measure gets the value that is equal to the difference between those first two calculated measures. Again, this evaluates to number of customers so far minus the number of customers up to the previous period. What's left are those that are acquired in the current time period.

Finally, the Loyal Customers calculated measure is defined as the count of customers in the current period minus the new customers; everything that's in the current period is either a new or a loyal customer.

The two special calculated measures are not required to be in the query or visible at all. They are used here only to show how the evaluation takes place.

### There's more...

It makes sense to put those definitions in the MDX script so that they can be used in the subsequent calculations by all users. Here's what should go in the MDX script:

```
Create Member CurrentCube.[Measures].[Agg TD] AS
    Aggregate( null : [Date].[Fiscal].CurrentMember,
               [Measures].[Customer Count] )
    , Format_String = '#,#'
    , Visible = 0
    , Associated_Measure_Group = 'Internet Customers';

Create Member CurrentCube.[Measures].[Agg TD prev] AS
    Aggregate( null : [Date].[Fiscal].PrevMember,
               [Measures].[Customer Count] )
    , Format_String = '#,#'
    , Visible = 0
    , Associated_Measure_Group = 'Internet Customers';

Create Member CurrentCube.[Measures].[Lost Customers] AS
```

```

[Measures].[Agg TD] - [Measures].[Customer Count]
, Format_String = '#,#'
, Associated_Measure_Group = 'Internet Customers';

Create Member CurrentCube.[Measures].[New Customers] AS
[Measures].[Agg TD] - [Measures].[Agg TD prev]
, Format_String = '#,#'
, Associated_Measure_Group = 'Internet Customers';

Create Member CurrentCube.[Measures].[Loyal Customers] AS
[Measures].[Customer Count] - [Measures].[New Customers]
, Format_String = '#,#'
, Associated_Measure_Group = 'Internet Customers';

```

This allows us to identify the customers in those statuses (lost, loyal, and new).

### Identifying loyal customers in a particular period

The period we'll use in this example will be **Q1 FY 2008**. We want to find out which were our first loyal customers. It is perhaps that we want to reward them or analyze their behavior further in order to determine what made them stay with us. We'll choose the latter here and prepare the query which gathers all sorts of measures about those customers:

```

WITH
SET [Loyal Customers] AS
    NonEmpty( [Customer].[Customer].[Customer].MEMBERS,
               [Measures].[Customer Count] ) -
(
    NonEmpty( [Customer].[Customer].[Customer].MEMBERS,
               [Measures].[Agg TD] ) -
    NonEmpty( [Customer].[Customer].[Customer].MEMBERS,
               [Measures].[Agg TD prev] )
)
SELECT
    { [Measures].[Internet Order Count],
      [Measures].[Internet Order Quantity],
      [Measures].[Internet Sales Amount],
      [Measures].[Internet Gross Profit],
      [Measures].[Internet Gross Profit Margin],
      [Measures].[Internet Average Sales Amount],
      [Measures].[Internet Average Unit Price] } ON 0,
    { [Loyal Customers] } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Fiscal].[Fiscal Quarter].&[2008]&[1] )

```

The query has **Q1 FY 2008** in slicer and several measures on columns. The set on rows is defined above as the difference among three sets.

You may wonder why such a complex definition of that set? Couldn't we have just a single `NonEmpty()` with the measure `Loyal Customers` as its second argument? The short answer is no, it wouldn't work. The long answer follows.

The calculated measure `Loyal Customers` is defined as a difference of two measures, one of which is also defined as a difference of the other two calculated measures. Those final calculated measures have the mention of the current member of the **Fiscal** hierarchy. At the moment when the set is being evaluated, the **Q1 FY 2008** member is taken inside those calculations and we end up with three measures, each representing a certain number of customers. In other words, those measures are not null for those customers.

When we use such a calculated measure like `Loyal Customers` (or any of those measures defined in this recipe), the `NonEmpty()` function does not evaluate them as an expression. It looks at each of them in order to determine the area that is not empty. What effectively happens is the union of those three sets of customers. Since they overlap in this case, we get the value for the largest set, which is the `Agg TD` set of customers.

We have to force the `NonEmpty()` function to take the elementary calculated members and apply set logic in form of the difference or union above, using the functions themselves. Therefore, we must be careful not to lose the order of the evaluation steps because these types of set operations are neither associative nor commutative.

If you take another look at the definition of that set, you'll notice the brackets, which serve to force the evaluation of the `Agg TD - Agg TD prev` sets. Only then are we allowed to subtract it from the first set.

The expression for the `Loyal Customers` measure was in the form of  $Y = A - (B - C)$ . All we have to do is replace the measures A, B, and C with the `NonEmpty()` function and use them as the second argument of that function. All the brackets, signs, and operations should remain the same. That shouldn't be hard to remember.

Yes, using that set we got 962 customers. Here's the screenshot with their values:

	Internet Order Count	Internet Order Quantity	Internet Sales Amount	Internet Gross Profit	Internet Gross Profit Margin
Aaron C. Diaz	1	2	\$2,451.30	\$893.38	36.45%
Aaron M. Young	1	5	\$2,445.46	\$1,132.91	46.33%
Aaron V. Wang	1	3	\$2,366.46	\$1,083.46	45.78%
Abby L. Sai	1	2	\$2,439.99	\$1,129.49	46.29%
Abby Subram	1	4	\$620.46	\$246.71	39.76%
Abigail Howard	1	4	\$2,369.97	\$1,089.95	45.99%
Abigail R. Foster	1	2	\$2,414.99	\$1,118.13	46.30%
Adam Hill	1	2	\$2,478.34	\$910.31	36.73%
Adam Lal	1	3	\$848.47	\$395.59	46.62%
Adam Ross	1	4	\$2,363.96	\$1,064.80	45.04%
Adrian C. Richardson	1	2	\$839.48	\$393.53	46.88%

It's worth mentioning that you can build several dynamic sets in the MDX script which would represent the customers in the particular status. Here's the expression for the **Loyal Customers set**:

```
Create Dynamic Set CurrentCube.[Loyal Customers] AS
    NonEmpty( [Customer].[Customer].[Customer].MEMBERS,
        [Measures].[Customer Count] ) -
    (
        NonEmpty( [Customer].[Customer].[Customer].MEMBERS,
            [Measures].[Agg TD] ) -
        NonEmpty( [Customer].[Customer].[Customer].MEMBERS,
            [Measures].[Agg TD prev] )
    );

```

The other two shouldn't be a problem.

### Did you know?

If you're wondering whether this applies only to the customer analysis, the answer is – no! It doesn't have to be the customer dimension, it can be any dimension. The principles apply in the same way.

### More complex scenario

There are other options we can do with the customer analysis. For example, we might want to combine two or more periods.

We know there were 962 loyal customers in the first fiscal quarter of the year 2008. A perfectly logical question arises – what happens to them afterwards? Are they still loyal in the Q4 of the same year? How many of them?

Let's find out:

```
SELECT
    { [Measures].[Internet Order Count],
      [Measures].[Internet Order Quantity],
      [Measures].[Internet Sales Amount],
      [Measures].[Internet Gross Profit],
      [Measures].[Internet Gross Profit Margin],
      [Measures].[Internet Average Sales Amount],
      [Measures].[Internet Average Unit Price] } ON 0,
    { Exists( [Loyal Customers],
        ( [Measures].[Customer Count],
          [Date].[Fiscal].[Fiscal Quarter].&[2008]&[4] ),
        'Internet Customers' ) } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Fiscal].[Fiscal Quarter].&[2008]&[1] )
```

The query is similar to the previous one except here we have an extra construct using the `Exists()` function, the variant with the measure group name. What this function does is it takes `Loyal Customers` and reduces it to a set of members who also have values in the Q4 of the same fiscal year.

The result shows 96 customers out of 962 of them a few quarters ago:

	Internet Order Count	Internet Order Quantity	Internet Sales Amount	Internet Gross Profit	Internet Gross Profit Margin
Abby Subram	1	4	\$620.46	\$246.71	39.76%
Adriana L. Gonzalez	2	4	\$4,891.40	\$1,830.59	37.42%
Alan Zheng	1	3	\$854.97	\$403.22	47.16%
Alejandro Beck	1	3	\$2,405.47	\$1,107.88	46.06%
Amy C. Sun	1	1	\$2,294.99	\$1,043.01	45.45%
Ariana D. Gray	1	5	\$2,501.96	\$1,168.28	46.69%
Audrey Blanco	1	1	\$539.99	\$196.34	36.36%
Billy J. Munoz	1	2	\$777.34	\$302.81	38.95%
Brad She	1	1	\$2,443.35	\$888.40	36.36%
Brandi D. Gill	3	8	\$7,219.32	\$2,890.65	40.04%
Brandy Saunders	1	3	\$554.97	\$254.79	45.91%

### The alternative approach

Chris Webb has an alternative solution for customer analysis on his blog:

<http://tinyurl.com/ChrisCountingCustomers>

## Implementing the ABC analysis

ABC analysis is a method of identifying and classifying items, based on their impact, into 3 regions: A, B, and C. It's an extension of the "80-20" rule, also known as Pareto principle, which states that for many events, roughly 80% of the effects come from 20% of the causes. In one definition of the ABC analysis, the top 20% of the causes, the important part, are further divided in two subgroups: A (the top 5%) and B (the subsequent 15%), and the 80% of effects they contribute to into segments of 30% and 50%. These are the ratios we're going to use in this recipe. There are, of course, other definitions like 10/20/70. It really depends on user needs and you're free to change it and experiment.

ABC analysis is a very valuable tool which can be found mostly in highly specialized applications, for example, in an inventory management application. This recipe will demonstrate how to perform the ABC analysis on the cube.

### Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Internet Gross Profit] } ON 0,
    NON EMPTY
    { [Product].[Product].[Product].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Status].&[Current],
    [Date].[Fiscal Year].&[2008] )
```

Once executed, the query returns all active products and their profit in Internet sales for the fiscal year 2008. The result shows 102 active products.

Now let's see how we can classify them.

### How to do it...

The solution consists of two calculated measure and four named sets. The last measure returns the group A, B, or C.

1. Add the WITH part of the query.
2. Define a new calculated measure as an alias for the measure on columns and name it **Measure for ABC**.

3. Define a named set Set for ABC which returns only products for which the Internet profit is not null using the previously defined alias measure:

```
NonEmpty( [Product].[Product].[Product].MEMBERS,  
          [Measures].[Measure for ABC] )
```

4. Using the following syntax, define three named sets, A, B, and C:

```
SET [A] AS  
    TopPercent( [Set for ABC], 30,  
                [Measures].[Measure for ABC] )  
SET [B] AS  
    TopPercent( [Set for ABC], 80,  
                [Measures].[Measure for ABC] ) - [A]  
SET [C] AS  
    [Set for ABC] - [A] - [B]
```

5. Finally, define a calculated measure ABC group which returns the letter **A**, **B**, or **C** based on the contribution of each product. Use the following expression:

```
iif( IsEmpty( [Measures].[Measure for ABC] ), null,  
     iif( Intersect( [A],  
                      [Product].[Product].CurrentMember  
                    ).Count > 0,  
         'A',  
         iif( Intersect( [B],  
                         [Product].[Product].CurrentMember  
                       ).Count > 0,  
             'B',  
             'C'  
           )  
     )  
   )
```

6. Add that last calculated measure on columns and replace the existing measure with the Measure for ABC calculated measure.

7. Run the query which should now look like this:

```
WITH  
MEMBER [Measures].[Measure for ABC] AS  
    [Measures].[Internet Gross Profit]  
SET [Set for ABC] AS  
    NonEmpty( [Product].[Product].[Product].MEMBERS,  
              [Measures].[Measure for ABC] )  
SET [A] AS  
    TopPercent( [Set for ABC], 30, [Measures].[Measure for ABC] )  
SET [B] AS  
    TopPercent( [Set for ABC], 80, [Measures].[Measure for ABC] ) -  
    [A]  
SET [C] AS
```

```

[Set for ABC] - [A] - [B]
MEMBER [Measures].[ABC Group] AS
    iif( IsEmpty( [Measures].[Measure for ABC] ), null,
        iif( Intersect( [A],
            [Product].[Product].CurrentMember
        ).Count > 0,
            'A',
            iif( Intersect( [B],
                [Product].[Product].CurrentMember
            ).Count > 0,
                'B',
                'C' ) ) )
SELECT
    { [Measures].[Measure for ABC],
        [Measures].[ABC Group] } ON 0,
    NON EMPTY
    { [Product].[Product].[Product].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Product].[Status].&[Current],
        [Date].[Fiscal Year].&[2008] )

```

8. Verify that the result matches the following image:

The screenshot shows the SSMS Results tab with a table of data. The table has three columns: Measure for ABC, ABC Group, and Product Name. The data is as follows:

Measure for ABC	ABC Group	Product
\$4,495.60	C	Long-Sleeve Logo Jersey, XL
\$20,331.92	C	ML Mountain Tire
\$13,922.89	C	ML Road Tire
\$12,138.43	C	Mountain Bottle Cage
\$9,083.72	C	Mountain Tire Tube
\$433,891.62	A	Mountain-200 Black, 38
\$445,364.71	A	Mountain-200 Black, 42
\$437,020.65	A	Mountain-200 Black, 46
\$444,944.35	A	Mountain-200 Silver, 38
\$413,313.24	B	Mountain-200 Silver, 42
\$422,802.57	A	Mountain-200 Silver, 46
\$51,757.32	C	Mountain-400-W Silver, 38
\$44,763.08	C	Mountain-400-W Silver, 40
\$45,112.80	C	Mountain-400-W Silver, 42
\$48,260.20	C	Mountain-400-W Silver, 46
\$11,779.69	C	Mountain-500 Black, 40

## How it works...

The alias for the measure `Internet Gross Profit` in form of the calculated measure `Measure for ABC`, together with the alias for the set on rows `Set for ABC` enables us to have a flexible and readable query. Flexible in a way that we can change the measure in a single spot and have the query running another ABC analysis, for example, the analysis of the revenue or the number of orders. Readable because we're using short but informative names in it instead of the long MDX-specific names for members and levels. Even more than that, we're keeping the syntax short by not having to repeat some expressions. As for the set on rows, we're not making the query absolutely flexible; there are still some parts of the query where the mention of a specific hierarchy was not replaced by something more general.

Anyway, let's analyze the main part of the query: sets A, B, and C and the calculated measure `ABC Group`.

The `TopPercent( )` function, in combination with a set and a measure, returns top members from that set. The threshold, required as the second argument of that function, determines which members are returned.

That's a pretty vague description of what this function does because it is not clear which members get returned or how many top members will be returned. Let's see the more detailed explanation.

The behavior of that function can be explained using the list of members sorted in descending order. We don't know in advance how many of them the function will return; the only thing we know is that it will stop at some point.

The members are included up to the point where the ratio (in form of the percentage) of the cumulative sum versus the total becomes equal to the value provided as the second argument. In our example we had two such values, 30 and 80. 80 because the second segment is 50 and the sum of 30 and 50 is 80. The function would use all top members up to the point where their sum reaches 30% or 80%, respectively, of the total value of the set.

So, set A will contain the top products that form 30% of the total.

Set B gets calculated as even more, 80% of the total. However, that would include also members of the set A. That's why we had to exclude them.

Set C can eventually be calculated as all the rest, meaning, the complete set of members excluding members in both set A and set B.

The calculated measure `ABC Group` basically takes the current member on rows and checks in which set of three named sets it is. This is done using the combination of the `Intersect( )` function and the `.Count` function. If there's an intersection, the count will show 1.

When the iteration on cells starts, each member gets classified as either A, B, or C, based on its score and in which of the three predefined groups it can be found.

## There's more...

There are many ways to calculate A, B, or C. I'd like to believe I've shown you the fastest one.

Here's another example to explain what I meant by that.

Use the same query as before, but replace the measure ABC group with these two calculations:

```
MEMBER [Measures].[Rank in set] AS
    iif( IsEmpty( [Measures].[Measure for ABC] ), null,
        Rank( [Product].[Product].CurrentMember,
              [Set for ABC],
              [Measures].[Measure for ABC] ) )
    , FORMAT_STRING = '#,#'

MEMBER [Measures].[ABC Group] AS
    iif( IsEmpty( [Measures].[Measure for ABC] ), null,
        iif( [Measures].[Rank in set] <= [A].Count,
            'A',
            iif( [Measures].[Rank in set] <= { [A] + [B] }.Count,
                'B', 'C' ) ) )
```

Run the query and observe the results. They should match the results from the previous query. What changes is the execution time.

In Adventure Works that change in time is not so obvious, especially in the context of this query. If you want to experience it, use the Reseller dimensions. There you'll see a small difference.

Why is the second approach slower?

The initial example illustrates a classic case of set-based thinking. First we've defined the set A, then we've used it in the definition of set B and finally, we've defined the third set using the previous two. Therefore, we've used a simple and effective set operation - the difference between the sets.

That's not all there is to it. Something else is important. Sets evaluate before the iteration on cells starts. This means by the time the engine starts to evaluate the expression in the measure ABC Group, sets are already evaluated and therefore sort of a constant. When the iteration starts, the engine compares each member with maximum two of those sets. That comparison is again performed using a fast set operation – intersection of two sets. In short, we completely avoided any iteration in this approach.

Now, consider the second example, the one which uses the Rank function.

The idea is to have a rank which tells us how good each member was — which position it took. We can use this rank and compare it with the count of members in set A or count of members in both sets A and B. If the rank is a smaller number, we get a match and the member gets the corresponding class as a result of the calculation.

True, sets A, B, and C are static again, pre-evaluated, but the rank operation, together with the process of counting the number of items in a particular set, takes time. Here we're not applying the set-based thinking; we're iterating, although internally, on a set to get the rank while we don't really need that rank at all.

Remember to always look for a set-based alternative if you catch yourself using iteration. Sometimes it will be possible, sometimes not. If you have a large cube, it's certainly worth a try.

### Tips and tricks

Always look if you can move everything that was on axis to a named set. Your calculations will be easier.

Consider defining everything in the MDX script because of the advantages of centrally-based calculations (speed, cache, availability).

### See also

The recipe *Identifying the best N members in a set* in chapter 3 covers the TopPercent function in more detail.

# 6

## When MDX is Not Enough

In this chapter, we will cover:

- ▶ Using a new attribute to separate members on a level
- ▶ Using a distinct count measure to implement histograms over existing hierarchies
- ▶ Using a dummy dimension to implement histograms over non-existing hierarchies
- ▶ Creating a physical measure as a placeholder for MDX assignments
- ▶ Using a new dimension to calculate the most frequent price
- ▶ Using a utility dimension to implement flexible display units
- ▶ Using a utility dimension to implement time-based calculations

### Introduction

So far we've been through the basics of MDX calculations. We learned few tricks regarding time calculations, practiced making concise reports, navigated hierarchies, and analyzed data by applying typical business calculations. As we're approaching the end of the book, more and more of the special topics arrive.

This is a book that follows the cookbook approach, the main topic being the MDX. In this chapter, however, we're going to discover that MDX calculations are not always the place to look for a solution to a problem. It is only one of the possible layers we can start from. The other two layers are the cube design and the underlying data warehouse (DW) model. One thing depends on the other. A good data model will enable a good cube design which in turn will enable simple MDX calculations.

## *When MDX is Not Enough* —————

Whenever we are given a request to get something from the cube, it is not only a request to make an adequate MDX query or calculation and then to return the data, it is something much deeper - a challenge to our cube design and the underlying dimensional model. If the MDX becomes too complex, that's a potential sign there's something wrong with the layers below and that they, perhaps, can be improved.

This chapter illustrates several techniques to optimize the query response times with a relatively simple change in cube structure, to simplify the cube maintenance and further MDX calculations, to enable new analyses or even to make the so-far impossible ones possible. The types of changes we're talking about here are adding new measure, new attribute, or even a complete dimension.

Calculations are performed by the formula engine at query time. Using regular measures we can avoid a query-time performance decrease. Regular measures have other advantages over the calculated ones which are discussed in this chapter.

Attributes not only enhance the dimension design allowing users to avoid multi-select and problems related to it, they are a potential place for aggregations and therefore a candidate for optimization of reports.

Utility dimensions play a vital role in providing powerful calculations that are easy to maintain in contrast to potential chaos with calculated measures. A variant of the utility dimensions, the dummy dimension, is a convenient structure which enables iteration, allocation, and other useful activities.

Finally, if you find yourself running a lot over leaves in your calculations, you should know that cubes are not designed to do that. Maybe a different granularity for the fact table should be applied or a new dimension should be considered.

These are the things we'll encounter in this chapter. If you can create the calculations in the DSV/DW layer or prepare data there without compromising the cube's flexibility, do it. That way whatever you implement will be resolved only once, during processing, instead of during every query execution. The idea is to push things down to the lower layers as much as possible so that queries run faster.

By the way, don't worry about MDX, there's still plenty of it here. The only difference, in contrast to the previous chapters, is that here the focus is on other aspects, not just MDX.

Let's start slowly.

## Using a new attribute to separate members on a level

In reporting and analysis, there are situations when a certain member (often named NA, Unknown, or similar) corrupts the analysis. It makes it hard for end users to focus on the rest of the data. It distracts them by making them think what this member represents, why it is here, and why it has data associated to it. Other times, the reason may be that the end users need a total without that member. In both of these situations, they remove that member from the result which generates a new MDX query.

True, a combination of a named set without that member and a calculated member as an aggregate of that set can be created in MDX script to simplify the rest of the calculations and the usage of that hierarchy in general. However, this is not the optimal solution.

Is there a better way? Yes, but it requires a dimension redesign. If that's applicable in your case, read on because this recipe shows how to keep your cube design simple and effective, all at the price of a bit of your time invested in the preparation of data.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Product** dimension, the dimension where the problematic member will be in this example.

We're going to use the **Color** attribute of the **Product** dimension. If you browse the dimension in the **Browser** tab, you'll see there are 10 colors, one of which is NA. The idea is to somehow exclude this color from the list of colors and keep it separate. To do this, we'll need another attribute to separate colors in two groups: colors with the exact name in one group and the NA color in another.

Attributes are built from one or more columns in the underlying dimension table or view. The preferred place for introducing this change is the data warehouse (DW), a view that represents the dimension table from which this dimension is built. Having all logic in one place increases the maintainability of the overall solution. However, in order to keep things simple and focus on what's important, we're going to disregard the best practice here and use the data source view (DSV) instead.

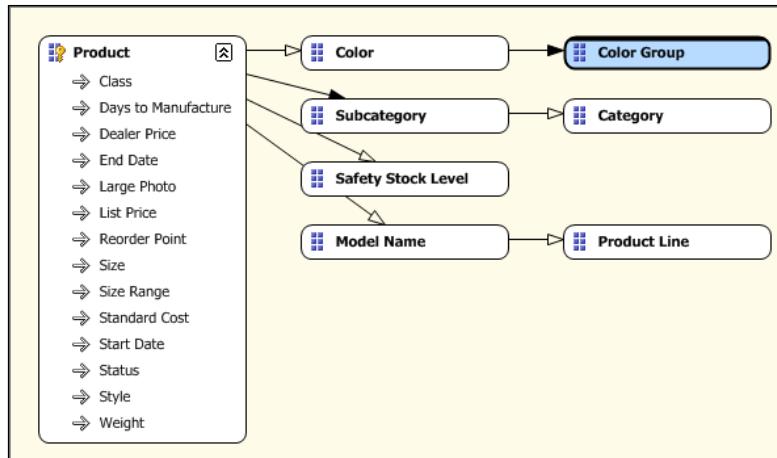
## How to do it...

Follow these steps to add a new attribute that will be used to separate members on another attribute.

1. Double-click the **Adventure Works** data source view.
2. Locate the **Product** dimension in the **Tables** pane on the left, then right-click and select **New Named Calculation**.
3. Enter **ColorGroupKey** for the **Column name** and this code for the expression:  

```
CASE Color WHEN 'NA' THEN 1 ELSE 0 END
```
4. Close the dialog and repeat the process for another column. This time name it **ColorGroupName** and use the following definition:  

```
CASE Color WHEN 'NA' THEN 'Unknown colors' ELSE 'Known colors'  
END
```
5. Close the dialog again and explore the **Product** table to see the result of those two new columns. They should be visible at the end of the table. If everything's OK, close the **Explore Product Table** tab.
6. Now return to the **Product** dimension again where you should see two new calculated columns in the end of that table in the **Data Source View** pane on the right.
7. Drag the **ColorGroupKey** column to the left and drop it in the **Attributes** pane.
8. Rename it to **Color Group**, then navigate to the **NameColumn** property and select the **ColorGroupName** column for that.
9. Set the **OrderBy** property to **Key** in order to preserve the order by key.
10. Drag the newly created **Color Group** attribute to the **Hierarchies** pane following by the **Color** attribute underneath it. The idea is to create a new user hierarchy.
11. Name the hierarchy **Product Colors** and set the **AllMemberName** in the **Properties** pane to **All Products**.
12. Notice the yellow warning sign. Go to the **Attributes Relationships** tab and set the correct relation between the **Color** and **Color Group** attributes by dragging the **Color** attribute over the **Color Group** attribute and releasing it.
13. Select the arrow that points to the **Color Group** attribute and change the **RelationshipType** property from **Flexible** to **Rigid** in the **Properties** pane. The tip of the arrow should turn black, just like in the image below:



14. Return to the **Dimension Structure** tab and check that the warning sign is gone.
15. Select the **Product Colors** user hierarchy and set the **DisplayFolder** property to **Stocking**.
16. Select the **Color Group** attribute and set its visibility to **False** using the **AttributeHierarchyVisible** property.
17. Process the dimension. When it's done, go to the **Browser** tab, reconnect, and verify that the new **Product Colors** user hierarchy works as expected, that colors are separated in two new nodes, **Known colors** and **Unknown colors**.
18. Now process the **Adventure Works** cube, reconnect and make a report using the new **Product Colors** user hierarchy which you'll find under **Stocking** folder of the **Product** dimension.

Color Group	Color	Order Quantity	Average Unit Price	Sales Amount	Gross Profit
Known colors	Black	81,856	\$610.08	\$38,236,124.06	\$4,971,374.95
	Blue	23,659	\$534.43	\$9,602,850.97	\$605,893.01
	Multi	25,059	\$27.07	\$649,030.25	\$13,160.89
	Red	29,187	\$990.68	\$21,597,890.81	\$2,720,764.26
	Silver	25,023	\$936.10	\$19,777,339.95	\$3,239,562.63
	Silver/Black	3,931	\$37.35	\$147,483.91	\$38,250.97
	White	5,217	\$6.85	\$29,745.13	\$12,163.31
	Yellow	32,556	\$731.52	\$18,669,505.22	\$83,372.60
Total	226,488				
Unknown colors	NA	48,288	\$18.44	\$1,099,303.91	\$466,823.63
	Total	48,288	\$18.44	\$1,099,303.91	\$466,823.63
Grand Total		274,776	\$465.18	\$109,809,274.20	\$12,551,366.25

## How it works...

The new attribute plays several roles. It enables natural, by-design subtotals, easy navigation, easy filtering, and keeps further MDX calculations simple and effective.

In order to create it, we've extended the **Product** table with two new columns. We did that in the DSV, although we could have done the same in the DW. In fact, it might have been a better decision to go to the DW, but to keep things the way that Adventure Works database is done, with calculated columns, we've used the DSV. In your real project, go to the DW and add those calculated columns there.

While we're there, it's good to notice one thing. By carefully planning the key column value we achieved the order we needed. In this example, we've deliberately placed the **Known colors** value first using the lower key value and ordering the attribute by key instead of by name later.

In the process of dimension redesign we did one extra step. We created a natural user hierarchy. This is good from the perspective of performance and easy navigation.

We also did something else which is important and should not be forgotten – defining relations between the new attribute and the old ones. This example showed how.

Other things we did were more or less cosmetic.

Dimensional processing was required because the structure has changed. The same goes for the cube.

You should also consider redesigning your aggregations. The new attribute has only two members, therefore the aggregation wizard may look for including it in many combinations with other hierarchies. In other words for the same amount of space dedicated for aggregations, a low-cardinality attribute can generate more aggregations than the one with more members in it. The more aggregations, the greater the chance of hitting one of them in queries that use this new attribute.

## There's more...

Notice that once you redesign your dimension like that, you can do many things relatively easily. One of those things we've already showed are natural subtotals and ordering of the members in user hierarchy. The next thing is filtering of members.

If you need to show only the known colors, you can put that single member in slicer. Having a single member is a much better solution than using multi-select. Not only the performance could be better, but you also avoid problems with tools that have problems with multi-select. Therefore, think about this solution in a much broader sense, not only as a way of isolating unwanted members, but also as a way of avoiding problems with multi-select, and to have simple MDX calculations.

So why not do it every time?

Well, you know that sometimes a modification like this is just not possible. You can't anticipate all the possible multi-selects that users may want, or you are not allowed to change the dimension structure, or to have the cube down for a certain amount of time. Yes, this can be handled, but your current configuration may be preventing you from doing that smoothly.

Redesigning the aggregations could be beneficial, but it's totally optional. It doesn't have to pull you off from implementing a new attribute. The existing aggregations on the `Color` attribute (if any) should be leveraged because of the established attribute relationships.

Nevertheless, the idea of this recipe was to show you how to do it. The final decision of whether you'll do it or not in the end is entirely yours. Weigh the pros and cons, especially if you have a development environment where you can play and make a decision based on your findings.

### **So, where's the MDX?**

There isn't any! At least not in this recipe. Remember, the best thing is to keep things simple. This recipe presented a way how making a small investment in the redesign of your dimension pays off by not having complex MDX calculations. Even better, by not having MDX at all sometimes!

It follows the main idea of the chapter that the solution isn't to make the best possible MDX calculation or query. Oftentimes it is better to look for an alternative in either a cube/dimension design or even further, in the dimensional model. The general rule of thumb is to always prefer a built-in Analysis Services feature over writing MDX. This was the first recipe in a series of recipes that showed how. The others follow, so read on.

### Typical scenarios

Every time you catch yourself using functions like `Except()`, `Filter()`, or `similar()` too often in your calculations, you should step back and consider whether that's a repeating behavior and whether it would pay off to have an attribute to separate the data you've been separating using MDX calculations. If it has a repeating pattern and it's not something unpredictable, there's your candidate.

Once you have a member (or more) separating two or more parts of the set, you can simply use them in the calculations. You can even call upon their children or other descendants. You could make ratios the easier way. Everything becomes simplified.

## Using a distinct count measure to implement histograms over existing hierarchies

Histograms are an important tool in data analysis. They represent the distribution of frequencies of an entity or event. In OLAP, those terms translate to dimensions and their attributes.

This recipe illustrates how to implement histograms over existing attributes.

In order to create them, we need a measure which counts distinct members of an attribute for any given context.

There are two solutions to this problem. One is to use the calculated measure; the other is to use a regular measure with the distinct count type of the aggregation.

Naturally, many BI developers lean towards the first option. Undoubtedly, it is a much faster-to-implement solution because deploying the MDX script doesn't require reprocessing the cube. All it takes is to open the cube, add a calculated measure, deploy, and that's it. In the case of a query, it's even simpler - define a calculated measure as a part of that query.

Like most things in life, a shortcut is not an optimal solution. Depending on various factors like the cube and dimension sizes, the calculation can turn out to be slow in certain scenarios or contexts.

When that's not fast enough, another approach is needed, and there is one in this recipe.

### Getting ready

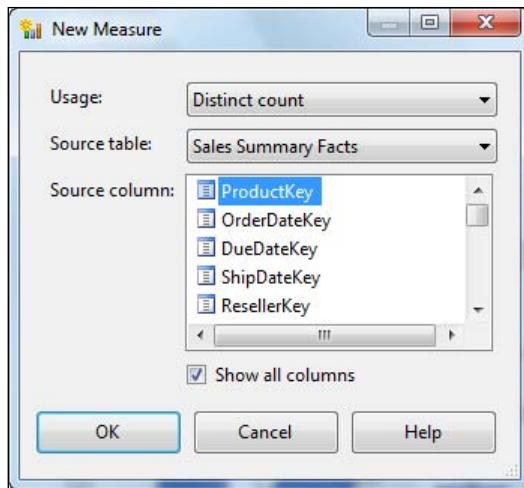
Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube.

In this example we're going to analyze products based on their characteristic. We're going to show how many red, green and blue products are there in a particular subcube.

### How to do it...

Follow these steps to create a distinct count measure:

1. Create a new **Distinct count** type of measure using the column **ProductKey** in the **Sales Summary Facts** table:



2. Name it **Product Count** and set the format string as # , # (without quotes) in the **Properties** pane.
3. The best practice for distinct count measures is to have them in a separate measure group. Notice how BIDS ensures this is done. Name the new measure group **Sales Products**.
4. Process the cube. Once it's done, go to the **Cube Browser** tab and reconnect.
5. Test this new measure using any attribute of the **Product** dimension or any other dimension, for example, the **Date** dimension as seen in the screenshot below. The measure aggregation function adjusts itself returning the distinct number of products in each cell:

Drop Filter Fields Here

	Fiscal Year ▾				
	[+ FY 2006]	[+ FY 2007]	[+ FY 2008]	[+ FY 2009]	Grand Total
Color ▾	Product Count				
Black	19	53	39	7	111
Blue	1	1	26	4	28
Multi	4	7	5	5	16
NA		12	35	18	47
Red	26	19	5	1	50
Silver	8	6	30	1	44
Silver/Black			7		7
White	2		2	2	4
Yellow		9	34	4	43
Grand Total	60	107	183	42	350

## How it works...

Attribute-based histograms are relatively easy to implement. All we need is a **Distinct count** type of measure over a dimension key column in the fact table. After that, everything is pretty straightforward; the new measure reacts to every attribute, directly or indirectly related to that fact table.

## There's more...

Here's the equivalent calculation, or should we say, the initial calculation that got replaced with the distinct count measure:

```
Create MEMBER CurrentCube.[Measures].[Product Count calc orig] AS
    Count( EXISTING
        Exists( [Product].[Product].[Product].MEMBERS, ,
            'Sales Summary' ) )
    , Visible = 0
    , Format_String = '#,#'
    , Associated_Measure_Group = 'Sales Summary'
    , Display_Folder = 'Histograms'
    ;

Create MEMBER CurrentCube.[Measures].[Product Count calc] AS
    iif( [Measures].[Product Count calc orig] = 0, null,
        [Measures].[Product Count calc orig] )
    , Format_String = '#,#'
    , Associated_Measure_Group = 'Sales Summary'
    , Display_Folder = 'Histograms'
    ;
```

What we're doing here is creating two calculated measures. One, hidden, returns the count of products relevant to the existing context which has data in the Sales Summary measure group. The other, visible, converts zeros to null because the result of the first calculated measure is never null and we don't want zeros in our result. We'd like to keep the data sparse.

Here's the image which shows that the distinct count measure works just as the calculation. In other words, they are equal in result, but different in performance:

Drop Filter Fields Here

	Fiscal Year ▾					
	[+] FY 2006	[+] FY 2007	[+] FY 2008			
Color ▾	Product Count calc	Product Count	Product Count calc	Product Count	Product Count calc	Product Count
Black	19	19	53	53	39	39
Blue	1	1	1	1	26	26
Multi	4	4	7	7	5	5
NA			12	12	35	35
Red	26	26	19	19	5	5
Silver	8	8	6	6	30	30
Silver/Black					7	7
White	2	2			2	2
Yellow			9	9	34	34
Grand Total	60	60	107	107	183	183

An additional measure group comes at the cost of having to process and store it, but it offers better performance, especially on very large dimensions. In scenarios where the distinct count column (i.e. **ProductKey**) has millions of distinct values, a distinct count measure may be a much, much faster solution than an MDX calculation. In scenarios where **ProductKey** has few distinct values (i.e. 100,000 members or less), another good approach might be to build a **ProductKey** grain measure group and add a many-to-many relationship on all the other dimensions in the **Sales Summary** measure group. Essentially, any distinct count can be reformulated as a many-to-many relationship which is elaborated in details in Marco Russo's *Many-to-Many dimensional modeling* paper:

<http://tinyurl.com/M2Mpaper>

## See also

The next recipe, *Using a dummy dimension to implementing histograms over non-existing hierarchies* covers a similar topic.

## Using a dummy dimension to implement histograms over non-existing hierarchies

As seen in the previous recipe, Analysis Services supports attribute-based histograms by design. All it takes is a distinct count measure and we're set to go.

This recipe illustrates how to implement more complex type of histograms – histograms over non-existing hierarchies.

The complexity comes from the fact that the hierarchy on which we'd like to base the calculation on does not exist. That's a very big issue when a multidimensional cube is concerned.

## *When MDX is Not Enough*

---

OLAP cubes operate on predetermined structures. It is not possible to build items on-the-fly. In other words, it is not possible to create a new hierarchy based on a calculation and use it the way we would use any other hierarchy. In OLAP, every hierarchy must be prepared in advance and must already be a part of the cube, otherwise it can't exist.

In this recipe, we're interested in the fact table. The Fact table represents a series of events that are taking place and are being recorded in a very consistent manner. Every row tells a story of an event. Various columns in that row represent dimensions related to those events.

In the previous recipe, where we calculated histograms over attributes, we created a measure based on one of the dimension columns in the fact table. That measure counted distinct dimension member keys that occur in a part of the fact table, the part determined by other dimensions in context. Those other dimensions limit the size of the fact table acting like a filter to it.

This time we're interested in something else. We're interested in combining several dimensions, for example counting the number of the distinct members of one of those dimensions inside the other.

Let's illustrate this with an example. Take resellers and their orders. We might be interested in orders only, as in the number of orders a particular reseller made in a certain period. The possible values are 0, 1, 2, 3, and so on, up to the maximum number of orders a single reseller ever made. If we want to analyze that, then that's a measure, the `Reseller Order Count` measure in the Adventure Works cube, to be precise. Resellers are simply a dimension we want to use on an axis in this case.

But what if we want to analyze both the resellers and the orders so that neither of them is on rows or columns? That's a different story. For example, we want to know how many resellers made 0 orders in a given time frame, how many of them made 1, 2, or more. That sequence of numbers (0, 1, 2, and so on) is what should go on rows (or columns), something we will iterate on. The measure in this case would be the `Reseller Count` measure, a measure showing the distinct count of resellers in a particular context. The only problem is – neither this special dimension exists, nor the distinct count measure does.

While it is relatively easy to make a distinct count type of measure, either using a calculated measure or a regular measure with the distinct count aggregation, creating a dimension takes preparation and cannot be done on-the-fly.

Now that we've explained the problem, let's see how it can be solved.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** data source view.

In this example we're going to analyze resellers based on the frequency of their orders. In other words, we're going to show how many resellers made zero, one, two, three, and so on orders in a given time frame and for given conditions.

## How to do it...

Follow these steps to implement the histogram over the hierarchy that doesn't exist in the cube.

1. Create a new named query **Reseller Order Frequency** with the following definition:

```

SELECT 0 AS Interval
UNION ALL
SELECT
    TOP(
        SELECT
            MAX(Frequency) AS Interval
        FROM
            ( SELECT
                ResellerKey,
                COUNT(SalesOrderNumber) AS Frequency
            FROM
                ( SELECT
                    DISTINCT
                    ResellerKey,
                    SalesOrderNumber
                FROM
                    dbo.FactResellerSales
                ) AS t1
            GROUP BY
                ResellerKey
            ) AS t2
            ) ROW_NUMBER()
        OVER (ORDER BY ResellerKey) AS Interval
    FROM
        dbo.FactResellerSales

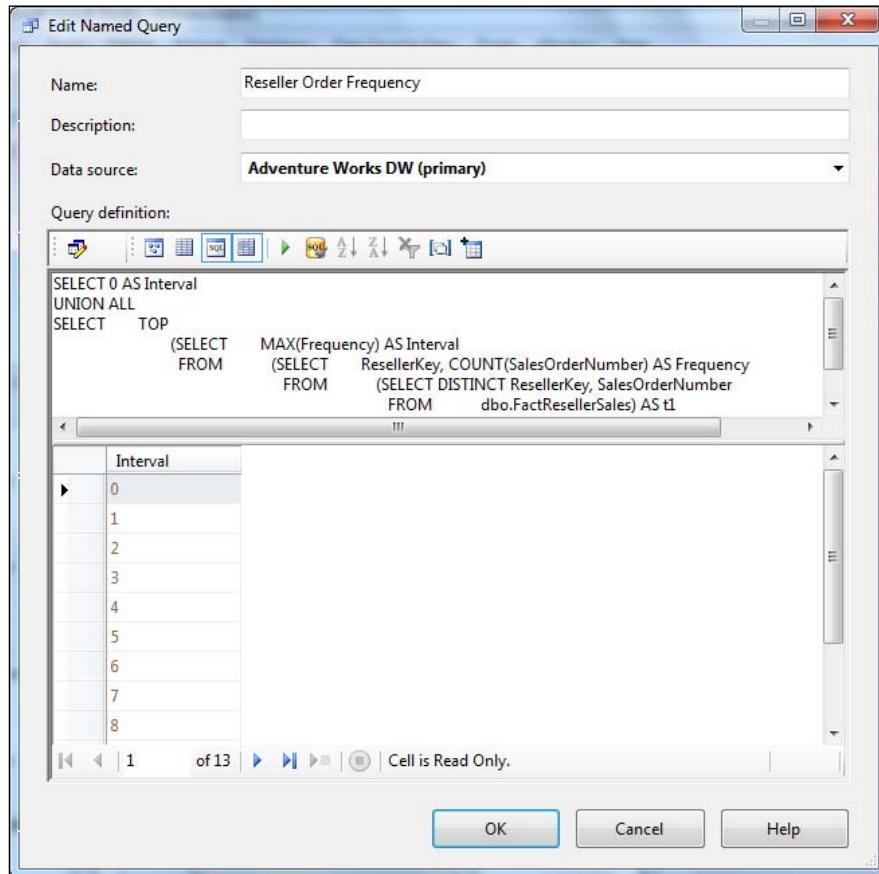
```

2. Turn off the **Show/Hide Diagram Pane** and **Show/Hide Grid Pane** and execute the previous query in order to test it.

An error might pop up when you run this type of query or when you open the named query to change it. The error basically says that this type of query cannot be represented visually using the diagram pane because it contains the OVER SQL clause. That's the reason why we've turned that pane off. If for whatever reason you see that error anyway, just acknowledge it and continue with the recipe. It is merely an information, not a problem.

*When MDX is Not Enough* —————

3. Once executed, the query returns 13 rows with numbers, the first one being zero and the last one being the maximum number of orders for a customer (here 12).



4. Close the named query editor, locate the **Reseller Order Frequency** named query, and mark the **Interval** column as a **Logical Primary Key**.
5. Create a new dimension using the previously defined named query and name it **Reseller Order Frequency**.

6. Use the **Interval** column for both the **KeyColumn** and the **ValueColumn** properties:

Source	
CustomRollupColumn	(none)
CustomRollupPropertiesColumn	(none)
KeyColumns	Reseller Order Frequency.Interval (BigInt)
NameColumn	(none)
ValueColumn	Reseller Order Frequency.Interval (BigInt)

7. Name the All member **All Intervals**.  
 8. Process-full that dimension.  
 9. Open the **Adventure Works** cube and add that dimension without linking it to any measure group.  
 10. Deploy the changes by right-clicking on the project name (in the **Solution Explorer** pane) and by choosing the **Deploy** action.  
 11. Go to the **Calculations tab** and add the following code to the end of the MDX script:

```

Create MEMBER CurrentCube.[Measures].[RO Frequency] AS
    Sum( EXISTING [Reseller].[Reseller].[Reseller].MEMBERS,
        iif( [Reseller Order Frequency].[Interval]
            .CurrentMember.MemberValue =
            [Measures].[Reseller Order Count],
            1,
            null
        )
    )
    , Format_String = '#,#'
    , Associated_Measure_Group = 'Reseller Orders'
    , Display_Folder = 'Histograms'
;

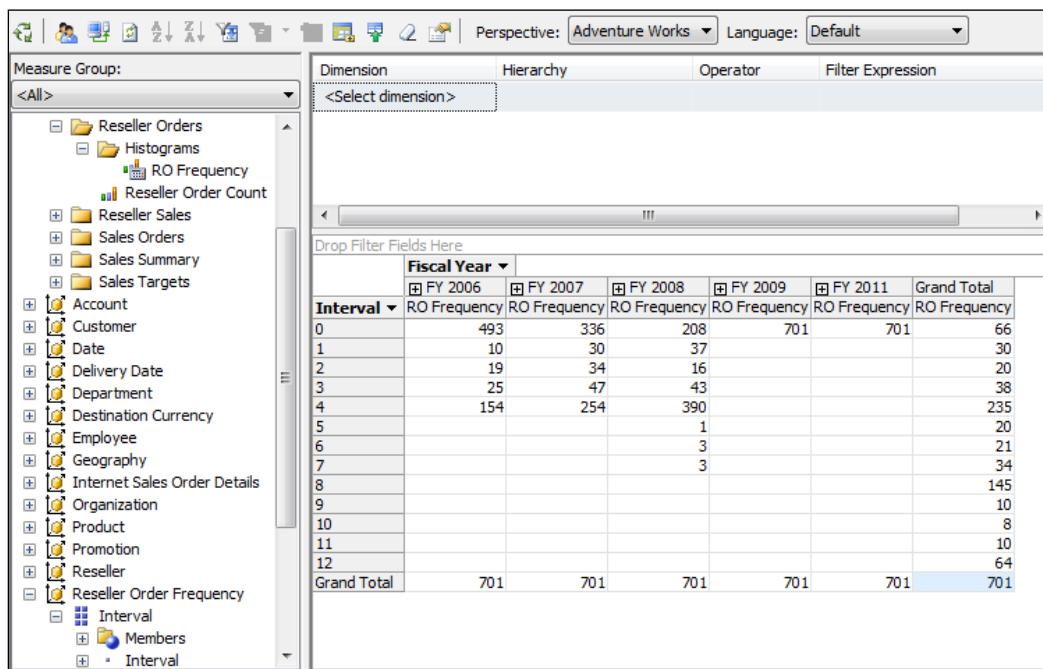
Scope( ( [Measures].[RO Frequency],
    [Reseller Order Frequency].[Interval].[All Intervals] ) );
    This = Sum( [Reseller Order Frequency].[Interval]
        .[Interval].MEMBERS,
        [Measures].[RO Frequency]
    );
End Scope;
  
```

12. Deploy the solution.

## When MDX is Not Enough

---

13. Go to the **Cube Browser** tab and build a pivot using the new **RO Frequency** measure. Then place the **Reseller Order Frequency** dimension on the opposite side from measures, on rows. Finally, add another dimension on columns, the **Date** dimension, to test if the calculation works for hierarchies on both axes. As seen in the following image, values don't repeat which is a sign that our calculation works. They represent the number of resellers who in a particular time period (here years) made as many orders as shown in the hierarchy on rows. The distribution of customers and their orders is in fact the histogram we've been looking for:



## How it works...

There are several important things in this recipe. We'll explain them one by one.

The solution starts with a particular T-SQL query that was used as a named query in the data source view. The purpose of that query is to return the maximum number of orders for any customer and all the integers up to that number, starting from zero.

The maximum frequency number is obtained by combining the distinct occurrence of the `ResellerKey` and `SalesOrderNumber` columns in the `ResellerSales` fact table and by taking the count of the orders for each reseller. That's the essence of that query.

As we saw, the query returns 13 rows, numbers starting from zero and ending with 12. Those values in turn become members of a new dimension which we'll build afterwards. The dimension is in no way associated with any fact table or dimension in that cube. What's the purpose then?

We should remember that our initial goal is to show how many distinct customers fall into each segment: zero orders, one order, two orders, and so on. That requirement is contradictory in OLAP terms because it would require two measures: one with the distinct number of customers and the other with the distinct number of orders, both somehow used one against the other in a pivot. The problem is OLAP supports measures on one axis only. Actually, that's true for any hierarchy.

So, what's the solution?

The idea is to build a special dimension and use it on one axis while we're keeping the other measure on another axis. The only thing this dimension has to have is a sequence of numbers so that we can assign values to them, somehow.

Now we've come to the calculations; the MDX script.

The first part of the calculation is the main part. That's where the assignment takes place. The idea is to use the `MemberValue` property value of the current member of the `Reseller Order Frequency` dimension and increment the measure **RO Frequency** only when `MemberValue` equals the number of orders a particular reseller made in the current slice. In other words, we're descending to the granularity of the resellers where we're counting how many of them are equal to the condition. The condition says that the number of orders must match the utility dimension member's value we're currently on. That way each reseller is thrown in one of the buckets, one of the members of that new dimension. Once we're done with the process, each bucket tells us how many resellers made that many orders. In other words, we have a complex histogram, a histogram based on an up-to-now non-existent dimension.

One more thing. Each member of the utility dimension got its value using the technique described above except for the `A11` member. No value was assigned to it because the calculation operated only on individual members. That's why there was a requirement to use another statement where we're basically saying, "collect all the individual values obtained during the allocation process and aggregate them."

### There's more...

The example presented here was using a dedicated utility dimension named **Reseller Order Frequency**. It had exactly 13 members. However, they were not fixed; they were the consequence of the data in the fact table. Meaning the number of members can grow with time. It will always be a sequence of natural numbers up to the maximum number of occurrence for the entire fact table. In this example, for this particular case of resellers and their orders.

## *When MDX is Not Enough* —————

It is quite reasonable to expect multiple event-based histograms in the same cube. One way to handle this would be to build a separate named query for each combination of dimensions we'd like to analyze, each returning different number of rows which are eventually translated to dimension members. But there's another approach to it.

It is possible to build a single utility dimension which contains enough members so that it can cover any, or most, types of histograms. The idea is to build The Tally table – a table with numbers starting from zero and ending with some big number. How big depends on your preference, could be 100, 1000, or more.

The name of the corresponding dimension should be unique, something neutral like the **Frequency** dimension.

The code presented in this recipe applies, but it must be modified to include the new dimension name. It can be repeated many times, once for each new distinct count type measure.

The only downside is that we have more members now than before, for a particular distinct count type measure. The query results and charts might look bad because of this. For example, if the dimension contains 100 members and we only have 13 frequencies like in the example presented in this recipe, most of the cells would be empty.

Turning the NON EMPTY switch wouldn't help either. It would only do damage by removing some of the members in the range. What we need is a list of numbers starting from zero and ending with the max value. If some of the members inside that range are empty, they must be preserved. The idea is to show the full range, not shrink it into something more compact by removing empty values.

How to solve this problem?

There are several solutions. One is to use the named sets defined in the MDX script defined as a range, 0 – max. Here's the example:

```
Create SET CurrentCube.[RO Frequency Set] AS
    { null : Tail( NonEmpty( [Frequency]
                                .[Interval].[Interval].MEMBERS,
                                [Measures].[RO Frequency] ),
                    1 ).Item(0)
```

In some tools, the set can be used on rows or columns instead of the complete Frequency dimension.

The other solution is to provide another scope in which you would convert nulls into zeros for members that form the range. Others would be null. That would work with NON EMPTY then.

## DSV or DW?

The initial T-SQL query could have been implemented in the relational DW database instead. After all, it is the best practice to keep everything related to business logic in one place and that place is the data warehouse (DW). However, the concept of the utility dimension is purely SSAS-related and hence falls in the domain of a BI developer, not a DW developer. He's the one who creates it, knows what it's for, and modifies it when required. That's the reason we've used the data source view (DSV) instead. Does it have to be DSV then? No, it is up to you to decide which place is better in your situation. Just be consistent.

### More calculations

Here's another calculation you may consider helpful. It calculates the percentage in total:

```
Create MEMBER CurrentCube.[Measures].[RO Frequency Total] AS
    ( [Measures].[RO Frequency],
      [Reseller Order Frequency].[Interval].DefaultMember )
, Visible = 0
, Format_String = '#,#'
, Associated_Measure_Group = 'Reseller Orders'
, Display_Folder = 'Histograms'
;

Create MEMBER CurrentCube.[Measures].[RO Frequency %] AS
    iif( [Measures].[RO Frequency Total] = 0,
        null,
        [Measures].[RO Frequency] /
        [Measures].[RO Frequency Total]
    )
, Format_String = 'Percent'
, Associated_Measure_Group = 'Reseller Orders'
, Display_Folder = 'Histograms'
;
```

The result looks like this:

Drop Filter Fields Here									
Business Type ▾									
	Specialty Bike Shop		Value Added Reseller		Warehouse		Grand Total		
Interval ▾	RO Frequency	RO Frequency %	RO Frequency	RO Frequency %	RO Frequency	RO Frequency %	RO Frequency	RO Frequency	RO Frequency %
0	31	13.42%	22	9.24%	13	5.60%	66	9.42%	
1	13	5.63%	11	4.62%	6	2.59%	30	4.28%	
2	8	3.46%	7	2.94%	5	2.16%	20	2.85%	
3	14	6.06%	14	5.88%	10	4.31%	38	5.42%	
4	71	30.74%	76	31.93%	88	37.93%	235	33.52%	
5	6	2.60%	7	2.94%	7	3.02%	20	2.85%	
6	10	4.33%	3	1.26%	8	3.45%	21	3.00%	
7	10	4.33%	12	5.04%	12	5.17%	34	4.85%	
8	44	19.05%	50	21.01%	51	21.98%	145	20.68%	
9	1	0.43%	5	2.10%	4	1.72%	10	1.43%	
10	1	0.43%	4	1.68%	3	1.29%	8	1.14%	
11	5	2.16%	1	0.42%	4	1.72%	10	1.43%	
12	17	7.36%	26	10.92%	21	9.05%	64	9.13%	
Grand Total	231	100.00%	238	100.00%	232	100.00%	701	100.00%	

It's visible that calculations work for other dimensions as long as they are related to the measure group of the **Reseller Order Count** measure, the measure used in the calculations.

### Other examples

The following link is a page describing the same thing in another way:

<http://tinyurl.com/OLAPHistograms>

### See also

The recipe *Using a distinct count measure to implementing histograms over existing hierarchies* covers a similar topic. *Creating a physical measure as a placeholder for MDX assignments* is directly related to this recipe and has an improved version of MDX assignments presented in this recipe.

## Creating a physical measure as a placeholder for MDX assignments

There can be many problems regarding calculated members. First, they don't aggregate up like regular members; we have to handle that by ourselves. Next, the drillthrough statements and security restrictions are only allowed for regular members. Not to mention the limited support for them in various client tools, for example in Excel. Finally, regular measures often noticeably beat their calculated measures counterparts in terms of performance or the ability to work with subselect.

On the other hand, calculated members can be defined and/or deployed very easily. They don't require any drastic change of the cube or the dimensions and therefore are good for testing, debugging, as a temporary solution, or even a permanent one. Anything goes as long as they don't become a serious obstacle in any of the ways mentioned earlier. When that happens, it's time to look for an alternative.

In the previous recipe, *Using a dummy dimension to implement histograms over non-existing hierarchies*, we used the EXISTING operator in the assignment made for the calculated measure **RO Frequency**. First, that expression tends to evaluate slowly which becomes noticeable on large dimensions. The **Reseller** dimension is not a large dimension and therefore, the provided expression will not cause significant performance issues there. However, the second problem is that the EXISTING operator will not react to the subselect and will therefore evaluate incorrectly if the subselect is used. That might become an issue with tools like Excel that extensively use the subselect whenever multiple members are selected in the filter area.

This recipe illustrates the alternative approach. It shows how a dummy physical measure can be used as a placeholder for calculations assigned to a calculated measure. In other words, how a calculated measure can be turned into a regular measure to solve a particular problem. As explained a moment ago, subselect is just one of the reasons we look for the alternative, which means the idea presented in this recipe can be applied in other cases too.

## Getting ready

This recipe depends heavily on the previous recipe, therefore you should implement the solution it presents. If you have read and practiced the recipes sequentially, you're all set. If not, simply read and implement the solution explained in the previous recipe in order to be able to continue with this one.

Once you have everything set up, you're ready to go with this one.

## How to do it...

Follow these steps to implement a physical measure as a placeholder for MDX assignments:

1. Double-click the **Adventure Works** data source view in the **Adventure Works DW 2008** solution opened in the **Business Intelligence Development Studio** (BIDS).
2. Locate the **Reseller Sales Fact** table in the pane with the list of tables on the left side.
3. Right-click it and add a new column by selecting **New Named Calculation** option.
4. The definition of the column should be:  
`CAST( null AS int )`
5. Name the column **NullMeasure**.
6. Specify **Histograms** for its **DisplayFolder** property and provide #, # for the **FormatString** property.
7. Double-click the **Adventure Works** cube and locate the **Reseller Sales** measure group.
8. Add a new physical measure in that measure group using the newly-added column in the underlying fact table and name it **R0 Frequency DSV**.
9. Use the **Sum** for the **AggregateFunction** and set the **NullProcessing** property (available inside the **Source** property) to **Preserve**.
10. Deploy changes.
11. Go to the **Calculations tab** and provide the adequate scope statement and MDX assignment for the new regular measure:  
`Scope( [Measures].[R0 Frequency DSV],  
[Reseller Order Frequency].[Interval].[Interval].Members,`

*When MDX is Not Enough* —————

```
[Reseller].[Reseller].[Reseller].Members  
);  
This = iif( [Reseller Order Frequency].[Interval]  
    .CurrentMember.MemberValue =  
    [Measures].[Reseller Order Count],  
    1,  
    null  
);  
End scope;
```

12. The **Reseller Order Frequency** dimension is not related to any fact table and automatic aggregations cannot work. Because of this, we have to perform additional steps by providing the assignment for the All member of the dummy dimension **Reseller Order Frequency**:

```
Scope( [Measures].[RO Frequency DSV],  
    [Reseller Order Frequency].[Interval].[All Intervals]  
);  
This = Sum( [Reseller Order Frequency].[Interval]  
    .[Interval].MEMBERS,  
    [Measures].[RO Frequency DSV]  
);  
End Scope;
```

13. Deploy the changes made in the MDX script.  
14. Now, go to the **Cube Browser** tab and build a pivot using the **Reseller Order Frequency** dimension on rows and both the new **RO Frequency DSV** measure and the old one, the **RO Frequency** measure on columns:

Drop Filter Fields Here		
Drop Column Fields Here		
Interval ▾	RO Frequency	RO Frequency DSV
0	66	66
1	30	30
2	20	20
3	38	38
4	235	235
5	20	20
6	21	21
7	34	34
8	145	145
9	10	10
10	8	8
11	10	10
12	64	64
Grand Total	701	701

15. As you can see in the image above, the calculation works.
16. Now, add the **Business Type** attribute hierarchy of the **Reseller** dimension in the subselect area of the **Cube Browser** and select **Specialty Bike Shop** for the **Filter Expression** there. The following image shows how it should look like:

The screenshot shows the Microsoft Analysis Services Cube Browser interface. At the top, there is a table for defining filters:

Dimension	Hierarchy	Operator	Filter Expression
Reseller	Business Type	Equal	{ Specialty Bike Shop }
<Select dimension>			

Below this is a large data grid titled "Drop Filter Fields Here". It contains two columns: "Interval" and "RO Frequency DSV". The data is as follows:

Interval	RO Frequency	RO Frequency DSV
0	66	31
1	30	13
2	20	8
3	38	14
4	235	71
5	20	6
6	21	10
7	34	10
8	145	44
9	10	1
10	8	1
11	10	5
12	64	17
Grand Total	701	231

17. Notice that the values for the **RO Frequency DSV** adjusted to the new context while the values of the **RO Frequency** measure (highlighted in the image) remained unchanged. The alternative approach worked!

### How it works...

A column, defined with the value of null, is created in the fact table and later used as a new cube measure. By default, the Analysis Services would turn the value of null into zero; however, we have prevented that by providing the **Preserve** option in the **NullProcessing** property. This produced a regular measure that is always null, in any context.

Now let's take a look at the definition of the measure from the previous recipe and see how it changed to the adequate scope statement. We'll repeat them here.

## *When MDX is Not Enough* ---

The calculated measure from the previous recipe is defined like this:

```
Create MEMBER CurrentCube.[Measures].[RO Frequency] AS
    Sum( EXISTING [Reseller].[Reseller].[Reseller].MEMBERS,
        iif( [Reseller Order Frequency].[Interval]
            .CurrentMember.MemberValue =
            [Measures].[Reseller Order Count],
            1,
            null
        )
    )
, Format_String = '#,#'
, Associated_Measure_Group = 'Reseller Orders'
, Display_Folder = 'Histograms'
;
```

The new measure from this recipe is scoped like this:

```
Scope( [Measures].[RO Frequency DSV],
    [Reseller Order Frequency].[Interval].[Interval].Members,
    [Reseller].[Reseller].Members
);
This = iif( [Reseller Order Frequency].[Interval]
    .CurrentMember.MemberValue =
    [Measures].[Reseller Order Count],
    1,
    null
);
End scope;
```

Notice the important difference in them. First, the need to detect existing resellers using the EXISTING operator is passed on to the server by adding the set of resellers inside the definition of the scope. Second, the need to summarize the values across the existing resellers is also passed on to the server, this time in the form of the `Sum()` aggregation function (not visible in the above expression) used for the new measure. In other words, the server does all the heavy work for us and makes the calculation work even with the subselect, as seen in the example. This is the benefit we get by using a dummy regular measure.

### There's more...

Improvement of this type comes with a price. In this case it is manifested in form of all the modifications you have to perform and maintain later, as well as the increased cube size because of the new measure. Because of that, it is up to you to decide whether it pays off to implement this alternative or not in your particular case. Hopefully you'll have the chance to test it in parallel to the existing solution and measure the costs/benefits ratio of the new solution.

Additional information can be found in Teo Lachev's article:

<http://tinyurl.com/TeoCalcAsRegular>

### Associated measure group

The same format string and display folder used for the calculated measure can be specified in the process of creating new measure. The only exception we made is that we used the Reseller Sales measure group. That's because the Reseller Orders measure group, the one to which the calculated measure was associated with, has a distinct count measure and distinct count measures should not be mixed with other regular measures.

We've decided to put the calculated measure in the best place we thought it should be. It made sense to put it in the same measure group as the Reseller Orders Count measure because they both count something related to resellers. Just for the record, we could have easily defined it on any measure group, it wouldn't make much difference.

### See also

Reading and implementing the recipe *Using a distinct count measure to implement histograms over existing hierarchies* is a prerequisite for this recipe.

## Using a new dimension to calculate the most frequent price

In this recipe we're going to analyze the products and their prices. The requirement is to find the most frequent price in any context.

Finding the price for a particular product doesn't look like a problem. We can either read it in its properties or calculate the average price based on the sales amount and the quantity sold.

Member properties are static and therefore choosing the first approach would be wrong. For example, the price in form of the member property doesn't change in time or with the territory. What we need is a dynamic expression.

The other option is to calculate the average price, but that's also not good enough. We need to know the exact price for each transaction, not the average value.

If we go low enough with the granularity of the query, the average price will eventually become the price used in the transaction, however, cubes are not optimized to be queried on leaf level. All in all, it's pretty obvious we need some help here. This can't be solved in MDX only.

The solution is to create a new dimension based on the price column in the fact table. That way we can have prices as an object we can use and manipulate within MDX calculations.

## *When MDX is Not Enough* —————

In addition to that, we need a measure that counts rows, but that's already there (or can be easily added) in every measure group (as long it's not the one with the distinct count measure). That measure can be used to isolate the price with the maximum number of rows for a given context. After that, we need to extract this price member's value which becomes the price we're after.

Let's see how it's done in this recipe.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** data source view.

### How to do it...

Follow these steps to change the model the cube is built from which in turn will enable simple and effective MDX calculations.

1. Create a new named query. Name it **Price** and use the following definition for it:

```
SELECT  
    DISTINCT  
    UnitPrice  
FROM  
    dbo.FactResellerSales
```

2. Execute the query in order to test it. It should return 233 rows.
3. Mark sure you've selected a single column in that table and then mark it as a **Logical Primary Key**.
4. Locate the **Reseller Sales** diagram in the **Diagram Organizer** pane.
5. Drag and drop the **Price** named query in the free area of that diagram and link the **Reseller Sales Facts** fact table to it using the **UnitPrice** columns in both objects.
6. Then create a new dimension using the previously defined **Price** named query and name it **Price**.
7. Use the **UnitPrice** column for the **KeyColumn**.
8. Name the attribute **Price**.
9. Once you finish with the wizard, the **Dimension Editor** will show.
10. Verify that the **OrderBy** property is set to **Key**.
11. Set the **ValueColumn** to **UnitPrice** column.
12. Name the All member **All Prices**.
13. Process-full the dimension.

14. When it's done, verify the dimension in the **Browser** tab of **Dimension Editor**. The prices should be sorted in ascending order.
15. Now, open the **Adventure Works** cube and go to the **Dimension Usage** tab of the cube to add the **Price** dimension.
16. Once added, the dimension will be automatically related to two measure groups: **Reseller Sales** and **Reseller Orders**. Verify that and then set its **Visible** property to **False** in the **Properties** pane.
17. Deploy the changes. When the processing is over, you're ready to write calculations.
18. Go to the **Calculations tab** and position the cursor in the end of the MDX script.
19. Enter the calculation for the Mode Price:

```
Create MEMBER CurrentCube.[Measures].[Mode Price] AS
    iif( IsEmpty( [Measures].[Reseller Transaction Count] ),
        null,
        TopCount( [Price].[Price].[Price].MEMBERS,
            1,
            [Measures].[Reseller Transaction Count]
        ).Item(0).MemberValue
    )
    , Format_String = 'Currency'
    , Display_Folder = 'Statistics'
    , Associated_Measure_Group = 'Reseller Sales' ;
```

20. Deploy changes in the MDX script and then go to the **Cube Browser** tab.
21. A comparison of the **Mode Price** and the **Reseller Average Unit Price** measures across the products subcategories can be seen in the following image:

Drop Filter Fields Here		Drop Column Fields Here	
Category	Subcategory	Reseller Average Unit Price	Mode Price
Accessories		\$21.67	\$20.19
Bikes	Mountain Bikes	\$1,085.21	\$647.99
	Road Bikes	\$779.14	\$469.79
	Touring Bikes	\$841.25	\$1,430.44
	Total	\$882.72	\$469.79
Clothing		\$28.21	\$28.84
Components	Bottom Brackets	\$56.39	\$72.89
	Brakes	\$63.88	\$63.90
	Chains	\$12.14	\$12.14
	Cranksets	\$179.76	\$242.99
	Derailleurs	\$61.36	\$54.89
	Forks	\$117.43	\$137.69
	Handlebars	\$44.73	\$33.77
	Headsets	\$59.78	\$74.84
	Mountain Frames	\$432.37	\$158.43
	Pedals	\$37.35	\$48.59
	Road Frames	\$335.10	\$202.33
	Saddles	\$26.17	\$31.58
	Touring Frames	\$425.50	\$602.35
	Wheels	\$127.53	\$67.54
	Total	\$251.40	\$202.33
Grand Total		\$444.43	\$469.79

## How it works...

By counting rows in the original fact table we're basically saying we want to know how many times the current price occurred in the context of other dimensions. That, of course, demands for the **Price** dimension which we have made in the initial steps of the solution presented in this recipe.

There's something else to it. The **Price** dimension is kept aside all the time by being invisible. Its only purpose is to spread the prices so that we can pick the most frequent one. The evaluation of the most frequent price is done in the definition of the calculated measure, where we're applying the `TopCount( )` function over all prices and therefore use the `Reseller Transaction Count` measure to count how many occurrences each price had in the given context. For the record, the `Reseller Transaction Count` measure is also invisible, but it can be used in the calculations.

If you want to verify the correctness of the calculation, turn both the **Price** dimension and the **Reseller Transaction Count** measure visible, deploy changes, and use them in the pivot. The price with the highest value of the **Reseller Transaction Count** measure is automatically selected as the **Mode Price**.

In the end, it's worth mentioning that once we had the most frequent price, we read its member value and showed it as the result of the calculation.

## There's more...

It's perfectly normal to encounter the situations where two or more prices have the same frequency - the same rate of occurrence. However, the calculation we made can take only a single one. How do we know which one?

Unlike datasets resulting from queries in the relational world, the default behavior of sets in the multidimensional world is that they are always sorted unless explicitly requested otherwise. Order is determined during the design of the dimensions. Each attribute has a property for that. It can be a **Key**, **Name**, or another attribute. The point is – members in are sorted one way or the other.

Because of this, in the case of a tie, the order of members determines the winner. If members are sorted by the key, a member with the lower key will win. If members are sorted by the name, the one that alphabetically comes sooner wins. If you need the opposite order, consider using another attribute or switch the keys to their negative values.

## Using a utility dimension to implement flexible display units

Measures are sometimes too precise for reporting needs. Either because they have decimals which distract us or because the numbers are very large, consisting of many digits, and therefore hard to memorize or compare with others. The phrase "can't see the forest for the trees" fits here perfectly.

One way to simplify this problem would be to divide those measures directly in the fact table by, let's say 1,000 or 1,000,000. We would of course have to specify the new unit in the title of the measure. However, that wouldn't be a good solution. In situations when we need the results to be as precise as possible, that would cause problems. As said, only sometimes, not always, there's a need to simplify the numbers.

The other way would be to generate a set of parallel calculated measures, one for each factor. That way we would have Sales Amount, then Sales Amount (000), and finally Sales Amount (000,000) or similar markings. Again, not a very good approach. End users would have a hard time finding the right measure in a set of so many calculated measures.

The next option is the best, although it can have its specifics that may demand our attention. It's the case of building a separate utility dimension to display values in thousands, millions, and so on. This way we can preserve the simplicity of the cube design while enjoying the benefits of being able to visualize measures in other metric. This recipe shows how to achieve such a solution. It also tackles problems specific to this type of solution.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** data source view.

### How to do it...

Follow these steps to create utility dimension which displays result in various formats.

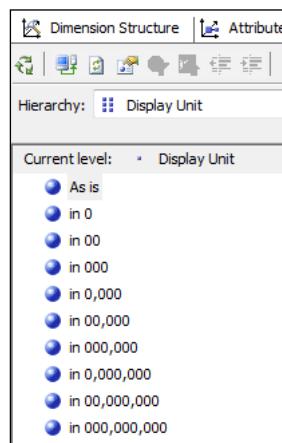
1. Create a new named query **Calculations - Display Units** with the following definition:

```
SELECT      0 AS UnitKey, 'As is' AS UnitName
UNION ALL
SELECT      1      , 'in 0'
UNION ALL
SELECT      2      , 'in 00'
UNION ALL
SELECT      3      , 'in 000'
UNION ALL
```

*When MDX is Not Enough* —————

```
SELECT      4      , 'in 0,000'  
UNION ALL  
SELECT      5      , 'in 00,000'  
UNION ALL  
SELECT      6      , 'in 000,000'  
UNION ALL  
SELECT      7      , 'in 0,000,000'  
UNION ALL  
SELECT      8      , 'in 00,000,000'  
UNION ALL  
SELECT      9      , 'in 000,000,000'
```

2. Execute the query in order to test it. It should return 10 rows.
3. Set the **UnitKey** column as a **Logical Primary Key** column.
4. Create a new dimension using the previously defined named query and name it **Calculations – Display Units**.
5. Use the **UnitKey** column for the **KeyColumn** and the **UnitName** column for the **NameColumn** properties.
6. Name the attribute **Display Unit**.
7. Set the **IsAggregatable** property of the attribute to **False**.
8. Set the **OrderBy** property to **Key**.
9. Set the **ValueColumn** to **UnitKey** column.
10. Process-full the dimension.
11. Set the default member for the attribute to the **As is** member. The expression in the property should be `[Calculations - Display Units].[Display Unit].&[0]` once you're done.
12. Deploy and verify in the **Browser** tab of **Dimension Editor** that your new utility dimension looks like this:



13. Open the **Adventure Works** cube and add that dimension without linking it to any measure group.
14. Deploy changes by right-clicking on the project name (in the **Solution Explorer** pane) and choosing the **Deploy** action.
15. Go to the **Calculations tab** and add the following code in the beginning of the MDX script:

```

Create Hidden SET CurrentCube.[Display Units Set] AS
    Except( [Calculations - Display Units].[Display Unit].MEMBERS,
            [Calculations - Display Units].[Display Unit].&[0] )
;
Scope( [Display Units Set] );
    This = [Calculations - Display Units].[Display Unit].&[0]
    /
    10 ^ [Calculations - Display Units].[Display Unit]
        .CurrentMember.MemberValue;
    Format_String( This ) = '#,##0';
End Scope;

```

16. Deploy the solution.
17. Go to the **Cube Browser** tab and build a pivot using the new dimension on columns where you select only some members, for example, display in thousands and **As is**. Put the **Gross Profit**, the **Sales Amount**, and the **Ratio to Parent Product** measures in the data area and product categories on rows. You should be able to see that the values are reduced as required for amounts, but not for the percentages:

Drop Filter Fields Here						
	Display Unit ▾		As is in 000			
Category ▾	Gross Profit	Sales Amount	Ratio to Parent Product	Gross Profit	Sales Amount	Ratio to Parent Product
Accessories	\$634,467.16	\$1,272,057.89	1.16%	\$634.47	1,272	1.16%
Bikes	\$10,515,096.61	\$94,620,526.21	86.17%	\$10,515.10	94,621	86.17%
Clothing	\$368,836.00	\$2,117,613.45	1.93%	\$368.84	2,118	1.93%
Components	\$1,032,966.48	\$11,799,076.66	10.75%	\$1,032.97	11,799	10.75%
Grand Total	\$12,551,366.25	\$109,809,274.20	100.00%	\$12,551.37	109,809	100.00%

### How it works...

The Utility dimension, a dimension not related to any fact table of the cube, can be understood as a cube parameter. The logic we have implemented using the scope statement in the MDX script acts like a modification of the cube. When any of the members of that dimension is in the context, the scope fires. The calculation specifies that the value of the measure will be reduced by factor 10 to the power of the member value of the current member. The formula can be so elegant because we used the `MemberValue` property, the same sequence of numbers as in the dimension key.

The definition of the scope says that the value of any measure will be divided by 10 to the power of N, where N is read from the `MemberValue` property of the current member of that utility dimension.

### There's more...

Things don't run that smoothly. There's a potential problem in this approach.

One is that we lose the format string of the measure (see the **Sales Amount** measure in the previous image); the other is that the scope is applied to all measures. Let's see what we can do about that.

#### Set-based approach

We can expand the scope and specify which measures we want in it. That way we can precisely control the format string for each set of measures.

Here's an example how it can be done. Add this new hidden set immediately after the previously defined hidden set and modify the scope as specified below:

```
Create Hidden SET CurrentCube.[Measures Set] AS
    { [Measures].[Extended Amount],
      [Measures].[Freight Cost],
      [Measures].[Sales Amount],
      [Measures].[Standard Product Cost],
      [Measures].[Total Product Cost],
      [Measures].[Tax Amount] }
;

Scope( [Display Units Set],
      [Measures Set] );
This = [Calculations - Display Units].[Display Unit].&[0]
/
10 ^ [Calculations - Display Units].[Display Unit]
    .CurrentMember.MemberValue;
Format_String(This) = 'Currency';
End Scope;
```

Here's the same example using one regular, one calculated, and one ratio measure with the scope applied to the **in 000** member:

Drop Filter Fields Here						
	Display Unit ▾		As is			
Category	Gross Profit	Sales Amount	Ratio to Parent Product	Gross Profit	Sales Amount	Ratio to Parent Product
	\$634,467.16	\$1,272,057.89	1.16%	\$634.47	\$1,272.06	1.16%
Accessories	\$10,515,096.61	\$94,620,526.21	86.17%	\$10,515.10	\$94,620.53	86.17%
Bikes	\$368,836.00	\$2,117,613.45	1.93%	\$368.84	\$2,117.61	1.93%
Clothing	\$1,032,966.48	\$11,799,076.66	10.75%	\$1,032.97	\$11,799.08	10.75%
Components	\$12,551,366.25	\$109,809,274.20	100.00%	\$12,551.37	\$109,809.27	100.00%
Grand Total						

Only regular cube measures can be explicitly specified in the scope, no calculated ones. The effect will manifest in all subsequently defined calculated measures too because their definition is dependent on the values of regular measures and since the regular measures will be reduced, so shall the calculated measure.

The scoped set solution can be repeated many times. We can group measures of the same type in multiple sets and use an explicit format string for each of those sets. However, that's not too practical of a solution. Moreover, the reduced values are still not readable. It would be great if we could lose decimals and currency symbols that distract us.

### Format string on a filtered set approach

A more general way to specify various measures in the cube without having to list them all in the scope or hidden sets heavily depends on the naming convention used in the cube. We can filter all measures, regular and calculated ones, based on some criteria and apply the format only to those measures that don't have the words "price", "margin", "ratio", etc. in them. Here's an example for that.

Modify the definition of the Measures Set according to the expression provided below. Modify the scope statement, but move it to the end of the MDX script this time, not the beginning. Also, make sure the Display Units Set is still available in the MDX script since the scope is referring to it:

```
Create Hidden SET CurrentCube.[Measures Set] AS
  Filter( Measures.AllMembers,
    --MeasureGroupMeasures('Sales Summary'),
    InStr( Measures.CurrentMember.Name, 'Price' ) = 0 AND
    InStr( Measures.CurrentMember.Name, 'Margin' ) = 0 AND
    InStr( Measures.CurrentMember.Name, 'Percentage' ) = 0 AND
    InStr( Measures.CurrentMember.Name, 'Growth' ) = 0 AND
    InStr( Measures.CurrentMember.Name, 'Rate' ) = 0 AND
    InStr( Measures.CurrentMember.Name, 'Ratio' )
```

When MDX is Not Enough

---

```
= 0
)
;

Scope( [Display Units Set] );
Format_String( [Calculations - Display Units].[Display Unit].&[3]
    * [Measures Set] ) = '#,##0,';
Format_String( [Calculations - Display Units].[Display Unit].&[6]
    * [Measures Set] ) = '#,##0,,';
Format_String( [Calculations - Display Units].[Display Unit].&[9]
    * [Measures Set] ) = '#,##0,,,';
End Scope;
```

When deployed, this solution works just right. The amounts are readable while the percentages remain percentages:

Drop Filter Fields Here							
Category	Display Unit ▾		in 000				
	As is		Gross Profit	Sales Amount	Ratio to Parent Product	Gross Profit	Sales Amount
Accessories	\$634,467.16	\$1,272,057.89	1.16%	634	1,272	1.16%	
Bikes	\$10,515,096.61	\$94,620,526.21	86.17%	10,515	94,621	86.17%	
Clothing	\$368,836.00	\$2,117,613.45	1.93%	369	2,118	1.93%	
Components	\$1,032,966.48	\$11,799,076.66	10.75%	1,033	11,799	10.75%	
Grand Total	\$12,551,366.25	\$109,809,274.20	100.00%	12,551	109,809	100.00%	

Notice that we didn't have to modify actual values in the scope statement; we only changed the format string for certain measures. This makes it a faster approach.

The other thing to notice is that we were able to use both calculated and regular measures. That is because in the `Format_String()` function we are allowed to use both types of measures, whereas in the scope's definition we are not allowed to use calculated measures. By carefully constructing our expression, we were able to achieve the goal.

The only downside is that we only made it work for some members of the utility dimension, not all of them. However, that should not be a problem because those members are used most of the time anyway (in 000, in 000,000, in 000,000,000 and so on). In other words, if we stick with the latest solution, we can remove all not-used members from the utility dimension.

The commented part of the scope statement shows that we can limit the filter to one or more measure groups.

## See also

The recipes *Setting special format for negative, zero and null values* and *Applying conditional formatting on calculations*, both in chapter 1, show additional formatting options.

## Using a utility dimension to implement time-based calculations

The MDX language implemented in SQL Server Analysis Services offers various time-related functions. Chapter 2 showed how they can be utilized to construct useful sets and calculations. The main focus there was to show how to make OLAP cubes time-aware; how to detect the member that represents today's date and then create related calculations.

In this chapter we meet time functions again. This time our focus is to generate time-based calculations in the most effective way. One way to do it is to use the built-in **Time Intelligence Wizard**, but that path is known to be a problematic one. The best practice says we need a separate utility dimension instead, a utility dimension with members representing common time-based calculations like the year-to-date, year-over-year, previous period, and so on.

The idea behind the utility dimension is to minimize the number of calculations in a way that measures are combined with members of the utility dimension. That effectively means we have a Cartesian product of measures and their possible time-related variants. A more elegant approach than having many calculated measures in the cube, not to mention maintaining them or presenting to end users.

Members on the utility dimension have more or less complex formulas. However, once we set them up in one project, we can use them in future ones. As long as we keep metadata the same: the names of the attributes and levels of the date dimension and this utility dimension, the process comes down to copy/paste and is done in a minute. Sounds tempting?

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** data source view.

Before we start, I should inform you that this recipe is heavy on MDX. Although it has the longest code in the whole book, it is a very useful one. Once adjusted to your date dimension, the code can be reused in every project that has the same date dimension. If the dimension is the same as in the Adventure Works DW database, then no further modification is needed. Here I am by no means saying you should design your date dimension the way it is done in Adventure Works. You don't. I'm merely saying you should start with that, read the explanations, analyze the code, and then adjust it to your needs.

## How to do it...

Follow these steps to create a utility dimension with relative time calculations.

1. Create a new named query **Time Calcs** using the following definition:

```
SELECT 0 AS ID, 'As is' AS Name

UNION ALL
SELECT 1      , 'YTD'
UNION ALL
SELECT 2      , 'Prev period'
UNION ALL
SELECT 3      , 'Year ago'
UNION ALL
SELECT 4      , 'YTD Year ago'

UNION ALL
SELECT 21     , 'Prev period i'
UNION ALL
SELECT 22     , 'Prev period %'
UNION ALL
SELECT 31     , 'Year ago i'
UNION ALL
SELECT 32     , 'Year ago %'
UNION ALL
SELECT 41     , 'YTD Year ago i'
UNION ALL
SELECT 42     , 'YTD Year ago %'
```

2. Execute the query in order to test it. It should return 11 rows. Then close the window.
3. Mark the **ID** column as the **Logical Primary Key** column.
4. Create a new dimension using the previously defined named query and name it **Time Calcs**.
5. Use the **ID** column for the **KeyColumn** and the **Name** column for the **NameColumn** properties.
6. Name the attribute **Calc**.
7. Once you finish with the wizard a **Dimension Editor** will show.
8. Set the **IsAggregatable** property of the attribute to **False**.
9. Set the **OrderBy** property to **Key**.
10. Set the **ValueColumn** to **ID** column.
11. Process-full the dimension.

12. Set the default member for the attribute to the **As is** member. The expression in that property should be `[Time Calcs].[Calc].&[0]` once you're done.
13. Deploy and verify the dimension in the **Browser** tab of the **Dimension Editor**.
14. Open the **Adventure Works** cube and add that dimension without linking it to any measure group, then deploy.
15. Now we're ready to write calculations. Go to the **Calculations tab** and position the cursor at the end of the MDX script.
16. Add the calculation for the YTD member. In the case of Adventure Works, the calculation should consist of three parts. The first two parts handle attribute hierarchies not related to the year level and thus enable them to be used in combination with the YTD member. The last part is a classic YTD calculation expanded to work for all visible attribute hierarchies that make sense to be used directly with the year level.

```

Scope( [Time Calcs].[Calc].[YTD] );
    -- focus on anything below and including year level
Scope( [Date].[Date].MEMBERS,
    [Date].[Calendar Year].[Calendar Year].MEMBERS );
    -- attribute hierarchies not related to year level
    -- and related to months
    -- i.e. day of month
Scope( [Date].[Month of Year].[Month of Year].MEMBERS );
    This = Aggregate(
        YTD( [Date].[Calendar].CurrentMember ) *
        YTD( [Date].[Calendar Weeks].CurrentMember ) *
        { null : [Date].[Month of Year].CurrentMember
    } *
        { null : [Date].[Day of Month].CurrentMember
    },
        [Time Calcs].[Calc].&[0]
    );
End Scope;
    -- attribute hierarchies not related to year level
    -- and related to weeks
    -- i.e. day of week & day name
Scope( [Date].[Calendar Week of Year]
    .[Calendar Week of Year].MEMBERS );
    This = Aggregate(
        YTD( [Date].[Calendar].CurrentMember ) *
        YTD( [Date].[Calendar Weeks].CurrentMember ) *
        { null : [Date].[Calendar Week of Year]
            .CurrentMember } *
        { null : [Date].[Day of Week].CurrentMember }
    *
        { null : [Date].[Day Name].CurrentMember },

```

```
[Time Calcs].[Calc].&[0]
);
End Scope;
-- user hierarchies
-- and attribute hierarchies related to year level
This = Aggregate(
    YTD( [Date].[Calendar].CurrentMember ) *
    YTD( [Date].[Calendar Weeks].CurrentMember ) *
    { null : [Date].[Calendar Semester of Year]
        .CurrentMember } *
    { null : [Date].[Calendar Quarter of Year]
        .CurrentMember } *
    { null : [Date].[Month of Year].CurrentMember
} *
    { null : [Date].[Calendar Week of Year]
        .CurrentMember } *
    { null : [Date].[Day of Year].CurrentMember }
*
{ null : [Date].[Date].CurrentMember },
[Time Calcs].[Calc].&[0]
);
End Scope;
-- performance is improved if we preserve empty cells
-- remove this to get continuous date ranges
This = iif( IsEmpty( [Time Calcs].[Calc].&[0] ),
            null,
            Measures.CurrentMember );
End Scope;
```

17. Notice the comments inside the calculation. They are there to help you know what's going on.
18. Add the calculation for the Prev period member. The idea is to allow hierarchies to expand by one member to the left, if possible, and then to apply the intersection. The first part of the scope should handle user hierarchies; the second one is dedicated to attribute hierarchies. Additionally, when the intersection subcube contains only a single member, a circle shift in combination with the previous year should occur. In case there is no previous year, an empty value should be returned.

```
Scope( [Time Calcs].[Calc].[Prev period] );
    -- user hierarchies, complete
Scope( [Date].[Calendar].MEMBERS,
    [Date].[Calendar Weeks].MEMBERS );
    -- previous member exists? (null for "no")
This = iif( Count( LastPeriods( 2, [Date].[Calendar]
                                .CurrentMember )
*
LastPeriods( 2, [Date].[Calendar Weeks]
                                .CurrentMember )
```

```

        ) <= 1,
        null,
        -- include one member to the left on each
        -- hierarchy and see what comes out as the
        -- intersection, then deduct the value in
        -- the current context from it
        Aggregate(
            LastPeriods( 2, [Date].[Calendar]
                .CurrentMember ) *
            LastPeriods( 2, [Date].[Calendar Weeks]
                .CurrentMember ),
            [Time Calcs].[Calc].&[0]
                ) -
            [Time Calcs].[Calc].&[0]
        );
    End Scope;
    -- attribute hierarchies directly related to year level
    Scope( [Date].[Calendar Year].[Calendar Year].MEMBERS );
        -- previous member exists? (test prev year for "no")
        This = iif( Count(
            LastPeriods( 2, [Date].[Calendar Semester of
Year]
                .CurrentMember ) *
            LastPeriods( 2, [Date].[Calendar Quarter of Year]
                .CurrentMember ) *
            LastPeriods( 2, [Date].[Month of Year]
                .CurrentMember ) *
            LastPeriods( 2, [Date].[Calendar Week of Year]
                .CurrentMember ) *
            LastPeriods( 2, [Date].[Day of Year]
                .CurrentMember ) *
            LastPeriods( 2, [Date].[Date]
                .CurrentMember ) ) <= 1,
            -- previous year exists? (null for "no")
            iif( Count( LastPeriods( 2, [Date].[Calendar
Year]
                .CurrentMember
            )
                ) <= 1,
            null,
            -- members shift in circle
            -- from the first position to the last
            -- hence we need the previous year
            (

```

```
[Date].[Calendar Year].PrevMember,  
[Date].[Calendar Semester of Year].  
LastSibling,  
[Date].[Calendar Quarter of Year].  
LastSibling,  
[Date].[Month of Year].LastSibling,  
[Date].[Calendar Week of Year].LastSibling,  
[Date].[Day of Year].LastSibling,  
[Date].[Date].LastSibling,  
[Time Calcs].[Calc].&[0]  
)  
,  
-- include one member to the left on each  
-- hierarchy and see what comes out as the  
-- intersection, then deduct the value in the  
-- current context from it  
Aggregate(  
    LastPeriods( 2, [Date].[Calendar Semester of  
        Year]  
            .CurrentMember ) *  
    LastPeriods( 2, [Date].[Calendar Quarter of  
        Year]  
            .CurrentMember ) *  
    LastPeriods( 2, [Date].[Month of Year]  
            .CurrentMember ) *  
    LastPeriods( 2, [Date].[Calendar Week of Year]  
            .CurrentMember ) *  
    LastPeriods( 2, [Date].[Day of Year]  
            .CurrentMember ) *  
    LastPeriods( 2, [Date].[Date].CurrentMember ),  
    [Time Calcs].[Calc].&[0]  
) -  
[Time Calcs].[Calc].&[0]  
);  
End Scope;  
End Scope;
```

19. The next calculation is relatively simple compared to those before. Define the calculation for the Year ago member, a member that returns the value for the same period in the previous year:

```

Scope( [Time Calcs].[Calc].[Year ago] );
    -- focus on anything below and including year level
    Scope( [Date].[Date].MEMBERS,
        [Date].[Calendar Year].[Calendar Year].MEMBERS );
        -- jump one year back on both user hierarchies
        This = ( ParallelPeriod( [Date].[Calendar].[Calendar
Year],
            1,
            [Date].[Calendar].CurrentMember
),
    ParallelPeriod( [Date].[Calendar Weeks]
        .[Calendar Year],
        1,
        [Date].[Calendar Weeks]
        .CurrentMember ),
    [Time Calcs].[Calc].&[0]
);
End Scope;
End Scope;

```

20. Adding the YTD variant of the previous calculation is also easy. To define the YTD Year ago member, simply use the **YTD** member of the utility dimension in the definition used for the Year ago member:

```

Scope( [Time Calcs].[Calc].[YTD Year ago] );
    -- focus on anything below and including year level
    Scope( [Date].[Date].MEMBERS,
        [Date].[Calendar Year].[Calendar Year].MEMBERS );
        This = ( ParallelPeriod( [Date].[Calendar].[Calendar
Year],
            1,
            [Date].[Calendar].CurrentMember
),
    ParallelPeriod( [Date].[Calendar Weeks]
        .[Calendar Year],
        1,
        [Date].[Calendar Weeks]
        .CurrentMember ),
        -- don't forget to use the YTD member this time
        [Time Calcs].[Calc].[YTD]
);
End Scope;

```

21. Now that we've defined the main calculations, we're ready to proceed with indexes and percentages. The first indicator (Prev period i) determines the difference from a previous period, the second one (Year ago i) from a year ago and finally, the third one (YTD Year ago i) also from a year ago, but cumulative (YTD). All of them are additionally formatted with a red font color for negative values. The i stands for index.

```
Scope( [Time Calcs].[Calc].[Prev period i] );
    This = [Time Calcs].[Calc].&[0] -
        [Time Calcs].[Calc].[Prev period];
    Fore_Color(This) = Abs( Measures.CurrentMember < 0 ) * 255;
End Scope;

Scope( [Time Calcs].[Calc].[Year ago i] );
    This = [Time Calcs].[Calc].&[0] -
        [Time Calcs].[Calc].[Year ago];
    Fore_Color(This) = Abs( Measures.CurrentMember < 0 ) * 255;
End Scope;

Scope( [Time Calcs].[Calc].[YTD Year ago i] );
    This = [Time Calcs].[Calc].[YTD] -
        [Time Calcs].[Calc].[YTD Year ago];
    Fore_Color(This) = Abs( Measures.CurrentMember < 0 ) * 255;
End Scope;
```

22. Add the code for 3 indicators in form of percentages: Prev period %, Year ago % and YTD Year ago %. The % sign stands for the value expressed in the form of percentage:

```
Scope( [Time Calcs].[Calc].[Prev period %] );
    This = iif( [Time Calcs].[Calc].[Prev period] = 0, null,
        [Time Calcs].[Calc].&[0] /
        [Time Calcs].[Calc].[Prev period] - 1);
    Format_String(This) = 'Percent';
    Fore_Color(This) = Abs( Measures.CurrentMember < 0 ) * 255;
End Scope;

Scope( [Time Calcs].[Calc].[Year ago %] );
    This = iif( [Time Calcs].[Calc].[Year ago] = 0, null,
        [Time Calcs].[Calc].&[0] /
        [Time Calcs].[Calc].[Year ago] - 1);
    Format_String(This) = 'Percent';
    Fore_Color(This) = Abs( Measures.CurrentMember < 0 ) * 255;
End Scope;

Scope( [Time Calcs].[Calc].[YTD Year ago %] );
    This = iif( [Time Calcs].[Calc].[YTD Year ago] = 0, null,
```

```
[Time Calcs].[Calc].[YTD] /
[Time Calcs].[Calc].[YTD Year ago] - 1);
Format_String(This) = 'Percent';
Fore_Color(This) = Abs( Measures.CurrentMember < 0 ) * 255;
End Scope;
```

23. You're done with calculations! Now, deploy changes in MDX script and test the solution in the **Cube Browser** tab.
24. Calculations should work correctly no matter which hierarchy you take in pivot, as long as you don't use fiscal hierarchies. See the following image with two attribute hierarchies on rows, years and months:

Drop Filter Fields Here									
Calc ▾									
Calendar Year ▾	Month of Year ▾	Sales Amount	YTD	Prev period	Year ago	YTD Year ago	Prev period i	Prev period %	Ye
CY 2005	July	\$962,716.74	\$962,716.74	\$0.00			\$962,716.74		
	August	\$2,044,600.00	\$3,007,316.75	\$962,716.74			\$1,081,883.26	112.38%	
	September	\$1,639,840.11	\$4,647,156.85	\$2,044,600.00			(\$404,759.89)	-19.80%	
	October	\$1,358,050.47	\$6,005,207.32	\$1,639,840.11			(\$281,789.64)	-17.18%	
	November	\$2,868,129.20	\$8,873,336.53	\$1,358,050.47			\$1,510,078.73	111.19%	
	December	\$2,458,472.43	\$11,331,808.96	\$2,868,129.20			(\$409,656.77)	-14.28%	
	Total	\$11,331,808.96	\$11,331,808.96				\$11,331,808.96		
CY 2006	January	\$1,309,863.25	\$1,309,863.25	\$2,458,472.43			(\$1,148,609.18)	-46.72%	
	February	\$2,451,605.62	\$3,761,468.88	\$1,309,863.25			\$1,141,742.37	87.17%	
	March	\$2,099,415.62	\$5,860,884.49	\$2,451,605.62			(\$352,190.01)	-14.37%	
	April	\$1,546,592.23	\$7,407,476.72	\$2,099,415.62			(\$552,823.39)	-26.33%	
	May	\$2,942,672.91	\$10,350,149.63	\$1,546,592.23			\$1,396,080.68	90.27%	
	June	\$1,678,567.42	\$12,028,717.05	\$2,942,672.91			(\$1,264,105.49)	-42.96%	
	July	\$2,894,054.68	\$14,922,771.73	\$1,678,567.42	\$962,716.74	\$962,716.74	\$1,215,487.26	72.41%	
	August	\$4,147,192.18	\$19,069,963.91	\$2,894,054.68	\$2,044,600.00	\$3,007,316.75	\$1,253,137.50	43.30%	
	September	\$3,235,826.19	\$22,305,790.10	\$4,147,192.18	\$1,639,840.11	\$4,647,156.85	(\$911,365.99)	-21.98%	
	October	\$2,217,544.45	\$24,523,334.55	\$3,235,826.19	\$1,358,050.47	\$6,005,207.32	(\$1,018,281.74)	-31.47%	
	November	\$3,388,911.41	\$27,912,245.97	\$2,217,544.45	\$2,868,129.20	\$8,873,336.53	\$1,171,366.97	52.82%	
	December	\$2,762,527.22	\$30,674,773.18	\$3,388,911.41	\$2,458,472.43	\$11,331,808.96	(\$626,384.20)	-18.48%	
	Total	\$30,674,773.18	\$30,674,773.18	\$11,331,808.96	\$11,331,808.96	\$11,331,808.96	\$19,342,964.22	170.70%	
CY 2007	January	\$1,756,407.01	\$1,756,407.01	\$2,762,527.22	\$1,309,863.25	\$1,309,863.25	(\$1,006,120.21)	-36.42%	
	February	\$2,873,936.93	\$4,630,343.93	\$1,756,407.01	\$2,451,605.62	\$3,761,468.88	\$1,117,529.92	63.63%	
	March	\$2,049,529.87	\$6,679,873.80	\$2,873,936.93	\$2,099,415.62	\$5,860,884.49	(\$824,407.05)	-28.69%	
	April	\$2,371,677.70	\$9,051,551.50	\$2,049,529.87	\$1,546,592.23	\$7,407,476.72	\$322,147.83	15.72%	
	May	\$3,443,525.25	\$12,495,076.75	\$2,371,677.70	\$2,942,672.91	\$10,350,149.63	\$1,071,847.55	45.19%	
	June		!!!						

## How it works...

The solution presented in this recipe is a long one. Therefore we'll break the explanation in several phases.

The first phase was creating the utility dimension. We created the utility dimension in DSV. It can also be created in DW.

The process of creating a utility dimension is pretty straightforward. The first member becomes the default member, the All member is disabled. Other actions help but are not mandatory. I do suggest you make them though.

The second phase started when we defined 4 main calculations and assigned them to members of the utility dimension. We used one or more scope statements for that, to precisely determine the context in which particular assignments should be applied.

## *When MDX is Not Enough* —————

The first of these calculations defined what should happen when the YTD member gets selected in the utility dimension. To remind you, this was done in step 16.

The outer scope in that calculation limits the cube space to anything including and below the year level which is exactly the subcube that YTD calculation should be applied to.

The first thing we did is provided the assignments for two attribute hierarchies – the Month of Year and the Calendar Week of Year hierarchy. This is not a mandatory step when defining YTD calculations and you'll rarely see it made. However, it pays off to implement it because those additional assignments enable the usage of all attribute hierarchies for the YTD calculation. In other words, they allow us to combine several attribute hierarchies just like they were levels of a natural user hierarchy.

Additional assignments like these come with a prerequisite - our guarantee that they make sense in the particular context. This is defined using the scope subcube.

If you take one more look at the YTD calculation, you'll notice that YTD values for the Day of Month hierarchy will only be calculated when months are also present in the context. Similarly, it makes sense to calculate the YTD values for Day of Week and Day Name hierarchies too, but only when weeks are in the context. This is how the invisible time sequence gets established.

The third assignment in that calculation is the one made for the user hierarchies and all attribute hierarchies that make sense to be used together with years. Just like in the previous assignments, two natural user hierarchies are used to collect the YTD range while all other hierarchies filter that context by interacting with it in form of intersections. What comes out of that intersection is a subcube that represents the YTD values for that particular context.

The Prev period member is also a complex one. Again, we have two scopes: one for user hierarchies and the other for attribute hierarchies. In order to keep things concise, we'll explain the logic behind this calculation by focusing on the main intentions here.

The idea behind calculating the value in the previous period is to extend every current member with its previous member, aggregate that, and deduct the value in the current context from that. What comes as the result is the value in the previous period.

Notice how we keep using the generic term "period". It's not a month, it's not a week; it can be anything depending on what we've put in the context. The shape of the subcube is not our concern. All we have to do is provide a **potential subcube** for this calculation. In the case of the Prev period member, we did that by specifying the current and previous member for each relevant hierarchy. So, the idea to chase the current or existing members of every hierarchy and then detect whether they've moved or not is not the way to go. We should think in terms of sets and allow the engine to do the magic by intersecting members in the provided subcube.

Two additional things were important in the previous period calculation. First, counting is a way of detecting when we're on the first member of a hierarchy when multiple hierarchies are involved. When we encounter the first member, we either have to shift the year by minus one (by referring to the previous member), or to provide `null` because there's nowhere to go. Second, when we shift the year to the previous one, we also need to move other attribute hierarchies to their last member. At first, this may seem strange, but it's actually not. It's something we do unconsciously every day. Here's an example proving that.

The year that precedes the year 2000 is the year 1999. In order to evaluate that, we do the following procedure. We slice the year by its digits and analyze them. We then realize that all the digits on the right are the first digits in the list of digits, 0 being the first digit in the decade system and 9 being the last. Because of that we change the 2 into its preceding digit, the digit 1. At the same time, we shift all other digits by one place, as if 0 and 9 were connected in a circle. In other words, all other digits jump from the first digit 0 to the last one, the 9. The same principle was applied here, year hierarchy being the leftmost digit 2 and other hierarchies being digits 0 on the right.

But what will happen in that calculation? Again, we're letting the current members decide what the result of the intersection will be. If any of them is on the root member, that member will not influence the intersection. That's a single member on its level which means the `LastSibling` function, used to shift members in a circle, will return the same member. When another member is active, we'll jump to the last member of that level, for example to Q4, December, and so on. Again, the most selective members of all hierarchies determine the final subcube.

In the third calculation, the `Year ago` member is calculated using two user hierarchies. The tuple is formed using two `ParallelPeriod()` functions that return the member in the previous year. Naturally, the last part of the tuple is the default member of the utility dimension.

As this was a relatively easy calculation, let's explain what the default member of the utility calculation does in expressions.

The cube is a multidimensional structure which means every time we make a reference to a particular subcube, the coordinate of that subcube is also multidimensional. Fortunately, we don't have to specify all hundreds or more hierarchies a typical cube has; the server does that for us by implicitly including all unmentioned hierarchies in every tuple we make. Tuple is just another word for the coordinate in the cube space.

Implicit hierarchies are included using their current members at the time of evaluation of the tuple.

In the case of the calculations presented in this recipe, now it becomes clear that the current measure is already there implicitly in every expression. There's no reason to be explicit about that.

## *When MDX is Not Enough* —————

What's not obvious at first is that the hierarchy of the member we're defining is also inside the tuple. Yes, the utility dimension. Now, guess what will be its current member at the time of that evaluation? The same `Year ago` member we're defining. That's the current member in that moment! That would lead to empty result as the value of calculated members is `null` in advance. We must replace it and the only way to do that is to be explicit and specify another member of that dimension inside the tuple. The member that should be there is the default member of the utility dimension. That's our `As is` member, a single member of that dimension which doesn't modify the result of the current measure. Remember, the dimension is in no way related to the cube and that member is the only one without a calculation on it. Therefore, it preserves the results of the cube and keeps it unmodified, so that it can be used as such in other calculations.

By placing the default member of the utility dimension inside the tuple (actually, in all calculations so far), we're basically saying, "calculate this expression using the unmodified current measure whatever it may be." In the beginning it may seem a bit strange, but once you understand the concept of the unrelated utility dimension, it's easy to read expressions because this is a standard way of writing calculations for members of the utility dimension.

We're progressing. Let's see what comes next.

The `YTD Year ago` member has the same definition like the `Year ago` member except a small detail instead of the default `As is` member, the `YTD` member was used in the calculation. How come?

A moment ago we explained the purpose of the `As is` member and its behavior in calculations. We said that it's there to force the unchanged measure. This time, however, we'd like to be smart and apply the `YTD` calculation not on the value of the current member, but on the value of the member in the previous year which is exactly what the `Year ago` member calculates. It jumps to the previous year and grabs the corresponding value.

Wouldn't it be nice to combine the `YTD` and the `Year ago` calculations? Definitely, and it's not that difficult. All we have to do is make the composition of calculations. The result of the inner calculation (the value of the `Year ago` member) is passed as an argument to the outer one, the `YTD` calculation. That's one of the benefits of implementing the utility dimension.

Slowly, we come to the last phase. In this part, six more calculations are created. The first three calculate the difference between previously defined members of the utility dimension. The other three express that difference in form of a percentage. All 6 calculations are relatively simple and easy to comprehend just by taking a look at them. This is because again, the calculations refer to other members of the utility dimension. In other words, the calculations for some of the members in the utility dimension are rather complex, however, all subsequent calculations become relatively simple and easy.

Finally, a word or two about highlighting the negative results. When the value of `True` gets converted to a number, it becomes `-1`. The numeric value of `False` is `0`. The `Abs( )` function was used to correct the negative sign.

## There's more...

This section contains additional information regarding the concept presented in this recipe.

### Interesting details

It's worth mentioning that both user hierarchies have the year level clearly marked as the Year type. This way we had no problem using the `YTD()` function. If it weren't the case, the alternative would be to use the `PeriodsToDate()` function and to manually specify the year level in it.

Next, the Adventure Works cube has a date dimension that starts with July. Therefore the values of the `Year ago` calculations appear shifted by a half year. Fortunately, that happens only in the second year and only when user hierarchies are used. I believe your date dimension follows the best practice and has the complete year starting from January and ending with December. In that case you won't experience any problems with these calculations.

Another interesting thing is that neither of the calculations uses any measure in the assignments. That's a feature of the utility dimension and how we make assignments using it. Any time we need to reference the current measure, we reference the default member of the utility dimension instead. When applicable, we can also reference other members of that dimension, for example, the way we defined the calculation for the `YTD Year ago` member.

You may be wondering why none of the fiscal hierarchies were included in the calculation. That's because they are not compatible with calendar hierarchies. They break years into two. The proper way to include them would be to define a separate YTD member on the same utility dimension or, to make another utility dimension just for fiscal hierarchies. Naturally, that would allow using the same time calculations only modified to fit the fiscal hierarchies.

Here's another interesting detail about the YTD calculation. We don't have to know, imagine, or visualize the result of the intersection that happens inside the `Aggregate()` function (although it's possible). All that matters is that we provide a way for this intersection to happen and we did that by specifying all attribute hierarchies there. The trick with that expression is to form ranges that span from the first member of a particular hierarchy (represented using `null` in the code) up to the current member of each attribute hierarchy. That way each attribute hierarchy has the potential to determine the subcube, but on the other hand, every other hierarchy is slicing that subcube with its range of members. The result is the smallest common subcube, the one which fits all hierarchies in the `Aggregate()` function.

This concept repeats, with few modifications, in other calculations.

## Fine-tuning the calculations

It is possible to add a couple of scope statements to hide values on future dates, as explained in chapter 2, the recipe *Hiding calculations values on future dates*. For example, this is how one of those scope statements would look like:

```
Scope( { Tail( NonEmpty( [Date].[Date].[Date].MEMBERS,
                           [Measures].[Transaction Count] ),
               1 ).Item(0).NextMember : null } );
  This = null;
End Scope;
```

The other things we can do to improve the solution is to provide a scope that forces the default member of the utility dimension in case a multi-select is made in the slicer. Here's the code for that.

```
Scope( [Time Calcs].[Calc].MEMBERS );
  This = iif( Count( EXISTING [Time Calcs].[Calc].MEMBERS ) > 1,
              [Time Calcs].[Calc].&[0],
              [Time Calcs].[Calc].CurrentMember );
End Scope;
```

## Other approaches

The following link presents another implementation of the time-based utility dimension, the *DateTool* dimension created by Marco Russo:

<http://tinyurl.com/MarcoDateTool>

Where these approaches differ from each other is that the one presented in this book supports multiple hierarchies and doesn't use string operations. Because of that its code grew much bigger and became more complex. Actually, the code for both approaches looks difficult if you're not used to these types of MDX expressions, with lots of scopes and assignments in them. However, that shouldn't stop you from experimenting and applying them in your solutions.

The approaches also differ in the use of single or multiple hierarchies of the utility dimension. This is just a matter of taste; both techniques can be switched to the other style, with a little bit of coding of course.

## See also

The recipes *Setting special format for negative, zero and null values* and *Applying conditional formatting on calculations*, both in *Chapter 1*, show additional formatting options.

The recipes *Calculating the YTD (Year-To-Date) value*, *Calculating the YoY (Year-over-Year) growth (parallel periods)* and *Hiding calculations values on future dates*, all in chapter 2, show additional information about YTD and year ago calculations. Actually, the complete chapter may be useful because the the majority of the topics in that chapter are time calculations.

# 7

## Context-aware Calculations

In this chapter, we will cover:

- ▶ Identifying the number of columns and rows a query will return
- ▶ Identifying the axis with measures
- ▶ Identifying the axis without measures
- ▶ Adjusting the number of columns and rows for OWC and Excel
- ▶ Identifying the content of axes
- ▶ Calculating row numbers
- ▶ Calculating the bit-string for hierarchies on an axis
- ▶ Preserving empty rows
- ▶ Implementing utility dimension with context-aware calculations

### Introduction

Context-aware calculations are calculations relative to the result of an MDX query.

First, let's explain what the term "relative" means in this case.

Calculations can be relative to the set on an axis or a part of it. The rank on set, percentage in set, percentage of parent, average among siblings, cumulative sum, cumulative product, and so on are the examples of such calculations.

Calculations can also be relative to vertically or horizontally positioned cells. In the difference or variance calculations, the value of the current cell is compared either to a fixed cell coordinate or to the previous cell. This way, we can track the base and the incremental growth, respectively.

To add to the complexity, in all these situations the "context" can range from the totally unpredictable (whatever there is on rows and columns), partially known (a particular measure or a hierarchy is expected), to the completely known context (for exact measures and hierarchies).

How does one create these generic calculations, which not only know how to traverse up and down the result of the query, but also know to calculate correctly no matter which context they've been thrown at?

At the moment, it's better to postpone the answer to that question and tackle the other problem.

As stated earlier, these calculations should be relative to the result of an MDX query, but calculations **are** part of a query. In other words, they evaluate when the query evaluates. Is it reasonable to expect they should somehow know in advance the result of the query?

A more logical place for context-aware calculations is the application itself, the front-end being used to analyze the results. After all, the application controls the context and knows exactly what was returned as the result of the query (among other things). It can operate on the data returned and calculate whatever is necessary.

While that's certainly a way to go, it may be of little comfort to an end user for several reasons. First, not many applications incorporate these features. In those that do, the provided functionality might not be satisfactory in every situation. Showing values as percentages, ranks and similar? No problem. Handling more complex scenarios? Probably not.

Last but not least, these new calculations take place on a special layer, the presentation layer. The calculations are aware of the results of the query, but the query might not be aware of them, and it usually isn't. Let's see why this is important.

There are two types of pivot features in every SSAS front-end: those that generate a new query and those that don't. When an application feature modifies the data without issuing a new query, it can be said it has operated on the presentation layer. When a feature generates a new query, it generates it using the original data, not their representation in form of ranks, percentages, and similar. To keep a long story short, various pivot features can be applied to values in the presentation layer, but they will rarely work as expected. In other words, the end user will be surprised to get an empty pivot when he tries to filter on values where the **Sales Amount** was less than 10% of the parent's value. Effectively, that's like asking where the **Sales Amount** is less than 0.1, which very easily could be nowhere. The 10% is just the visual representation of an another amount which generated that percentage in a particular context. The query doesn't know that the 10% is not a constant but an expression from a layer unknown to the query, the presentation layer. This means the presentation layer is only half useful, offering no further analysis but on raw data.

By now you should have a better understanding of the reasons why context-aware calculations are such a complex topic. The problems have arisen wherever we turned.

As we concluded, the ideal place for them is the front-end. In addition to that, the front-end should also either prevent some features or push the calculations down to the query when required. Needless to say, implementing restrictions is never a popular method, which means some form of MDX calculations matching expressions used for modifying the visual information about the raw data should end up in the new query.

Building calculations? That's something you can do by yourself! Whenever you need context-aware calculations not available in your front-end, whenever you need them to work even after the query had been modified by a user action, or you need to perform an action based on the result of the context-aware calculation, you can implement special MDX calculations inside the cube or a particular query. That's what this chapter is about and you'll learn it in the series of recipes that follow.

The chapter starts simple, with a series of recipes to a particular problem. The initial recipes are not complete; they solve one problem and warn you about the next one which is solved in later sections or in the further recipes. After the base is built, we can apply the knowledge to generate typical reports. Finally, the chapter ends with a recipe that shows the best practice of implementing context-aware calculations.

I recommend reading the recipes in sequential order in order to grasp the topic in the best way possible.

## Identifying the number of columns and rows a query will return

Knowing the number of columns or rows the query will return can be handy. That way you can calculate the average value of a measure and compare individual values with it. You can also analyze the results of ABC analysis and see if the number of members in groups A, B, and C matches the expected percentage. In other words, what percentage of members contributes to 80% of the results, the expected 20%, or another percentage? These are just some of the situations where knowing the number of rows or columns is essential. How difficult is to get it when we need it?

If we're the one writing the MDX query, nothing stops us from defining a set with whatever was supposed to go on rows, count the members of that set using a calculated measure and finally putting that set on rows instead of the original members. That way we can know in advance the number of rows or in the opposite scenario, the number of columns. Sets are usually found on rows, that's why I used the "rows" example.

Other situations where counting the members does not present a problem are predefined named sets from the MDX script, or sets where the complete level of a hierarchy is used. Again, all it takes is to define a calculated measure that counts members of a sets, optionally wrapping the set with the `NonEmpty()` function to avoid empty rows or columns, as shown in the recipe *Optimizing MDX queries using NonEmpty() function in chapter 1*.

The common thing in all these situations is the fact that we knew what the set consists of in advance. In other words, getting the number of columns or rows is easy for **known** sets.

But what happens when the set is not known, when users exclude members from columns or rows, apply a function, or similar action? How do we count then?

This recipe shows a way.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to get the number of columns and rows of the query:

1. Define the # of columns calculated measure using the following definition:

```
Create MEMBER CurrentCube.[Measures].[# of columns] AS  
    iif( IsError( Axis(0).Count ), null,  
        Axis(0).Count )  
, Visible = 1  
, Display_Folder = 'Context-aware calcs';  
Define the # of rows calculated measure using the following  
definition:
```

```
Create MEMBER CurrentCube.[Measures].[# of rows] AS  
    iif( IsError( Axis(1).Count ), null,  
        Axis(1).Count )  
, Visible = 1  
, Display_Folder = 'Context-aware calcs';
```

2. Deploy.
3. Now, open the **SQL Server Management Studio** (SSMS) and connect to the SSAS server with **Adventure Works DW 2008R2** database.

4. Select the **Adventure Works** cube in that database and click on the **New Query** button. Then type the following MDX query and execute it:

```
SELECT
    { [Promotion].[Promotions].[Category].MEMBERS } *
    { [Measures].[# of columns],
      [Measures].[# of rows] }
    ON 0,
    { [Product].[Product Model Lines].[Product Line].MEMBERS }
    ON 1
FROM
    [Adventure Works]
```

5. The result of the query is shown in the following image:

	Customer	Customer	No Discount	No Discount	Reseller	Reseller
	# of columns	# of rows	# of columns	# of rows	# of columns	# of rows
Accessory	6	5	6	5	6	5
Components	6	5	6	5	6	5
Mountain	6	5	6	5	6	5
Road	6	5	6	5	6	5
Touring	6	5	6	5	6	5

### How it works...

The # of columns and # of rows calculations are relatively simple calculations.

The Axis() function returns a set on a particular axis of the query. The Count function counts members (or tuples, to be precise) of a set. The IsError() function detects errors, and therefore enables error handling. In case the required axis cannot be found in the query, we have the option to return null or something else instead of an error.

In short, the calculation returns the number of tuples on an axis after it is confirmed that the axis was indeed present in the query.

The Axis() function takes only one parameter - an integer that represents the 0-based index of an axis in a query.

The query cross-joins promotion categories with two measures we've just defined and puts the product model lines on the opposite axis.

There are 6 columns and 5 rows in the result, just as the measures tell us. In other words, calculations work.

There's more...

**SQL Server Management Studio** returned the result as expected. However, that will rarely be so in SSAS front-ends. For example, look what the **Cube Browser** in BIDS returns in the same case:

Drop Filter Fields Here									
Product Line	Category		No Discount		Reseller		Grand Total		# of rows
	# of columns	# of rows							
Accessory	4	6	4	6	4	6	4	6	
Components	4	6	4	6	4	6	4	6	
Mountain	4	6	4	6	4	6	4	6	
Road	4	6	4	6	4	6	4	6	
Touring	4	6	4	6	4	6	4	6	
Grand Total	4	6	4	6	4	6	4	6	

Cube Browser, also known as OWC, returned 8 columns, but our `# of columns` measure detected only 4 of them, as highlighted on the previous image. Additionally, there are 6 rows, not 5 this time, because of the **Grand Total** row. The `# of rows` measure detected this correctly.

What's the situation in other tools?

Row Labels	Column Labels		# of columns		# of rows		# of columns		# of rows	
	Customer		No Discount		Reseller		Total # of columns		Total # of rows	
Accessory			8	6	8	6	8	6	8	6
Components			8	6	8	6	8	6	8	6
Mountain			8	6	8	6	8	6	8	6
Road			8	6	8	6	8	6	8	6
Touring			8	6	8	6	8	6	8	6
Grand Total			8	6	8	6	8	6	8	6

Both Excel 2007 and Excel 2010 return the correct number of rows and columns for this particular query. But if we modify it a little, by removing one of the measures or one of the hierarchies, the result will be incorrect.

What's going on?

SSAS front-ends have MDX query generators. Every time end users put measures on the data area and hierarchies on rows, columns and the filter area, the front-end generates an MDX query. The front-end has absolute freedom how to generate the query as long as the result is correct. The problem occurs with the use of the `Axis()` function. That function doesn't tolerate what certain front-ends do – switching axes of an MDX query at will. In other words, the visual layout of the result doesn't have to correspond to axes of the query. What is on rows might have been retrieved on columns in the MDX query and vice versa. Not only that, but the result can be a composition of two queries. Measures can also move from an axis to another axis depending of the complexity of the query.

This represents a new challenge for us and is something we are going to deal with in the following recipes. But before we do, check the front-end you're using, maybe it behaves correctly. If so, you're lucky. This will significantly simplify the final solution with context-aware calculations in this chapter because you won't have to provide very complex calculations and scopes that detect and try to correct what's wrong in the queries.

## Visibility of the calculations

The **Visible** property specified in the calculations is a reminder for us to make the calculations invisible once we're sure they work as expected. Setting the property to **0** will make measures disappear in front-ends, but you'll still be able to use them in your calculations.

Just be aware that scope statements that apply to all measures don't apply to invisible calculated measures. See this link for more information about it:

<http://tinyurl.com/InvisibleCalcsAndScope>

If you want the scope to work on an invisible calculated measure, you should explicitly refer to that measure as part of the scope subcube, as explained in the previously mentioned blog article by Teo Lachev.

### See also

The recipes *Identifying the axis with measures*, *Identifying the axis without measures*, and *Adjusting the number of columns and rows for OWC and Excel* are a natural sequence to this recipe because they explain how to deal with problems we have encountered in this recipe.

## Identifying the axis with measures

As we saw in the previous recipe, *Identifying the number of columns and rows a query will return*, the relatively simple calculations we used don't always work. If the SSAS front-end being used changes the position of the measures in the query, we might have a problem detecting things. This recipe explains how to eliminate that problem. In other words, this recipe shows the magical calculation for detecting the axis with measures.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to identify the axis with measures:

1. Define the Axis with measures calculated measure using the following definition:

```
Create MEMBER CurrentCube.[Measures].[Axis with measures] AS
    case
        when NOT IsError(Extract( Axis(0), Measures ).Count) then 0
        when NOT IsError(Extract( Axis(1), Measures ).Count) then 1
        when NOT IsError(Extract( Axis(2), Measures ).Count) then 2
        else -1
    end
    , Visible = 1
    , Display_Folder = 'Context-aware calcs';
```

2. Deploy.

## How it works...

The `Axis()` function returns a set on a particular axis of the query. The `Extract()` function extracts a hierarchy from the multidimensional set. In this case, that was the **Measures** hierarchy. The `Count` function counts tuples of a set. The `IsError()` function detects errors and therefore enables error handling.

In short, the calculation returns the ordinal number of axes with the **Measures** hierarchy. Possible values are **0** for columns, **1** for rows, and **2** for pages. In case the hierarchy is not found, it can be assumed it's in slicer, either explicitly or implicitly. The value of **-1** is given in that situation.

Here are few images with the Axis with measures calculation in action.

The following image shows **2** because OWC puts measures on pages whenever there is something on rows and columns:

Drop Filter Fields Here				
	Category ▼			
	[+ Customer	[+ No Discount	[+ Reseller	Grand Total
Product Line ▼	Axis with measures	Axis with measures	Axis with measures	Axis with measures
[+ Accessory	2	2	2	2
[+ Components	2	2	2	2
[+ Mountain	2	2	2	2
[+ Road	2	2	2	2
[+ Touring	2	2	2	2
Grand Total	2	2	2	2

The following image shows **1** because OWC puts measures on rows whenever there is something on rows or columns, but not on both axes:

Drop Filter Fields Here	
Drop Column Fields Here	
Product Line ▾	Axis with measures
[+] Accessory	1
[+] Components	1
[+] Mountain	1
[+] Road	1
[+] Touring	1
Grand Total	1

Yes, you read well, measures were actually on rows, not columns, although from the image above it looks the opposite. Axes are simply switched in the presentation layer.

The following image shows **0** because OWC puts measures on columns whenever there is nothing else in the query:

Drop Filter Fields Here	
Drop Column Fields Here	
Axis with measures	
	0

Here's the special case of **-1**. It can be seen in Excel 2007 and Excel 2010. No matter what you put on rows and columns, if that's the only measure in the result, the measure is put in slicer.

Axis with measures		Column Labels ▾			
Row Labels		[+] Customer	[+] No Discount	[+] Reseller	Grand Total
[+] Accessory		-1	-1	-1	-1
[+] Components		-1	-1	-1	-1
[+] Mountain		-1	-1	-1	-1
[+] Road		-1	-1	-1	-1
[+] Touring		-1	-1	-1	-1
Grand Total		-1	-1	-1	-1

Feel free to experiment with your front-end, it might be interesting.

## There's more...

Here are some additional things you should know about this type of calculation.

### Visibility of the calculations

The `Visible` property is a reminder to make the calculation invisible once we're sure it works as expected. Setting the property to `0` will make the measure disappear in front-ends, but you'll still be able to use it in your calculations.

Everything said about invisible calculated measures and scopes in the previous recipe stands here too – you have to explicitly mention them inside the scope definition. See the *There's more* section of the recipe *Identifying the number of columns and rows a query will return*, for link to additional information.

### Additional axes

In this recipe we've limited ourselves to 3 axes only: columns, rows, and pages. That's because most of the front-ends use only 2 or 3 axes. In case your front-end uses more, extending the `case` statement appropriately should not be a problem.

## See also

The recipe *Identifying the axis without measures* shows just the opposite. For that reason, it is recommended that you read it after this recipe.

## Identifying the axis without measures

We learned how to detect the axis with measures, now let's see how to detect the axis without them. If you've read the previous recipe, *Identifying the axis with measures*, then you know what this is about. If you haven't, you're strongly advised to do so before reading any further.

You might be wondering why it is important to detect the axis without measures. Isn't it enough to know where the measures are?

No, unfortunately not. If your front-end switches axes and generates non-consistent MDX queries based on the number of measures or hierarchies in the report, then you must also know where the axis you want to analyze is. There are hierarchies on that axis. Knowing which one is it is crucial for applying context-aware calculations.

Let's see how to do it.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to identify the axis without measures:

1. Define the Non-measures axis calculated measure using the following definition:

```
Create MEMBER CurrentCube.[Measures].[Non-measures axis] AS
    iif(
        NOT IsError(Axis(1).Count) AND
        [Measures].[Axis with measures] <> 1,
        1,
        iif(
            [Measures].[Axis with measures] <> 0 AND
            NOT IsError(Axis(0).Count),
            0,
            null
        )
    )
, Visible = 1
, Display_Folder = 'Context-aware calcs';
```

2. Deploy.

## How it works...

Are there rows in the query? Are the measures there? If not, the result is **1**. This means rows is the axis without the measures.

Are there columns in the query? Are the measures there? If not, return **0**. In other words, the column axis is the axis without measures.

In case nothing was as expected, it will return null because the query doesn't have the axis without measures. Either no hierarchy was used or other hierarchies were cross-joined with measures on the same axis.

## Context-aware Calculations

---

Here are two images that show how this measure behaves in **Cube Browser**:

Drop Filter Fields Here				
	Category ▼			
	[+] Customer	[+] No Discount	[+] Reseller	Grand Total
Category ▼	Non-measures axis	Non-measures axis	Non-measures axis	Non-measures axis
[+] Accessories	1	1	1	1
[+] Bikes	1	1	1	1
[+] Clothing	1	1	1	1
[+] Components	1	1	1	1
Grand Total	1	1	1	1

The above image shows **1** because rows is the axis without measures. Actually, if you remember from the previous recipe, the measures are on pages axis in this case. Both the columns and rows axes are free of measures. However, since only one value can be returned, the preference was to use rows because of how OWC displays the result.

The image below shows **0**. OWC always tries to push the measures on the axis number with the highest ordinal and never cross-joins them with anything else. Therefore, product categories are on columns, measures on rows in this case. The result shows that measures are not on columns, although it looks like they are. That's why we need these calculations – one thing is seeing, the other is knowing.

Drop Filter Fields Here	
Drop Column Fields Here	
Category ▼	Non-measures axis
[+] Accessories	0
[+] Bikes	0
[+] Clothing	0
[+] Components	0
Grand Total	0

Feel free to experiment with your front-end.

There's more...

Here are some additional things you should know about this type of calculation.

### Visibility of the calculations

The **Visible** property is a reminder to make the calculation invisible once we're sure it works as expected. Setting the property to **0** will make the measure disappear in front-ends, but you'll still be able to use it in your calculations.

Everything said about invisible calculated measures and scopes in the first recipe goes here too – you have to explicitly mention them inside the scope definition. See the *There's more* section of the recipe *Identifying the number of columns and rows a query will return* for a link to additional information.

### Additional axes

In this recipe we've limited ourselves to 2 axes only: columns and rows. That's because most of the front-ends use only 2 or 3 axes. In case there's a need to extend that, you should detect the behavior of your front-end by tracking which MDX queries it generates in various situations. Once you get the pattern, you'll be able to modify the calculation.

### More information on the behavior of OWC

Here's the link to a short explanation how **Cube Browser** generates MDX queries:  
<http://tinyurl.com/MDXinOWC>. As this control is incorporated in BIDS, I think you should be aware of its specifics.

### See also

The recipe *Identifying the axis with measures* shows the opposite case.

The recipe *Capturing MDX queries generated by SSAS front-ends* in chapter 9 shows how to detect MDX queries generated by front-ends.

## Adjusting the number of columns and rows for OWC and Excel

We started the chapter with calculating the number of rows and columns a query returns. We saw that the correct numbers are not always easy to get. We stopped there and focused ourselves on detecting the axis with measures followed by the opposite case, the axis without them. Once we know how to do that, we can make adjustments to the initial calculation for number of columns and rows, which brings us to this recipe.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to adjust the number of columns and rows. Depending on the front-end you're using, use one of the calculations below. Remember, only one, not both of them.

1. To see the correct number of columns and rows in **Cube Browser**, put these calculations in your MDX script:

```
-- OWC
Scope([Measures].[# of columns]);
    This = iif( [Measures].[Axis with measures] = 1 OR
                [Measures].[Axis with measures] = -1,
                Axis(1).Count,
                [Measures].CurrentMember );
End Scope;

Scope([Measures].[# of rows]);
    This = iif( [Measures].[Axis with measures] = 1 OR
                [Measures].[Axis with measures] = -1,
                Axis(0).Count,
                [Measures].CurrentMember );
End Scope;
```

2. If you want the adjusted number of rows in Excel, put this calculation in your MDX script:

```
-- Excel
Scope([Measures].[# of rows]);
    This = iif( [Measures].[Axis with measures] = 0 AND
                IsEmpty([Measures].[Non-measures axis]),
                Axis(0).Count,
                iif( IsError( Axis(1).Count ), null,
                    Axis(1).Count ) );
End Scope;
```

3. Deploy.

## How it works...

The calculations for the number of columns and rows are already defined in the first recipe in this chapter using the correct syntax. In case the front-end displays something else in particular situations, we need to adjust that. The most natural way to implement adjustments in MDX is to use the **Scope** command.

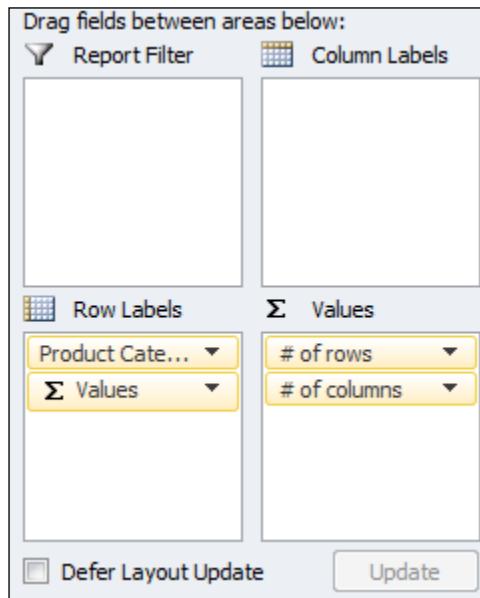
The basic idea is simple; we scope a measure and test for a situation that needs to be adjusted. In the case of **Cube Browser**, we need to switch the axis when OWC puts measures on rows, because they are never displayed there in the pivot.

For that, we've used the `Axis with measures` measure defined in the second recipe in this chapter. That measure, as its name says, detects the axis with measures and returns the ordinal position of the axis. In case of rows, that's **1**. Therefore, whenever the measures are on rows, we need to switch rows and columns in our calculation and count the number of tuples on the opposite axis.

In Excel, we applied the scope to rows. Whenever we cannot detect the axis without measures, they are either cross-joined with measures or not there at all. In case of the first, we need to adjust the calculation and count columns, not rows. Otherwise, we should stay on rows but not before we perform a test that they actually exist in the query.

### There's more...

There are situations in Excel when it is impossible to detect the position of hierarchies in the pivot. For example, if we have a cross-join of several measures and various hierarchies on rows and nothing on columns, the same query is generated as when we put all that on columns and leave nothing on rows. The reason is that in both cases the cross-join is placed on axis 0 of the query, the columns, making it impossible to detect which of these two combinations triggered the query. In other words, one combination will return the correct number of columns, the other will not. The following image shows the combination which will return the wrong result.



The situation is not perfect in **Cube Browser** either. Because of the way it generates queries using multiple MDX commands, the results might sometimes be out-of-sync on subtotals and **Grand Total**.

## Other SSAS front-ends

In case you're using another SSAS client, you should look at the scopes provided in this recipe and try to modify them to fit your scenario. You should detect the behavior of your front-end by tracking which MDX queries it generates in various situations. Once you understand the pattern, you'll be able to modify the calculation.

### See also

The recipes *Identifying the axis with measures*, *Identifying the axis without measures*, and *Identifying the number of columns and rows for OWC and Excel* are related to this recipe. If you haven't read them yet, I advise you to do that; you'll understand the topic presented in this recipe much better.

The recipe *Capturing MDX queries generated by SSAS front-ends* in chapter 9 book shows how to detect MDX queries generated by front-ends.

## Identifying the content of axes

Now that we've solved the axis problem, it's time to see the contents of axes on a particular axis. This recipe teaches you how to break down and reconstruct the content of an axis back the way it was. Here we're going to learn how to count the hierarchies on axes, how to identify them and their current members. We'll also learn how to form a tuple which will play an important role in the subsequent recipe where we're dealing with the row number, the crucial ingredient for vertical navigation. But first the easier one, horizontal navigation in form of discovering the content of an axis.

To make the recipe more compact, we're going to focus on one axis only – rows. Later sections of this recipe explain what needs to be done in order to have the same code for columns.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

### How to do it...

Follow these steps to find out various information about the content of an axis:

1. Define the calculated measure that detects the dimensionality of the set on rows using the following definition:

```
Create MEMBER CurrentCube.[Measures].[Dim rows] AS  
    iif( IsError( Axis(1).Count ), null,
```

```

        Axis(1).Item(0).Count )
, Visible = 1
, Display_Folder = 'Context-aware calcs';

```

2. In case your front-end switches axes, apply the correct scope command, otherwise skip this step. For example, here's what the scope should look like for a Cube Browser:

```

-- OWC
Scope([Measures].[Dim rows]);
This = iif( [Measures].[Axis with measures] = 1 OR
            [Measures].[Axis with measures] = -1,
            Axis(0).Item(0).Count,
            [Measures].CurrentMember );
End Scope;

```

3. Define the calculated measure that displays the name of the first hierarchy on rows using the following definition:

```

Create MEMBER CurrentCube.[Measures].[First H on rows] AS
    iif( IsError( Axis(1).Count ), null,
          Axis(1).Item(0).Item(0).Hierarchy.Name )
, Visible = 1
, Display_Folder = 'Context-aware calcs';

```

4. Define the calculated measure that displays the name of the last hierarchy on rows using the following definition:

```

Create MEMBER CurrentCube.[Measures].[Last H on rows] AS
    iif( IsError( Axis(1).Count ), null,
          Axis(1).Item(0).Item( Axis(1).Item(0).Count - 1 )
                  .Hierarchy.Name )
, Visible = 1
, Display_Folder = 'Context-aware calcs';

```

5. In case your front-end switches axes, apply the correct scope command. Otherwise, skip this step. For example, here's what the scope should look like for a Cube Browser:

```

-- OWC
Scope([Measures].[First H on rows]);
This = iif( [Measures].[Axis with measures] = 1 OR
            [Measures].[Axis with measures] = -1,
            Axis(0).Item(0).Item(0).Hierarchy.Name,
            [Measures].CurrentMember );
End Scope;

```

```

Scope([Measures].[Last H on rows]);
This = iif( [Measures].[Axis with measures] = 1 OR

```

```

[Measures].[Axis with measures] = -1,
Axis(0).Item(0).Item(
    Axis(0).Item(0).Count - 1 )
    .Hierarchy.Name,
[Measures].CurrentMember );
End Scope;

```

6. Define the calculated measure that displays the name of the current member of the first hierarchy on rows using the following definition:

```

Create MEMBER CurrentCube.[Measures].[Current member on First H on
rows] AS
    iif( IsError( Axis(1).Count ), null,
        Axis(1).Item(0).Item(0).Hierarchy.CurrentMember.Name )
    , Visible = 1
    , Display_Folder = 'Context-aware calcs';

```

7. Define the calculated measure that displays the name of the current member of the last hierarchy on rows using the following definition:

```

Create MEMBER CurrentCube.[Measures].[Current member on Last H on
rows] AS
    iif( IsError( Axis(1).Count ), null,
        Axis(1).Item(0).Item( Axis(1).Item(0).Count - 1 )
            .Hierarchy.CurrentMember.Name )
    , Visible = 1
    , Display_Folder = 'Context-aware calcs';

```

8. In case your front-end switches axes, apply the correct scope command. Otherwise, skip this step. For example, here's what the scope should look like for a Cube Browser:

```

-- OWC
Scope([Measures].[Current member on First H on rows]);
    This = iif( [Measures].[Axis with measures] = 1 OR
        [Measures].[Axis with measures] = -1,
        Axis(0).Item(0).Item(0)
            .Hierarchy.CurrentMember.Name,
        [Measures].CurrentMember );
End Scope;

Scope([Measures].[Current member on Last H on rows]);
    This = iif( [Measures].[Axis with measures] = 1 OR
        [Measures].[Axis with measures] = -1,
        Axis(0).Item(0).Item(
            Axis(0).Item(0).Count - 1 )
            .Hierarchy.CurrentMember.Name,
        [Measures].CurrentMember );
End Scope;

```

9. Define the calculated measure that generates the current tuple on rows using the following definition:

```
Create MEMBER CurrentCube.[Measures].[Current tuple on rows] AS
    iif( IsError( Axis(1).Count ), null,
        TupleToStr(
            StrToTuple("(" +
                Generate( Head( [Measures].ALLMEMBERS,
                    Axis(1).Item(0).Count ) AS RN,
                    "Axis(1).Item(0).Item(" +
                    CStr(RN.CurrentOrdinal - 1) +
                    ").Hierarchy.CurrentMember",
                    ",") +
                    ")"))
        )
    , Visible = 1
    , Display_Folder = 'Context-aware calcs';
```

10. In case your front-end switches axes, apply the correct scope command. Otherwise, skip this step. Here's what the scope should look like for a Cube Browser:

```
-- OWC
Scope([Measures].[Current tuple on rows]);
    This = iif( [Measures].[Axis with measures] = 1 OR
                [Measures].[Axis with measures] = -1,
                TupleToStr(
                    StrToTuple("(" +
                        Generate( Head( [Measures].ALLMEMBERS,
                            Axis(0).Item(0).Count ) AS RN,
                            "Axis(0).Item(0).Item(" +
                            CStr(RN.CurrentOrdinal - 1) +
                            ").Hierarchy.CurrentMember",
                            ",") +
                            ")"),
                    [Measures].CurrentMember );
    End Scope;
```

11. Deploy.

## How it works...

Dimensionality of rows is calculated as the count of hierarchies in the first tuple on rows. Since all the tuples on rows must be of the same dimensionality, we were allowed to pick any. Naturally, the easiest one to select was the first one. That's what the `Axis(1).Item(0)` expression represents.

The next two calculated measures were measures which extract the name of the first and the last hierarchy on rows. Again, it was easy to select the first hierarchy using this expression:  
`Axis(1).Item(0).Item(0)`

The last hierarchy was more difficult, but still possible to extract. The trick was to count the number of hierarchies, subtract 1 because of zero-based indexes, and then point to that hierarchy in the tuple using the `Item()` function.

The same code was used for extracting current members on those hierarchies, but this time it was the name of the member, not hierarchy, that we asked for.

Finally, the current tuple on rows was generated. The expression should be read inside-out. There we have a `Generate()` function, which, as we've learned in chapter 1, is used to generate a new set from the previous one. In this case, we've used the measures to perform the loop which repeats as many times as there are hierarchies on rows. The proper way would be to use a dummy dimension, but in order to preserve simplicity as much as possible we'll stick with what we have.

The purpose of the loop is to build a string using the static expression `"Axis(1).Item(0).Item( "` and the value of the loop counter. In other words, in the case of 3 hierarchies on rows, this string is being built:

```
Axis(1).Item(0).Item(0).Hierarchy.CurrentMember,  
Axis(1).Item(0).Item(1).Hierarchy.CurrentMember,  
Axis(1).Item(0).Item(2).Hierarchy.CurrentMember
```

That string is additionally wrapped in brackets, converted into a tuple, and then back into a string.

The idea of double conversion can be understood as taking a picture. First, we carefully build a string that can be converted to tuple and then when that gets evaluated, we convert it back to a string thereby capturing the content of the axis.

The tuple is constructed in a way that when it is evaluated, it captures current members of hierarchies on rows. Each hierarchy is referenced using its relative position on the axis. Relative positions were created in the string building process, using the counter in the loop.

The following image shows the result of these new calculated measures:

Drop Filter Fields Here		Drop Column Fields Here					
Color	Calendar Year	Dim rows	# of rows	First H on rows	Current member	Current tuple on rows	# of columns
[+] Black		2	61	Color	Black	((Product].[Color].&[Black].[Date].[Calendar].[All Periods])	8
[+] Blue		2	61	Color	Blue	((Product].[Color].&[Blue].[Date].[Calendar].[All Periods])	8
[+] Grey		2	61	Color	Grey	((Product].[Color].&[Grey].[Date].[Calendar].[All Periods])	8
[+] Multi		2	61	Color	Multi	((Product].[Color].&[Multi].[Date].[Calendar].[All Periods])	8
[+] NA		2	61	Color	NA	((Product].[Color].&[NA].[Date].[Calendar].[All Periods])	8
[+] Red	[+] CY 2005	2	61	Color	Red	((Product].[Color].&[Red].[Date].[Calendar].[Calendar Year].&[2005])	8
	[+] CY 2006	2	61	Color	Red	((Product].[Color].&[Red].[Date].[Calendar].[Calendar Year].&[2006])	8
	[+] CY 2007	2	61	Color	Red	((Product].[Color].&[Red].[Date].[Calendar].[Calendar Year].&[2007])	8
	[+] CY 2008	2	61	Color	Red	((Product].[Color].&[Red].[Date].[Calendar].[Calendar Year].&[2008])	8
	[+] CY 2010	2	61	Color	Red	((Product].[Color].&[Red].[Date].[Calendar].[Calendar Year].&[2010])	8
	Total	2	61	Color	Red	((Product].[Color].&[Red].[Date].[Calendar].[All Periods])	8
[+] Silver	[+] CY 2005	2	61	Color	Silver	((Product].[Color].&[Silver].[Date].[Calendar].[Calendar Year].&[2005])	8
	[+] CY 2006	2	61	Color	Silver	((Product].[Color].&[Silver].[Date].[Calendar].[Calendar Year].&[2006])	8
	[+] CY 2007	2	61	Color	Silver	((Product].[Color].&[Silver].[Date].[Calendar].[Calendar Year].&[2007])	8
	[+] CY 2008	2	61	Color	Silver	((Product].[Color].&[Silver].[Date].[Calendar].[Calendar Year].&[2008])	8
	[+] CY 2010	2	61	Color	Silver	((Product].[Color].&[Silver].[Date].[Calendar].[Calendar Year].&[2010])	8
	Total	2	61	Color	Silver	((Product].[Color].&[Silver].[Date].[Calendar].[All Periods])	8
[+] Silver/Black		2	61	Color	Silver/Black	((Product].[Color].&[Silver/Black].[Date].[Calendar].[All Periods])	8
[+] White		2	61	Color	White	((Product].[Color].&[White].[Date].[Calendar].[All Periods])	8
[+] Yellow		2	61	Color	Yellow	((Product].[Color].&[Yellow].[Date].[Calendar].[All Periods])	8
Grand Total		2	61	Color	All Products	((Product].[Color].[All Products].[Date].[Calendar].[All Periods])	8

## There's more...

A word or two about scopes. If you're going to use SSMS or a front-end where you have control over MDX being generated, then there's no need for scopes mentioned in the *How to do it* section. They were provided to show how calculations work in **Cube Browser**. In case of another front-end, you should analyze its MDX generation patterns and make the correct scope statement.

## What about the other axis?

In order to keep the recipe short as much as possible we've focused on one axis only. In case you need to perform the same calculations for columns, just replace `Axis(1)` with `Axis(0)` in all expressions and change the name of calculated measures so that it matches their new definition (columns become rows and vice versa). Remember to change the names in scope, too.

## Potential problems

The loop was performed using all measures, regular and calculated. The potential problem with this is that there might not be enough members to perform the loop. Hardly likely, but still possible. In order to prevent this from happening, create a separate dummy dimension.

## More info

Examples of context-aware calculations based on the content of axes can be found here:

<http://tinyurl.com/AverageAmongSiblings>

<http://tinyurl.com/GoldfishRank>

## See also

The recipe *Calculating row numbers* shows why it is useful to have the current tuple on rows.

The recipe *Implementing the Tally table utility dimension* in chapter 8 shows how to create a dummy dimension with a sequence of numbers to iterate on and hence have safe loops in your calculations.

## Calculating row numbers

Those who know the row numbers can navigate the results of the query. Those who don't can only jump to a certain position, such as the first tuple, the last tuple, and so on.

This recipe is based on the previous one. It continues using the current tuple on an axis in order to determine its position in the whole set on rows. Once we know the position, we can capture the previous or the next tuple on rows and make interesting relative calculations that work in any context.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to calculate the row number:

1. Define the calculated measure that calculates the row number using the following definition:

```
Create MEMBER CurrentCube.[Measures].[Row number] AS
    iif( IsError( Axis(1).Count ), null,
        Rank( StrToTuple( [Measures].[Current tuple on rows] ),
              Axis(1) )
    )
    , Visible = 1
    , Format_String = '#,##0'
    , Display_Folder = 'Context-aware calcs';
```

2. In case your front-end switches axes, apply the correct scope command. Otherwise, skip this step. For example, here's what the scope should look like for a Cube Browser:

```
-- OWC
Scope( [Measures].[Row number]);
```

```

This = iif( [Measures].[Axis with measures] = 1 OR
           [Measures].[Axis with measures] = -1,
           Rank( StrToTuple(
                   [Measures].[Current tuple on rows] ),
               Axis(0) ),
           [Measures].CurrentMember );
End Scope;

```

### 3. Deploy.

## How it works...

The calculation uses the measure defined in the previous recipe, which returns the current tuple on rows in form of a string. When that string is converted to a real tuple, in other words, when it is evaluated, the rank of that tuple in the set found on axis 1 is the row number. The initial check prevents errors to appear when axis 1 is not a part of the query, when the query contains only the columns axis or less.

In the case of **Cube Browser**, we need to adjust the calculation using the scope.

## There's more...

Here's the image that shows how the result looks like in **Cube Browser**:

Color	Drop Column Fields Here		
	Dim rows	# of rows	Row number
Black	1	11	2
Blue	1	11	3
Grey	1	11	4
Multi	1	11	5
NA	1	11	6
Red	1	11	7
Silver	1	11	8
Silver/Black	1	11	9
White	1	11	10
Yellow	1	11	11
<b>Grand Total</b>	1	11	1

Notice the first row is in fact the last row, the one that says **Grand Total**. All other rows increment naturally, but they start from number 2.

The reason for this is that the query contains a reference to the **All** member of that dimension, of any dimension to be precise, not just to the members of a required level. Moreover, that member is returned as the first member in the result. However, it is displayed last and therefore, the sequence looks awkward.

If you want, you can turn the subtotals down, there's a button for that. But first, you have to turn the toolbar on by right-clicking on the pivot, choosing the **Commands and Options...** function, navigating to the last tab in that window, and checking the **Toolbar** checkbox under the **Hide/Show elements**. Then click on the column you want to turn the subtotal on, here the Color column, and notice the active icon.



Make it inactive and the **Grand Total** goes away, but row numbers don't change. We need to make one more small change in the calculation – decrease the row number by one in case of OWC. Here's the improved scope for that:

```
Scope([Measures].[Row number]);
This = iif( [Measures].[Axis with measures] = 1 OR
           [Measures].[Axis with measures] = -1,
           Rank( StrToTuple(
                  [Measures].[Current tuple on rows] ),
                  Axis(0) ) - 1,
           [Measures].CurrentMember - 1 );
End Scope;
```

### Performance of the calculation

The performance of this calculation will not be great. This is because it uses the `Rank()` function and moreover, because the argument used in that function is a complex one. For anything bigger than 1000 rows, you'll experience delays in getting the result back.

### Related calculations

Once that you have a row number, you can move forward and backward in the result of the query. Simply take the row number and add or subtract 1 in order to get the next or previous row, respectively. That way you can perform calculations that track incremental changes in results no matter which members are on rows and also no matter what order.

Again, in the case of columns, the calculations are similar. All it takes is to replace the index in the `Axis()` function.

### Visibility of the calculations

The `Visible` property is a reminder to make the calculation invisible once we're sure it works as expected. Setting the property to `0` will make the measure disappear in front-ends, but you'll still be able to use it in your calculations.

Everything said about invisible calculated measures and scopes in the first recipe goes here too – you have to explicitly mention them inside the scope definition. See the *There's more* section of the recipe *Identifying the number of columns and rows a query will return* for a link to additional information.

## Other SSAS front-ends

In case you're using another SSAS client, you should look at the scopes provided in this recipe and try to modify them to fit your scenario. You should detect the behavior of your front-end by tracking which MDX queries it generates in various situations. Once you get the pattern, you'll be able to modify the calculation.

### See also

The recipe *Capturing MDX queries generated by SSAS front-ends* in chapter 9 shows how to detect MDX queries generated by front-ends.

## Calculating the bit-string for hierarchies on an axis

We've been up and down the result so far. We've traversed the result horizontally and identified the current tuple, its hierarchies, and current member of those hierarchies. We've also identified the row number which allows vertically-oriented relative calculations, for example, the increment from the previous row.

In this recipe we're going to focus a bit more on the horizontal part. Although we already know a lot about the current tuple and its content, there are situations where we can benefit from having a bit more. For example, if there are multidimensional sets on rows, it might be interesting to calculate the ratio of the rightmost hierarchy members against their immediate parent or their top ancestor. Percentages are another example.

Detection of the context can be a very exhausting operation in case of these multidimensional sets on an axis. In other words, knowing which hierarchy is currently not on its root member and which one is, which ratio, rank, or percentage formula to apply in respect to all the hierarchies there, and so on.

The trick is always in good preparation. If we build an indicator that returns an expression, we can decode that expression and apply the appropriate formula in our calculation. The indicator, of course, has to carry as little information as possible, but still the information which undoubtedly identifies the context. Needless to say, the format of the information must suit us perfectly so that we can use it with minimal effort.

Meet the bit-strings!

The bit-string is a string that carries information coded with symbols **0** and **1**. The length of the string should (in this case) be equal to the number of hierarchies on rows. Each symbol identifies whether the current member of that hierarchy meets our condition or not. In this scenario, when the current member is the A11 member of that hierarchy. If the answer is yes, we'd put **1** on that place. In the case of no, we'd leave the zero.

This recipe shows how to build that string indicator. The later sections of the recipe highlight some interesting examples. A bit-string indicator has no purpose by itself; it is always used as an intermediate tool for another calculation. It is the key ingredient in the final calculation and therefore interesting to be explained in a recipe dedicated exclusively to it.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to calculate the bit-string for hierarchies on rows:

1. Define the Bit-string calculated measure using the following definition:

```
Create Member CurrentCube.[Measures].[Bit-string] AS
    Generate(
        Head( [Measures].AllMembers,
              [Measures].[Dim rows] ) AS L,
        -( Axis(1).Item(0).Item( L.CurrentOrdinal - 1 )
            .Hierarchy.CurrentMember.Level.Ordinal = 0 )
        ),
        Visible = 1
        , Display_Folder = 'Context-aware calcs';
```

2. In case your front-end switches axes, apply the correct scope command. Otherwise, skip this step. For example, here's what the scope should look like for a **Cube Browser**:

```
-- OWC
Scope([Measures].[Bit-string]);
    This =
        iif( [Measures].[Axis with measures] = 1 OR
            [Measures].[Axis with measures] = -1,
        Generate(
            Head( [Measures].AllMembers,
                  [Measures].[Dim rows] ) AS L,
            -( Axis(0).Item(0).Item( L.CurrentOrdinal - 1 )
                .Hierarchy.CurrentMember.Level.Ordinal = 0 )
            ),
            [Measures].CurrentMember );
End Scope;
```

3. Define the Reversed Bit-string calculated measure using the following definition:

```
Create Member CurrentCube.[Measures].[Reversed Bit-string] AS
    Generate(
        Head( [Measures].AllMembers,
            [Measures].[Dim rows] ) AS L,
        -( Axis(1).Item(0).Item(
            [Measures].[Dim rows] - L.CurrentOrdinal )
            .Hierarchy.CurrentMember.Level.Ordinal = 0 )
        ),
        , Visible = 1
        , Display_Folder = 'Context-aware calcs';
```

4. In case your front-end switches axes, apply the correct scope command. Otherwise, skip this step. For example, here's what the scope should look like for a **Cube Browser**:

```
-- OWC
Scope([Measures].[Reversed Bit-string]);
This =
    iif( [Measures].[Axis with measures] = 1 OR
        [Measures].[Axis with measures] = -1,
    Generate(
        Head( Measures.AllMembers,
            [Measures].[Dim rows] ) AS L,
        -( Axis(0).Item(0).Item(
            [Measures].[Dim rows] -
            L.CurrentOrdinal )
            .Hierarchy.CurrentMember.Level.Ordinal = 0 )
        ),
        [Measures].CurrentMember );
End Scope;
```

5. Define the Rightmost 0 calculated measure using the following definition:

```
Create Member CurrentCube.[Measures].[Rightmost 0] AS
    InStr( [Measures].[Reversed Bit-string], '0' )
    , Visible = 1
    , Display_Folder = 'Context-aware calcs';
```

6. Define the Inner 1 calculated measure using the following definition:

```
Create Member CurrentCube.[Measures].[Inner 1] AS
    CInt( [Measures].[Bit-string] ) = 1
    , Visible = 1
    , Display_Folder = 'Context-aware calcs';
```

7. Deploy.

## How it works...

Since this is a relatively complex recipe, let's see the result of it first.

Three hierarchies were put on rows: **Sales Territories**, **Product Colors**, and **Calendar Years**. All measures defined in this recipe were placed on columns. The following image shows the result:

Drop Filter Fields Here			Drop Column Fields Here				
Group	Color	Calendar Year	Dim rows	Bit-string	Reversed Bit-string	Rightmost 0	Inner 1
Europe	[+] Black		3	001	100	2	True
	[+] Blue	[+] CY 2005	3	000	000	1	False
		[+] CY 2006	3	000	000	1	False
		[+] CY 2007	3	000	000	1	False
		[+] CY 2008	3	000	000	1	False
		[+] CY 2010	3	000	000	1	False
		Total	3	001	100	2	True
	[+] Grey		3	001	100	2	True
	[+] Multi		3	001	100	2	True
	[+] NA		3	001	100	2	True
	[+] Red		3	001	100	2	True
	[+] Silver		3	001	100	2	True
	[+] Silver/Black		3	001	100	2	True
	[+] White		3	001	100	2	True
	[+] Yellow		3	001	100	2	True
		Total	3	011	110	3	False
	[+] NA		3	011	110	3	False
	[+] North America		3	011	110	3	False
	[+] Pacific		3	011	110	3	False
	Grand Total		3	111	111	0	False

The first calculated measure called **Bit-string** is a measure that returns a string with zeros and ones. **1** represents the root member of the hierarchy on that position, **0** represents a non-root member.

The idea is to perform a loop that runs **N times** where N is the number of hierarchies on rows. In other words, the loop will repeat proportionally to the dimensionality of axis 1. Normally, we were supposed to use a dummy dimension for that, but we've simplified the solution and used measures instead. In case you have problems with that because there are not enough measures in your cube, use another dimension instead or build the dummy dimension mentioned earlier.

The measure **Dim rows** determines how many times the loop will repeat. That's the measure we defined in one of the earlier recipes, so if you don't remember what it is about, take a look at the recipe *Identifying the content of axes*.

The purpose of the loop is to navigate through hierarchies on rows. Each pass of the loop positions us on another hierarchy starting from the first from the left and ending with the last one on the right. The `CurrentOrdinal` function is a pointer that runs from 1. Therefore, we need to subtract 1 from it in order to match the zero-based indexes in `Item( )` function.

We test whether the current member of the hierarchy in focus is its root member. The test is performed using the ordinal number of the current member's level. The top level has an ordinal of 0.

The result of that test is a *Boolean* value. Therefore, we need to change the sign for the value of `True`. The minus in front of the expression serves that purpose.

Additionally, the scope for the Bit-string measure modifies the result in case the axes are switched by OWC.

The Reversed Bit-string measure is essentially the same as the Bit-string measure except for a small detail – we're collecting hierarchies from the right side. This was achieved by changing the expression for the pointer. In this case, the pointer is calculated as the difference between the total number of hierarchies on rows and the ordinal of the current hierarchy in the loop. Everything else is the same, including the logic of the subsequent scope expression.

You might be wondering why we need the reversed bit-string. The reason is simple although not obvious. Most of the times we will need to know the state of the hierarchies looking inside-out, from right, not left. For example, we need to know the rightmost hierarchy that is currently not on its root member because the calculations should be applied on that hierarchy. We need to know which one is it for every row of the result.

For that purpose, we have another measure called `Rightmost 0`. That measure detects the rightmost hierarchy not on its root member by searching for the first position of zero in the reversed string. The search is performed using the `InStr( )` function, one of the VBA functions that are optimized to perform well in MDX. That function looks from the left side of the string which is the technical reason why we needed the reversed bit-string. Actually, both versions of the bit-string, so that we can navigate/search from left and right.

The last measure in this recipe, the `Inner 1` measure, is a measure that detects the combination where only the rightmost hierarchy is on its root member. It's here to show you how to test for a particular combination in case you need to handle combinations individually. All you have to do in that case is convert the string or part of it into its numeric value and compare it with the value that represents the wanted combination. Alternatively, you can compare the strings that's easier for you.

### There's more...

Here are some additional things you should know about this type of calculation.

## Visibility of the calculations

The **Visible** property is a reminder to make the calculation invisible once we're sure it works as expected. Setting the property to 0 will make the measure disappear in front-ends, but you'll still be able to use it in your calculations.

Everything said about invisible calculated measures and scopes in the first recipe goes here too – you have to explicitly mention them inside the scope definition. See the *There's more* section of the recipe *Identifying the number of columns and rows a query will return* for link to additional information.

### Other SSAS front-ends

In case you're using another SSAS client, you should look at the scopes provided in this recipe and try to modify them to fit your scenario. You should detect the behavior of your front-end by tracking which MDX queries it generates in various situations. Once you get the pattern, you'll be able to modify the calculation.

### More info

An example of using the bit-string calculation can be found here:

<http://tinyurl.com/UniversalPercentage>

The recipe *Capturing MDX queries generated by SSAS front-ends* in chapter 9 shows how to detect MDX queries generated by front-ends.

## Preserving empty rows

In recipes so far we've used the context-aware calculated measures only. We never made a report using another measure, be it a cube or calculated measure. The idea was to show that these special measures, the context-aware measures, can be used independently of the cube structure. But what happens when we combine them in the report with other measures?

Other measures are sometimes empty. They cause empty rows or columns to appear. On the other hand, the measures we've been defining so far in this chapter are rarely null, that is, only when the required axis is missing or something similar. The point is, they are never null when other cube measures are null. In other words, these new measures return more cells than necessary.

This has two consequences. First, it slows down the response from the server already slowed down by the nature of these context-aware calculations (they cannot be executed in bulk mode which is another name for fast query execution). Second, it corrupts the result, introduces new rows or columns, and prevents the `NON EMPTY` clause to compact the result.

This recipe offers a solution to this problem. It shows how to detect rows that should be left empty and provides the calculation to keep them such. It will all be more understandable once we start, so let's go!

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** cube and go to the **Calculations** tab. Turn on the **Script View** and navigate to the end of the MDX script.

## How to do it...

Follow these steps to detect empty rows and preserve them:

1. Define the Row with data calculated measure using the following definition:

```
Create MEMBER CurrentCube.[Measures].[Row with data] As
    iif( IsEmpty( [Measures].[Order Count] ), null, 1 )
, Visible = 1
, Display_Folder = 'Context-aware calcs';
```

2. Deploy.
3. Put some hierarchies on rows, for example: **Sales Territories**, **Product Colors**, and **Calendar Years**.
4. Put the measure Order Count on columns and next to it, the new Row with data measure.

Turn the **Show Empty Cells** option so that we can see the effect in action. It can be found on the toolbar of the **Cube Browser**.



5. The following image shows the result of that query:

Drop Filter Fields Here		Drop Column Fields Here	
Group	Color	Calendar Year	Order Count Row with data
Europe	Black	CY 2005	3,002 1
	Blue	CY 2006	25 1
		CY 2007	613 1
		CY 2008	836 1
		CY 2010	
		Total	1,474 1
	Grey		
	Multi		1,533 1
	NA		4,902 1
	Red		1,684 1
	Silver		1,224 1
	Silver/Black		136 1
	White		184 1
	Yellow		1,637 1
	Total		8,504 1
NA			
North America			16,108 1
Pacific			6,843 1
Grand Total			31,455 1

## How it works...

The calculated measure is defined to be empty wherever the original measure is empty. Otherwise it should return 1. The combination of the `iif()` and `IsEmpty()` functions were used.

As seen in the above image, the calculation works as expected.

## There's more...

Here are some additional things you should know about this type of calculation.

### Multiple measures

When there is more than one measure in the report, use the appropriate Boolean logic to test whether there's at least one non-empty measure for a particular row or column. In case all measures return null, you should too.

Here's an example that tests two measures:

```
Create MEMBER CurrentCube.[Measures].[Row with data 2] As
    iif( IsEmpty( [Measures].[Internet Order Count] ) AND
        IsEmpty( [Measures].[Reseller Order Count] ),
        null, 1 )
    , Visible = 1
    , Display_Folder = 'Context-aware calcs' ;
```

As you can see, the `AND` operator was used. Only when both measures are null should the final calculation be null.

You can deploy this new calculated measure, reconnect, and then drag it in the previous report in **Cube Browser** together with measures **Internet Order Count** and **Reseller Order Count** to see how it reacts.

### When the measure is not known in advance

The example in this recipe worked because we knew which measure are going to be put in the pivot. If it weren't the case, we would need a dynamic expression which tests for measures that are currently in the query. In addition to that, we would have to accept all the context-aware measures from that expression. Finally, we would have to implement that in a calculated measure.

This scenario is very complicated because the calculations are supposed to be implemented on measures and there's a need to isolate context-aware measures from other measures. A much better way to do it is using a utility dimension. This way calculations are separated from measures, so expressions become much simpler. The last recipe in this chapter illustrates how to make such utility dimension with context-aware calculations on it.

## Implementing utility dimension with context-aware calculations

This is the last recipe in this chapter. It shows how to build a dimension, a utility dimension, to be precise, and use it to store context-aware calculations. Since we've already covered a lot of material about these type of calculations, let's get started right away, there's a lot of MDX calculations to explain here.

### Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads, double-click the **Adventure Works** data source.

### How to do it...

Follow these steps to implement utility dimension with context-aware calculations on it:

1. Define a new named query ContextAwareCalculations using the following definition:

```
SELECT 0 AS ID, 'Value' AS Name UNION ALL
SELECT 1, 'Row with data' UNION ALL
SELECT 2, 'Number of rows' UNION ALL
SELECT 3, 'Number of potential rows' UNION ALL
SELECT 4, 'Number of Hierarchies in rows' UNION ALL
SELECT 5, 'Current Member in 1st Hierarchy in rows' UNION ALL
SELECT 6, 'Current Member in last Hierarchy in rows' UNION ALL
SELECT 7, 'Current Tuple in rows' UNION ALL
SELECT 8, 'Row Number' UNION ALL
SELECT 9, 'Potential Row Number' UNION ALL
SELECT 10, 'Previous Tuple in rows' UNION ALL
SELECT 11, 'Next Tuple in rows' UNION ALL
SELECT 12, 'Previous Member in 1st Hierarchy in rows' UNION ALL
SELECT 13, 'Next Member in 1st Hierarchy in rows' UNION ALL
SELECT 14, 'Previous Member in last Hierarchy in rows' UNION ALL
SELECT 15, 'Next Member in last Hierarchy in rows' UNION ALL
SELECT 16, 'Rank in set for rows' UNION ALL
SELECT 17, 'Cumulative sum for rows' UNION ALL
SELECT 18, 'Cumulative percentage for rows' UNION ALL
SELECT 19, 'ABC result for rows' UNION ALL
SELECT 20, 'Cumulative product for rows' UNION ALL
SELECT 21, 'Normalized value for rows'
```

- 
2. Define a new dimension using the `ContextAwareCalculations` named query and name it `Calculation`. Use the `ID` columns for the **Key** and the `Name` column for the **Name**.
  3. Name the single attribute in that dimension `Calculation`. Set order by **Key**.
  4. Specify the default member: `[Calculation].[Calculation].&[0]`
  5. Turn the **IsAggregatable** property to **False**.
  6. Process the dimension and verify that everything went right by browsing it later.
  7. Double-click the **Adventure Works** cube.
  8. Add this new dimension there, but don't link it to any measure group.
  9. Go to the **Cube Structure** tab and select that dimension in the lower left pane.
  10. Find the property named **HierarchyUniqueNameStyle** in the **Properties** pane and change the value to **ExcludeDimensionName**.
  11. Deploy.
  12. Go to the **Calculations** tab and turn on the **Script View**. Navigate to the end of the MDX script and enter the script for utility dimension. For clarity, the script will be broken in steps from this point on, so that you can easily follow what happens.
  13. Scope the first calculated member of the utility dimension - `Row with data`. It should detect whether the current row has data or not:

```
Scope([Calculation].[Row with data]);
  This =
    iif(IsError(Axis(1).Count), null,
        iif( NOT IsError(Extract(Axis(1), [Calculation])),
            null,
            iif(IsEmpty([Calculation].[Value]), null, 1)));
  Format_String(This) = '#,#';
End Scope;
```

14. Scope the second and the third calculated members of the utility dimension (`Number of rows` and `Number of potential rows`, respectively). The first of them counts all rows and the latter one counts potential rows, that is, it includes empty rows as well:

```
Scope([Calculation].[Number of rows]);
  This =
    iif(IsEmpty([Calculation].[Row with data]), null,
        NonEmpty(Axis(1), [Calculation].[Value]).Count);
  Format_String(This) = '#,#';
End Scope;

Scope([Calculation].[Number of potential rows]);
  This =
    iif(IsEmpty([Calculation].[Row with data]), null,
```

---

```

        Axis(1).Count);
Format_String(This) = '#,#';
End Scope;

```

15. The Number of Hierarchies in rows is the next member that is scoped. It calculates the dimensionality of the set on rows:

```

Scope([Calculation].[Number of Hierarchies in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Axis(1).Item(0).Count);
Format_String(This) = '#,#';
End Scope;

```

16. The next two members to be scoped detect the current member in the first and the last hierarchy on rows:

```

Scope([Calculation].[Current Member in 1st Hierarchy in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Axis(1).Item(0).Hierarchy.CurrentMember.Name);
-- or .UniqueName instead of .Name
End Scope;

```

```

Scope([Calculation].[Current Member in last Hierarchy in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Axis(1).Item(0).Item( Axis(1).Item(0).Count - 1 )
        .Hierarchy.CurrentMember.Name);
-- or .UniqueName instead of .Name
End Scope;

```

17. The Current tuple in rows is the next member to be scoped. It returns the current tuple on rows, as its name says:

```

Scope([Calculation].[Current Tuple in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    TupleToStr(StrToTuple("(" +
Generate( Head(Axis(1), Axis(1).Item(0).Count) AS RN,
                    "Axis(1).Item(0).Item(" +
                    CStr(RN.CurrentOrdinal - 1) +
                    ").Hierarchy.CurrentMember",
                    ",") +
                    ")"));
End Scope;

```

18. The Row Number and the Potential Row Number members count rows. The difference among them is that Row Number counts only empty rows:

```
Scope([Calculation].[Row Number]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Rank(StrToTuple( [Calculation].[Current Tuple in rows] ),
        NonEmpty(Axis(1), [Calculation].[Value]) ));
Format_String(This) = '#,#';
End Scope;

Scope([Calculation].[Potential Row Number]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Rank(StrToTuple( [Calculation].[Current Tuple in rows] ),
        Axis(1) ));
Format_String(This) = '#,#';
End Scope;
```

19. Previous Tuple in rows and Next Tuple in rows are calculations that look vertically and take the tuples from previous or next row, respectively:

```
Scope([Calculation].[Previous Tuple in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    iif([Calculation].[Row Number] = 1, null,
        TupleToStr(NonEmpty(Axis(1), [Calculation].[Value])
            .Item([Calculation].[Row Number] - 1 - 1)));
End Scope;

Scope([Calculation].[Next Tuple in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    iif([Calculation].[Row Number] =
        [Calculation].[Number of rows], null,
        TupleToStr(NonEmpty(Axis(1), [Calculation].[Value])
            .Item([Calculation].[Row Number] - 1 + 1)));
End Scope;
```

20. The next four calculations show how to perform both horizontal and vertical navigation on results of the query in respect to the current position:

```
Scope([Calculation].[Previous Member in 1st Hierarchy in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    NonEmpty(Axis(1), [Calculation].[Value])
        .Item([Calculation].[Row Number] - 1 - 1).Item(0).Name);
```

```

-- or .UniqueName instead of .Name
End Scope;

Scope([Calculation].[Next Member in 1st Hierarchy in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    NonEmpty(Axis(1), [Calculation].[Value])
    .Item([Calculation].[Row Number] - 1 + 1).Item(0).Name);
-- or .UniqueName instead of .Name
End Scope;

Scope([Calculation].[Previous Member in last Hierarchy in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    NonEmpty(Axis(1), [Calculation].[Value])
    .Item([Calculation].[Row Number] - 1 - 1)
    .Item(Axis(1).Item(0).Count - 1).Name);
-- or .UniqueName instead of .Name
End Scope;

Scope([Calculation].[Next Member in last Hierarchy in rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    NonEmpty(Axis(1), [Calculation].[Value])
    .Item([Calculation].[Row Number] - 1 + 1)
    .Item(Axis(1).Item(0).Count - 1).Name);
-- or .UniqueName
End Scope;

```

21. The next member is Rank in set for rows. It calculates the rank of the current tuple against everything on rows the complete set.

```

Scope([Calculation].[Rank in set for rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Rank(StrToTuple([Calculation].[Current Tuple in rows]),
        NonEmpty(Axis(1), [Calculation].[Value]),
        [Calculation].[Value]));
Format_String(This) = '#,#';
End Scope;

```

22. Cumulative sum for rows and Cumulative percentage for rows are similar members. The first one summarizes all the rows up to the current row. The second one shows the percentage of that cumulative sum against the total:

```
Scope([Calculation].[Cumulative sum for rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    Sum(Head(NonEmpty(Axis(1), [Calculation].[Value]),
        [Calculation].[Row Number]),
    [Calculation].[Value]));
End Scope;

Scope([Calculation].[Cumulative percentage for rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    [Calculation].[Cumulative sum for rows] /
    Sum(NonEmpty(Axis(1), [Calculation].[Value]),
        [Calculation].[Value]));
Format_String(This) = 'Percent';
End Scope;
```

23. ABC result for rows is a way of calculating the indicator for each row:

```
Scope([Calculation].[ABC result for rows]);
This =
iif(IsEmpty([Calculation].[Row with data]), null,
    case when [Calculation].[Rank in set for rows] <=
        TopPercent(NonEmpty(Axis(1),
            [Calculation].[Value]),
            30,
            [Calculation].[Value]
        ).Count
        then 'A'
    when [Calculation].[Rank in set for rows] <=
        TopPercent(NonEmpty(Axis(1),
            [Calculation].[Value]),
            80,
            [Calculation].[Value]
        ).Count
        then 'B'
    else 'C'
    end);
End Scope;
```

24. Cumulative product for rows uses recursion to calculate the running product:

```
Scope([Calculation].[Cumulative product for rows]);
  This =
    iif(IsEmpty([Calculation].[Row with data]), null,
        iif([Calculation].[Row Number] <= 1,
            [Calculation].[Value],
            ( [Calculation].[Cumulative product for rows],
              StrToTuple([Calculation].[Previous Tuple in rows]) ) *
              [Calculation].[Value] ));
End Scope;
```

25. Finally, Normalized value for rows divides the individual row's result with the best result on rows:

```
Scope([Calculation].[Normalized value for rows]);
  This =
    iif(IsEmpty([Calculation].[Row with data]), null,
        [Calculation].[Value] /
        Max( NonEmpty(Axis(1), [Calculation].[Value]),
             [Calculation].[Value] ) );
  Format_String(This) = '#,##0.00000';
End Scope;
```

26. Great, you're done with providing definition for members of the utility dimension.  
There should be 21 scope statements.

27. Deploy the script.

28. Put some hierarchies on rows, for example: **Calendar Years**.

29. Put the utility dimension on rows.

30. Observe the results. Notice that the utility dimension respects empty rows; you shouldn't be able to see **CY 2010** unless you turn the option to see empty rows.

## How it works...

The process of creating a utility dimension should already be familiar to you if you've read previous chapters. The idea is to have a set of members on a dimension not related to any measure group. That way they are supposed to show the original value of any measure. We'll leave only the default member, the first in that dimension as is and scope all others in MDX script later, providing them various calculation tasks.

The recipe shows how to implement more than 20 various calculations. You can of course implement additional calculations in the same manner – by creating the required members in the named query and by providing appropriate calculations for them later in MDX script.

The first member in that script is the `Row with data` member. It detects whether the current row has data or not by testing for the existence of rows in the first place. Then it checks that the utility dimension is not on rows, and finally performs the test on emptiness for every measure found in the query by referring to the default member of the utility dimension, the only one which doesn't change its value in calculations.

All subsequent calculations refer to this member and return a value only if the row is not empty.

The next two members count rows. The first of them counts all rows, the latter one counts potential rows, that is, it includes empty rows as well. The `NonEmpty()` function is applied over the default member of the utility dimension.

`Number of Hierarchies in rows` is the member that calculates the dimensionality of the set on rows. It calculates that similarly to the `Dim rows` member we had earlier in this chapter.

The next two members have calculations similar to members we had earlier in this chapter. They detect current members on the first and the last hierarchy on rows by referring to the name of those members. Optionally, you can change that and get their unique names if you need it in subsequent calculations. The comments show how.

`Current Tuple in rows` and `Row Number` are the type of calculations we had in previous recipes, so there should be no mystery in it. The only difference is that the `Row Number` counts rows with data only. That is achieved by applying the `NonEmpty()` function over the set on rows. The returned rank has therefore consecutive numbers. In contrast to that, the `Potential Row Number` calculation simply uses the `Axis(1)` expression for a set to determine the rank. Here, the row numbers will have holes in their sequence. Actually, members like this one are here only to show you what happens if you don't filter empty rows in your calculations.

`Previous Tuple in rows` and `Next Tuple in rows` are calculations we haven't had before. We've just mentioned the possibility of having them.

Their definition should be read from inside out. The `NonEmpty()` function applied on rows serves the purpose of eliminating empty rows. The `Item()` function is used to pick a particular tuple on rows. Increasing or decreasing the current row number by 1 means we're referring to the previous and next row, respectively. The `-1` in both calculations adjusts the row number to a zero-based index used in the `Item()` function.

The vertical stoppers are implemented by comparing the current row number with 1 and with the total number of rows.

The next four members are there to show you how you can navigate in both the hierarchies and their members on rows. The idea is the same as with tuples except that here we're also adding the `.Item().Name` part in the end of the calculation. The set returned by the `Axis()` function consists of tuples and they consist of members. The first `Item()` function extracts the tuple; in other words, it enables vertical movement. The second `Item()` function extracts members from that tuple. In other words, it enables horizontal movement inside the tuple. The `Row Number` measure is used as a variable position from where we're either going up or down. The first hierarchy has ordinal of 0, the last hierarchy's ordinal can be computed using the total number of hierarchies on rows minus 1 because of the zero-based index.

The next measure is `Rank in set for rows`. It calculates the rank of the current tuple against everything on rows, the complete set. The rank is, of course, calculated against non-empty rows only for which the `NonEmpty()` function took care. It's worth noticing that the extended form of `Rank()` function was used, the one that has the third argument.

`Cumulative sum for rows` and `Cumulative percentage for rows` are similar measures. The first one summarizes all the rows up to the current row. This was achieved using the `Head()` function and the `Row Number` measure. Again, only non-empty rows were collected because the `Row Number` is referring to non-empty rows. Had we failed in collecting the non-empty rows only, we would have wrong result.

The other measure uses the `Cumulative sum for rows` measure and divides it with the total on rows. That way we got the increasing percentage which hits 100% in the last row.

`ABC result for rows` is a way of calculating the indicator for each row. The A, B, or C group is calculated based on current row's percentage. Those rows that make up to the top 30% of results are assigned letter "A", those who are not in that group but made in the top 80% group are assigned letter "B" and finally, all the rest are group "C".

`Cumulative product for rows` uses recursion and `Previous tuple in rows` as the input parameter to recursion.

`Normalized value for rows` divides the individual row's result with the best result on rows. The best result is obtained using the `Max()` function over non-empty results on rows. Naturally, since the values are lower than 1, an appropriate decimal format was used; otherwise we wouldn't see anything but the row with the best score.

### There's more...

Here are some additional things you should know about this recipe and context-aware calculations in general.

#### Performance of the calculation

The performance of some calculations will not be great. This is because the expressions in them are so complex that they evaluate slowly.

Test the behavior of the utility dimension on your server before you apply it in your reports in order to understand limitations in the number of rows it can serve without generating annoying delays.

## Visibility of the calculations

Some of the calculations can be made invisible by setting the property `Visible` to 0. That will make the measure disappear in front-ends, but you'll still be able to use it in your calculations.

Everything said about invisible calculated measures and scopes in the first recipe goes here too – you have to explicitly mention them inside the scope definition. See the *There's more* section of the recipe *Identifying the number of columns and rows a query will return* for link to additional information.

## In case of problems

In case you're getting wrong or empty results in your SSAS client, you should consider either adding new scopes or extending the current ones in order for them to support things like switching axis, moving measures on axes, and similar tricks front-ends do.

You can detect the behavior of your front-end by tracing which MDX queries it generates in various situations. Once you get the pattern, you'll be able to modify the calculation.

## More info

Additional examples of context-aware calculation can be found on my blog:

<http://tomislav.piasevoli.com>

Look for topics about universal calculations, dynamic calculations, and similar.

The recipe *Capturing MDX queries generated by SSAS front-ends* in chapter 9 shows how to detect MDX queries generated by front-ends.

# 8

## Advanced MDX Topics

In this chapter, we will cover:

- ▶ Displaying members without children (leaves)
- ▶ Displaying members with data in parent-child hierarchies
- ▶ Implementing the Tally table utility dimension
- ▶ Displaying random values
- ▶ Displaying random sample of hierarchy members
- ▶ Displaying a sample from random hierarchy
- ▶ Performing complex sorts
- ▶ Using recursion to calculate cumulative values

### Introduction

This chapter contains some of the more advanced MDX topics.

Here we'll learn how to isolate and display members without children, how to isolate members with data in parent-child hierarchies, how to display their individual values, and other useful information regarding non-balanced hierarchies. Non-balanced hierarchies are hierarchies that appear to have holes inside their dimensional structure which is manifested in members missing from the bottom up.

Next, we're going to learn how to make the Tally table dimension. This multipurpose utility dimension will be used in the next part of the chapter, a part where we'll be dealing with randomness. We're going to show how to display random numbers, but also how to work with random samples or a sample from a random hierarchy. The first of those can be used to extract a portion of members from their hierarchy, a portion which is random in nature. Second is the case of a single report that always shows different information and as such, can be used for monitoring purposes.

We'll close the chapter with two complex recipes. The first one explains how to perform multiple sorts or how to use several conditions in order to sort members of a hierarchy. The second features recursion. In it we'll see how to construct calculations that traverse horizontally - calculations that enable navigation inside sets and thereby cumulate the values.

As the title states, the chapter is filled with complex topics. Before you start, make sure you've familiarized yourself with some of the earlier chapters of this book. You'll benefit from it.

## Displaying members without children (leaves)

Parent-child hierarchies, also known as unbalanced or recursive hierarchies, are hierarchies with flexible-level depth. Unlike regular user hierarchies, the depth in a parent-child hierarchy is not determined in advance. It is a direct consequence of recursive relation inside the dimension table. Members, irrespective of the level they are at, are not required to have descendants. The Adventure Works database has several such dimensions: the **Employee** dimension, the **Account** dimension, and the **Organization** dimension.

Ragged hierarchies are another type of non-balanced hierarchies. Their depth is fixed, known in advance, but some of the members on intermediate levels are hidden. The final effect is identical to parent-child hierarchies with one difference – members really do exist, they are just not made visible.

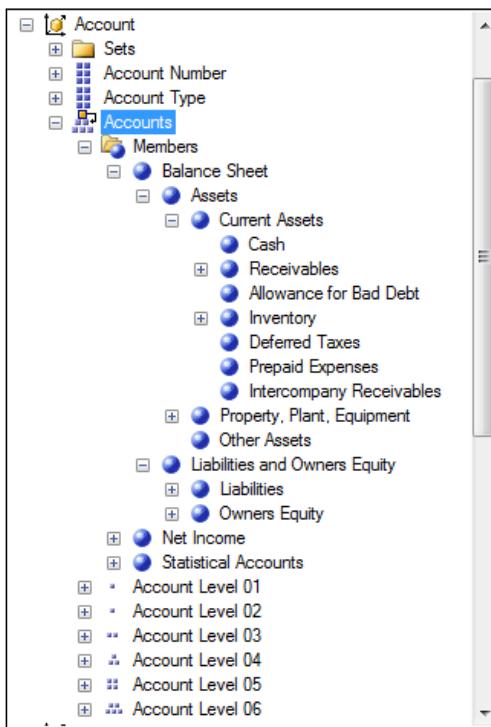
Several requests occur in relation to the non-balanced hierarchies. Showing members without children is one of them, particularly in parent-child hierarchies such as **Account** with values recorded on the leaf level. If values can appear on non-leaf members too, then it might be interesting to show members with data, as for example, it is done in the **Employee** and **Organization** hierarchies. Finally, it is possible to "show" hidden members in ragged hierarchies if the circumstances require so.

The bottom line is this - we need to provide a complex expression. Specifying a single level name followed by **.MEMBERS** won't do this time. First, we need to collect members all over the hierarchy because the ones we want might appear on different levels. Second, we need to flatten the hierarchy somehow. Curious enough to learn how? Read on.

### Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Expand the **Accounts** hierarchy of the **Account** dimension in the cube structure tree. Notice how members form navigation paths of different lengths. That's a parent-child hierarchy:



Now, let's see how to display only members without children.

### How to do it...

Follow these steps to show parent-child hierarchy members that don't have any children underneath them:

1. Write the following query:

```

SELECT
{ [Measures].[Amount] } ON 0,
{ { [Account].[Account Type].&[Assets],
    [Account].[Account Type].&[Liabilities] } *
[Account].[Account Number].[Account Number].MEMBERS *
Descendants( [Account].[Accounts]
                .[Account Level 01].MEMBERS, ,
                LEAVES )
} ON 1
FROM
[Adventure Works]

```

2. Execute it. The result should match the following image:

			Amount
Assets	1110	Cash	\$3,236,799.00
Assets	1130	Trade Receivables	\$3,371,580.00
Assets	1140	Other Receivables	\$104,343.00
Assets	1150	Allowance for Bad Debt	\$67,429.00
Assets	1162	Raw Materials	\$2,007,586.00
Assets	1164	Work in Process	\$1,393,582.00
Assets	1166	Finished Goods	\$742,230.00
Assets	1170	Deferred Taxes	\$505,424.00
Assets	1180	Prepaid Expenses	\$341,992.00
Assets	1185	Intercompany Receivables	\$674,663.00
Assets	1210	Land & Improvements	\$93,843.00
Assets	1220	Buildings & Improvements	\$299,043.00
Assets	1230	Machinery & Equipment	\$92,606.00
Assets	1240	Office Furniture & Equipment	\$523,016.00
Assets	1250	Leasehold Improvements	\$80,759.00
Assets	1260	Construction In Progress	\$33,127.00
Assets	1300	Other Assets	\$172,709.00
Liabilities	2210	Notes Payable	\$148,316.00
Liabilities	2220	Current Installments of Long-term Debt	\$149,571.00
Liabilities	2230	Accounts Payable	\$1,286,664.00
Liabilities	2310	Salary & Other Comp	\$459,471.00

3. Compare it with the cube structure tree. Notice that none of the members with "+" is in the result. The query works.

### How it works...

As explained in the introduction, the idea is to navigate the hierarchy one way or the other and to collect members that match certain criteria. The solution in this case was to use the `Descendants( )` function in combination with the `LEAVES` flag as its third argument. Let's analyze this in more detail.

The `Descendants( )` function can either take a set or a member as its first argument. That argument determines the part of the hierarchy from which we'll descend. For the complete hierarchy, it's natural to look for and use the `All` member of that hierarchy. However, the **Account.Accounts** hierarchy doesn't have that member (it's not there in the **SQL Server Management Studio**) because it was disabled for that hierarchy. Therefore, we've had to use the highest available level of that hierarchy to achieve the same effect.

Both the second and the third argument of the `Descendants()` function are optional. As you probably noticed, we didn't use the second one, only the third one for which we've specified the `LEAVES` flag, which instructs SSAS to return leaf members, that is, members without children.

The second argument specifies the level which, in combination with the first argument, determines members to be returned by this function. For example, when the first argument is the **Category** level members of the **Product Categories** user hierarchy, the second the **Subcategory** level, and the third the **SELF\_AND\_BEFORE** flag, the `Descendants()` function returns all members between (and including) the **Category** and the **Subcategory** levels. More information and examples about the `Descendants()` function can be found on this MSDN site:

<http://tinyurl.com/MDXDescendants>

For the reasons explained above, it made no sense to use the second argument because we needed all members without children, no matter what level they are at. Therefore, we omitted that argument which, in the case of the `LEAVES` flag, SSAS interprets as all possible levels. In other words, the `Descendants()` function from our example instructed SSAS to return all leaf members starting from (and including) the top level of that hierarchy.

Additional hierarchies on rows are there to make the result more comprehensible and to show you the possibilities for enhancing the result. They are not required to display members without children. They merely provide interesting information, enable grouping and sorting of leaf members, and therefore were included in this detailed balance sheet example.

### There's more...

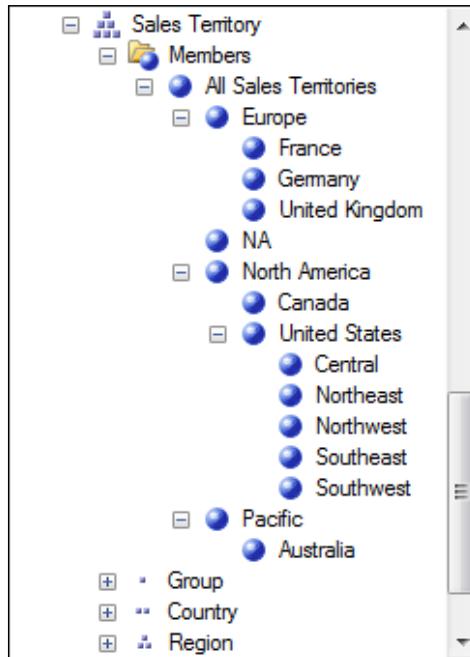
The approach works with ragged hierarchies too. In the following query, the solution from the previous example is applied on the ragged **Sales Territory** user hierarchy, only this time the **All** member is used. We've used it instead of the top level to show that it works in this case too:

```
SELECT
    { [Measures].[Sales Amount] } ON 0,
    { Descendants( [Sales Territory].[Sales Territory]
                    .[All Sales Territories],
                    , LEAVES )
    } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns this result:

	Sales Amount
France	\$7,251,555.65
Germany	\$4,878,300.38
United Kingdom	\$7,670,721.04
NA	(null)
Canada	\$16,355,770.46
Central	\$7,909,009.01
Northeast	\$6,939,374.48
Northwest	\$16,084,942.55
Southeast	\$7,879,655.07
Southwest	\$24,184,609.60
Australia	\$10,655,335.96

If you compare this with the structure of the hierarchy in the cube tree, you'll see that the query collected the exact members as required, those that form the so-called **leaf** of the hierarchy:



What happens when optional arguments are not used?

When the second argument is not used, it gets interpreted as the level of the member in the first argument of the `Descendants()` function. In case the first argument was a set, the level of each individual member in the set is used in turn for the evaluation of that function. The only exception to this rule happens when the `LEAVES` flag is used. Providing no level means you want all levels below the level of the member or set in the first argument.

When the third argument is not used, the default `SELF` flag is used.

### A reverse case

If you need to display all non-leaf members, simply put a minus sign in front of the `Descendants()` function. This gets interpreted as the opposite set in respect to the set of all members. What gets returned is the set of non-leaf members, those without children.

Bear in mind that additional hierarchies, as those used in this recipe, will make the result look confusing in the reverse case scenario. Try and you'll see what I mean by that. Therefore, consider removing them from the query.

### Possible problems with ragged hierarchies

Ragged hierarchies, contrary to parent-child hierarchies and balanced hierarchies, might not display properly in some front-end applications depending on how their underlying dimension tables were built. The comparison of ways to build ragged hierarchies, together with their advantages and disadvantages, is explained in this article written by Chris Webb:

<http://tinyurl.com/ChrisRaggedHierarchies>

Helpful information for working with ragged hierarchies can be found on MSDN too:

<http://tinyurl.com/MSDNRaggedHierarchies>

### See also

The recipe *Displaying members with data in parent-child hierarchies* shows other options for flattening non-balanced hierarchies.

## Displaying members with data in parent-child hierarchies

In the previous recipe we saw how to display members without children. The idea was to separate members with direct, fact table values from those whose value is a result of an aggregation. Quite a common thing if you're working with assignments and allocations in order to have correct calculations.

However, some business scenarios require direct values on non-leaf members. For example, a manager of salespeople generates sales himself too. Of course, this feature is supported in the parent-child hierarchy model built into SQL Server Analysis Services. The key properties to be used in this scenario are the **MembersWithData** and **MembersWithDataCaption** properties.

Separating the leaves is not good enough in the case of parent-child hierarchies with non-leaf members with data. In addition to leaves, we need to include all non-leaf members with direct data. This recipe shows how to do this.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

All parent-child hierarchies in the Adventure Works cube that have non-leaf members with data have those members hidden. That's why nothing special about those hierarchies can be seen in the cube structure tree. Anyway, we're going to use the **Employees** hierarchy of the **Employee** dimension and apply a small trick to show you that some of the members do indeed have their own data, although not visible at first. Here's what we'll do.

Run this query:

```
WITH
MEMBER [Measures].[Level Ordinal] AS
    [Employee].[Employees].CurrentMember.Level.Ordinal
MEMBER [Measures].[Subtotal OK] AS
    iif( IsLeaf( [Employee].[Employees].CurrentMember ), null,
        Sum( [Employee].[Employees].CurrentMember.CHILDREN,
            [Measures].[Sales Amount Quota] ) =
            [Measures].[Sales Amount Quota] )
SELECT
    { [Measures].[Level Ordinal],
        [Measures].[Sales Amount Quota],
        [Measures].[Subtotal OK] } ON 0,
    { [Employee].[Employees].MEMBERS } ON 1
FROM
    [Adventure Works]
```

The query returns all employees and their sales quota. Two additional calculated measures are here to inform us about the level of a particular member and whether the number shown next to each member matches the sum of its children. As seen in the following image, members on the third level have different values from the sum of their descendants. The reason is that they have their own values not visible in the result. That's why the amounts don't match.

	Level	Ordinal	Sales Amount Quota	Subtotal OK
All Employees	0		\$114,253,550.00	True
Ken J. Sánchez	1		\$114,253,550.00	True
Brian S. Welcker	2		\$114,253,550.00	True
Amy E. Alberts	3		\$24,202,000.00	False
Jae B. Pak	4		\$12,547,100.00	(null)
Rachel B. Valdez	4		\$3,269,800.00	(null)
Ranjit R. Varkey Chudukatil	4		\$7,260,700.00	(null)
Stephen Y. Jiang	3		\$87,336,050.00	False
David R. Campbell	4		\$4,876,850.00	(null)
Garrett R. Vargas	4		\$5,049,450.00	(null)
Jillian Carson	4		\$13,778,850.00	(null)
José Edvaldo. Saraiva	4		\$8,516,850.00	(null)
Linda C. Mitchell	4		\$13,844,750.00	(null)
Michael G. Blythe	4		\$13,288,250.00	(null)
Pamela O. Anzman-Wolfe	4		\$3,981,650.00	(null)
Shu K. Ito	4		\$9,066,250.00	(null)
Tete A. Mensa-Annan	4		\$3,449,600.00	(null)
Tsvi Michael. Reiter	4		\$9,823,500.00	(null)
Syed E. Abbas	3		\$2,715,500.00	False
Lynn N. Tsolfias	4		\$2,332,300.00	(null)

Now let's see how to make those missing values visible.

### How to do it...

Follow these steps to show parent-child hierarchy members with direct data:

1. Delete the last calculated measure, the Subtotal OK measure. Also remove it from columns axis.
2. Add the Data Member Quota calculated measure using this expression and apply the Currency format string to it:

```
MEMBER [Measures].[Data Member Quota] AS
    iif( [Measures].[Level Ordinal] = 0, null,
        ( [Employee].[Employees].DataMember,
          [Measures].[Sales Amount Quota] )
    )
, FORMAT_STRING = 'Currency'
```

3. Add that measure on the columns axis.
  4. Apply the `NonEmpty()` function over the set on rows using the newly created calculated measure **Data Member Quota**:
- ```
{ NonEmpty( [Employee].[Employees].MEMBERS,
            [Measures].[Data Member Quota] ) } ON 1
```
5. Execute the query and then notice two things about the result returned. First, only members with data are returned. Second, the **Data Member Quota** measure displays the individual value of members even for non-leaf members (see the highlighted rows in the following image):

|                           | Level Ordinal | Sales Amount Quota | Data Member Quota |
|---------------------------|---------------|--------------------|-------------------|
| Amy E. Alberts            | 3             | \$24,202,000.00    | \$1,124,400.00    |
| Jae B. Pak                | 4             | \$12,547,100.00    | \$12,547,100.00   |
| Rachel B. Valdez          | 4             | \$3,269,800.00     | \$3,269,800.00    |
| Ranjit R. Varkey Chudu... | 4             | \$7,260,700.00     | \$7,260,700.00    |
| Stephen Y. Jiang          | 3             | \$87,336,050.00    | \$1,660,050.00    |
| David R. Campbell         | 4             | \$4,876,850.00     | \$4,876,850.00    |
| Garrett R. Vargas         | 4             | \$5,049,450.00     | \$5,049,450.00    |
| Jillian Carson            | 4             | \$13,778,850.00    | \$13,778,850.00   |
| José Edvaldo. Saraiva     | 4             | \$8,516,850.00     | \$8,516,850.00    |
| Linda C. Mitchell         | 4             | \$13,844,750.00    | \$13,844,750.00   |
| Michael G. Blythe         | 4             | \$13,288,250.00    | \$13,288,250.00   |
| Pamela O. Ansman-Wolfe    | 4             | \$3,981,650.00     | \$3,981,650.00    |
| Shu K. Ito                | 4             | \$9,066,250.00     | \$9,066,250.00    |
| Tete A. Mensa-Annan       | 4             | \$3,449,600.00     | \$3,449,600.00    |
| Tsvi Michael. Reiter      | 4             | \$9,823,500.00     | \$9,823,500.00    |
| Syed E. Abbas             | 3             | \$2,715,500.00     | \$383,200.00      |
| Lynn N. Tsolfias          | 4             | \$2,332,300.00     | \$2,332,300.00    |

### How it works...

The `.DataMember` function is the key to this solution. It can be used either to separate members with data from those with only an aggregation, or to get the direct value of members with data, combined with a measure (as in this case). In other words, we formed a tuple which played both roles – it isolated data members on rows and it displayed their individual values in the calculated measure we've created for this purpose.

The only extra thing we had to take care of was to eliminate the `All` member, otherwise it would be present in the result too.

## There's more...

If you only want to see the non-leaf members with data, simply use the `IsLeaf()` function from the initial query as an additional condition for the Data Member Quota expression:

```
MEMBER [Measures].[Data Member Quota] AS
    iif( [Measures].[Level Ordinal] = 0
        OR IsLeaf( [Employee].[Employees].CurrentMember ),
        null,
        ( [Employee].[Employees].DataMember,
          [Measures].[Sales Amount Quota] )
    )
, FORMAT_STRING = 'Currency'
```

That will remove all leaves from the result, leaving only 3 rows highlighted in the previous image. Convenient, if you only need to display non-visible data or members having it.

### Alternative solution

If circumstances allow, you can go to BIDS and change the **MemberWithData** property of your parent-child hierarchy to **NonLeafDataVisible**. You should also use the **MembersWithDataCaption** property to make the distinction between members and their data part. For example, you can use this: "\* - data (without quotes).

After deploying the project and running the query again, you'll see additional members whose values will match the difference seen previously. You will also be able to notice those additional members in the hierarchy displayed in the cube structure pane on your left in SSMS.

## See also

The recipe *Displaying members without children (leaves)* shows other options for flattening non-balanced hierarchies.

## Implementing the Tally table utility dimension

The Tally table, also known as the Number table, is a single-column table filled with natural numbers, numbers starting with 1 and ending with a number equal to the number of rows in that table. The Tally table can also start with 0 if the circumstances require so.

Having a collection of numbers, often up to a very large number, is just one aspect of the Tally table. The other, more important aspect is that the numbers form a sequence. Having a sequence of numbers in a table enables set-based operations because such a table can be used instead of much slower iterations using loops and navigation using ranks. That's the reason why many relational databases incorporate the Tally table.

SQL Server Analysis databases can also profit from the utility dimension built from the Tally table. First, this utility dimension can enable iteration when required and as many times as required. You see, the MDX language enables iteration on sets only. If we need to perform an operation 100 times, we have to use a set containing that many members. In other words, we have to use one of the dimensions (attributes to be precise) in the cube. Now, there's a problem with that. The attribute might not have enough members, but what's worse, the attribute will mess with the current context if we use it for the loop. True, we could force it to a value outside of the loop, but getting that value might be a problem. Instead, having a separate, unrelated utility dimension to iterate on is a much more convenient solution.

Next, this type of utility dimension can be used as an X-axis in various charts where we need to put measures on a Y-axis, histograms in particular.

Finally, the sequence of numbers transformed into members of a dimension enables advanced navigation on other dimensions as you're about to see in this chapter. But first, let's see how to implement the Tally table utility dimension.

## Getting ready

Open the **Business Intelligence Development Studio** (BIDS) and then open the **Adventure Works DW 2008** solution. Once it loads double-click the **Adventure Works** data source view.

In this example, we're going to create a utility dimension using a Tally table with 1 million rows. That can be changed, of course, in your projects. Simply use any number you want to limit the table.

## How to do it...

Follow these steps to create a Tally table utility dimension.

1. Create a new named query **Tally Table** with the following definition:

```
SELECT  
    TOP 1000000  
    ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS n  
FROM  
    master.dbo.SysColumns t1,  
    master.dbo.SysColumns t2
```

2. Turn off the **Show/Hide Diagram Pane** and **Show/Hide Grid Pane** and execute the previous query in order to test it.

 An error might pop up when you run this type of query or when you open the named query to change it. The error basically says that this type of query cannot be represented visually using the diagram pane because it contains the OVER SQL clause. That's the reason why we've turned that pane off. If for whatever reason you see that error anyway, just acknowledge it and continue with the recipe. It is just an information, not a problem.

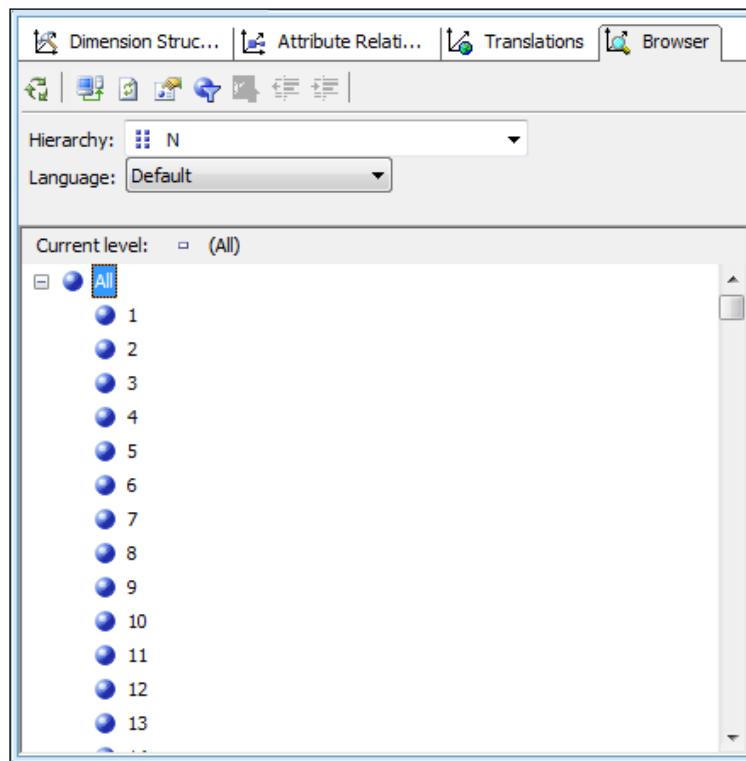
3. Once executed, the query starts returning rows with numbers starting with 1 and continuing in the sequence of 1. Stop it at some point; this was just a test to check that it works.
4. Mark the **n** column as a key column.
5. Create a new dimension using the previously defined named query and name it **Tally Table**.
6. Use the **n** column for both the **KeyColumns** and the **ValueColumn** properties:

|                          |                        |
|--------------------------|------------------------|
| Source                   |                        |
| CustomRollupColumn       | (none)                 |
| CustomRollupPropertiesCo | (none)                 |
| KeyColumns               | Tally Table.n (BigInt) |
| NameColumn               | (none)                 |
| ValueColumn              | Tally Table.n (BigInt) |

7. Optionally, mark the **Type** property of that dimension and its attribute as **Utility**. This might help show this dimension appropriately in some client tools:

|             |             |
|-------------|-------------|
| Basic       |             |
| Description |             |
| ID          | Tally Table |
| Name        | Tally Table |
| Type        | Utility     |

8. Process that dimension completely. Then verify that it looks correctly in the **Browser** tab, as displayed in the following image:



9. Open the **Adventure Works** cube and add that dimension without linking it to any measure group.
10. In the **Cube Structure** tab, select the **Tally Table** dimension and change the **HierarchyUniqueNameStyle** property to **ExcludeDimensionName**.
11. In the same pane, change the **Visible** property to **False**.
12. Deploy.

### How it works...

The query we used generates one million rows using cross joins of two significantly large system tables. The `ROW_NUMBER` operator is what creates incremental values. In the `ORDER BY` part, no reference was made to underlying tables in order to speed up the query and to get exactly what was needed – an incremental sequence of numbers.

The process of creating a utility dimension should be straightforward to you if you have experience building dimensions. The important part is to use the `MemberValue` property because we're going to reference that in the upcoming recipes. Besides, it's good practice to always put something meaningful in that property so that we can use it in calculations later, be it a utility dimension or not.

The utility dimension is usually not connected to any measure group. We can also simplify calculations by shortening the unique name of its members. Finally, depending on the circumstances, we either make the utility dimension visible or not. Here we've made it hidden because it is a very large dimension, there's no reason to expose it to end-users.

Basically, there's nothing to see as the result of this recipe. If everything went well, you're ready for some of the following recipes where you'll learn how to make use of this hidden dimension.

### There's more...

There are many ways to use the Tally table. We can create a real table in the database and populate it with numbers. We can also create a table-valued function and use it later. Finally, we can just execute the `SELECT` statement without the need to create the table. The following links provide good examples for building the Tally table:

- ▶ Jeff Moden: <http://tinyurl.com/JeffTallyTable>
- ▶ Michael Coles: <http://tinyurl.com/MichaelTallyTable>
- ▶ Itzik Ben-Gan: <http://tinyurl.com/ItzikTallyTable>
- ▶ Stack Overflow:
  - <http://tinyurl.com/SOTallyTable1>
  - <http://tinyurl.com/SOTallyTable2>

While the last (`SELECT` only) option is the least desirable in the relational database world, in multidimensional SSAS databases it shouldn't cause a performance problem because the result will eventually become a dimension, a materialized structure. Moreover, it will be ordered (by key, which is the default option). That's why we used a named query approach in this recipe. In case you want to have everything in the DW, feel free to create a table, cluster index, select the appropriate view that limits the result up to a particular number of rows, and then build a dimension from that view. In fact, this is how you can build multiple Tally table dimensions when required. You might need several small utility dimensions and one big Tally table utility dimension. The maintenance is easier if it all originates from the single table in DW.

## See also

The recipe *Using a dummy dimension to implement histograms over non-existing hierarchies* in Chapter 6 illustrates a solution to a problem using a small Tally table starting with 0 and ending with a particular number.

## Displaying random values

Random values are fun, but there's something more to them. They are great for learning how SSAS engine evaluates a query and its parts. Even if you're not interested in getting random values, you'll still profit from reading this recipe by learning important concepts in MDX that you can apply in other situations.

This recipe shows how to get random values in queries that operate on cubes. Subsequent recipes show how to use those random values to sample hierarchy members or generate interesting reports.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example, we're going to use the **Product** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Sales Amount] } ON 0,
    { [Product].[Product Model Lines].[Model].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns 120 models. Most of the models have data associated with them; only a few models have no sales.

## How to do it...

Follow these steps to display random values in rows:

1. Add the **WITH** part of the query.
2. Create three calculated measures, **Min Value**, **Max Value** and **Value Range**, which will be used as constants describing random value range:

```
MEMBER [Measures].[Min Value] AS
    10
MEMBER [Measures].[Max Value] AS
    99
MEMBER [Measures].[Value Range] AS
```

```
[Measures].[Max Value] -  
[Measures].[Min Value]
```

3. Create a calculated measure named Const Value using the following definition:

```
MEMBER [Measures].[Const Value] AS  
    iif( IsEmpty( [Measures].[Sales Amount] ), null,  
        Int( [Measures].[Min Value] +  
            [Measures].[Value Range] * Rnd( ) ) )
```

4. Create a calculated measure named Random number per row using the following definition:

```
MEMBER [Measures].[Random number per row] AS  
    Rnd( Rank( [Product].[Product Model Lines].CurrentMember,  
                [Product].[Product Model Lines].CurrentMember  
                .Level.MEMBERS ) )
```

5. Create a calculated measure named Random Value using the following definition:

```
MEMBER [Measures].[Random Value] AS  
    iif( IsEmpty( [Measures].[Sales Amount] ), null,  
        Int( [Measures].[Min Value] +  
            [Measures].[Value Range] *  
            [Measures].[Random number per row] ) )
```

6. Include the Const Value and the Random Value measures in the query and execute it.

7. If the result of your query **doesn't** match the following image, you're good. Remember, we were supposed to get **different** results because of random values:

The screenshot shows a table with four columns: Sales Amount, Const Value, and Random Value. The Sales Amount column lists various product names with their corresponding sales amounts. The Const Value column contains the value 52 for all rows except 'Assembly Components' and 'Chain', which have null values. The Random Value column contains random integers between 14 and 95 for each row.

|                             | Sales Amount | Const Value | Random Value |
|-----------------------------|--------------|-------------|--------------|
| Bike Wash                   | \$18,406.97  | 52          | 89           |
| Cable Lock                  | \$16,225.22  | 52          | 16           |
| Classic Vest                | \$259,488.37 | 52          | 54           |
| Cycling Cap                 | \$51,229.45  | 52          | 12           |
| Half-Finger Gloves          | \$113,948.30 | 52          | 35           |
| Hitch Rack - 4-Bike         | \$237,096.16 | 52          | 24           |
| Hydration Pack              | \$105,826.42 | 52          | 11           |
| Long-Sleeve Logo Jersey     | \$431,061.10 | 52          | 54           |
| Men's Bib-Shorts            | \$166,739.71 | 52          | 70           |
| Men's Sports Shorts         | \$81,898.62  | 52          | 70           |
| Minipump                    | \$13,514.69  | 52          | 86           |
| Patch kit                   | \$8,232.60   | 52          | 65           |
| Short-Sleeve Classic Jersey | \$321,198.29 | 52          | 64           |
| Sport-100                   | \$484,048.53 | 52          | 29           |
| Water Bottle                | \$28,654.16  | 52          | 18           |
| Women's Tights              | \$201,833.01 | 52          | 95           |
| Assembly Components         | (null)       | (null)      | (null)       |
| Chain                       | \$9,377.71   | 52          | 14           |

8. Execute the same query a few more times:

|                             | Sales Amount | Const Value | Random Value |
|-----------------------------|--------------|-------------|--------------|
| Bike Wash                   | \$18,406.97  | 25          | 30           |
| Cable Lock                  | \$16,225.22  | 25          | 39           |
| Classic Vest                | \$259,488.37 | 25          | 62           |
| Cycling Cap                 | \$51,229.45  | 25          | 47           |
| Half-Finger Gloves          | \$113,948.30 | 25          | 14           |
| Hitch Rack - 4-Bike         | \$237,096.16 | 25          | 37           |
| Hydration Pack              | \$105,826.42 | 25          | 71           |
| Long-Sleeve Logo Jersey     | \$431,061.10 | 25          | 59           |
| Men's Bib-Shorts            | \$166,739.71 | 25          | 13           |
| Men's Sports Shorts         | \$81,898.62  | 25          | 83           |
| Minipump                    | \$13,514.69  | 25          | 11           |
| Patch kit                   | \$8,232.60   | 25          | 25           |
| Short-Sleeve Classic Jersey | \$321,198.29 | 25          | 54           |
| Sport-100                   | \$484,048.53 | 25          | 16           |
| Water Bottle                | \$28,654.16  | 25          | 43           |
| Women's Tights              | \$201,833.01 | 25          | 30           |
| Assembly Components         | (null)       | (null)      | (null)       |
| Chain                       | \$9,377.71   | 25          | 35           |

9. Notice the Const Value measure changes, though only per query.  
 10. Notice the Random Value measure changes per row in every query.  
 11. The explanation follows.

### How it works...

The Rnd( ) function, a VBA function, generates random values between 0 and 1. The random value is naturally multiplied in order to make the range bigger. For example, in this recipe, the measure Value Range was used as the multiplier. Additionally, the range was offset not to start with zero using the first calculated measure, the Min Value measure.

The boundaries for the random values can be set either using measures as constants for the minimum and maximum values, or directly in the formula for the random value. We used the first approach to show how it can be done. Using the latter should not be a problem, just replace those measures later in the query with their values.

The Const Value calculated measure was used to illustrate an important principle – random values are random per query, not per row, unless we do something about it. The next two calculated measures show what we must do in order to get random values per row.

In the `Random number per row` measure, a seed, was used in the `Rnd()` function. That seed has one characteristic – it is unique for each row. Having a unique seed in the `Rnd()` function generates a unique random value in each row. That's the solution to the problem of repeating random values in the `Const Value` measure. The `Rank()` function applied over the current member of the set on rows generated unique seeds which in turn generated non-repeating random values. This was noticeable in both screenshots.

Why did we have to take that extra step? Why didn't we get the random values the way we wanted in the first attempt, in the `Const Value` measure, just by referring to the `Rnd()` function?

The SQL Server Analysis Services engine, just like the SQL Server database engine, tries to optimize the queries by evaluating expressions and determining which ones behave like a constant and which don't. A reference to the current member of a hierarchy on an axis is definitely a sign of a non-constant expression.

Non-constant expressions can be more or less complex. The engine will evaluate them in order to see if it can calculate the expression in *bulk* mode, a mode where everything is calculated in one go. That's much faster than *cell-by-cell* mode, a mode where every cell is calculated individually, in one big loop.

If the engine determines it can perform a bulk evaluation, it will do so. That's what happened in the `Const Value` calculated measure. Remember, nothing was there in that expression which explicitly requested row-by-row random values. All that we requested was a random value and we got it, different in each execution of the query. But that's not what we needed.

If we want random values per row, we have to explicitly say so. We have to insert part of the expression that will turn this calculation into a slow calculation, a *cell-by-cell* calculation. Yes, it will slow down the execution of the query, but that's exactly what we need in this case. Sometimes we need to pull the brakes and this is one of those cases. The power of OLAP is not only in its speed, it is also in knowing how to use the brakes when required. Hopefully this recipe showed how.

### There's more...

You might have noticed the use of `IsEmpty()` function inside the definition of `Const Value` and the `Random Value` measures. This is used to preserve empty rows. The idea, already explained in previous chapters, is to keep the effect of the `NON EMPTY` keyword in the query. In short, if the original value was empty, so shall be the calculated measure.

### See also

The recipe, *Displaying a random sample of hierarchy members* shows how to make use of the random values once we know how to generate them.

## Displaying a random sample of hierarchy members

In the previous recipe you learned how to generate random values per row. This recipe teaches you how to take it one step further and use those random values to sample the data.

### Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to sample the **Product** dimension, the same one used in the previous recipe. To remind you, this was the query we started from:

```
SELECT
    { [Measures].[Sales Amount] } ON 0,
    { [Product].[Product Model Lines].[Model].MEMBERS } ON 1
FROM
    [Adventure Works]
```

It returned 120 models. Now let's see how to get a sample from that pool.

### How to do it...

Follow these steps to display a random sample of hierarchy members:

1. Add the **WITH** part of the query.
2. Create the **H on axis** calculated set using the **set on rows**:

```
SET [H on axis] AS
    { [Product].[Product Model Lines].[Model].MEMBERS }
```

3. Create a calculated measure named **Cardinality** that will count members in the previously defined calculated set:

```
MEMBER [Measures].[Cardinality] AS
    [H on axis].Count
```

4. Create a calculated measure named Random number per row using the same definition as in the previous recipe:

```
MEMBER [Measures].[Random number per row] AS  
    Rnd( Rank( [Product].[Product Model Lines].CurrentMember,  
        [Product].[Product Model Lines].CurrentMember  
        .Level.MEMBERS ) )
```

5. Create a calculated measure named Random Value using the same definition as in the previous recipe:

```
MEMBER [Measures].[Random Value] AS  
    Iif( IsEmpty( [Measures].[Sales Amount] ), null,  
        Int( [Measures].[Min Value] +  
            [Measures].[Value Range] *  
            [Measures].[Random number per row] ) )
```

6. Create a calculated measure named Percentage that will determine the percentage of members from the previously defined calculated set to be returned by the query. Make it 10 for now, change later if you'd like to see another percentage.

```
MEMBER [Measures].[Percentage] AS  
    10
```

7. Create the H on axis sample calculated set using this definition:

```
SET [H on axis sample] AS  
    TopCount( [H on axis],  
        [Percentage] / 100 * [Cardinality],  
        [Random value] )
```

8. Include the Random Value measure in the query. Include the H on axis sample calculated set on rows instead of the existing set on rows.

```
SELECT  
    { [Measures].[Sales Amount],  
        [Measures].[Random Value] } ON 0,  
    NON EMPTY  
    { [H on axis sample] } ON 1  
FROM  
    [Adventure Works]
```

9. Execute the query. If the result of your query **doesn't** match the following image, you're good. Remember, we were supposed to get **different** results because of random values. What should match though is the number of rows. Both of us should get 12 rows:

|                       | Sales Amount   | Random Value |
|-----------------------|----------------|--------------|
| ML Crankset           | \$10,464.79    | 118          |
| ML Mountain Tire      | \$34,818.39    | 116          |
| HL Road Frame         | \$1,344,209.60 | 116          |
| LL Mountain Frame     | \$521,864.42   | 115          |
| ML Mountain Frame     | \$343,785.29   | 114          |
| Cable Lock            | \$16,225.22    | 112          |
| ML Mountain Frame-W   | \$482,953.16   | 112          |
| Touring-3000          | \$2,447,377.26 | 112          |
| Fender Set - Mountain | \$46,619.58    | 111          |
| Mountain Bike Socks   | \$6,573.39     | 108          |
| ML Road Pedal         | \$24,624.89    | 108          |
| ML Road Front Wheel   | \$78,986.43    | 105          |

10. Execute the same query a few more times. Verify the number of rows returned; it should be 12.

### How it works...

There were 120 members in the initial set. We specified that we want 10 percent of them. Consequently, each query returned 12 members, and each time those were 12 different members. Let's see how this was done.

By assigning a random value to each member of a hierarchy we can use the `TopCount()` function to return a predefined number of random members in that hierarchy.

The core part is the definition of the `H` on axis sample calculated set. The second argument determines the size of the sample. The third orders the members by the random value each of them got using the `Random` value calculated measure. Simple, isn't it?

## There's more...

As in the previous recipe, the `IsEmpty()` function is used to preserve empty rows. If the original value was empty, so should be the value of the calculated measure.

### Alternative solution

Stored procedures are often a valid alternative to some MDX problems. The *Analysis Services Stored Procedure Project (ASSP)* at <http://tinyurl.com/ASSPCodePlex> contains the `RandomSample` function which can be used to get a sample of N members from a set.

## See also

The recipe *Displaying random values* shows how to generate random values in the query which is very important for understanding this recipe completely.

## Displaying a sample from a random hierarchy

In the previous recipe you learned how to generate a random sample from a predefined hierarchy. In this recipe you'll learn how to generate a sample from a random hierarchy. The idea is to make a single report that can be used over and over again, similar to news ticker on TV. All that is required is a mechanism for refreshing it. In case of a web page or a widget, it can be implemented relatively easy.

Apart from that, this recipe is interesting from a didactical point of view. It shows how to operate with hierarchies, how to navigate them, and get information about them, all without actually specifying any of the cube's hierarchy, other than measures. Even if you don't plan on implementing it, it can be helpful to understand the methods used.

## Getting ready

The prerequisite for this recipe is the recipe *Implementing the Tally table utility dimension*. If you haven't already read it and implemented the solution presented there, do it now.

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to limit the sample to 10 members, the best 10 to be precise.

## How to do it...

Follow these steps to display a sample from a random hierarchy:

1. Add the WITH part of the query.
2. Create the # of H in cube calculated measure using the following definition:

```
MEMBER [Measures].[# of H in cube] AS  
    Dimensions.Count
```

3. Create the Random H ordinal calculated measure using the following definition:

```
MEMBER [Measures].[Random H ordinal] AS  
    Int( [# of H in cube] * Rnd() )
```

4. Create the Random H set calculated set using this definition:

```
SET [Random H set] AS  
    TopCount(  
        Descendants(  
            Dimensions( [Random H ordinal] ).Levels(0)  
                .ALLMEMBERS,  
            1 ),  
        10,  
        [Measures].DefaultMember )
```

5. Create three calculated measures to display information about the hierarchy being displayed in the result of the query. Remember, this will be a random hierarchy, so we need to know what we're looking at.

```
MEMBER [Measures].[Dimension] AS  
    Mid( [Random H set].Item(0).Hierarchy.UniqueName, 2,  
        InStr( [Random H set].Item(0).Hierarchy.UniqueName,  
            '[' ) - 2 )
```

```
MEMBER [Measures].[Hierarchy] AS  
    [Random H set].Item(0).Hierarchy.Name
```

```
MEMBER [Measures].[Cardinality] AS  
    [Random H set].Item(0).Hierarchy.CurrentMember  
        .Level.Members.Count
```

6. Create another three calculated measures to display information about the measure we're analyzing. Here we'll use the default measure:

```

MEMBER [Measures].[Measure] AS
    [Measures].DefaultMember.Name

MEMBER [Measures].[Valid context] AS
    ( [Random H set].Item(0).Hierarchy.Levels(0).Item(0),
        [Measures].DefaultMember ) <>
    Avg( [Random H set], [Measures].DefaultMember )

MEMBER [Measures].[Value] AS
    iif( [Measures].[Valid context],
        [Measures].DefaultMember,
        'N/A' )
    , FORMAT_STRING = 'Currency'

```

7. Create the Ratio calculated measure using the following definition:

```

MEMBER [Measures].[Ratio] AS
    iif( [Measures].[Valid context],
        iif( ( [Measures].[Value],
            [Random H set].Item(0).Hierarchy.CurrentMember
                .Parent ) = 0,
            null,
            [Measures].[Value] /
            ( [Measures].[Value],
                [Random H set].Item(0).Hierarchy.CurrentMember
                    .Parent ) ),
        'N/A' )
    , FORMAT_STRING = 'Percent'

```

8. Put the Random H set on rows.

9. Put the Hierarchy, Dimension, Cardinality, Measure, Value, and Ratio measures on columns:

```

SELECT
    { [Measures].[Hierarchy],
        [Measures].[Dimension],
        [Measures].[Cardinality],
        [Measures].[Measure],
        [Measures].[Value],
        [Measures].[Ratio] } ON 0,
    { [Random H set] } ON 1
FROM
    [Adventure Works]

```

10. Execute the query.
11. If the result of your query **doesn't** match the following image, you're good. Remember, we were supposed to get **different** results because of random values. What should match is the number of rows. Both of us should get up to 10 rows, no more than that. The following image shows 10 rows out of 11 for the **Product.Size Range** hierarchy together with the percentages of each size range in respect to the **Reseller Sales Amount** value of the root member in that hierarchy.

|          | Hierarchy  | Dimension | Cardinality | Measure               | Value           | Ratio  |
|----------|------------|-----------|-------------|-----------------------|-----------------|--------|
| 42-46 CM | Size Range | Product   | 11          | Reseller Sales Amount | \$28,414,387.97 | 35.32% |
| 48-52 CM | Size Range | Product   | 11          | Reseller Sales Amount | \$18,389,433.47 | 22.86% |
| 38-40 CM | Size Range | Product   | 11          | Reseller Sales Amount | \$14,370,341.92 | 17.86% |
| 60-62 CM | Size Range | Product   | 11          | Reseller Sales Amount | \$8,160,130.13  | 10.14% |
| 54-58 CM | Size Range | Product   | 11          | Reseller Sales Amount | \$7,173,941.24  | 8.92%  |
| NA       | Size Range | Product   | 11          | Reseller Sales Amount | \$2,130,544.01  | 2.65%  |
| L        | Size Range | Product   | 11          | Reseller Sales Amount | \$603,570.48    | 0.75%  |
| S        | Size Range | Product   | 11          | Reseller Sales Amount | \$504,976.54    | 0.63%  |
| M        | Size Range | Product   | 11          | Reseller Sales Amount | \$455,723.49    | 0.57%  |
| XL       | Size Range | Product   | 11          | Reseller Sales Amount | \$182,028.99    | 0.23%  |

12. Execute the same query few more times. Verify the number of rows returned; it should never be greater than 10. Also, the time to execute the query will vary by size of the hierarchy you've hit and its aggregation design.

### How it works...

The idea of this query is to have an ever-changing report that shows the top 10 members of a random hierarchy from the pool of all hierarchies in the cube.

The first calculated measure, # of H in cube, counts the number of hierarchies in the cube. In the case of the **Adventure Works DW 2008R2** cube, that number is 213 for the Enterprise Edition (EE) and 191 for the Standard Edition (SE).

Once we know our limits in terms of the dimensionality of the cube, we can use that number to span the random values generated by the Rnd( ) function, as implemented in the second calculated measure, the Random H ordinal measure. That measure generates random integers between 0 and 212 (0 and 190 for Standard Edition).

Then comes the calculated set. It takes the top 10 members below the root member from the random hierarchy using the default cube measure. Naturally, it doesn't have to be that particular measure; you can use any measure in its place. This measure was used to illustrate something else. First, let's see what the next two calculated measures do.

The Dimension calculated measure takes the unique name of the hierarchy and extracts the first part of it. In short, it removes square brackets.

The Hierarchy calculated measure displays the name of the hierarchy.

The Cardinality calculated measure counts number of leaf members on that random hierarchy.

The Measure calculated measure displays the name of the measure to be used in the TopCount( ) function.

The next measure is a complex one. It is used to detect non-existing cube space.

The idea is to test if the average value in the top 10 members is the same as the value of the root member. If it is, then that hierarchy is not related to the measure being used in the query – the default cube measure in this example. The values repeat.

If the average value is not the same, values don't repeat which means we can calculate the next two measures, the Value and Ratio measures.

The Value measure shows the value of the default measure in case the context is valid. In case it is not, the **not** available indicator will be there as a notice.

Finally, the Ratio measure shows how we can calculate something useful. If the context is valid and the denominator is not null or zero, we can display the ratio of the current row against all rows. If any of those are not true, we'll show **N/A**. Here's an example of such a query.

|   | Hierarchy      | Dimension | Cardinality | Measure               | Value | Ratio |
|---|----------------|-----------|-------------|-----------------------|-------|-------|
| 0 | Total Children | Customer  | 6           | Reseller Sales Amount | N/A   | N/A   |
| 1 | Total Children | Customer  | 6           | Reseller Sales Amount | N/A   | N/A   |
| 2 | Total Children | Customer  | 6           | Reseller Sales Amount | N/A   | N/A   |
| 3 | Total Children | Customer  | 6           | Reseller Sales Amount | N/A   | N/A   |
| 4 | Total Children | Customer  | 6           | Reseller Sales Amount | N/A   | N/A   |
| 5 | Total Children | Customer  | 6           | Reseller Sales Amount | N/A   | N/A   |

The **Reseller Sales Amount** measure, which is the cube's default measure, is not related to the **Customer** dimension. The **N/A** notice makes this visible.

There's more...

The initial pool of hierarchies can naturally be reduced. There's absolutely no need to use every hierarchy of the cube.

For example, you can define a set of hierarchies to be used in the pool. This can be achieved either by referring to their names or ordinal positions. The former is less prone to errors and easier to maintain. The other way is to apply a filter to all hierarchies based on a condition, for example, their cardinality, to avoid hierarchies that are either too large or too small to be useful in this report.

Let's see how it could be done.

Change the first part of the query to the following text. The rest of the query can remain as before, in the previous section.

```
WITH

MEMBER [Measures].[# of H in cube] AS
    Dimensions.Count

SET [H pool] AS
    Filter( Head( [Tally Table].[N].[N].MEMBERS,
        [Measures].[# of H in cube] ) AS MyLoopSet,
        NOT Dimensions( MyLoopSet.CurrentOrdinal - 1) Is
        [Measures]
        AND
        { Descendants(
            Dimensions( MyLoopSet.CurrentOrdinal - 1)
            .Levels(0).ALLMEMBERS,
            1 ) AS MyLevelSet }.Count <= 100
        AND
        MyLevelSet.Count >= 3
        AND
        (
            ( Dimensions( MyLoopSet.CurrentOrdinal - 1)
            .Levels(0).Item(0),
            [Measures].DefaultMember ) <>
            Avg( MyLevelSet,
            [Measures].DefaultMember )
        )
    )

MEMBER [Measures].[# of H in pool] AS
    [H pool].Count

MEMBER [Measures].[Random H ordinal] AS
    Int( [Measures].[# of H in pool] * Rnd() )

SET [Random H set] AS
```

---

```

TopCount(
    Descendants(
        Dimensions( [H pool].Item(
            [Measures].[Random H ordinal] )
            .MemberValue - 1 ).Levels(0).ALLMEMBERS,
        1 ),
        10,
        [Measures].DefaultMember )
)

```

The first measure, # of H in cube is the same, but immediately after it comes a new set. The purpose of that set is to limit the number of hierarchies to be displayed in the report. This is achieved using the Tally Table dimension defined in the third recipe in this chapter. Now we're about to see its usefulness.

The Tally Table dimension is used to iterate on all cube hierarchies so that the final set H pool contains only those hierarchies that meet the condition. The condition says:

- ▶ Avoid the Measures hierarchy
- ▶ Avoid hierarchies whose first level contains less than 3 members and more than 100 members
- ▶ Avoid unrelated hierarchies

It's a relatively complex expression, so let's analyze it a bit further.

We took only a small portion of members on the Tally Table dimension and used them to perform the loop. We also named that as MyLoopSet in order to be able to use that set later in expression.

The CurrentOrdinal function is a 1-based loop counter. That's why we need to deduct 1 from it when we want to use it inside the Dimensions() function which is 0-based.

MyLevelSet is used in the same manner as MyLoopSet. We've defined it to be able to refer to it in later parts of the expression.

The Levels() function returns members on a particular level. Level 0 represents the root level, the topmost level. The ALLMEMBERS function returns all members on a particular level, including the calculated members. The Item(0) part used in the combination of the Levels(0) part is a way of specifying the root member.

The # of H in pool is a new measure that we need to calculate the size of the new pool. In the case of the **Adventure Works DW 2008R2** database (Enterprise Edition), that number is 117 which can easily be verified by including that measure in the query, on columns.

Once we know that number, we can limit the span of the generated random numbers.

Finally, the definition of the `Random H` set is also modified to fit our intention. This time we're using the set containing the members of the Tally Table dimension. Those members are random this time. In other words, we're selecting a random member from the pool, reading its `MemberValue` property (which is an integer) and using it to traverse to the original hierarchy to which ordinal that member is pointing to.

It's probably not necessary to emphasize that a query modified this way will never return the `N/A` value or an error when it hits the **Measures** dimension by accident. The latter cannot be said for the example described in the *How to do it* section. The chances are very small but they still exist. Therefore, the Tally Table solution is the recommended solution.

### The N/A sign

In the first chapter of this book there was a recommendation to use the `FORMAT_STRING` property instead of providing string values for null. In this case we had to make a distinction between null as a result of the query and **N/A** as an indicator of non-existing cube space.

### How to use another measure?

The recipe showed how to use the default cube measure which is the `Reseller Sales Amount` measure. If you want to use another measure, simply search-replace all occurrences of the `[Measures].DefaultMember` with the measure of your choice.

### See also

The recipe *Displaying random values* shows how to generate random values in the query which is very important for understanding this recipe completely. The recipe *Displaying a random sample of hierarchy members* shows how to make use of the random values once we know how to generate them.

## Performing complex sorts

Sorting is one of those often-requested operations. To sort a hierarchy by a measure is not a problem. Neither is to sort a hierarchy using its member properties. The MDX language has a designated function for that operation and a pretty straightforward one too. Yes, we're talking about the `Order( )` function.

Difficulties appear when we need to sort two or more hierarchies, one inside the other, or when we need to use two or more criteria. Not to mention the confusion when one of the members on columns is supposed to be the criteria for sorting a related hierarchy on rows. These are complex sort operations, operations we'll cover in this recipe.

Let's build a case and see how it should be solved.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Product** dimension, **Sales Territory** dimension and the **Date** dimension. Here's the query we'll start from:

```
SELECT
    NON EMPTY
        { [Date].[Fiscal].[Fiscal Year].MEMBERS *
            [Measures].[Sales Amount] } ON 0,
    NON EMPTY
        { [Sales Territory].[Sales Territory Country]
            .[Sales Territory Country].MEMBERS *
            [Product].[Color].[Color].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns 54 country-color combinations on rows, four fiscal years on columns, and the value of sales shown in the grid. No particular sort operation was applied. The countries and colors are returned in their default order – alphabetically, as visible in the following image:

|           |              | FY 2006        | FY 2007        | FY 2008        | FY 2009      |
|-----------|--------------|----------------|----------------|----------------|--------------|
|           |              | Sales Amount   | Sales Amount   | Sales Amount   | Sales Amount |
| Australia | Black        | \$386,170.96   | \$966,208.67   | \$1,688,269.24 | \$1,427.65   |
| Australia | Blue         | (null)         | (null)         | \$1,328,411.04 | \$870.86     |
| Australia | Multi        | (null)         | (null)         | \$21,488.04    | \$693.73     |
| Australia | NA           | (null)         | (null)         | \$101,951.10   | \$4,115.47   |
| Australia | Red          | \$1,876,531.32 | \$603,914.25   | \$200,424.34   | \$454.87     |
| Australia | Silver       | \$305,999.10   | \$447,426.63   | \$768,442.84   | \$384.93     |
| Australia | Silver/Black | (null)         | (null)         | \$1,702.39     | (null)       |
| Australia | White        | (null)         | (null)         | \$961.93       | \$44.95      |
| Australia | Yellow       | (null)         | \$82,035.88    | \$1,866,163.99 | \$1,241.77   |
| Canada    | Black        | \$1,065,837.48 | \$2,858,563.62 | \$2,086,940.76 | \$2,033.03   |
| Canada    | Blue         | \$2,947.23     | \$8,774.37     | \$1,023,374.50 | \$1,004.34   |
| Canada    | Multi        | \$14,319.69    | \$76,377.18    | \$47,572.38    | \$1,211.61   |
| Canada    | NA           | (null)         | \$52,132.54    | \$140,230.37   | \$5,037.15   |
| Canada    | Red          | \$1,910,459.19 | \$1,511,459.17 | \$150,759.84   | \$734.79     |
| Canada    | Silver       | \$658,333.56   | \$885,178.06   | \$1,314,349.15 | \$274.95     |
| Canada    | Silver/Black | (null)         | (null)         | \$28,242.00    | (null)       |
| Canada    | White        | \$1,010.64     | (null)         | \$3,464.39     | \$71.92      |
| Canada    | Yellow       | (null)         | \$527,694.91   | \$1,976,895.77 | \$485.91     |

To sort the rows by a particular value, you could simply wrap the `Order( )` function around them. For example, to sort the previous result by the **Sales Amount** in **FY 2009**, you would have to change the set on rows like this:

```
Order(
    [Sales Territory].[Sales Territory Country]
        .[Sales Territory Country].MEMBERS *
    [Product].[Color].[Color].MEMBERS,
    ( [Date].[Fiscal].[Fiscal Year].&[2009],
        [Measures].[Sales Amount] ),
    BDESC )
```

The following image shows the result of the modified query:

|                |       | FY 2006        | FY 2007         | FY 2008        | FY 2009      |
|----------------|-------|----------------|-----------------|----------------|--------------|
|                |       | Sales Amount   | Sales Amount    | Sales Amount   | Sales Amount |
| United States  | NA    | (null)         | \$165,385.50    | \$375,195.04   | \$8,338.18   |
| Canada         | NA    | (null)         | \$52,132.54     | \$140,230.37   | \$5,037.15   |
| Australia      | NA    | (null)         | (null)          | \$101,951.10   | \$4,115.47   |
| United States  | Black | \$5,053,651.68 | \$10,104,109.58 | \$8,022,261.19 | \$4,027.57   |
| United States  | Multi | \$45,434.46    | \$199,260.72    | \$120,426.86   | \$2,364.24   |
| United Kingdom | NA    | (null)         | \$15,912.81     | \$75,647.63    | \$2,138.44   |
| Canada         | Black | \$1,065,837.48 | \$2,858,563.62  | \$2,086,940.76 | \$2,033.03   |
| United States  | Blue  | \$10,384.35    | \$21,558.91     | \$4,244,913.01 | \$1,980.17   |
| Germany        | NA    | (null)         | (null)          | \$61,959.99    | \$1,709.83   |
| France         | NA    | (null)         | \$12,287.01     | \$75,754.33    | \$1,508.53   |
| Australia      | Black | \$386,170.96   | \$966,208.67    | \$1,688,269.24 | \$1,427.65   |

This time the rows are returned in the descending order in respect to the last column, highlighted in the image. But, notice one thing – countries and colors are mixed. Breaking their sequence like that will rarely be asked for. More often, the request will be to sort one hierarchy inside the other, the rightmost inside those on its left. In other words, to sort colors inside each country. Now let's see how this can be done.

### How to do it...

Follow these steps to sort hierarchies on rows one inside the other.

1. Modify the set on rows like this:

```
{ Generate(
    [Sales Territory].[Sales Territory Country]
```

```

    .[Sales Territory Country].MEMBERS,
    [Sales Territory].[Sales Territory Country]
        .CurrentMember *
    Order( [Product].[Color].[Color].MEMBERS,
    ( [Date].[Fiscal].[Fiscal Year].&[2009],
        [Measures].[Sales Amount] ),
    BDESC
    )
    )
}

```

2. Execute the query. The result should match the following image:

The screenshot shows a query results grid with two tabs: 'Messages' and 'Results'. The 'Results' tab is selected and displays a table with data from four columns: 'Country/Region' (Country), 'Color', 'FY 2006' (Sales Amount), 'FY 2007' (Sales Amount), 'FY 2008' (Sales Amount), and 'FY 2009' (Sales Amount). The data is grouped by country, and colors are ordered from highest to lowest sales amount within each country.

|           |              | FY 2006        | FY 2007        | FY 2008        | FY 2009      |
|-----------|--------------|----------------|----------------|----------------|--------------|
|           |              | Sales Amount   | Sales Amount   | Sales Amount   | Sales Amount |
| Australia | NA           | (null)         | (null)         | \$101,951.10   | \$4,115.47   |
| Australia | Black        | \$386,170.96   | \$966,208.67   | \$1,688,269.24 | \$1,427.65   |
| Australia | Yellow       | (null)         | \$82,035.88    | \$1,866,163.99 | \$1,241.77   |
| Australia | Blue         | (null)         | (null)         | \$1,328,411.04 | \$870.86     |
| Australia | Multi        | (null)         | (null)         | \$21,488.04    | \$693.73     |
| Australia | Red          | \$1,876,531.32 | \$603,914.25   | \$200,424.34   | \$454.87     |
| Australia | Silver       | \$305,999.10   | \$447,426.63   | \$768,442.84   | \$384.93     |
| Australia | White        | (null)         | (null)         | \$961.93       | \$44.95      |
| Australia | Silver/Black | (null)         | (null)         | \$1,702.39     | (null)       |
| Canada    | NA           | (null)         | \$52,132.54    | \$140,230.37   | \$5,037.15   |
| Canada    | Black        | \$1,065,837.48 | \$2,858,563.62 | \$2,086,940.76 | \$2,033.03   |
| Canada    | Multi        | \$14,319.69    | \$76,377.18    | \$47,572.38    | \$1,211.61   |
| Canada    | Blue         | \$2,947.23     | \$8,774.37     | \$1,023,374.50 | \$1,004.34   |
| Canada    | Red          | \$1,910,459.19 | \$1,511,459.17 | \$150,759.84   | \$734.79     |
| Canada    | Yellow       | (null)         | \$527,694.91   | \$1,976,895.77 | \$485.91     |
| Canada    | Silver       | \$658,333.56   | \$885,178.06   | \$1,314,349.15 | \$274.95     |
| Canada    | White        | \$1,010.64     | (null)         | \$3,464.39     | \$71.92      |
| Canada    | Silver/Black | (null)         | (null)         | \$28,242.00    | (null)       |

3. Notice colors ordered this time descending inside each country. Notice also their sequence changes from country to country (**Yellow** is the 3<sup>rd</sup> in **Australia**, **Multi** is the 3<sup>rd</sup> in **Canada**).

## How it works...

Do you remember a recipe in the first chapter about creating a new set from the old one using iteration *Iterating on a set in order to create a new one*. The function used in that recipe is the same one used in this recipe – the `Generate()` function, which takes a set (single or multi-dimensional one) and creates a new set the way we specify in the second argument of that function.

What's important to note is that we've used only the first hierarchy on rows, not both of them. The `Generate()` function has no problem in creating a multi-dimensional set from the single-dimensional one. In fact, that's exactly what was needed in this case. We had to preserve the outer hierarchy's order. We've used it as the first argument of the `Generate()` function and cross-joined each of its members with the set of colors ordered by the required criteria. It may not be obvious, but the criteria also included the current country, implicitly. That's because `Generate()` is a loop function that sets its own context instead of modifying the existing. Consequently, the colors came ordered differently in each country.

## There's more...

Sorting the rightmost hierarchy inside the one on the left is fine, but what if we also need to sort the outer one? What if the requirement says, "return the countries in descending order by the same criteria and then only return colors sorted inside each country". Can we deliver that as well? Yes, and here's how.

Modify the set on rows this way:

```
{ Generate(
    Order( [Sales Territory].[Sales Territory Country]
          .[Sales Territory Country].MEMBERS,
          ( [Date].[Fiscal].[Fiscal Year].&[2009],
            [Measures].[Sales Amount] ),
        BDESC
      ),
    [Sales Territory].[Sales Territory Country].CurrentMember *
    Order( [Product].[Color].[Color].MEMBERS,
          ( [Date].[Fiscal].[Fiscal Year].&[2009],
            [Measures].[Sales Amount] ),
        BDESC
      )
    ) }
```

If you take a close look, you'll notice not much has changed. All that we have done extra this time is that we have ordered the initial set in the `Generate()` function so that it preserves the order when we cross join its members with the other set.

The result of this modification is shown on the following image:

The screenshot shows the SSMS Results grid with two tabs: 'Messages' and 'Results'. The 'Results' tab is selected and displays a table with six columns: 'FY 2006', 'FY 2007', 'FY 2008', 'FY 2009', 'Sales Amount', and 'Sales Amount'. The data is sorted by 'Sales Amount' in descending order. The first few rows show data for 'United States' and 'Canada' across various product categories like NA, Black, Multi, Blue, Yellow, Red, Silver, White, and Silver/Black.

|               |              | FY 2006        | FY 2007         | FY 2008        | FY 2009      |
|---------------|--------------|----------------|-----------------|----------------|--------------|
|               |              | Sales Amount   | Sales Amount    | Sales Amount   | Sales Amount |
| United States | NA           | (null)         | \$165,385.50    | \$375,195.04   | \$8,338.18   |
| United States | Black        | \$5,053,651.68 | \$10,104,109.58 | \$8,022,261.19 | \$4,027.57   |
| United States | Multi        | \$45,434.46    | \$199,260.72    | \$120,426.86   | \$2,364.24   |
| United States | Blue         | \$10,384.35    | \$21,558.91     | \$4,244,913.01 | \$1,980.17   |
| United States | Yellow       | (null)         | \$1,849,153.43  | \$7,807,453.88 | \$1,079.80   |
| United States | Red          | \$6,559,542.49 | \$5,334,256.87  | \$662,928.77   | \$1,049.70   |
| United States | Silver       | \$3,986,235.29 | \$3,232,560.30  | \$5,070,819.13 | \$549.90     |
| United States | White        | \$5,562.75     | (null)          | \$15,535.54    | \$44.95      |
| United States | Silver/Black | (null)         | (null)          | \$91,526.48    | (null)       |
| Canada        | NA           | (null)         | \$52,132.54     | \$140,230.37   | \$5,037.15   |
| Canada        | Black        | \$1,065,837.48 | \$2,858,563.62  | \$2,086,940.76 | \$2,033.03   |
| Canada        | Multi        | \$14,319.69    | \$76,377.18     | \$47,572.38    | \$1,211.61   |
| Canada        | Blue         | \$2,947.23     | \$8,774.37      | \$1,023,374.50 | \$1,004.34   |
| Canada        | Red          | \$1,910,459.19 | \$1,511,459.17  | \$150,759.84   | \$734.79     |
| Canada        | Yellow       | (null)         | \$527,694.91    | \$1,976,895.77 | \$485.91     |
| Canada        | Silver       | \$658,333.56   | \$885,178.06    | \$1,314,349.15 | \$274.95     |
| Canada        | White        | \$1,010.64     | (null)          | \$3,464.39     | \$71.92      |
| Canada        | Silver/Black | (null)         | (null)          | \$28,242.00    | (null)       |

Look at the query one more time and you'll notice there are two criteria in it, both the same. It doesn't take a lot of imagination to conclude that they don't have to be the same. Yes, you can have different criteria, simply modify any of them and test. Having learned this much about sorting, you can confidently perform complex sorts.

### Things to be extra careful about

Write this query, but don't execute it yet!!

```

SELECT
NON EMPTY
{ [Product].[Product Line].[Product Line].MEMBERS *
[Measures].[Sales Amount] } ON 0,
NON EMPTY
{ Generate(
    [Sales Territory].[Sales Territory Country]
    .[Sales Territory Country].MEMBERS,
    [Sales Territory].[Sales Territory Country]
    .CurrentMember *)

```

```

        Order( [Product].[Model Name].[Model Name].MEMBERS,
        ( [Product].[Product Line].&[M],
        -- [Product].[Model Name].CurrentMember,
        [Measures].[Sales Amount] ),
        BDESC )
    ) } ON 1
FROM
[Adventure Works]

```

If you analyze the code, you'll notice that the same idea is used to sort the results based on one of the columns. This time however, the hierarchies on rows and columns are related. The **Model Name** and the **Product Line** attribute hierarchies can be found in the **Product Model Lines** user hierarchy. In short, models are grouped by the product lines.

Now, run the query and observe the result:

|           |                        | Accessory    | Components   | Mountain     | Road         | Touring      |
|-----------|------------------------|--------------|--------------|--------------|--------------|--------------|
|           |                        | Sales Amount |
| Australia | All-Purpose Bike Stand | (null)       | (null)       | \$10,335.00  | (null)       | (null)       |
| Australia | Bike Wash              | \$2,286.42   | (null)       | (null)       | (null)       | (null)       |
| Australia | Chain                  | (null)       | \$850.08     | (null)       | (null)       | (null)       |
| Australia | Classic Vest           | \$21,065.62  | (null)       | (null)       | (null)       | (null)       |
| Australia | Cycling Cap            | \$4,828.75   | (null)       | (null)       | (null)       | (null)       |
| Australia | Fender Set - Mountain  | (null)       | (null)       | \$7,143.50   | (null)       | (null)       |
| Australia | Front Brakes           | (null)       | \$3,770.10   | (null)       | (null)       | (null)       |
| Australia | Front Derailleur       | (null)       | \$5,214.93   | (null)       | (null)       | (null)       |
| Australia | Half-Finger Gloves     | \$9,771.75   | (null)       | (null)       | (null)       | (null)       |
| Australia | Hitch Rack - 4-Bike    | \$18,406.70  | (null)       | (null)       | (null)       | (null)       |
| Australia | HL Bottom Bracket      | (null)       | \$3,426.02   | (null)       | (null)       | (null)       |

Oops, it doesn't look good - there's no trace of any sort in it. Now uncomment the commented line and run it again. All good, the result is ordered by the middle column, the **Mountain** model:

|           |                         | Accessory    | Components   | Mountain       | Road         | Touring      |
|-----------|-------------------------|--------------|--------------|----------------|--------------|--------------|
|           |                         | Sales Amount | Sales Amount | Sales Amount   | Sales Amount | Sales Amount |
| Australia | Mountain-200            | (null)       | (null)       | \$2,171,361.25 | (null)       | (null)       |
| Australia | Mountain-100            | (null)       | (null)       | \$670,498.02   | (null)       | (null)       |
| Australia | Mountain-400-W          | (null)       | (null)       | \$77,718.49    | (null)       | (null)       |
| Australia | Mountain-500            | (null)       | (null)       | \$53,177.24    | (null)       | (null)       |
| Australia | HL Mountain Frame       | (null)       | (null)       | \$21,997.62    | (null)       | (null)       |
| Australia | Women's Mountain Shorts | (null)       | (null)       | \$15,649.76    | (null)       | (null)       |
| Australia | All-Purpose Bike Stand  | (null)       | (null)       | \$10,335.00    | (null)       | (null)       |
| Australia | HL Mountain Tire        | (null)       | (null)       | \$8,400.00     | (null)       | (null)       |
| Australia | LL Mountain Frame       | (null)       | (null)       | \$7,600.51     | (null)       | (null)       |
| Australia | Fender Set - Mountain   | (null)       | (null)       | \$7,143.50     | (null)       | (null)       |
| Australia | ML Mountain Tire        | (null)       | (null)       | \$5,938.02     | (null)       | (null)       |

### What's going on?

Remember what we said about current members being implicit in the criteria for sort? The same applies here. Both the country and the model are in the tuple that determines the sort.

In the examples we started this recipe with, all the hierarchies were unrelated and no problem was noticed. This time they were related and behaved differently because related hierarchies interfere with each other. In other words, the **Mountain** member of the **Product Line** attribute hierarchy pushed the current member of the **Model Name** attribute hierarchy to its root member. The relation between them is 1:N, models are below the product lines. Consequently, all of the models evaluated the same in that tuple, as the value of the **Mountain** product line for a particular country. Sorting members by a constant value leaves them in their existing order. That's the result we got in the first image.

On the other hand, when we're explicit about the current member of the **Model Name** attribute hierarchy in the tuple for sort criteria, we get the correct result.

The difference is that this time we have specified the intersection of related hierarchies. In other words, we were referring to the individual cells found in the intersection of the models and the product lines. Those cells are exactly what we needed, each different from another and hence returning results sorted the way we wanted.

Remember this and don't forget to force the coordinate in case there are related hierarchies, when the hierarchy on columns is above the hierarchy on rows in terms of attribute paths.

### A costly operation

Sorting is a costly operation. If you have large dimensions, always look for an alternative solution. For example, if you don't need the entire set, use set-limiting functions like `NonEmpty()`, `TopCount()`, and others.

### See also

Refresh your memory about the `Generate()` function by reading the recipe *Iterating on a set in order to create a new one* in chapter 1.

## Using recursion to calculate cumulative values

This recipe shows how to calculate a cumulative product and sum it using standard range techniques and recursion.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

In this example we're going to use the **Subcategory** hierarchy of the **Product** dimension. Here's the query we'll start from:

```
SELECT
    { [Measures].[Gross Profit],
      [Measures].[Gross Profit Margin] } ON 0,
    { [Product].[Subcategory].[Subcategory].MEMBERS } ON 1
FROM
    [Adventure Works]
```

Once executed, the query returns 37 product subcategories and their corresponding profits and margins. Our task is to create calculations that will perform cumulative sums for the products.

## How to do it...

Follow these steps to create cumulative calculations.

1. Create a calculated set **Axis Set** based on what was on the rows axis:

```
SET [Axis Set] AS
    [Product].[Subcategory].[Subcategory].MEMBERS
```

2. Define the Rank on Set calculated measure using this definition:

```
MEMBER [Measures].[Rank on Set] AS
    Rank( [Product].[Subcategory].CurrentMember,
          [Axis Set] )
```

3. Define the following cumulative calculated measures and include them on the columns axis:

```
MEMBER [Measures].[Cumulative Sum] AS
    Sum( Head( [Axis Set],
               [Measures].[Rank on Set] ),
         [Measures].[Gross Profit] )
```

```
MEMBER [Measures].[Cumulative Product] AS
    Exp( Sum( Head( [Axis Set],
                    [Measures].[Rank on Set] ),
              Log( [Measures].[Gross Profit Margin] )
              )
        )
    , FORMAT_STRING = 'Percent'
```

4. Define the following recursive calculated measures and include them on the columns axis:

```

MEMBER [Measures].[Recursive Sum] AS
    iif( [Measures].[Rank on Set] = 1,
        [Measures].[Gross Profit],
        [Measures].[Gross Profit] +
        ( [Measures].[Recursive Sum],
            [Axis Set].Item( [Measures].[Rank on Set] - 1 - 1 )
        )
    )
, FORMAT_STRING = 'Currency'

MEMBER [Measures].[Recursive Product] AS
    iif( [Measures].[Rank on Set] = 1,
        [Measures].[Gross Profit Margin],
        [Measures].[Gross Profit Margin] *
        ( [Measures].[Recursive Product],
            [Axis Set].Item( [Measures].[Rank on Set] - 1 - 1 )
        )
    )
, FORMAT_STRING = 'Percent'

```

5. Execute the query. The result should match the following image:

|                   | Gross Profit | Cumulative Sum | Recursive Sum | Gross Profit Margin | Cumulative Product | Recursive Product |
|-------------------|--------------|----------------|---------------|---------------------|--------------------|-------------------|
| Bib-Shorts        | \$51,256.59  | \$51,256.59    | \$51,256.59   | 30.74%              | 30.74%             | 30.74%            |
| Bike Racks        | \$95,006.08  | \$146,262.66   | \$146,262.66  | 40.07%              | 12.32%             | 12.32%            |
| Bike Stands       | \$24,783.97  | \$171,046.63   | \$171,046.63  | 62.60%              | 7.71%              | 7.71%             |
| Bottles and Cages | \$38,233.69  | \$209,280.32   | \$209,280.32  | 59.48%              | 4.59%              | 4.59%             |
| Bottom Brackets   | \$13,474.82  | \$222,755.15   | \$222,755.15  | 26.00%              | 1.19%              | 1.19%             |
| Brakes            | \$17,077.70  | \$239,832.85   | \$239,832.85  | 25.87%              | 0.31%              | 0.31%             |
| Caps              | (\$1,203.20) | \$238,629.65   | \$238,629.65  | -2.35%              | -1.#IND            | -0.01%            |
| Chains            | \$2,422.08   | \$241,051.73   | \$241,051.73  | 25.83%              | -1.#IND            | 0.00%             |
| Cleaners          | \$8,538.59   | \$249,590.32   | \$249,590.32  | 46.39%              | -1.#IND            | 0.00%             |
| Cranksets         | \$52,778.81  | \$302,369.13   | \$302,369.13  | 25.88%              | -1.#IND            | 0.00%             |
| Derailleurs       | \$18,147.48  | \$320,516.61   | \$320,516.61  | 25.85%              | -1.#IND            | 0.00%             |

6. Notice the cumulative values displayed using the calculated measures created in this query. The **Sum** is identical whereas the **Product** experiences a problem with negative values when recursion is not used.

## How it works...

Two types of algorithms to calculate cumulative values were presented in this recipe.

Both approaches require a calculated measure that serves as an indicator of the current position of the set on axis. That's why we defined both the set and the rank.

Once we know the position of the current row during the iteration, we can calculate the sum of members up to that position.

We can also perform the sum of logarithm values of the measure and then take the exponential value of it in order to calculate the cumulative product value. Here we've used a known mathematical rule which says that the logarithm of a product is equal to the sum of individual logarithms. In other words, logarithms reduce products to sum. See Wikipedia for more info:

<http://tinyurl.com/WikiLogarithm>

The opposite operation of logarithm is the exponential function, therefore we've applied it on the sum of algorithm in order to get the cumulative product.

The problem we've encountered with negative values will probably not happen when you use a proper measure in the cumulative product calculation, the one that is never negative or zero. For example, a measure that states the likelihood of a risk to happen, status to occur, or similar. If you want to be absolutely sure you don't get **-1.#IND** errors, avoid negative and zero values the same way you avoid division by zero problems.

The next set of calculated measures is also a cumulative type of measure. What differentiates them from the previous ones is that they use recursion.

Recursive measures are almost identical except that one uses the sum operation and the other the product operation. Of course, the measure they use is different, but that's only to make the result look better.

Their expression says this: If we're on the first member, we'll take the value of the measure. If we're somewhere else in the set, we'll take the measure's value and add/multiply it with the value of the measure in the previous position. The `.Item()` function is zero-based, `Rank()` is not. Therefore we need to reduce the rank by one. The second reduction, the second `-1` in the expression, is where we instruct the engine to move to the previous position in the set.

Additionally, format strings had to be restored on measures that lost their inherited format due to complex expressions.

## There's more...

It is important to define a set and operate exclusively on that set, not on the members of the hierarchy, and there are two reasons for that. First, members on rows don't have to be positioned in their original hierarchical order. Second, not all members need to be present on rows; the end-user might have excluded some of them from the query.

The calculations defined in this recipe work no matter what the order or exclusion of members is because they rely on a named set. However, the definition of that set should always match the set on rows. There are several ways how it can be done.

The first step is to sync the named set with the set on rows, as shown in this example. The other way is to keep the definition of what comes on rows in the named set exclusively, not in two places like rows and the set. This means we should specify the named set on rows instead of the set of members, like this:

```
SELECT
    { [Measures].[Gross Profit],
      [Measures].[Cumulative Sum],
      [Measures].[Recursive Sum],
      [Measures].[Gross Profit Margin],
      [Measures].[Cumulative Product],
      [Measures].[Recursive Product] } ON 0,
    { [Axis Set] } ON 1
FROM
    [Adventure Works]
```

Finally, there's an option to define the calculated set using the `Axis()` function, as explained in chapter 7. Here's what it **might** look like:

```
SET [Axis Set] AS
    Axis(1)
```

I said "look like", because it largely depends on the front-end being used, a very unfortunate but important detail discussed in detail in chapter 7.

Let's check if the query works for an arbitrary set of members.

Here's the modified part of the query:

```
{ [Product].[Subcategory].&[1],
  [Product].[Subcategory].&[3],
  [Product].[Subcategory].&[2] } ON 1
```

We're using the last definition of the set, the one with the `Axis(1)` as the definition.

Here's what the result looks like:

The screenshot shows a query results grid with columns: Gross Profit, Cumulative Sum, Recursive Sum, Gross Profit Margin, Cumulative Product, and Recursive Product. The rows represent different bike categories: Mountain Bikes, Touring Bikes, and Road Bikes. The data shows how recursive and non-recursive calculations differ across these categories.

|                | Gross Profit   | Cumulative Sum  | Recursive Sum   | Gross Profit Margin | Cumulative Product | Recursive Product |
|----------------|----------------|-----------------|-----------------|---------------------|--------------------|-------------------|
| Mountain Bikes | \$5,932,916.14 | \$5,932,916.14  | \$5,932,916.14  | 16.28%              | 16.28%             | 16.28%            |
| Touring Bikes  | \$217,277.71   | \$6,150,193.86  | \$6,150,193.86  | 1.52%               | 0.25%              | 0.25%             |
| Road Bikes     | \$4,364,902.75 | \$10,515,096.61 | \$10,515,096.61 | 9.95%               | 0.02%              | 0.02%             |

It's clear that both recursive and non-recursive calculations work as expected.

### A simplified version of the solution

In situations where all members of a level are used and when you don't need to worry about their order (because it can't be changed for instance), you can simplify the recursive calculation a lot and often gain on performance too.

Here's an updated query that shows the parts that need to change in boldface:

```

WITH
SET [Axis Set] AS
[Product].[Subcategory].[Subcategory].MEMBERS
MEMBER [Measures].[Rank on Set] AS
Rank( [Product].[Subcategory].CurrentMember,
[Axis Set] )
MEMBER [Measures].[Cumulative Sum] AS
Sum( null : [Product].[Subcategory].CurrentMember,
[Measures].[Gross Profit] )
MEMBER [Measures].[Cumulative Product] AS
Exp( Sum( null : [Product].[Subcategory].CurrentMember,
Log( [Measures].[Gross Profit Margin] )
)
)
, FORMAT_STRING = 'Percent'
MEMBER [Measures].[Recursive Sum] AS
iif( [Measures].[Rank on Set] = 1,
[Measures].[Gross Profit],
[Measures].[Gross Profit] +
( [Measures].[Recursive Sum],
[Product].[Subcategory].CurrentMember.PrevMember )
)
, FORMAT_STRING = 'Currency'
MEMBER [Measures].[Recursive Product] AS
iif( [Measures].[Rank on Set] = 1,
[Measures].[Gross Profit Margin],

```

```

[Measures].[Gross Profit Margin] *
( [Measures].[Recursive Product],
  [Product].[Subcategory].CurrentMember.PrevMember )
)
, FORMAT_STRING = 'Percent'
SELECT
{ [Measures].[Gross Profit],
  [Measures].[Cumulative Sum],
  [Measures].[Recursive Sum],
  [Measures].[Gross Profit Margin],
  [Measures].[Cumulative Product],
  [Measures].[Recursive Product] } ON 0,
{ [Product].[Subcategory].[Subcategory].MEMBERS } ON 1
FROM
[Adventure Works]

```

As you can see, there was no reference to the `Axix()` function. In cumulative calculations, this was replaced with the range from the first member (here null) to the current member, which is exactly what the cumulative value represents. In recursive calculations, the expression that determines the previous member in the set using the `Axix()` function is replaced with its simplification in terms of the `.PrevMember` function. Easy to remember, easy to use, but as said earlier, it works only in the special case of the complete set of members of a level.

### Which type of calculation to choose?

Recursive calculations can perform well, but not always, so they have to be used carefully.

In this recipe we showed how to build both types of calculations, recursive and cumulative. Armed with this knowledge, it's up to you to test which version performs better in your particular scenario because there's no general rule to it.

In the case of the Adventure Works cube, we can test this by modifying the query so that it uses the **Reseller** hierarchy which is the biggest hierarchy in that cube. A test should be performed by first clearing the cache, initializing the MDX script, and then running the query with one type of measure only. After that, the process should be repeated with the other types of measures. More information about clearing the cache will be given in the next chapter.

### See also

Refresh your memory about recursions by reading the recipe *Iterating on a set using recursion* in chapter 1.



# 9

## On the Edge

In this chapter, we will cover:

- ▶ Clearing the Analysis Services cache
- ▶ Using Analysis Services stored procedures
- ▶ Executing MDX queries in T-SQL environments
- ▶ Using SSAS Dynamic Management Views (DMVs) to fast-document a cube
- ▶ Using SSAS Dynamic Management Views (DMVs) to monitor activity and usage
- ▶ Capturing MDX queries generated by SSAS front-ends
- ▶ Performing custom drillthrough

### Introduction

The last chapter in this book is a special chapter. We're going to talk about topics which didn't fit into the previous chapters, topics where MDX mixes with other areas. These areas will expand our horizon and motivate us to explore.

We're starting with clearing the cache. Clearing the cache is a very important technique in performance tuning the cube. The response of consecutive queries is influenced by the ones before if we don't clear the cache, particularly if we don't clear it thoroughly.

Stored procedures are another interesting area. Whether you have developer skills or not, there are cool stored procedures available in community assemblies which means you're only a step away from exploring the benefits they bring to your project.

Using the technique of distributed queries, you can execute MDX queries inside the relational database environment. The third recipe in this chapter explains the procedure and settings you must perform in order for it to work.

Dynamic Management Views (DMVs) are another interesting and not yet thoroughly explored area. They can be used to get information about your project structure, cube, dimensions, you name it. Everything is there, in tabular format, so that you can use it with great ease. Two recipes cover that topic, although there are plenty of useful examples to be explored by yourself once you get the initial boost.

The last part of the chapter covers capturing the MDX queries that are generated by a front-end used to analyze cube data, and the drillthrough.

Remember, these recipes are here to encourage you to explore further, to go over the edge.

## Clearing the Analysis Services cache

MDX query performance tuning is a process in which various query attempts are made in order to see which combination of calculations backed up by the existing cube aggregations runs faster. Having the same initial conditions for every query is a must. Only then can we truly measure how slow they actually are.

The problem in preserving the initial condition lies in the fact that Analysis Services caches the result of each query, making every subsequent query potentially run faster than it normally would.

Normally, caching is a great thing. Hitting a cached value is a goal we're trying to achieve in our everyday cube usage because it speeds up the result. Here, we're trying to do the opposite - we're clearing the cache on purpose in order to have the same conditions for every query.

This recipe introduces the process of clearing the cache. It begins by showing the standard way of clearing the Analysis Services cache and then continues by pointing out other steps required to ensure all conditions are preserved, not just a part of them. The other part is partially covered in the next recipe. Therefore, make sure you read both of them.

Let's start with the first one.

### Getting ready

Start the **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance, then click on the **New XMLA Query** button.

Expand the **Databases** item in the **Object Explorer** so that you can see the **Adventure Works DW 2008R2** database and its **Adventure Works** cube.

In this example, we're going to show how to clear the cache for that cube.

## How to do it...

Follow these steps to clear the Analysis Services cube cache:

1. Write the following XMLA query:

```
<Batch xmlns =
  "http://schemas.microsoft.com/analysisservices/2003/engine">
  <ClearCache>
    <Object>
      <DatabaseID></DatabaseID>
      <CubeID></CubeID>
    </Object>
  </ClearCache>
</Batch>
```

2. Right-click the **Adventure Works DW 2008R2** database in the **Object Explorer** and select **Properties**:

|                                |                                                                                 |
|--------------------------------|---------------------------------------------------------------------------------|
| <b>General</b>                 |                                                                                 |
| Name                           | <b>Adventure Works DW 2008R2</b>                                                |
| <b>ID</b>                      | Adventure Works DW 2008R2                                                       |
| Description                    | A Unified Dimensional Model that encompasses all business data for the company. |
| Create Timestamp               | 12/22/2010 10:47:18 AM                                                          |
| Last Schema Update             | 2/1/2011 12:30:39 AM                                                            |
| Last Update                    | 2/1/2011 12:31:10 AM                                                            |
| Read-Write Mode                | ReadWrite                                                                       |
| <b>Security Settings</b>       |                                                                                 |
| Data Source Impersonation Info | <b>Default</b>                                                                  |
| <b>Status</b>                  |                                                                                 |
| Last Processed                 | 1/30/2011 10:45:05 PM                                                           |
| Estimated Size                 | 130.30 MB                                                                       |
| <b>Storage</b>                 |                                                                                 |
| Storage Location               |                                                                                 |

3. Notice the **ID** property highlighted in the previous image. Copy the value, close the **Database Properties** window, and paste it inside the **DatabaseID** tags:

```
<DatabaseID>Adventure Works DW 2008R2</DatabaseID>
```

4. Now right-click the **Adventure Works** cube in the **Object Explorer** and select **Properties**:

| General                      |                        |
|------------------------------|------------------------|
| Name                         | Adventure Works        |
| ID                           | Adventure Works        |
| Description                  |                        |
| Create Timestamp             | 12/22/2010 10:47:18 AM |
| Last Schema Update           | 2/1/2011 12:30:41 AM   |
| Storage Location             |                        |
| Processing Mode              | Regular                |
| Script Cache Processing Mode | Regular                |
| Status                       |                        |
| State                        | Processed              |
| Last Processed               | 2/1/2011 12:31:08 AM   |

5. Again, notice the **ID** property highlighted in the previous image. Copy the value, close the **Cube Properties** window and paste it inside the **CubeID** tags:

```
<CubeID>Adventure Works</CubeID>
```

6. Run the XMLA query. If everything went fine, you should see the result as shown in the following image. The cube cache is successfully cleared.

```
<return xmlns="urn:schemas-microsoft-com:xml-analysis">
  <results xmlns="http://schemas.microsoft.com/analysisservices/2003
    <root xmlns="urn:schemas-microsoft-com:xml-analysis:empty" />
  </results>
</return>
```

### How it works...

The `ClearCache` command clears the Analysis Services cache for the object specified in the command.

Several objects can be used inside that command. A common scenario is to use the cube object, like in this example. Other objects are specified later in this recipe.

Inside the `ClearCache` command, we had to specify the object's ID, a property that uniquely identifies the object (among its sibling objects only, not on the entire SSAS instance). The ID is not visible in the **Object Explorer**, only the object's name. When we need the ID, we have to open the **Properties** window for that object. It's there among other properties.

The uniqueness of the object on the entire SSAS instance is enforced by the requirement to specify all parent objects up to the database object, an object which must be unique on an SSAS instance. That's why we had to specify both the `DatabaseID` and the `CubeID` in this example.

If the query returns no errors it means it has successfully cleared the Analysis Services cache for that object.

### There's more...

Clearing the Analysis Services cache is not enough to guarantee the same performance testing conditions. There's also the file system cache which in turn consists of the Active Cache and the Standby Cache. To test MDX query performance on a true cold cache we have to either reboot the server or clear those caches too. Greg Galloway, one of the reviewers of this book, wrote a breakthrough article about that here:

<http://tinyurl.com/GregClearCache>

There are several ways, according to that article, of how you can clear those caches. One of them (and probably the most convenient one), is by using the stored procedure.

The next recipe shows how to register an assembly and use its stored procedures on the Analysis Services server. The example will feature the `ClearAllCaches` stored procedure of the *Analysis Services Stored Procedure Project*, a procedure which clears both the Analysis Services cache and the file system cache (Active and Standby). Here's a link to that site:

<http://tinyurl.com/ASSPCodePlex>

Once you register the `ASSP.dll` on your server (or database), call the procedure (in the MDX query window) as follows:

```
call ASSP.ClearAllCaches()
```

There can be permission issues with clearing the cache depending on the identity of the caller. It is well documented on the ASSP site.

### Objects whose cache can be cleared

Cube is just one of the four objects whose cache can be cleared. Other objects are database, measure group, and dimension. The appropriate commands are as follows:

- ▶ Database:

```
<Batch xmlns = "http://schemas.microsoft.com/
analysiservices/2003/engine">
  <ClearCache>
    <Object>
      <DatabaseID>Adventure Works DW 2008R2</DatabaseID>
```

```
</Object>
</ClearCache>
</Batch>

▶ Measure group:
<Batch xmlns =
  "http://schemas.microsoft.com/analysisservices/2003/engine">
  <ClearCache>
    <Object>
      <DatabaseID>Adventure Works DW 2008R2</DatabaseID>
      <CubeID>Adventure Works</CubeID>
      <MeasureGroupID>Sales Summary</MeasureGroupID>
    </Object>
  </ClearCache>
</Batch>

▶ Dimension:
<Batch xmlns =
  "http://schemas.microsoft.com/analysisservices/2003/engine">
  <ClearCache>
    <Object>
      <DatabaseID>Adventure Works DW 2008R2</DatabaseID>
      <DimensionID>Dim Product</DimensionID>
    </Object>
  </ClearCache>
</Batch>
```

Simply insert the ID of the object between its tags. All of the IDs can be found in the **Properties** windows for those objects except for the measure groups' object. The **Properties** window is not implemented for them. Luckily, that's the only object where it is allowed to use both the name and the ID without getting an error.

If you still want to use the ID for measure groups, right-click on the measure group in the **Object Explorer** and select **Script Measure Group As | DELETE To | New Query Editor Window**. There you'll find the ID for that measure group: Fact Sales Summary.

Use the **CREATE To** or **ALTER To** option instead of the **DELETE To** option and you'll wait a bit longer for the script to appear.

You can use this trick for any object. Just make sure you don't run the DELETE query!

Or, play it safe by sticking to the **Properties** window as explained earlier.

## Additional information

The following MDX query is recommended as the next step immediately after you clear the cache:

```
SELECT {} ON 0  
FROM [Adventure Works]
```

It forces the loading of the MDX script of the cube specified in the query. That also means no data is loaded in the cache unless the MDX script has named sets which, in order to be evaluated, require evaluation of measures. In that case some data could be loaded into the cache.

## Tips and tricks

It is possible to execute both commands in the same MDX query if you separate them using the GO statement. In other words, the MDX query accepts this specific XMLA batch; you also don't have to use the XMLA query.

The only difference is that the XMLA query formats the code and shows a more verbose result. That's why we've used it in this example.

This is also applicable to stored procedures.

### See also

The related recipe is *Using Analysis Services stored procedures*.

## Using Analysis Services stored procedures

Analysis Services supports both COM and CLR assemblies (DLL files) as an extension of its engine in which developers can write custom code. Once written and compiled, assemblies can be deployed to an Analysis Services instance.

Though the process of creating the assemblies is outside the scope of this book, using them is not outside its scope because of the benefits they bring you: the stored procedures implemented in those assemblies can be called from MDX queries, used in calculations, or triggered when a particular event occurs on the server.

If you haven't already read the previous recipe, do it now because this recipe continues where the previous one has ended. It shows you how to register a popular community assembly available on the Internet and use its stored procedures. The recipe focuses on the ClearAllCaches stored procedure in particular, which clears both the Analysis Services cache and the file system cache, therefore allowing BI developers and testers to perform an accurate query tuning.

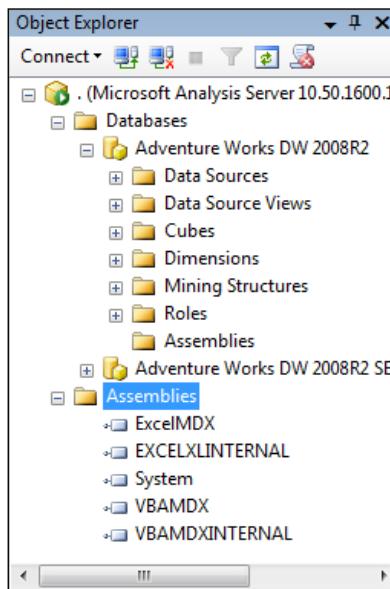
For those of you who are not familiar with stored procedures, it's worth saying that later sections of this recipe illustrate other types of usage of stored procedures. Ready to learn this? Then read on!

## Getting ready

Start the **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance.

Expand the **Databases** item in the **Object Explorer** so that you can see the **Adventure Works DW 2008R2** database. Expand the database and its **Assemblies** folder. It should be empty. No assemblies are registered there.

Now, expand the **Assemblies** folder of the server as displayed on the following image:



These assemblies, deployed and registered on the server during the installation, are available to the entire SSAS instance.

Let's see how to register a new assembly on this server.

As mentioned in the introduction, we're going to use the *Analysis Services Stored Procedure Project* assembly which can be downloaded from this location:

<http://tinyurl.com/ASSPCodePlex>

Notice the green **DOWNLOAD** button in the top-right part of the page. Click on it, read the license and if you agree, click on the **I Agree** button. If you're doing this on a server, make sure the site is added to the list of secured sites; otherwise you won't see the pop-up dialog.

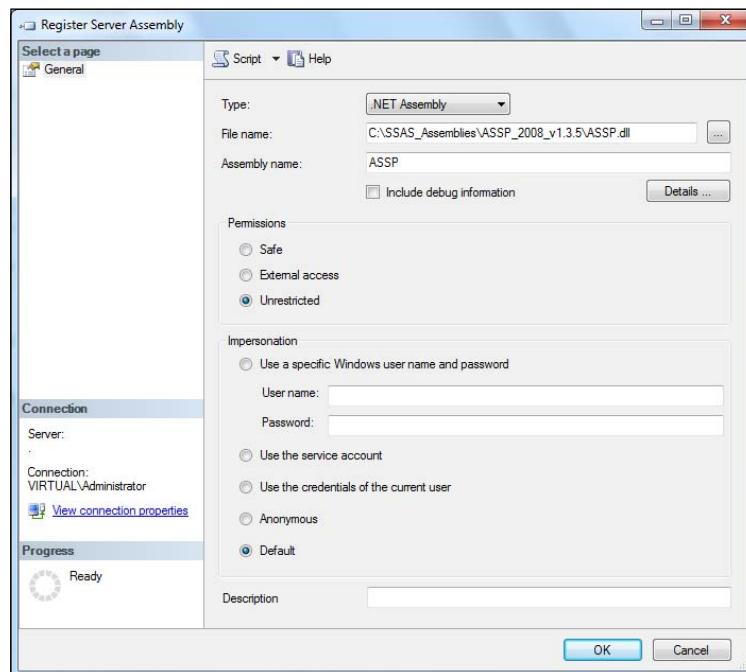
Save the file somewhere on your server (for example, to a folder like C:\SSAS\_Assemblies) and extract its contents once the download is finished. Then verify that the ASSP.dll file is inside.

You're ready to begin.

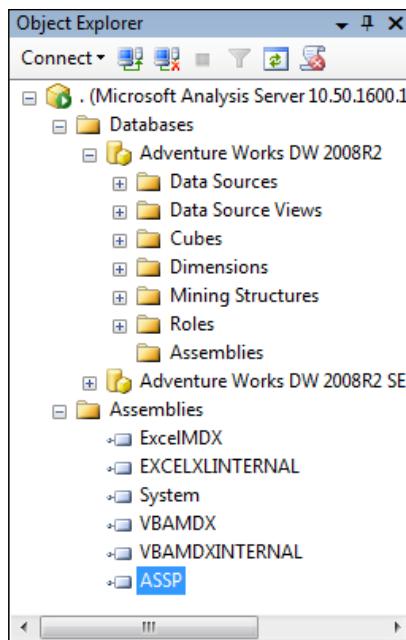
## How to do it...

Follow these steps to add a custom assembly to the Analysis Services instance:

1. Right-click the server's **Assemblies** folder and select the **New Assembly...** option.
2. Click on the button next to the **File Name** box and navigate to the ASSP.dll file you downloaded a moment ago.
3. Verify the assembly's properties using the **Details...** button. Then close the **Assembly Details** window.
4. The **Assembly name** can be changed if required. Here we'll leave it as it is.
5. Change the **Permissions** option to **Unrestricted** if company policy allows you to. Otherwise, some functions will be unavailable. The ASSP home page on CodePlex lists the permissions which are required by each of the functions.
6. Optionally, add a description for this assembly so that you know what it does.
7. The final configuration should look like this:



8. Close the window by clicking the **OK** button.
9. The assembly will be visible in the list of server's assemblies which means you're ready to use it:



10. Now start a new MDX query by right-clicking the **Adventure Works DW 2008R2** database in the **Object Explorer** and selecting **New Query**, then **MDX**.
11. Type this and then execute it.

```
call ASSP.ClearAllCaches()
```
12. The query might take a while to complete. Once it finishes, the result should say that the execution is complete.

You're done! You have successfully registered a custom assembly and tested it, proving that it works by clearing all cache that might corrupt the performance tuning process.

### How it works...

The process of registering an Analysis Services assembly is pretty straightforward. You select the file, set the required permissions, and optionally provide or change other information in the dialog. Once you're done, you're ready to use the assembly and the procedures it implements.

The assembly can be registered as a server-based assembly or as a database-specific assembly. Stored procedures in the server-based assembly can be used in any database on that SSAS instance. Database-specific assemblies can be used only in the database they've been registered at.

Some procedures can be activated using the `call` method, others as part of a query or calculation, depending on the result they return. In this recipe, we've used a procedure which executes a specific task, that's why we used the `call` method. In the following section we'll show how to use stored procedures as part of the query.

### There's more...

Here's an MDX query, actually two of them separated with a GO statement. Run them in SSMS:

```

WITH
MEMBER [Measures].[Sales Amount $] AS
    [Measures].[Sales Amount],
    BACK_COLOR = RGB(255, 255, 200)
SELECT
NON EMPTY
{ [Date].[Fiscal].[Fiscal Year].MEMBERS } *
{ [Measures].[Sales Amount $],
  [Measures].[Gross Profit Margin] } ON 0,
NON EMPTY
{ [Sales Territory].[Sales Territory Country]
  .[Sales Territory Country].MEMBERS } ON 1
FROM
[Adventure Works]
CELL PROPERTIES
VALUE,
FORMATTED_VALUE,
BACK_COLOR

GO

WITH
MEMBER [Measures].[Sales Amount $] AS
    [Measures].[Sales Amount],
    BACK_COLOR = RGB(255, 255, 200)
SELECT
NON EMPTY
Hierarchize(
ASSP.InverseHierarchility(
{ [Date].[Fiscal].[Fiscal Year].MEMBERS } *

```

## On the Edge

---

```
{ [Measures].[Sales Amount $],  
  [Measures].[Gross Profit Margin] } ) ) ON 0,  
NON EMPTY  
{ [Sales Territory].[Sales Territory Country]  
  .[Sales Territory Country].MEMBERS } ON 1  
FROM  
[Adventure Works]  
CELL PROPERTIES  
VALUE,  
FORMATTED_VALUE,  
BACK_COLOR
```

The first MDX query contains a crossjoin of fiscal years and measures. In the second, the crossjoin is reversed using the ASSP.InverseHierarchility() stored procedure. That procedure performs the commutation of members in the resulting tuple.

Commutation was not enough to make the result appealing in this case. We had to group measures by applying the built-in Hierarchize() function.

	FY 2006	FY 2006	FY 2007	FY 2007	FY 2008	FY 2008	FY 2009	FY 2009
	Sales Amount \$	Gross Profit Margin						
Australia	\$2,568,701.39	40.51%	\$2,099,585.43	41.91%	\$5,977,814.92	27.63%	\$9,234.23	54.30%
Canada	\$3,652,907.79	5.16%	\$5,920,179.83	6.16%	\$6,771,829.14	6.63%	\$10,853.70	56.41%
France	\$414,245.32	40.13%	\$2,061,420.07	15.04%	\$4,772,398.31	11.96%	\$3,491.95	57.46%
Germany	\$513,353.17	40.27%	\$593,247.24	40.77%	\$3,768,095.13	16.60%	\$3,604.83	57.89%
United Kingdom	\$550,507.33	40.33%	\$2,103,086.93	17.14%	\$5,012,905.37	16.20%	\$4,221.41	55.28%
United States	\$15,660,811.02	6.45%	\$20,906,285.30	6.65%	\$26,411,059.89	7.70%	\$19,434.51	55.58%

	Gross Profit Margin	Gross Profit Margin	Gross Profit Margin	Gross Profit Margin	Sales Amount \$	Sales Amount \$	Sales Amount \$	Sales Amount \$
	FY 2006	FY 2007	FY 2008	FY 2009	FY 2006	FY 2007	FY 2008	FY 2009
Australia	40.51%	41.91%	27.63%	54.30%	\$2,568,701.39	\$2,099,585.43	\$5,977,814.92	\$9,234.23
Canada	5.16%	6.16%	6.63%	56.41%	\$3,652,907.79	\$5,920,179.83	\$6,771,829.14	\$10,853.70
France	40.13%	15.04%	11.96%	57.46%	\$414,245.32	\$2,061,420.07	\$4,772,398.31	\$3,491.95
Germany	40.27%	40.77%	16.60%	57.89%	\$513,353.17	\$593,247.24	\$3,768,095.13	\$3,604.83
United Kingdom	40.33%	17.14%	16.20%	55.28%	\$550,507.33	\$2,103,086.93	\$5,012,905.37	\$4,221.41
United States	6.45%	6.65%	7.70%	55.58%	\$15,660,811.02	\$20,906,285.30	\$26,411,059.89	\$19,434.51

This example showed how to use a stored procedure inside an MDX query. We can apply it directly on a set on an axis or use other functions on top of it. The same is true for using procedures in MDX calculations. Try it!

## Tips and tricks

A good practice in creating custom SSAS assemblies is to create a procedure which returns all the available stored procedures in that assembly if there are many of them. This way you don't have to look at the documentation, as it's right there in front of you.

The ASSP assembly incorporates such procedure. It should be called as follows:

```
Call ASSP.ListFunctions()
```

The output should be as follows:

Assembly	Class	Method	ReturnType	Parameters
ASSP	StrToSet	KeysStrToSet	Set	Hierarchy as String, KeysComm...
ASSP	StrToSet	CompositeKeysStrToSet	Set	Hierarchy as String, KeysComm...
ASSP	WritebackWithAssignments	AssignValue	Void	subCube as String, valueToAssi...
ASSP	WritebackWithAssignments	AssignMDXExpression	Void	subCube as String, expressionT...
ASSP	LinkMember	HierarchyLinkMember	Member	m as Member, sHierarchyUniqu...
ASSP	LinkMember	LevelLinkMember	Member	m as Member, sLevelUniqueNa...
ASSP	PartitionHealthCheck	DiscoverPartitionSlices	DataTable	cubeName as String, measureG...
ASSP	PartitionCreator	CreatePartitions	Void	CubeName as String, MeasureG...
ASSP	PartitionCreator	CreatePartitions	Void	CubeName as String, MeasureG...
ASSP	PartitionCreator	CreateDistinctCountPartitions	Void	CubeName as String, MeasureG...
ASSP	PartitionCreator	CreateDistinctCountPartitions	Void	CubeName as String, MeasureG...
ASSP	PartitionCreator	CreateStringDistinctCountPartitions	Void	CubeName as String, MeasureG...

## Existing assemblies

The list of existing VBA functions can be found here:

<http://tinyurl.com/VBAFunctions>

It's worth repeating one more time – VBA functions are available to any cube once the SSAS instance is started. The list of functions in that document is applicable for SSAS 2008 R2 too.

VBA functions evaluate much slower than regular MDX functions. The SSAS team made an assessment of the often-used VBA functions and implemented them as part of the SSAS code base. In the previously mentioned document, these functions are marked in bold. They have better performance than the other functions on that list and hence can be considered "internal VBA functions."

Optimized VBA functions are also listed in the Books Online:

<http://tinyurl.com/Improved2008R2>

Excel functions, although registered, are unavailable until Excel is installed on the server. Then only do they become valid and applicable.

The System assembly contains data-mining functions.

Existing assemblies don't have a procedure which lists the functionality provided. However, there are certain DMV queries which can return the same. DMVs are covered later in this chapter.

### Additional information

Chapter 14 of the book *Microsoft SQL Server 2008 Analysis Services Unleashed* contains additional information about stored procedures as well as chapter 11 of the *Professional Microsoft SQL Server Analysis Services 2008 with MDX*. Finally, here's the MSDN reference for stored procedures:

<http://tinyurl.com/SSASStoredProcedures>

### See also

The recipe *Clearing Analysis Services cache* is related to this recipe.

## Executing MDX queries in T-SQL environments

Throughout this book, numerous recipes showed how to create various calculations, either directly in MDX queries or inside the MDX script. Prior to writing and running the queries, you naturally had to establish the connection to your Analysis Services server instance and click on the **New Query** icon which opened the SQL Server Management Studio's built-in MDX editor. The other option for running those queries which we didn't show in this book was to use an Analysis Services front-end/tool that allows writing and executing MDX queries.

To connect to data sources like Analysis Services, applications use providers. Relational database environments, on the other hand, allows us to use those providers to run distributed queries, also known as pass-through queries. This feature opens the window of possibilities for us. We can combine results from the cube with those in the data warehouse or simply get the flattened result of an MDX query and use it like any other result of T-SQL queries. True, we won't have a nice pane with the cube structure on our left to help us write MDX queries, but nothing stops us from writing them in MDX editor and then copy/paste working MDX queries inside the pass-through T-SQL queries.

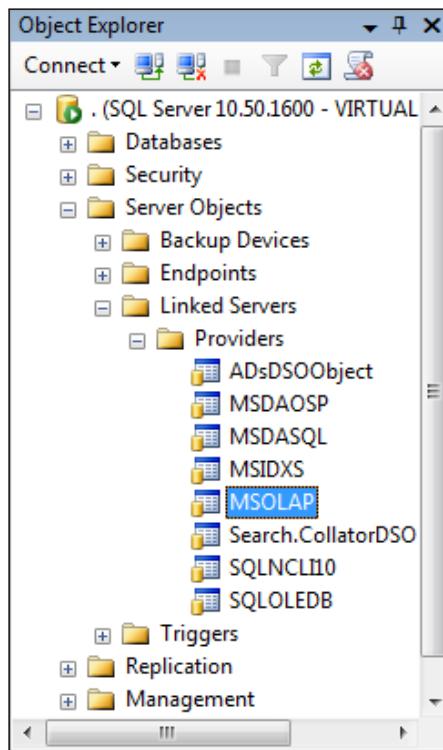
Let's see how it's done.

### Getting ready

Start the **SQL Server Management Studio** and connect to your SQL Server 2008 R2 database engine instance.

Start a new query by clicking on the **New Query** button.

Expand the **Server Objects** item in the **Object Explorer**. Expand the **Linked Servers** item and then the **Providers** item. Verify that the **MSOLAP** provider is there:



It should be there if you have installed at least the client connectivity for Analysis Services on your computer. If not, you need to install it from here:

<http://tinyurl.com/SQL2008R2FeaturePack>

Look for the OLE DB provider for SSAS there and choose the version of the provider (32-bit or 64-bit) that matches your computer's OS.

Next, double-click the **MSOLAP** provider and make sure that the **Allow inprocess** option is checked. Optionally check the **Dynamic parameter** option and close the window. Now you're ready to query your SSAS servers.

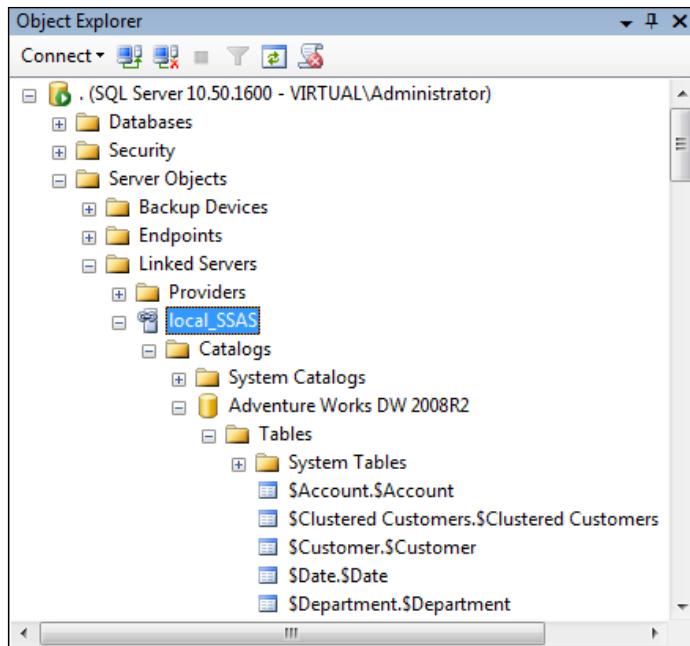
## How to do it...

Follow these steps to perform a SQL Server distributed query with your SSAS instance:

1. Register your SSAS instance as a linked server by running the following command:

```
EXEC sp_addlinkedserver
    @server='local_SSAS',
    @srvproduct='',
    @provider='MSOLAP',
    @datasrc='.',
    @catalog='Adventure Works DW 2008R2'
```

2. Refresh the **Linked Servers** item in the **Object Explorer** by right-clicking on it and selecting **Refresh**.
3. If everything is successful, you should see your linked server there. You should also be able to navigate that server and its tables as shown in the following image:



4. Run your MDX query as a distributed query like this:

```
Select * From OpenQuery(local_SSAS,
'
SELECT
NON EMPTY
{ [Measures].[Sales Amount],
```

```

[Measures].[Gross Profit Margin] } *
{ [Product].[Category].MEMBERS } ON 0,
NON EMPTY
{ [Product].[Color].MEMBERS } ON 1
FROM
[Adventure Works]
WHERE
( [Date].[Fiscal].[Fiscal Year].&[2007] )
'
)

```

5. The result will look like this:

	[Product].[Color].[Color], [MEMBER_CAPTION]	[Measures].[Sales Amount] [Product].[Category].[All Products]	[Measures].[Sales Amount] [Product].[Category]&[4]
1	NULL	33683804.815100692	124433.3514000001
2	Black	16189726.403700007	31866.829899999993
3	Blue	33795.263500000008	33795.263500000008
4	Multi	311717.5149000003	NULL
5	NA	245717.8570000005	29739.907299999999
6	Red	8801936.6286001001	29031.350699999995
7	Silver	5286484.9835999962	NULL
8	Yellow	2814426.1638000011	NULL

6. You've successfully executed your MDX query against your SSAS server from the database engine environment and the server returned the result set in tabular format. The specifics of this output will be discussed in later sections of this recipe.

### How it works...

The best way to use an SSAS server in the database engine environment is to register it as a linked server. That can be done either by running the `sp_addlinkedserver` stored procedure, as we did in this example, or by manually registering the server using the **New Linked Server...** option on the **Linked Servers** item. Of course, you have to right-click the item first in order to see that option.

Registration is a one-time process which requires the following information:

1. The name for the server, the way it will be referred to later in the code (here we've named it **local\_SSAS**)
2. The provider for the server (in this case, that's always the **MSOLAP** provider)
3. SSAS server and the instance (here we've used the local server with the default instance which can be specified as . (dot), **localhost**, **local**, **(local)**, or simply the name of the computer)
4. SSAS database we want to connect to (here, we use the **Adventure Works DW 2008R2** database, the one we've used throughout this book)

If you enter the information correctly and have the correct MSOLAP provider on your computer, then you should be able to see and browse the new linked server. The idea of browsing is merely a test to see whether everything works as expected with the registered linked server.

The MDX that goes into the `OpenQuery` command should be created and tested in a separate query, the one that connects to the SSAS instance, not the relational database instance. The MDX should then be copy-pasted as the second argument of the `OpenQuery` command. To keep the recipe short, we've skipped that part and wrote that MDX query directly.

The result of the `OpenQuery` over MSOLAP provider is a bit different than the result returned by the same query executed against SSAS instance directly. The `All` member is specified only for hierarchies on columns; in rows, it is returned as a `NULL` value (see the first column in the previous image, that's the `All Products` member of the **Color** hierarchy).

Next, notice that values are flattened to fit the relational form. The columns are actually a combination (a tuple) of a measure and a dimension. In respect to that, here are some optimization tips.

Try to put measures on columns and everything else on rows, just like Reporting Services expects it when it uses SSAS as a data source. In this example, we've deliberately made a bad type of query in order to show you what happens if you put more hierarchies on columns (notice the names of the columns).

Formats are not preserved, but you can always convert column values in database types that suit you. Naturally, you would do that in the outer T-SQL query, not in the inner MDX query.

What's important is that you've been able to get the data from your cube in another environment. In case you need to bring the data back to the, let's say Data Warehouse, now you know you can do it.

### There's more...

If you don't want to register a linked server, you must configure some of the server options in order to be able to query the SSAS instance. This can be done either visually by changing some of the server and provider options, or using the script. We'll show them both, in that order.

Right-click on the server in the **Object Explorer** pane and then choose **Facets**. In the **Facet** dropdown, select the last item, **Surface Area Configuration**. Then change the first property, the **AdHocRemoteQueriesEnabled**, to **True**. This enables the SQL Server to run distributed queries without a linked server.

Close the window using **OK** button.

The visual option presented earlier is relatively easy to remember. For those of you who need or prefer the script version, here's the alternative:

```
USE [master]
GO
sp_configure 'Show Advanced Options', 1
GO
RECONFIGURE WITH OverRide
GO
sp_configure 'Ad Hoc Distributed Queries', 1
GO
RECONFIGURE WITH OverRide
GO
EXEC master.dbo.sp_MSset_oledb_prop
    'MSOLAP',
    'AllowInProcess', 1
GO
EXEC master.dbo.sp_MSset_oledb_prop
    'MSOLAP',
    'DynamicParameters', 1
GO
```

Now you can query your SSAS instance using the OpenRowset syntax also:

```
Select * From OpenRowset('MSOLAP',
    'Data Source=.;Initial Catalog=Adventure Works DW 2008R2;',
    'SELECT [Measures].DefaultMember ON 0
     FROM [Adventure Works]')
```

### Additional information

Here are useful links regarding the OpenQuery and OpenRowset commands, linked servers, and other terms mentioned inside this recipe:

- ▶ Accessing External Data: <http://tinyurl.com/ExternalData>
- ▶ Adding a linked server: <http://tinyurl.com/sp-addlinkedserver>
- ▶ Linked Server properties: <http://tinyurl.com/LinkedServerProperties>
- ▶ OpenQuery: <http://tinyurl.com/OpenQueryTSQL>
- ▶ OpenRowset: <http://tinyurl.com/OpenRowsetTSQL>
- ▶ Ad hoc distributed queries: <http://tinyurl.com/DistributedQueryOptions>

### Useful tips

Removing a linked server is easy. Just right-click on that linked server in the **Object Explorer** and choose the **Delete** option. Confirm and you're done.

You can also script the linked server for future use using the **Script Linked Server as...** option. That feature is available in the context menu (by right-clicking the linked server).

### Accessing Analysis Services 2000 from a 64-bit environment

The technique presented in this recipe can also be useful to make the SSAS 2000 server accessible from a 64-bit server. The 32-bit SQL Server 2005 or later can be used as a gateway by using a linked server.

### Troubleshooting the linked server

If you have trouble using a linked server to SSAS, here are the things you should check:

- ▶ Are you using the correct MSOLAP provider to connect to your server?
- ▶ Have you entered the correct SSAS instance?
- ▶ Have you used the correct database?
- ▶ Do you have all the necessary server options enabled for the type of the query you're trying to make?
- ▶ Is the SSAS database processed?
- ▶ Do you have permissions to connect and use that database?

You can eliminate those questions one by one by verifying them, that is, using SSMS to connect to SSAS database, opening configuration dialogs, and so on). This way you will narrow the potential causes and soon be able to focus on solving the rest of the items.

### See also

The recipe *Using SSAS Dynamic Management Views (DMVs) to fast-document a cube* is related to this recipe because it shows the opposite – how to run SQL-like queries in an MDX query environment.

## Using SSAS Dynamic Management Views (DMV) to fast-document a cube

Dynamic Management Views are Analysis Services schema rowsets (XML metadata) exposed as tables which can be queried like any other relational table, using SQL queries. There are restrictions in what we can do, what types of queries and data manipulations are available to us when we use DMVs. In general, the idea is to expose several collections of information about SSAS and to be able to browse those using SQL queries.

The first collection of DMVs is the DBSCHEMA collection which consists of 4 DMVs. Its purpose is to provide various metadata about tables and other relational objects that SSAS databases are built from. For example, the list of columns in dimension tables in DSV.

The second collection is the MDSchema collection. This collection exposes various metadata about objects in SSAS databases. For example, we can find out which databases, cubes, dimensions, measures, levels, members, actions, and so on. there are and a lot of information about them.

A part of the MDSchema is dedicated to data mining, the part which has the MINING term inside the DMV's name.

Lastly, the biggest collection is the DISCOVER collection. It tells us how many active connections the server has, which queries are running at the moment, which sessions are active, and various information about the memory, CPU usage, and other server resources.

DMVs can be very useful for documenting SSAS databases, monitoring usage and activity, being the source for operational monitoring reports, and other purposes. In short, they are important enough to be covered in a recipe. In fact, two of them. This recipe shows how to use them to document the cube. The next one deals with monitoring the activity and usage.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

This is the environment where you normally write MDX queries. However, the SSAS server recognizes SQL queries that use DMVs and redirects them to the part of the engine that knows how to interpret them and what result to return.

Here's the first query we're going to run:

```
select * from $system.Discover_Schema_Rowsets
```

## On the Edge

---

When run, this query returns all available schema rowsets. It can be a good starting point to use other DMVs because it lists them and their restrictions.

The screenshot shows a SQL Server Management Studio (SSMS) results grid. The title bar has tabs for 'Messages' and 'Results'. The results grid displays a table with the following columns: SchemaName, SchemaGuid, Restrictions, Description, and RestrictionsMask. The data rows include various system schema names like DBSCHEMA\_CATALOGS, DBSCHEMA\_TABLES, etc., each with its corresponding GUID, restriction count, description, and mask value.

SchemaName	SchemaGuid	Restrictions	Description	RestrictionsMask
DBSCHEMA_CATALOGS	c8b52211-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	1	
DBSCHEMA_TABLES	c8b52229-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	31	
DBSCHEMA_COLUMNS	c8b52214-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	31	
DBSCHEMA_PROVIDER_TYPES	c8b5222c-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	3	
MDSHEMA_CUBES	c8b522d8-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	31	
MDSHEMA_DIMENSIONS	c8b522d9-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	127	
MDSHEMA_HIERARCHIES	c8b522da-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	511	
MDSHEMA_LEVELS	c8b522db-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	1023	
MDSHEMA_MEASURES	c8b522dc-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	255	
MDSHEMA_PROPERTIES	c8b522dd-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	8191	
MDSHEMA_MEMBERS	c8b522de-5cf3-11ce-adef-500aa0044773d	[+] Restrictions	16383	
MDSHEMA_FUNCTIONS	a07cc07-8148-11d0-87bb-00c04fc33942	[+] Restrictions	15	
MDSHEMA_ACTIONS	a07cc08-8148-11d0-87bb-00c04fc33942	[+] Restrictions	511	
MDSHEMA_SETS	a07cc0b-8148-11d0-87bb-00c04fc33942	[+] Restrictions	255	
DISCOVER_INSTANCES	20518699-2474-4c15-9885-0e947ec7a7e3	[+] Restrictions	1	
MDSHEMA_KPIS	2ae44109-ed3d-4842-b1f6-b694d1cb0e3f	[+] Restrictions	63	
MDSHEMA_MEASUREGROUPS	e1625ebff96-42fdbea6-db90adafdf96b	[+] Restrictions	15	
MDSHEMA_MEASUREGROUP_DIMENSIONS	a07cc03-8148-11d0-87bb-00c04fc33942	[+] Restrictions	63	
MDSHEMA_INPUT_DATASOURCES	a07cc02-8148-11d0-87bb-00c04fc33942	[+] Restrictions	15	
DMSHEMA_MINING_SERVICES	3add8a95-d8b9-11d2-8d2a-00e029154fde	[+] Restrictions	3	

In this recipe we're going to use the MDSHEMA collection starting with the **MDSHEMA\_LEVELS** DMVs which list all the attributes in a particular cube.

### How to do it...

Follow these steps to get detailed information about attributes in an SSAS database, cube, or dimension, depending upon how you set the restrictions.

1. Execute the following query:

```
SELECT
    *
FROM
    $system.mdschema_LEVELS
WHERE
    [CATALOG_NAME] = 'Adventure Works DW 2008R2'
    AND LEFT(CUBE_NAME, 1) <> '$' -- avoid dimension cubes
    AND [LEVEL_TYPE] <> 1 -- avoid root levels
    AND LEVEL_IS_VISIBLE -- avoid hidden levels
```

2. A table with many columns and rows is returned. This represents detailed information about the attributes visible in the **Adventure Works DW 2008R2** database and its cube and dimensions:

CATALOG_NAME	SCHEMA_NAME	CUBE_NAME	DIMENSION_UNIQUE_NAME	HIERARCHY_UNIQUE_NAME
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Account Number]
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Account Type]
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Accounts]
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Accounts]
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Accounts]
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Accounts]
Adventure Works DW 2008R2		Adventure Works	[Account]	[Account].[Accounts]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[City]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Commute Distance]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Country]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Customer]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Customer Geography]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Customer Geography]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Customer Geography]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Customer Geography]
Adventure Works DW 2008R2		Adventure Works	[Customer]	[Customer].[Education]

3. Scroll the table in both directions to familiarize yourself with the contents of that table.
4. Right-click on the result, **Select All**, copy the contents, and paste it somewhere where you will assemble this data into documentation.
5. Repeat the process with the other MDSchema DMVs. To remove the conditions that don't fit, run the query with no restrictions and then add those restrictions that you think will help you get the result you need. Skip the MEMBERS DMV; it might be huge and it's not like you will need it in the documentation.

### How it works...

Running a query with a DMV inside is simple. The biggest challenge is to find out, or remember, which schema rowsets can be used. The good thing is that you need to memorize only a single DMV, the `$system.Discover_Schema_Rowsets` DMV.

The `$system` schema is common to all DMVs, so that shouldn't be hard to remember. The `Discover_Schema_Rowsets` is the phrase we need to memorize in order to use DMVs without the documentation. Running this DMV returns the names and restrictions for all other DMVs.

As we said in the introduction, there are 3 collections of DMVs. In this recipe we've used the `MDSchema` which is a shortcut for multidimensional schema. This collection tells us everything about SSAS databases and their objects. By repeatedly collecting results from DMVs in that collection which are important to us, we can assemble the documentation quickly and relatively easily.

Conditions applied to a particular DMV serve the purpose of limiting the result to only those rows that we need. For example, in this recipe we've deliberately removed dimension cubes, cubes starting with the \$ prefix. We've also focused ourselves on a single database and we've excluded all root and hidden levels, so that our documentation becomes smaller and less distracting. It is up to you whether you'll use the same conditions or not.

Conditions for other DMVs can be set after each of them is run without any conditions and by analyzing which of them can be applied in a particular case.

The good thing is that once you make a set of MDSHEMA DMV queries that suits your needs, you can use them on any SSAS database. In other words, it's all about preparation. This recipe showed you how.

### There's more...

You can execute queries with DMVs as distributed queries (see previous recipe). All you have to do is either register a linked server or enable the *Ad Hoc Distributed Queries* option using the T-SQL script presented in the previous recipe and then run an `OpenQuery` or `OpenRowset` query.

Here's the query that uses the linked server `local_SSAS` defined in the previous recipe and returns lots of information about dimensions in the Adventure Works DW 2008R2 database.

```
Select * From OpenQuery(local_SSAS,
'
SELECT *
FROM
$system.mdschema_DIMENSIONS
WHERE
[CATALOG_NAME] = ''Adventure Works DW 2008R2''
AND LEFT(CUBE_NAME, 1) <> '$' -- avoid dimension cubes
AND DIMENSION_IS_VISIBLE -- avoid hidden dimensions
AND DIMENSION_TYPE <> 2 -- avoid measures
')
'
```

Notice that you have to use double quotes around the database name and the \$ sign because the MDX query is one big string now. We don't want to terminate it, we want to emphasize there's a quote inside and the way we do it is by using another quote next to it.

This is what you get when you run the query:

	CATALOG_NAME	SCHEMA_NAME	CUBE_NAME	DIMENSION_NAME	DIMENSION_UNIQUE_NAME
1	Adventure Works DW 2008R2	NULL	Adventure Works	Account	[Account]
2	Adventure Works DW 2008R2	NULL	Adventure Works	Customer	[Customer]
3	Adventure Works DW 2008R2	NULL	Adventure Works	Date	[Date]
4	Adventure Works DW 2008R2	NULL	Adventure Works	Delivery Date	[Delivery Date]
5	Adventure Works DW 2008R2	NULL	Adventure Works	Department	[Department]
6	Adventure Works DW 2008R2	NULL	Adventure Works	Destination Currency	[Destination Currency]
7	Adventure Works DW 2008R2	NULL	Adventure Works	Employee	[Employee]
8	Adventure Works DW 2008R2	NULL	Adventure Works	Geography	[Geography]
9	Adventure Works DW 2008R2	NULL	Adventure Works	Internet Sales Order Details	[Internet Sales Order Details]
10	Adventure Works DW 2008R2	NULL	Adventure Works	Organization	[Organization]
11	Adventure Works DW 2008R2	NULL	Adventure Works	Product	[Product]
12	Adventure Works DW 2008R2	NULL	Adventure Works	Promotion	[Promotion]
13	Adventure Works DW 2008R2	NULL	Adventure Works	Reseller	[Reseller]
14	Adventure Works DW 2008R2	NULL	Adventure Works	Reseller Sales Order Det...	[Reseller Sales Order Details]
15	Adventure Works DW 2008R2	NULL	Adventure Works	Sales Channel	[Sales Channel]

Again, only a small portion of information is visible in this image. You will be able to scroll in both directions and see what else is available.

### Tips and tricks

When you run the `$system.Discover_Schema_Rowsets` DMV, you can expand the **Restrictions** item for a particular DMV in order to find out which columns can be used to set the filter for a particular DMV.

You can also use column names instead of \* in order to avoid empty columns.

You can't convert data types directly in DMVs, but you can do that in the outer part of the OpenQuery query. Remember that for certain joins you will need to convert the ntext data type to nvarchar(max) data type.

You can join multiple DMVs using the OpenQuery statement and create interesting reports.

### Warning!

The connection determines the context!

In other words, when you connect to, let's say the Adventure Works DW 2008R2 SE database (Standard Edition sample), many DMVs will return metadata about that database only. You won't be able to see Adventure Works DW 2008R2 database objects.

For example, if you use the condition `where [CATALOG_NAME] = 'Adventure Works DW 2008R2'` and get nothing as the result of your DMV query, it may be a clue that you're not connected to the right database. Verify that in the drop-down combo.

Always make sure you're connected to the right database when you use DMVs.

## More information

The MSDN library contains more information about DMVs and the columns they return:

<http://tinyurl.com/MSDN-DMVs>

Vincent Rainardi has a good overview of DMVs here:

<http://tinyurl.com/VincentDMVs>

Vidas Matelis aggregates DMV-based articles on his SSAS-Info portal here:

<http://tinyurl.com/SSAS-InfoDMVs>

Finally, there are a few interesting DMV-related community samples at Codeplex (see everything related to monitoring or DMVs):

<http://tinyurl.com/SSASCodePlex>

## See also

The recipe *Using SSAS Dynamic Management Views (DMVs) to monitor activity and usage* is related to this recipe as well as the *Executing MDX queries in T-SQL environment* recipe.

# Using SSAS Dynamic Management Views (DMVs) to monitor activity and usage

The previous recipe explained what DMVs are and illustrated how they can be used to get various information about SSAS objects using the MDSHEMA collection of DMVs.

In this recipe we're shifting the focus to another collection of DMVs, the DISCOVER collection which is the largest collection of DMVs. DMVs in that collection can be used to monitor the usage of the cube across all its aspects.

## Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

As explained in the previous recipe, the MDX query environment supports SQL queries that use DMVs. We should be in the right spot, so let's run a query to verify that:

```
select * from $system.Discover_Schema_Rowsets
```

When run, this query returns all available schema rowsets. As mentioned in the previous recipe, this can be a good starting point for using other DMVs because it lists them and their restrictions.

Now, scroll through that list and notice the names of the DMVs from the **DISCOVER** collection. As a matter of fact, why not show another query we can write in order to isolate DMVs in that schema:

```
select * from $system.Discover_Schema_Rowsets
where left(SchemaName, 8) = 'DISCOVER'
order by SchemaName
```

The previous query lists all DMVs in that schema. The list is big, so only a portion of it is shown in the next image, but it should be enough to give you an idea of what you can get using the DMVs in this collection:

SchemaName
DISCOVER_COMMAND_OBJECTS
DISCOVER_COMMANDS
DISCOVER_CONNECTIONS
DISCOVER_DATASOURCES
DISCOVER_DB_CONNECTIONS
DISCOVER_DIMENSION_STAT
DISCOVER_ENUMERATORS
DISCOVER_INSTANCES
DISCOVER_JOBS
DISCOVER_KEYWORDS
DISCOVER_LITERAL
DISCOVER_LOCATIONS
DISCOVER_LOCKS
DISCOVER_MASTER_KEY
DISCOVER_MEMORYGRANT
DISCOVER_MEMORYUSAGE
DISCOVER_OBJECT_ACTIVITY
DISCOVER_OBJECT_MEMORY_USAGE
DISCOVER_PARTITION_DIMENSION_STAT
DISCOVER_PARTITION_STAT
DISCOVER_PERFORMANCE_COUNTERS

OK, let's see some of the most interesting ones in action.

## How to do it...

Follow these steps to see information about the activity on your SSAS server. The queries will return unique results based on your activity on the server. Therefore only the queries are presented; there are no screenshots.

1. Execute the following query to see the list of connections:

```
select * from $system.DISCOVER_CONNECTIONS
```

2. Execute the following query to see the list of sessions. You can additionally limit it using a particular connection:

```
select * from $system.DISCOVER_SESSIONS
```

3. Execute the following query to see the list of commands. You can additionally limit the list using a particular session:

```
select * from $system.DISCOVER_COMMANDS
```

4. Execute the following query to see the list of commands in more detail, by showing which objects they use. Naturally, you can limit this by session.

```
select * from $system.DISCOVER_COMMAND_OBJECTS
```

5. Execute the following query to see the accumulated results of users' activity since the last restart of the service. Here, for example, you can order the query to see which aggregations and objects are hit or missed the most as well as the CPU activity spent on each object so far (since the restart of the SSAS service).

```
select * from $system.DISCOVER_OBJECT_ACTIVITY
```

6. Execute the following query to see the detailed list of memory usage per object:

```
select * from $system.DISCOVER_MEMORYUSAGE
```

7. Execute the following query to see the spread of shrinkable and non-shrinkable memory usage per object:

```
select * from $system.DISCOVER_OBJECT_MEMORY_USAGE
```

8. Explore the other DMVs in the **DISCOVER** collection in order to see what they return and if you can benefit from that information in the monitoring solution you plan to build. If you encounter an error, skip that DMV. The explanation will be covered in later sections of this recipe.

## How it works...

Managing the DISCOVER collection DMVs is easy when you learn what they represent and how are they related. Until then you might be a little lost.

Unlike the MDSHEMA DMVs which follow the natural path of objects and their descending objects (for example, the database, dimensions and cubes, hierarchies of dimensions, levels of hierarchies, and so on.), here we have several entry points and their paths to a more detailed DMV.

We started the example with the list of connections returned by the DISCOVER\_CONNECTIONS DMV. Connections have sessions; therefore, the next DMV we covered was the DISCOVER\_SESSIONS DMV. Naturally, we can create a query that joins those two results in one if we want to and this can be done using the OpenQuery statement (joins will be covered in later sections of this recipe).

The DISCOVER\_COMMANDS and the more detailed DISCOVER\_COMMAND\_OBJECTS DMVs are here to give us more details about the queries running in each session as well as objects associated to returning the result of a particular command (query).

The DISCOVER\_OBJECT\_ACTIVITY DMV is great for performance tuning your cube. It includes information about aggregation hits, aggregation misses, object hits and misses, CPU time spent so far (from the restart of SSAS service) on a particular object, and so on. Monitoring this DMV on a regular basis can give you valuable information whether there's a place for improving your cube by building additional aggregations or by warming up the cache.

The last two DMVs covered in this recipe were DMVs that inform us about the memory usage in case we want to fine-tune SSAS settings regarding memory consumption and its limits.

### There's more...

Some DMVs require restrictions. They cannot be run without them. Here is the list of DMVs that require restrictions and that are part of the DISCOVER collection:

- ▶ \$system.DISCOVER\_DIMENSION\_STAT
- ▶ \$system.DISCOVER\_INSTANCES
- ▶ \$system.DISCOVER\_PARTITION\_DIMENSION\_STAT
- ▶ \$system.DISCOVER\_PARTITION\_STAT
- ▶ \$system.DISCOVER\_PERFORMANCE\_COUNTERS
- ▶ \$system.DISCOVER\_XML\_METADATA

Here's the error we get when we run the first DMV in that list:

```
Executing the query ...
Errors from the SQL query module: The 'DIMENSION_NAME' restriction
is required but is missing from the request. Consider using
SYSTEMRISTRICTSCHEMA to provide restrictions.
```

```
Execution complete
```

The list of parameters for restrictions is visible when you run the DISCOVER\_SCHEMA\_ROWSETS DMV, the first one we mentioned in this and in the previous recipe. Yes, that's the starting DMV, the one which tells you which DMVs are available in the system.

The usage is the following:

We need to wrap the DMV and its required parameters inside the SystemRestrictSchema function. Here's how:

```
select * from
SystemRestrictSchema(
    $system.DISCOVER_DIMENSION_STAT,
    DATABASE_NAME='Adventure Works DW 2008R2',
    DIMENSION_NAME='Customer')
order by ATTRIBUTE_NAME
```

This query returns all attributes in the Customer dimension, real attributes and properties. Additionally, the cardinality of each attribute is displayed.

You could think that this type of DMV fits better in the MDSHEMA, not the DISCOVER schema. Quite right. There, in the MDSHEMA, we have two similar DMVs, but none of them is the same as this DMV. The MDSHEMA\_LEVELS returns real attributes and their cardinality (and a bunch of other information), but it doesn't return the properties. On the other hand, they can be returned together with real attributes in another MDSHEMA DMV, the MDSHEMA\_PROPERTIES DMV. However, here we don't have any cardinality.

To summarize, some DMVs overlap. Look for the DMV that returns the exact information you need. In case none of them do, see if you can join them and then take whatever you need.

Speaking of joins, here's how you would do that.

You can execute queries with DMVs being the pass-through queries (see the recipe *Executing MDX queries in T-SQL environments*). All you have to do is either register a linked server or enable the *Ad Hoc Distributed Queries* option using the T-SQL script shown in that recipe and then running an OpenQuery or OpenRowset query.

Here's the query that uses the linked server local\_SSAS defined in the recipe *Executing MDX queries in T-SQL environments*, two recipes back from this recipe. Don't forget to run it in the **Database Engine Query** window, not the **Analysis Services MDX Query** window.

```
Select * from OpenQuery(local_SSAS,
    'select * from $system.DISCOVER_CONNECTIONS') c
inner join OpenQuery(local_SSAS,
    'select * from $system.DISCOVER_SESSIONS') s
    on s.SESSION_CONNECTION_ID = c.CONNECTION_ID
inner join OpenQuery(local_SSAS,
    'select * from $system.DISCOVER_COMMAND_OBJECTS') co
    on co.SESSION_SPID = s.SESSION_SPID
```

The result will be unique to the usage pattern of your cube and will show information about objects that have recently been queried.

## See also

The recipe *Using SSAS Dynamic Management Views (DMVs) to fast-document a cube* is related to this recipe as well as the *Executing MDX queries in T-SQL environments* recipe.

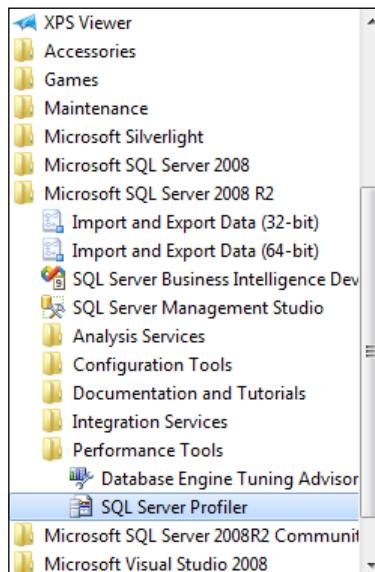
## Capturing MDX queries generated by SSAS front-ends

Some tools allow you to write your own MDX queries, others generate them for you. If you want to know what these other MDX queries look like, you need another tool which will tell you that. One such tool is the SQL Server Profiler which comes as a part of the SQL Server installation. Others might come as an add-in to the application that generates MDX queries. In other words, there's always a way.

In this recipe we're going to show how to capture the MDX query that has been sent by an application to the server.

## Getting ready

**SQL Server Profiler** can be found in the **Performance Tools** folder of the **Microsoft SQL Server 2008 R2** folder in the list of programs available on your computer.



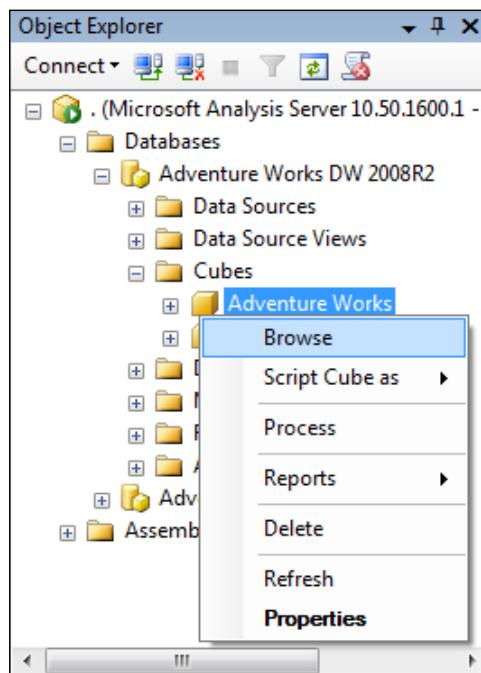
Start the Profiler, click on the **File** menu, **Templates**, and then choose the **New Template...** item.

Select **Microsoft SQL Server 2008 R2 Analysis Services** for the **Server Type**. Provide the name for the template: **MDX queries**. If you rarely perform other activities in the Profiler, you can optionally make this template the default template by selecting the appropriate checkbox in that window.

Next, go to the other tab in this window, the **Events Selection** tab. Expand **Queries Events** and select all items there. If you want to know more internal details about how the query performs, select items in the **Query Processing** category. See *Using SQL Profiler* in the MSDN library by clicking the **Help** button if you need additional information.

Finally, click on the **Save** button. The template is saved now and ready to be used every time we need it in the future.

Before we can see how it would go, we have to prepare something else. We need to simulate a working environment where users are browsing the cubes and analyze the data in their front-ends which in turn generate MDX queries we can capture. Since we're not in this situation, we're going to simulate it using the **SQL Server Management Studio** (SSMS) and its **Cube Browser**. Start the SSMS and open the Adventure Works cube for browsing:

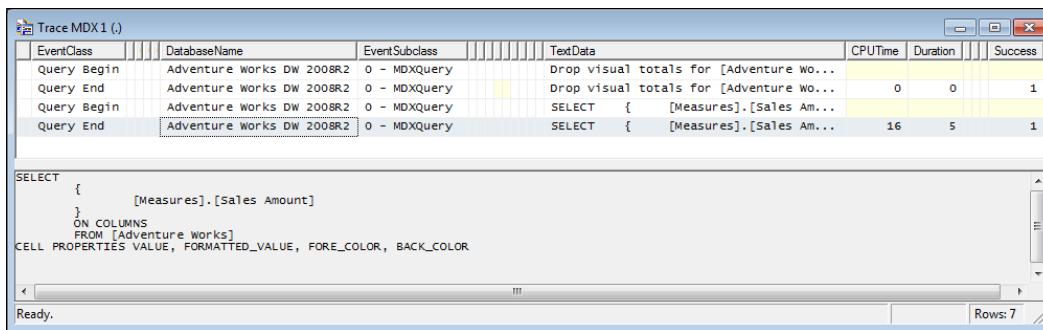


Now we're ready to go.

## How to do it...

Follow these steps to capture the MDX query sent to a cube on an Analysis Services instance:

1. Click on the **File** menu and then choose the **New Trace...** item.
2. Verify that the server type is set to **Analysis Services** (or change it if it's not) and verify the name of the server you're connecting to. Then click the **Connect** button.
3. In the **Trace Properties** window select your MDX queries template and name the trace appropriately, for example, **Trace MDX 1**.
4. Optionally, you can choose to save the trace results in a file or a table by selecting the appropriate option in the **General** tab of this window. Pay attention to performance when you write log information into a table.
5. In the **Events Selection** tab, you can change the events you're planning to track by selecting the **Show all events** option (and **Show all columns** option, for more customization of this trace). We are going to let it be as it is this time and click **Run**.
6. Maximize the inner window to see the results better.
7. If you're alone on this server, nothing will happen. That's why we're going to use SSMS and its **Cube Browser** to simulate users querying the cube. If on the other hand you're tracking an active server, you'll see events appearing in your running trace.
8. Expand the **Measures** item in SSMS's **Cube Browser** control and drag the **Sales Amount** measure, found under the **Sales Summary** folder, in the **Drop Totals or Details Field Here** part of the pivot table.
9. There are new events captured in your trace, as displayed in this image. It looks like there were 2 queries issued, one dropping the visual totals (visible in the **TextData** column in the first two rows) and the other returning the value for the measure we've put in the pivot.



The screenshot shows the 'Trace MDX 1' window with a grid of captured events and their corresponding MDX queries below.

EventClass	DatabaseName	EventSubclass	TextData	CPUTime	Duration	Success
Query Begin	Adventure Works DW 2008R2	0 - MDXQuery	Drop visual totals for [Adventure wo...]			
Query End	Adventure Works DW 2008R2	0 - MDXQuery	Drop visual totals for [Adventure wo...]	0	0	1
Query Begin	Adventure Works DW 2008R2	0 - MDXQuery	SELECT { [Measures].[Sales Am...]			
Query End	Adventure Works DW 2008R2	0 - MDXQuery	SELECT { [Measures].[Sales Am...]	16	5	1

Below the grid, the captured MDX queries are displayed:

```

SELECT
{
    [Measures].[Sales Amount]
}
ON COLUMNS
FROM [Adventure Works]
CELL PROPERTIES VALUE, FORMATTED_VALUE, FORE_COLOR, BACK_COLOR
  
```

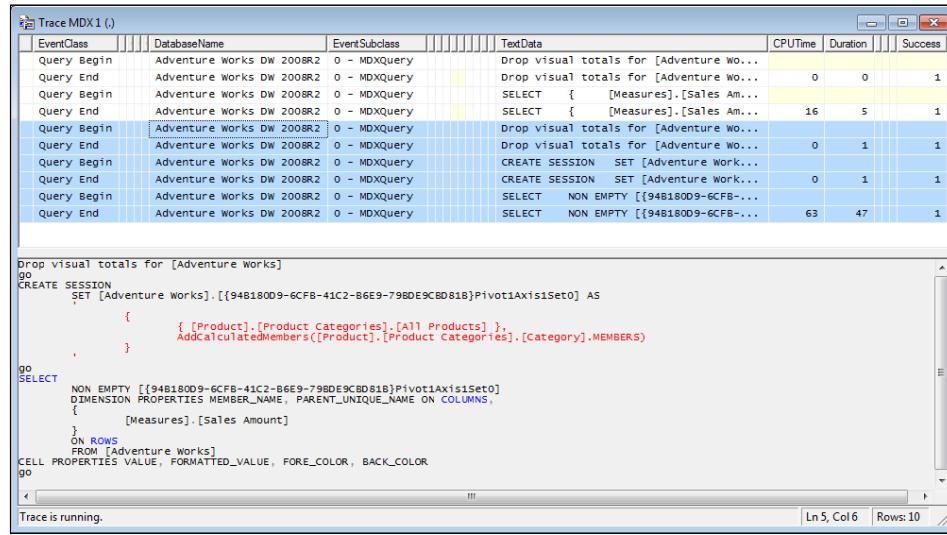
Ready. Rows: 7

10. Expand the **Product** dimension. Drag and drop the **Product Categories** user hierarchy on rows.

On the Edge

---

11. There are again new trace events, 6 of them this time. In total 3 new queries were issued. If you want to see them together, select those rows like the following image suggests:



The screenshot shows the Trace MDX1 window with 6 trace events listed in the grid. The events are:

- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery
- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery
- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery

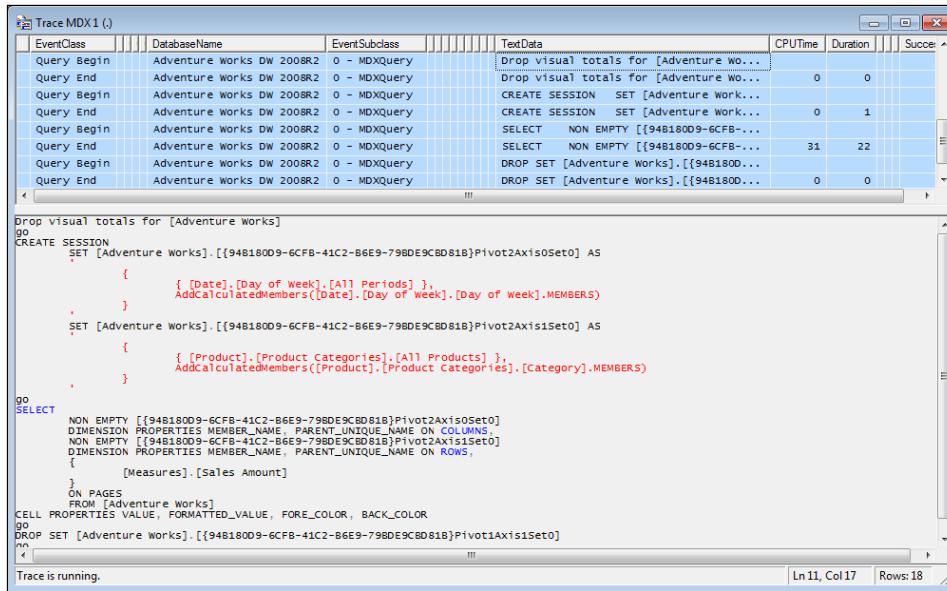
The MDX code in the bottom pane is:

```

Drop visual totals for [Adventure Works]
go
CREATE SESSION
    SET [Adventure Works].[{[94B180D9-6CFB-41C2-B6E9-798DE9CBD81B]}Pivot1Axis1Set0] AS
        [
            { [Product].[Product Categories].[All Products] },
            AddCalculatedMembers([Product].[Product Categories].[Category].MEMBERS)
        ]
    go
SELECT
    NON EMPTY {[94B180D9-6CFB-41C2-B6E9-798DE9CBD81B]}Pivot1Axis1Set0
    [
        [Measures].[Sales Amount]
    ]
    ON ROWS
    FROM [Adventure Works]
CELL PROPERTIES VALUE, FORMATTED_VALUE, FORE_COLOR, BACK_COLOR
go
    
```

Trace is running.

12. Now, expand the **Date** dimension and drag and drop the **Day of Week** attribute on columns.
13. We have 8 trace events which means 4 queries:



The screenshot shows the Trace MDX1 window with 8 trace events listed in the grid. The events are:

- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery
- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery
- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery
- Query Begin Adventure works DW 2008R2 0 - MDXQuery
- Query End Adventure works DW 2008R2 0 - MDXQuery

The MDX code in the bottom pane is:

```

Drop visual totals for [Adventure Works]
go
CREATE SESSION
    SET [Adventure Works].[{[94B180D9-6CFB-41C2-B6E9-798DE9CBD81B]}Pivot2Axis0Set0] AS
        [
            { [Date].[Day of Week].[All Periods] },
            AddCalculatedMembers([Date].[Day of week].MEMBERS)
        ]
    go
SET [Adventure Works].[{[94B180D9-6CFB-41C2-B6E9-798DE9CBD81B]}Pivot2Axis1Set0] AS
    [
        { [Product].[Product Categories].[All Products] },
        AddCalculatedMembers([Product].[Product Categories].[Category].MEMBERS)
    ]
    go
SELECT
    NON EMPTY {[94B180D9-6CFB-41C2-B6E9-798DE9CBD81B]}Pivot2Axis0Set0
    [
        [Measures].[Sales Amount]
    ]
    ON PAGES
    FROM [Adventure Works]
CELL PROPERTIES VALUE, FORMATTED_VALUE, FORE_COLOR, BACK_COLOR
go
DROP SET [Adventure Works].[{[94B180D9-6CFB-41C2-B6E9-798DE9CBD81B]}Pivot1Axis1Set0]
go
    
```

Trace is running.

14. The process can continue as long as you want or need, but this is where we'll stop with the trace. Click on the **Stop Selected Trace** button in the toolbar.
15. Congratulations, you've managed to capture all MDX queries sent to your server in a given time period.
16. Finally, save the trace if you want to preserve its contents. Otherwise, simply close the trace and the Profiler after it.

## How it works...

The Profiler is a tool which captures a lot of events regarding the SQL Server. In this case we were interested in the SSAS-related events, more precisely, the MDX query events.

There are two events in that group of events: **Query Begin** and **Query End** events. We tracked them both because it allowed us to see the start and the duration of captured queries. In practice, you can choose to track only the **Query End** event, the one which has the information about the duration of queries.

Multi-selection of events allowed us to merge queries in one pane.

As seen in this example, one user action can lead to one or many MDX queries generated in the background by an application. This can cause problems to special context-sensitive calculations presented in chapter 7 because those calculations depend on each report being a single MDX statement. In fact, this is how you would debug what's wrong with your calculations – you could see what an application did with them and how they were used in the query. In short, it benefits knowing that you can capture and later analyze any query that's in the background of an analytical report created in an application.

## There's more...

There is much more information you can capture regarding MDX queries, but this goes out of the scope of this book. For more information, see the *Expert Cube Development with Microsoft SQL Server 2008 Analysis Services* book or SQLCAT's whitepaper *Identifying and Resolving MDX Query Performance Bottlenecks in SQL Server 2005 Analysis Services*.

## Alternative solution

There are add-ins for some SSAS front-ends which enable you to see the MDX right inside the add-in, without the need to use Profiler. One such popular add-in is the Excel add-in which can be downloaded from here:

<http://tinyurl.com/PivotTableExtensions>

The site also contains instructions for its usage. Other add-ins might be available too, so make sure you've searched the Internet and found the one that's right for you.

### Tips and tricks

You can create several templates in the Profiler and use each in the appropriate situation. That way you will work faster not having to worry about selecting the right events manually every time.

### See also

The next recipe, *Performing custom drillthrough* is related to this recipe.

## Performing custom drillthrough

Multidimensional database systems like Analysis Services naturally support the top-down analysis approach. Data can be browsed using multilevel hierarchies, starting from the top level and descending down as the circumstances require. The whole process is backed up with aggregates, which are mostly calculated for upper levels, levels with fewer members, although this can be influenced in the aggregation designed.

There is another side of the story. Users occasionally want to see every single detail about an aggregated value represented in a single cell in pivot. The mechanism that supports this is the **DRILLTHROUGH** mechanism.

Drillthrough can either be issued as a standalone command, or it can be predefined as a cube-based action of that type. This recipe focuses on the first. It shows you how you can issue your own drillthrough command whenever you need to analyze the data behind a certain cell. Having a predefined action is nice, but if you don't want to use a predefined action and want to test the drillthrough or simply customize it the way you need, all it takes is to create a **DRILLTHROUGH** type of MDX query and execute it in any tool that supports MDX queries, for example, the SQL Server Management Studio. Let's see how it can be done.

### Getting ready

Start **SQL Server Management Studio** and connect to your SSAS 2008 R2 instance. Click on the **New Query** button and check that the target database is **Adventure Works DW 2008R2**.

Suppose this is our pivot:

	Actual	Budget	Forecast	Budget Variance	Budget Variance %
	Amount	Amount	Amount	Amount	Amount
Corporate	\$3,250,075.00	\$5,583,900.00	(null)	(\$2,333,825.00)	-41.80%
Corporate	(\$2,664,340.00)	(\$2,586,360.00)	(null)	(\$77,980.00)	3.02%
Executive General and Administration	(\$152,358.00)	(\$145,800.00)	(null)	(\$6,558.00)	4.50%
Inventory Management	(\$425,483.00)	(\$412,800.00)	(null)	(\$12,683.00)	3.07%
Manufacturing	(\$282,069.00)	(\$279,840.00)	(null)	(\$2,229.00)	0.80%
Quality Assurance	(\$167,079.00)	(\$160,890.00)	(null)	(\$6,189.00)	3.85%
Research and Development	\$7,044,358.00	\$9,272,850.00	(null)	(\$2,228,492.00)	-24.03%
Sales and Marketing	(\$102,954.00)	(\$103,260.00)	(null)	\$306.00	-0.30%

The highlighted cell represents the cell for which we want to analyze the background. We want to see the data the way it is stored in the cube, at it leaves.

The solution is to issue a drillthrough statement. Before we do so, let's see the MDX behind this pivot:

```

SELECT
    { [Scenario].[Scenario].[Scenario].ALLMEMBERS } *
    { [Measures].[Amount] } ON 0,
    { [Department].[Departments].MEMBERS } ON 1
FROM
    [Adventure Works]
WHERE
    ( [Date].[Fiscal].[Fiscal Year].&[2006] )
CELL PROPERTIES
    VALUE,
    FORMATTED_VALUE,
    FORE_COLOR,
    BACK_COLOR
  
```

The coordinates for the highlighted cell are: Year = 2006, Scenario = Budget Variance, Department = Research and Development, and Measure = Amount.

We cannot perform a drillthrough command on a calculated member which, in this case, is the Budget Variance scenario. Therefore, we're going to bypass that member in the coordinate and execute the drillthrough query using regular members only.

## How to do it...

Follow these steps to execute a drillthrough query against a cube on an Analysis Services instance.

1. Execute the following MDX query:

```
DRILLTHROUGH  
--MAXROWS 10  
SELECT  
FROM  
    [Adventure Works]  
WHERE  
    ( [Date].[Fiscal].[Fiscal Year].&[2006],  
      [Department].[Departments].&[6],  
      [Measures].[Amount] )
```

2. The previous query returns all rows in the underlying Financial Reporting table. Notice the column names. They will be explained in the next section.

The screenshot shows the SSMS Results pane with a table titled 'Financial Reporting'. The table has columns for various dimensions and measures. The data includes rows for different fiscal years (FY 2006), dates (July 1, 2005), calendar years (CY 2005), account types (Gross Sales, Net Sales, Total Cost of Sales, Labor Expenses, Operating Expenses, Travel Expenses), organization types (USA Operations), and scenario types (Actual). The table contains approximately 20 rows of data.

[Financial Reporting].[Amount]	[\$Date].[Fiscal Year]	[\$Date].[Date]	[\$Date].[Calendar Year]	[\$Account].[Accounts]	[\$Organization].[Organizations]	[\$Scenario].[Scenario]
3059	FY 2006	July 1, 2005	CY 2005	Gross Sales	USA Operations	Actual
2294	FY 2006	July 1, 2005	CY 2005	Net Sales	USA Operations	Actual
33968	FY 2006	July 1, 2005	CY 2005	Total Cost of Sales	USA Operations	Actual
3186	FY 2006	July 1, 2005	CY 2005	Total Cost of Sales	USA Operations	Actual
16320	FY 2006	July 1, 2005	CY 2005	Labor Expenses	USA Operations	Actual
1632	FY 2006	July 1, 2005	CY 2005	Labor Expenses	USA Operations	Actual
979	FY 2006	July 1, 2005	CY 2005	Labor Expenses	USA Operations	Actual
1912	FY 2006	July 1, 2005	CY 2005	Operating Expenses	USA Operations	Actual
220	FY 2006	July 1, 2005	CY 2005	Travel Expenses	USA Operations	Actual
200	FY 2006	July 1, 2005	CY 2005	Travel Expenses	USA Operations	Actual
104	FY 2006	July 1, 2005	CY 2005	Travel Expenses	USA Operations	Actual
76	FY 2006	July 1, 2005	CY 2005	Travel Expenses	USA Operations	Actual
22	FY 2006	July 1, 2005	CY 2005	Travel Expenses	USA Operations	Actual

3. Notice the commented part of the query. Uncomment it and then run the query again. This time it returned only 10 rows, just like we've specified using the MAXROWS keyword.

## How it works...

The DRILLTHROUGH command can be issued on a single cell only. That cell must not have calculated members or an error will occur.

We can limit the amount of rows to be returned by this type of query using the MAXROWS keyword. In case we don't, the server-based setting **OLAP \ Query \ DefaultDrillthroughMaxRows** determines the limit. This setting is set to 10000 by default which is visible amongst the advanced settings of the server, where you can change it.

Column names tell us the origin of that column. Names starting with \$ denote a dimension table. Others can come from a single fact table (measure group) only. The simple form of a drillthrough query returns all measures in that fact table along with all the dimension keys. Besides the fact columns, all columns that represent the coordinate for the drillthrough are also returned.

The custom type of drillthrough query, the one that returns only the specified columns is explained in the next section.

### There's more...

It is possible to change the columns that a drillthrough command returns. This type of drillthrough is a custom drillthrough.

Using the `RETURN` keyword we can specify which columns we want to get as the result of a drillthrough query. In addition to that, we can use 9 functions dedicated to extract more metadata from the available columns. Here's an example to illustrate that:

```
DRILLTHROUGH
SELECT
FROM
    [Adventure Works]
WHERE
    ( [Date].[Fiscal].[Fiscal Year].&[2006],
      [Department].[Departments].&[6],
      [Measures].[Amount] )
RETURN
    [$Date].[Date],
    MemberValue([$Date].[Date]),
    [$Organization].[Organizations],
    UnaryOperator([$Account].[Account]),
    [Financial Reporting].[Amount],
    [$Destination Currency].[Destination Currency Code],
    [$Account].[Accounts],
    [$Account].[Account]
```

This query differs from the one in the previous section in several ways. First, it uses a specific set of columns to be returned. Next, it uses two functions: `MemberValue()` and `UnaryOperator()`.

The result of the query looks like this:

Messages		Results						
[\$Date].[Date]	[\$Date].[Date]	[\$Organization] [...]	[\$Account].[Acc...]	Financial Report...	[\$Destination Cu...]	[\$Account].[Acc...]	[\$Account].[Acc...]	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	3059	USD	Gross Sales	Intercompany Sales	
July 1, 2005	7/1/2005 12:00...	USA Operations	-	2294	USD	Net Sales	Returns and Adjustments	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	33968	USD	Total Cost of Sa...	Standard Cost of Sales	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	3186	USD	Total Cost of Sa...	Variances	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	16320	USD	Labor Expenses	Salaries	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	1632	USD	Labor Expenses	Payroll Taxes	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	979	USD	Labor Expenses	Employee Benefits	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	1912	USD	Operating Expe...	Commissions	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	220	USD	Travel Expenses	Travel Transportation	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	200	USD	Travel Expenses	Travel Lodging	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	104	USD	Travel Expenses	Meals	
July 1, 2005	7/1/2005 12:00...	USA Operations	+	76	USD	Travel Expenses	Entertainment	

`UnaryOperator()` returns the sign for the Amount, the way it aggregates upwards.

`MemberValue()` returns the date in the datetime format; normally, member names are strings (for example, the first column). Sometimes it's good to have original datatypes. This is how to do it.

Naturally, you can wrap the drillthrough query in the `OpenQuery` statement and use it in the relational environment. That way you can change the names it returns. However, there are some issues with duplicate names that `OpenQuery/OpenRowset` doesn't tolerate. Notice the first two columns in the previous image. One of them you'll have to remove. You might say, not that bad, they represent the same thing and I can keep the better one which is the second column in this case because it preserved the datatype. True, but there's another duplicate column, `UnaryOperator()`. All special functions preserve the name of the column inside them. This time you will have to choose between the sign and the value of the account.

### Allowed functions and potential problems about them

The 9 functions mentioned earlier can be found here:

<http://tinyurl.com/Drillthrough>

For the reasons explained a moment ago, the use of the drillthrough command is limited in distributed queries.

### More info

Drillthrough-related problems are covered in-depth in the *Expert Cube Development with Microsoft SQL Server 2008 Analysis Services* book.

## Other examples

The *Analysis Services Stored Procedure Project* assembly has an interesting Drillthrough class which can bypass some of the restrictions imposed on the usage of the drillthrough command. As mentioned earlier in this chapter, this assembly can be downloaded from here:

<http://tinyurl.com/ASSPCodePlex>

### See also

The recipe *Capturing MDX queries generated by SSAS front-ends* is related to this recipe.



# Conclusion

The cookbook has reached its end. However, this doesn't have to be the end of your MDX journey. It's simply means that it's time for you to continue alone. MDX is a rich language intended for querying multidimensional databases. It is accepted as a standard by a plethora of OLAP vendors. Microsoft SQL Server Analysis Services is known for its capabilities and is respected as one of the most advanced OLAP databases, if not the most advanced OLAP database. I'm sure that's the reason why you've chosen it in the first place, and you've made the right choice. There's more to it than just technology that makes it superb and unique.

There's the community, for example. We've already mentioned active CodePlex projects for Microsoft SQL Server Analysis Services in some of the recipes. If you need the sample databases, code samples, and other good tools to make your life easier, simply look for the term SSAS on  
[www.codeplex.com](http://www.codeplex.com).

If you prefer reading and learning from people working with SSAS, there's a portal maintained by Vidas Matelis called SSAS-Info:  
[www.ssas-info.com](http://www.ssas-info.com).

Vidas collects articles about Analysis Services, collects important news, tips and tricks, scripts, maintains a detailed list of SSAS front-ends, tools, webcasts, books, demos and everything else that might interest you. I suggest you bookmark that site if you haven't done so already and see if you can expand what you learned in this book by searching among articles in the MDX category or by looking among article tags. You might be surprised how active the community is. The site also features a list of consultants that may assist you when you encounter a problem that you can't solve alone.

Another very thorough directory of articles, tools, products, videos, books, and personas active around SSAS, built mostly from the contents found on Vidas' site, is the SSAS-Wiki site: [ssas-wiki.com](http://ssas-wiki.com).

## Conclusion

---

Microsoft is also doing its share in spreading the knowledge of how to make the most of its analytical platform. The SQLCAT team, a group of Microsoft SQL Server experts that work with some of the world's largest BI projects, blogs about their experiences on this site: [sqlcat.com](http://sqlcat.com).

If you're interested in learning the best practices regarding SSAS, read their articles. There you'll find many good tips for improving the performance of your queries and modeling and architectural advices. If you're interested in learning more about MDX, you should definitely look for the latest version of these two whitepapers: *Identifying and Resolving MDX Query Performance Bottlenecks* and *The Analysis Services Performance Guide*.

When you have a problem and no one to turn to, here's a place to go – the Microsoft SSAS forum:

<http://tinyurl.com/SSASForum>.

Actually, that's just one of the hundreds of forums on the MSDN site, which means you'll probably find it by searching for the term *SSAS forum* sooner than remembering its long URL (hence the tinyurl shortcut here). Nevertheless, bookmark it the first time you come there, it's definitely worth it.

The forum is very active and it's moderated by some of the best Analysis Services experts from around the world. Many people there are keen in learning something new or putting their SSAS knowledge to the test by helping you solve your problems. No matter which part of the globe you come from, there will always be someone awake willing to help you. Give it a try and you'll be amazed by the speed with which someone answers your question.

The forum is also a good place to look for already-solved problems. Make sure you perform a search before you post anything, it may save you some time. And yes, you can rely on the quality of answers, it is among the best found on the Internet for SSAS.

Sooner or later you'll need to take a look at the MSDN site regarding the definition of a function or simply to read a part of the MDX fundamentals again. They are hidden inside this link:  
<http://tinyurl.com/MDXFundamentals>.

Prefer TechNet? Here's the recommended starting point for SSAS:

<http://tinyurl.com/SSASonTechNet>.

Finally, there's Moshav Pasumansky, the MDX guru who no longer works on the SSAS team, but whose legacy in form of the blog and the tool called MDX Studio continues to help many people around the world. His blog is a great resource of articles about interesting MDX tricks and useful MDX techniques. MDX Studio is a tool for analyzing and improving MDX queries. The respective links are here:

- ▶ <http://tinyurl.com/MoshaBlog>
- ▶ <http://tinyurl.com/MDXStudioPage>

The *MDXStudioPage* page mentioned above does not contain the link to the latest version of that tool, only the general information about MDX Studio. The latest version is the one with the highest number here:

<http://tinyurl.com/MDXStudioFolder>.

There's also the online version of the MDX Studio which can be found here:  
<http://mdx.mosha.com>.

As you can see, there are many contents about the MDX language that it's simply impossible to cover everything there is about MDX in a single book.

This book was based on the premise that you already know the basics of MDX and want to take them further. As a cookbook, the book offered you a structured but a free-to-explore way of learning new things. The recipes categorized in chapters covered the tasks you encounter in your daily work. I have tried to give you as much useful information as possible by keeping the recipes concise. This should have built your confidence well enough to be able to continue on your own. However, given all the resources mentioned earlier, it looks like you won't be alone.

There's so much more to learn, practice, and explore.



# Glossary of Terms

MDX is a powerful, yet complex language. Many terms and concepts need to be understood well enough if you want to master it. In order to help you in that mission, this book ends with a short explanation of all important terms used throughout the book.

The glossary of terms is divided in six parts. Each part covers one aspect of the MDX language. We start by explaining what MDX queries are made of, followed by the terms and concepts specific to the execution of MDX queries. The middle part of the Appendix covers the most important things related to dimension design, cube design, and MDX script. Next, we explain terms related to query optimization and finally finish the Appendix with types of queries that can be used with SQL Server Analysis Services. All explanations are brief and concise. If you don't understand them at first, try to read them slowly, maybe several times, or simply look in the list of related recipes laid out in the end of each explanation (if available). There you will find examples which should help you understand it better.

## Parts of an MDX query

This section contains the brief explanation of the basic elements of MDX queries: members, sets, tuples, axes, properties, and so on.

### Regular members

Regular dimension members are members sourced from the underlying dimension tables. They are the building blocks of dimensions, fully supported in any type of drill operations, drillthrough, scopes, subqueries, and probably all SSAS front-ends. They can have children and be organized in multilevel user hierarchies. Some of the regular member's properties can be dynamically changed using scopes in MDX script or cell calculations in queries - color, format, font, and so on.

## Glossary of Terms

---

Measures are a type of regular members, found on the Measures dimension/hierarchy.

The other type of members is calculated members.

*Related recipes:*

- ▶ *Creating a physical measure as a placeholder for MDX assignments (chapter 6)*
- ▶ *Using a new attribute to separate members on a level (chapter 6)*

## Calculated members

Calculated members are **artificial members** created in a query, session, or MDX script. They do not exist in the underlying dimension table and as such are not supported in drillthrough and scopes. In subqueries, they are only supported if the connection string includes one of these settings: Subqueries = 1 or Subqueries = 2. See here for examples:

<http://tinyurl.com/ChrisSubquerySettings>

They also have a limited set of properties compared to regular members and worse support than regular members in some SSAS front-ends.

An often practiced workaround is creating **dummy regular members** in a dimension table and then using MDX script assignments to provide the calculation for them. They are referred to as "Dummy" because they never occur in the fact table which also explains the need for assignments.

*Related recipes:*

- ▶ *Creating a physical measure as a placeholder for MDX assignments (chapter 6)*

## Tuples

A tuple is a coordinate in the multidimensional cube. That coordinate can be huge and is often such. For example, a cube with 10 dimensions each having 5 attributes is a 51 dimensional object (measures being that extra one). To fully define a coordinate we would have to reference every single attribute in that cube. Fortunately, in order to simplify their usage, tuples are allowed to be written using a part of the full coordinate only. The rest of the coordinate inside the tuple is implicitly evaluated by SSAS engine, either using the current members (for unrelated hierarchies) or through the mechanism known as *strong relationships* (for related hierarchies). It's worth mentioning that the initial current members are cube's default members. Any subsequent current members are derived from the current context of the query or the calculation.

Evaluation of implicit members can sometimes lead to unexpected problems. We can prevent those problems by explicitly specifying all the hierarchies we want to have control over and thereby not letting the implicit evaluation to occur for those hierarchies.

Contrary to members and sets, tuples are not an object that can be defined in the `WITH` part of the query or in MDX script. They are **non-persistent**. Tuples can be found in sets, during iteration or in calculations. They are often used to set or overwrite the current context, in other words, to jump out of the current context and get the value in another coordinate of the cube.

Another important aspect of tuples is their **dimensionality**. When building a set from tuples, two or more tuples can be combined only if they are built from the same hierarchies, specified in the exact same order. That's their dimensionality. You should know that rearranging the order of hierarchies in a tuple doesn't change its value. Therefore, this can be the first step we can do to make the tuples compatible. The other thing is adding the current members of hierarchies present only in the other tuple, to match the other tuple's dimensionality.

*Related recipes:*

- ▶ *Implementing logical OR on members from different hierarchies* (chapter 1)
- ▶ *Getting values on the last date with data* (chapter 2)
- ▶ *Combining two hierarchies into one* (chapter 3)
- ▶ *Performing complex sorts* (chapter 8)

## Named sets

A named set is a user-defined collection of members, more precisely, tuples. Named sets are found in queries, sessions, and MDX scripts. Query-based named sets are equivalent to dynamic sets in MDX script. They both react to the context of subquery and slicer. Contrary to them, static sets are constant, independent of any context.

Only the sets that have the **same dimensionality** can be combined together because what we really combine are the tuples they are built from (see the definition of a tuple above).

It is possible to extract one or more hierarchies from the set. It is also possible to expand the set by crossjoining it with hierarchies not present in its tuples. These processes are known as reducing and increasing the dimensionality of a set.

*Related recipes:*

- ▶ *Analyzing fluctuation of customers* (chapter 5)
- ▶ *Forecasting using linear regression* (chapter 5)
- ▶ *Iterating on a set in order to reduce it* (chapter 1)
- ▶ *Iterating on a set in order to create a new one* (chapter 1)
- ▶ *Iterating on a set using recursion* (chapter 1)
- ▶ *Calculating today's date using string functions* (chapter 2)
- ▶ *Isolating the best N members in a set* (chapter 3)
- ▶ *Isolating the worst N members in a set* (chapter 3)

## Glossary of Terms

---

- ▶ *Identifying the best/worst members for each member of another hierarchy*  
(chapter 3)
- ▶ *Displaying few important members, others as a single row, and the total at the end*  
(chapter 3)
- ▶ *Calculating various ranks* (chapter 4)

## Set alias

Set aliases can be defined **in calculations only**, as a part of that calculation and not in the WITH part of the query as a named set (explained earlier). This is done by identifying a part of the calculation that represents a set and giving a name to that expression inside the calculation, using the AS keyword. This way that set can be used in other parts of the calculation or even other calculations of the query or MDX script.

Set aliases enable **true dynamic evaluation** of sets in a query because they can be evaluated for each cell if used inside a calculated measure. The positive effect is that they are cached, calculated only once and used many times in the calculation or query. The downside is that they prevent block-computation mode because the above mentioned evaluation is performed for each cell individually.

In short, set aliases can be used in long calculations, where the same set appears multiple times or when that set needs to be truly dynamic. At the same time, they are to be avoided in iterations of any kind.

Related recipes:

- ▶ *Iterating on a set in order to create a new one* (chapter 1)
- ▶ *Identifying the content of axes* (chapter 7)
- ▶ *Calculating the bit-string for hierarchies on an axis* (chapter 7)
- ▶ *Implementing the utility dimension with context-aware calculations* (chapter 7)
- ▶ *Displaying a sample from a random hierarchy* (chapter 8)

## Axis

An axis is a part of the query where a set is projected at. A query can have up to 128 axes although most queries have 1 or 2 axes. A query with no axis is also a valid query but almost never used.

The important thing to remember is that axes are evaluated independently. SSAS engine knows in which order to calculate them if there is a dependency between them. One way to create such a dependency is to refer to the current member of a hierarchy on the other axis. The other option would be to use the Axis( ) function.

Some SSAS front-ends generate MDX queries that break the axes dependencies established through calculations. The workaround calculations can be very hard if not impossible.

*Related recipes:*

- ▶ Chapter 7 in its entirety

## Slicer

The slicer, also known as the filter axis or the `WHERE` clause, is a part of the query which sets the context for the evaluation of members and sets on axes and in the `WITH` part of the query.

The slicer, which can be anything from a single tuple up to the multidimensional set, interacts with sets on axes. A single member of a hierarchy in slicer forces the coordinate and reduces the related sets on axes by removing all non-existing combinations. Multiple members of the same hierarchy are not that strong. In their case, individual members in sets on axes overwrite the context of the slicer during their evaluation.

Finally, the context established by the slicer can be overwritten in the calculations using tuples.

*Related recipes:*

- ▶ *Calculating parallel periods for multiple dates in slicer* (chapter 2)
- ▶ *Implementing logical AND on members from the same hierarchy* (chapter 1)
- ▶ *Implementing NOT IN set logic* (chapter 1)
- ▶ *Implementing logical OR on members from different hierarchies* (chapter 1)
- ▶ *Iterating on a set in order to reduce it* (chapter 1)
- ▶ *Finding the last date with data* (chapter 2)
- ▶ *Isolating the best N members in a set* (chapter 3)
- ▶ *Analyzing fluctuation of customers* (chapter 5)

## Subquery

The subquery, also known as the subselect, is a part of the query which executes first and determines the cube space to be used in the query. Unlike slicer, the subquery doesn't set the coordinate for the query. In other words, current members of all hierarchies (related, unrelated, and even the same hierarchy used in the subquery) remain the same. What the subquery does is it applies the `VisualTotals` operation on members of hierarchies used in the subquery. The `VisualTotals` operation changes each member's value with the aggregation value of its children, but only those present in the subquery.

## Glossary of Terms

---

Because the slicer and the subquery have different behaviors, one should not be used as a replacement for the other. Whenever you need to set the context for the whole query, use the slicer. That will adjust the total for all hierarchies in the cube. If you only need to adjust the total for some hierarchies in the cube and not for the others, subquery is the way to go; specify those hierarchies in the subquery. This is also an option if you need to **prevent** any attribute interaction between your subquery and the query.

The areas where the subquery is particularly good at are grouping of non-granular attributes, advanced set logic, and restricting members on hierarchies.

*Related recipes:*

- ▶ *Implementing logical AND on members from the same hierarchy* (chapter 1)
- ▶ *Calculating parallel periods for multiple dates in slicer* (chapter 2)
- ▶ *Creating a physical measure as a placeholder for MDX assignments* (chapter 6)

## Cell properties

Cell properties are properties that can be used to get specific behaviors for cells. For example: colors, font sizes, types and styles, and so on. Unless explicitly asked for, only the `Cell.Ordinal`, `Value`, and `Formatted_Value` properties are returned by an MDX query.

*Related recipes:*

- ▶ *Setting special format for negative, zero, and null values* (chapter 1)
- ▶ *Applying conditional formatting on calculations* (chapter 1)
- ▶ *Highlighting siblings with the best/worst values* (chapter 3)
- ▶ *Implementing bubble-up exceptions* (chapter 3)

## Dimension properties

Dimension properties are a set of member properties that return extra information about members on axes. Intrinsic member properties are `Key`, `Name`, and `Value`; the others are those defined by the user for a particular hierarchy in the **Dimension Designer**. In client tools, dimension properties are often shown in the grid next to the attribute they are bound to or in the hint over that attribute.

*Related recipes:*

- ▶ *Implementing NOT IN set logic* (chapter 1)

## MDX query in action

In this part we cover topics relevant to execution of MDX queries: explicit and implicit members, current context, attribute overwrites, recursion, and so on.

### Explicit members

Explicit members are either regular or calculated members specified in a tuple using their unique name or a function. Members not being specified in the tuple are called implicit members. Explicit regular members are **never overwritten** by attribute relation mechanisms, unless they refer to the current member, that is, unless they use the `.CurrentMember` function, and there's also a related attribute in the same tuple. In that case they might get overwritten.

*Related recipes:*

- ▶ *Performing complex sorts (chapter 8)*

### Implicit members

Implicit members are regular members not specified in the current context. In order to get the full coordinate of the current context, these members are evaluated by the SSAS engine and internally added to every tuple.

Current members of dimensions not specified in the tuple remain on their previous positions, positions established by an outer context, be it one of the outer parts of that calculation (starting from the innermost one), the slicer, or MDX script. Current members of dimensions specified in the tuple shift the current members of all related hierarchies not specified in the tuple to their implicit positions. The implicit position is the lowest position in the hierarchy that doesn't interfere with the context. This is evaluated in respect to the dimension design, not the outer context which varies.

If there is a calculated member inside the tuple, current members of all related hierarchies not present in the tuple are shifted to their default member, the `All` member if not designed differently.

*Related recipes:*

- ▶ *Performing complex sorts (chapter 8)*

## Data members

Data members are system-generated, non-leaf members that occur in parent-child hierarchies whose value is not an aggregate of their descendants but their own contribution coming from the underlying fact table. These additional members may or may not be visible, depending on the option chosen for the `MembersWithData` property: `NonLeafDataVisible` or `NonLeafDataHidden`. When the system-generated data members are left visible (which is the default option), a name suffix or prefix can be used to distinguish data members from their parents. Otherwise, they will have the same name. The asterisk in the name is as a placeholder for the parent's name, for example, like this: "(\* data)".

The `.DataMember` function can get a member's individual value regardless of it being hidden or not.

*Related recipes:*

- ▶ *Displaying members with data in parent-child hierarchies* (chapter 8)

## Current context

Current context is a coordinate (more generally a subcube) formed by combining existing members from all cube hierarchies. It is a very important concept because knowing what is currently in context in each particular step of the query evaluation is crucial for writing good MDX queries, evaluating them, and debugging problematic calculations. Current context can be detected using the `EXISTING` function in every situation and using the `.CurrentMember` function in the case of a single member only.

*Related recipes:*

- ▶ *Calculating parallel periods for multiple dates in slicer* (chapter 2)
- ▶ *Finding related members in the same dimension* (chapter 4)

## Dimensionality

Dimensionality is a vector that shows which hierarchies are explicitly referred to in a tuple or a set and in what order. Only the sets and tuples of the same dimensionality can be combined together.

*Related recipes:*

- ▶ *Implementing logical OR on members from different hierarchies* (chapter 1)

## Attribute overwrites

Attribute overwrites is a mechanism where explicit members in a tuple overwrite the previously established context by forcing themselves inside the current coordinate. Therefore, they shift the current members of all related hierarchies not present in that tuple to their lowest compatible position, compatible in a sense that implicit members don't change the coordinate but merely adjust to it. The latter behavior is also known as strong relationships algorithm.

*Related recipes:*

- ▶ *Performing complex sorts* (chapter 8)

## Visual Totals

Visual Totals is a SSAS feature for modifying a non-leaf member's value so that it matches the aggregated value of the subcube formed only from descendants specified in that subcube. In case of the `VisualTotals()` function, members must be sorted hierarchically, the upper level members must be specified before the lower level members, otherwise it will not work as expected.

## Iteration

Iteration is a process of repeating a task by going through each item in the collection, usually a set. All set functions incorporate that mechanism although most of them manage to avoid the iteration by turning to the block-computation mode whenever possible. In situations when this is not possible, a much slower process of cell-by-cell evaluation happens.

*Related recipes:*

- ▶ *Iterating on a set in order to reduce it* (chapter 1)
- ▶ *Iterating on a set in order to create a new one* (chapter 1)
- ▶ *Iterating on a set using recursion* (chapter 1)

## Recursion

Recursion is an iterative process in which the value to be evaluated is requesting another call to the same object. Once this process starts, it keeps repeating itself until a stopping condition occurs. The stopping condition occurs when the value being evaluated does not trigger another recursive call to the same object.

It's important to notice that we don't know in advance how many times the process will repeat as it will vary. This characteristic makes the recursion very useful when we don't want to iterate over the complete set. When we need to stop as soon as the condition is met, recursion is potentially our friend.

Related recipes:

- ▶ *Iterating on a set using recursion* (chapter 1)
- ▶ *Using recursion to calculate cumulative values* (chapter 8)

## Context-aware calculations

Context-aware calculations are calculations which detect the current context and evaluate accordingly. This evaluation of the current context often triggers the cell-by-cell evaluation mechanism which means these calculations are slow on large sets. On the positive side, they can be made independent of the cube structure or a part of it and as such used in every cube.

Related recipes:

- ▶ The complete chapter 7

## Set-based operations

Set-based operations are mechanisms for operating with sets in a way that members in them can be intersected, unionized, or subtracted as a whole, not individually. Set operations are therefore very fast operations and should be the preferred way of performing complex calculations, preferred against much slower iterations where members are treated individually.

Related recipes:

- ▶ *Analyzing fluctuation of customers* (chapter 5)
- ▶ *Implementing the ABC analysis* (chapter 5)

## Errors

Errors occur for various reasons. However, that can be prevented by intercepting them with the `IsError()` function. In case of SSMS or BIDS, it's worth remembering that the error message is visible in the hint of the cell with an error (you just have to wait long enough).

Related recipes:

- ▶ *Identifying the number of columns and rows a query will return* (chapter 7)
- ▶ *Calculating the YTD (Year-To-Date) value* (chapter 2)

## Cube and dimension design

This part explains things you encounter during designing your dimensions and cubes: default members, attribute relationships, natural hierarchies, unnatural hierarchies, parent-child hierarchies, utility dimensions, granularity, and so on.

### Default member

The default member is a member defined as such in a hierarchy or MDX script. It sets the context for queries and calculations unless overwritten by another member of the same hierarchy or implicitly shifted by a member from a related hierarchy. The latter case can lead to unexpected results, therefore, setting the default member should only be done after a careful consideration of which attribute combinations are allowed to be made by end-users.

A default member should be specified whenever the `All` member of that hierarchy is disabled (whenever the `IsAggregatable` property is set to `False`).

*Related recipes:*

- ▶ *Setting the default member of a hierarchy in MDX script (chapter 1)*

### Attribute relationships

Attribute relationships initially exist only between the key attribute of a dimension and all other attributes of that dimension. Creating additional attribute relationships, whenever possible, improves the dimension design. That way, the SSAS engine knows what we as humans know – which attributes can be used to form multi-level structures called natural user hierarchies. This allows for many optimizations, however, it has to be done with extra care. Implicit members in tuples heavily depend on this, these relationships are used by the engine to determine them.

*Related recipes:*

- ▶ *Using a new attribute to separate members on a level (chapter 6)*

### Natural versus unnatural hierarchies

Natural hierarchies are user hierarchies where members of lower levels have one (and only one) member above them, and so on for each level. Their neighboring levels are built from related attributes. Unnatural hierarchies are those built from unrelated attributes which means that members from lower levels might have more than one parent. For example, combining product color and size in a single user hierarchy would create an unnatural user hierarchy. Natural user hierarchies are those found in Adventure Works.

## Parent-child hierarchies

Parent-child hierarchies are a type of unbalanced hierarchy where the number of levels can vary. However, that comes with a price in performance – aggregations can only be built on the root member and the leaves. If the number of levels can somehow be fixed, it is worth considering converting them into ordinary user hierarchies with the **HideMemberIf** option turned on to simulate the parent-child effect. This way, aggregates can be built on any level which should increase the performance. Small parent-child hierarchies can stay as such if the flexibility of levels they offer suits you. A significant performance boost (as a result of the conversion) will be noticeable on parent-child hierarchies with many members primarily.

Related recipes:

- ▶ *Displaying members without children (leaves)* (chapter 8)
- ▶ *Displaying members with data in parent-child hierarchies* (chapter 8)

## Utility dimension

The utility dimension is a special dimension used for calculation purposes. It is often designed with the disabled root member and the default member being the first member of that dimension. Additionally, that dimension should not be linked to any measure group in the cube because its attribute doesn't exist in any measure group. Finally, members have calculations relative to the dimension's default member, assigned to them in the MDX script.

The assignments for members of the utility dimension apply to all measures, unless specified otherwise. Therefore, instead of having a lot of calculated measures, the utility dimension (hence the name) allows to limit that by an order of magnitude. Namely, calculated measures based on the repeating principle or formula (a ratio) can be replaced with a single member on a utility dimension with a same formula (written in a more generic way) assigned to them. In other words, we're transforming M measures \*N variations into M measures + N members of the utility dimension.

Using regular members in utility dimension will prevent problems that some SSAS front-ends have with calculated members. For that you'll have to create a table in DW or a named query in DSV.

**Asymmetrical reporting**, that is having arbitrary shaped sets on axis, might be another problem introduced with this approach. If that's the case with your tool, you should consider creating additional calculated measures so that those arbitrary shaped sets can be presented using calculated measures only and hence become non-arbitrary.

Related recipes:

- ▶ *Using a utility dimension to implement flexible display units* (chapter 6)
- ▶ *Using a utility dimension to implement time-based calculations* (chapter 6)
- ▶ *Implementing a utility dimension with context-aware calculations* (chapter 7)
- ▶ *Displaying a sample from a random hierarchy* (chapter 8)

## Dummy dimension

Dummy dimension is similar to the utility dimension, yet different. The root member remains this time enabled and no calculations are assigned to individual members.

The idea of having a dummy dimension is being able to show its members in the result of a query or iterate on that dimension in order to achieve some goal. Remember, SSAS operates on **predefined structures only** (dimensions, hierarchies, levels, and so on). If we have a set of artificial members, members not belonging to any dimension but still required in the result of the query, we can either define them, one by one, as calculated members on a hierarchy that is not used very often in queries (so that it doesn't interfere with other hierarchies), or we can build a new dimension with those members, a dimension not related to any measure group and hence very convenient for browsing and displaying result on it. The first option is a hack, the other is the dummy dimension.

In the case of a dummy dimension, assignments are not made on its members. Instead, they are made on existing cube structures, for example on one or more measures. This way we can allocate measure's value across the dummy dimension members, based on a particular calculation. Other measures will not change across dimension members. This suggests controlled exposure to end-users, for example, in a perspective or similar.

Related recipes:

- ▶ *Implementing the Tally table utility dimension* (chapter 8)
- ▶ *Using a dummy dimension to implementing histograms over non-existing hierarchies* (chapter 6)

## Granularity

Granularity is the term that describes the resolution at which dimensions are captured in a fact table or loaded into the cube. In other words, it represents the level of detail at which a particular process is recorded, both in DW and in the cube. The good thing about SSAS is that different measure groups can be mapped to the same dimension at different granularity.

Related recipes:

- ▶ *Implementing logical AND on members from the same hierarchy* (chapter 1)

## Deployment versus processing

Deployment is a process of transferring the structure of a SSAS database or one of its major objects from a development computer to the SSAS server. Processing is a task issued on an already-deployed processable major SSAS object which fills that object with the underlying data. These are databases, cubes, dimensions, measure groups, and partitions.

If an operation modifies the structure of an SSAS database in a way that doesn't affect the aggregations, cube dimensionality, or other properties that invalidate existing dimensions and cubes, it is enough to deploy the changes; processing might be an unnecessary waste of time. An example of such operation is adding new calculations in MDX script. Don't click on the **Process** icon every time you do something simply because it's there. Yes, it is very convenient, but sometimes it might be faster to check whether deployment of changes is enough, particularly on large objects.

Here's a hint: by installing BIDS Helper, you'll get an icon on the toolbar that deploys changes you make in MDX script. Consider using that free tool available on CodePlex.

## MDX script

This part covers things you encounter when writing calculations in the MDX script: scopes, assignments, and similar.

### Calculate statement

The `Calculate` statement, usually the first statement of the MDX script, aggregates the values of leaf members to their descendants. Whenever you need to do something before that aggregation process, put those corrections before the `Calculate` statement. Otherwise, put the calculations after it. The latter is something you'll do in majority of cases; the former will rarely be required.

### Scopes

The scope is a part of MDX script in which we modify the value for a specified subcube. It can be imagined as something similar to computed columns or triggers in the relational database, not in the sense of performance but in terms of behavior, of what it does. The definition of the scope determines when the scope will activate. Inside the scope, we can apply various calculations and assignments.

When scoping the utility dimension, we must remember to specify the default member in every tuple mentioned in the assignment part of that scope, otherwise calculations will be empty because individual members of the utility dimension don't occur in any measure group. Only the default member has a value equal to the value when that dimension is not present in the query.

Scopes can be nested inside each other. **Nesting scopes** of the same dimension can cause unpredictable results because of attribute relationships, unless done with great care. It is therefore always better to define a single scope for related hierarchies. Non-related hierarchies don't experience those problems and can be freely nested.

*Related recipes:*

- ▶ *Hiding calculation values on future dates (chapter 2)*

## Assignments

Assignments are a mechanism for providing values inside the MDX script. They can be performed either inside a scope statement or outside of it, directly in the MDX script. Whatever the case is, they are valid for that particular context only. In the case of a scope, that context is the subcube defined by that scope. In the case of the whole MDX script, it is the whole cube. Their definition is preserved as long as they are not modified inside another subcube, be it inside an inner scope or a scope that follows in the MDX script.

Whenever there's a reference to an object of the cube (a member for example, not necessarily the current member to be precise), an assignment is merely a **pointer** to that member's value, not the value itself. In other words, assignments are functions which are evaluated in runtime. The control over that evaluation can be partially taken over using the `Freeze()` statement, though performance-wise it's better not to use it. It's better to redesign the assignment process, if possible, and let the assignments evaluation process flow without any intervention. The right hand side of an assignment is evaluated at query time, but the left hand side of an assignment and scope statements surrounding it are evaluated when the first user with a particular set of security roles connects to the cube.

A typical case where assignments are used, besides the scope of course, is to define a new calculated measure with the value of null and later scope it to something else for a particular subcube.

One of the most important things to realize about assignments is that assignments against regular members will subsequently aggregate up, but assignments against calculated members don't aggregate up. For example, if you run the following assignment and then query the All member, it will have a value:

```
( [Customer].[Customer].[Customer].Members,  
[Measures].[Physical Measure] ) = 100;
```

But the All member will not have a value due to this assignment:

```
( [Customer].[Customer].[Customer].Members,  
[Measures].[Calculated Measure] ) = 100;
```

Related recipes:

- ▶ *Creating a physical measure as a placeholder for MDX assignments* (chapter 6)
- ▶ *Using a utility dimension to implement flexible display units* (chapter 6)
- ▶ *Using a utility dimension to implement time-based calculations* (chapter 6)
- ▶ *Implementing a utility dimension with context-aware calculations* (chapter 7)

## Dynamic versus static sets

Sets defined in the MDX script can be either dynamic or static. Dynamic sets are re-evaluated against the subquery and the slicer of every query. Static sets are constant. Once evaluated in the MDX script when a user first connects, they are used as such in any query. The default option when creating a set in an MDX script is a static set. This means we have to use the prefix DYNAMIC if we want to define a dynamic set.

Both types of sets have their purpose. Use the one that's appropriate in a particular case. Also, see the explanation for *Set aliases* if you need sets that can evaluate differently for each cell.

Related recipes:

- ▶ *Analyzing fluctuation of customers* (chapter 5)
- ▶ *Forecasting using linear regression* (chapter 5)
- ▶ *Iterating on a set in order to reduce it* (chapter 1)
- ▶ *Iterating on a set in order to create a new one* (chapter 1)
- ▶ *Iterating on a set using recursion* (chapter 1)
- ▶ *Calculating today's date using the string functions* (chapter 2)
- ▶ *Isolating the best N members in a set* (chapter 3)
- ▶ *Isolating the worst N members in a set* (chapter 3)
- ▶ *Identifying the best/worst members for each member of another hierarchy* (chapter 3)
- ▶ *Displaying few important members, others as a single row and the total at the end* (chapter 3)
- ▶ *Calculating various ranks* (chapter 4)

## Query optimization

The Query optimization section is dedicated to explaining concepts relevant to the process of optimizing queries: block computation, cell-by-cell mode, arbitrary shaped sets, varying attributes, sparse and dense calculations, types of cache, and so on.

## Block-computation versus cell-by-cell evaluation mode

Block-computation mode, also known as subspace mode, is a process of evaluating cells in blocks. It is a mode that is many times faster than the cell-by-cell mode. Most MDX functions (<http://tinyurl.com/Improved2008R2>) are optimized to work in that mode. However, by applying non-optimized calculations we can prevent that, accidentally or on purpose.

Related recipes:

- ▶ *Iterating on a set in order to reduce it* (chapter 1)
- ▶ *Calculating moving averages* (chapter 2)
- ▶ *Finding the last date with data* (chapter 2)
- ▶ *Calculating the difference between two dates* (chapter 2)
- ▶ *Calculating parallel periods for multiple dates in a set* (chapter 2)
- ▶ *Calculating various ranks* (chapter 4)

## Arbitrary shaped sets

Arbitrary shaped sets are sets which don't form a compact space. Usually, they result from operations with tuples, not sets.

What's specific about them is that they cannot be used in scopes. The solution is to cut them into several smaller non-arbitrary sets on which scopes can be applied. Deciding how to perform that cut should not be a problem and here's how. First, detect the outer boundaries of the scope, that is, hierarchies and the range that their min and max referred to members form. That's a potential space the scope's subcube can occupy. Now, visualize the part of the space that the specified scope occupies. If the two don't match, the shape is arbitrary. In other words, if there are holes inside the potential space, you need to break the scope's subcube into as little as possible smaller but compact subcubes (subcubes with no space in them) and use each in its own scope statement

Related recipes:

- ▶ *Detecting members on the same branch* (chapter 4)

## Varying attribute

Varying attribute is an attribute that is referred to inside a calculation and which depends on another cube object, for example, another dimension or a measure. The problem is that the varying attribute can prevent block-computation mode if the dependency is too complex to be resolved by the SSAS engine. The solution is, if possible at all, to expand the tuple carrying the varying attribute with the root member of dimensions not used in that tuple but mentioned elsewhere in the query. This can help SSAS to decide to evaluate the expression using the block-computation. Not always, but it's worth a try.

The concept of varying attribute is explained in more detail in the Analysis Services Performance Guide:

<http://tinyurl.com/PerfGuide2008>

## Static versus dynamic calculation

A calculation is estimated as static if it returns the same value for every cell. The named set defined in the query is an example of such calculation because it is evaluated only once and used as such in all cell calculations. Dynamic calculations are those estimated to return different results per cell. These functions typically reference the current members of hierarchies.

If SSAS estimates a calculation as being static and we want it to become dynamic, we can correct that estimate by inserting a non-deterministic expression inside the calculation. For example, using the `Rank()` function over the current member. This will naturally slow down the calculation, but we will have achieved what we wanted.

*Related recipes:*

- ▶ *Displaying random values (chapter 8)*
- ▶ *Displaying random sample of hierarchy members (chapter 8)*

## Late binding functions

Late-binding functions like the `StrToMember()` function are functions that cannot be evaluated in advance. SSAS naturally always tries to determine the result to be returned in advance order to estimate whether the block-computation mode is a viable solution or not. If the engine doesn't know what the result will look like, it can only use the cell-by-cell mode. What might help to know the result in advance in this case are two things. First, that the string passed in is a static expression. Second, that an optional argument is supplied, the keyword `CONSTRAINED`.

*Related recipes:*

- ▶ *Getting values on the last date with data (chapter 2)*
- ▶ *Calculating today's date using the string functions (chapter 2)*
- ▶ *Calculating parallel periods for multiple dates in a set (chapter 2)*

## Sparse versus dense expressions

Sparse expressions are expressions where only a small part of the subcube contains the values. This is where SSAS tries to use the block-computation mode, hence they are evaluated fast.

Dense expressions are expressions which are always (or most of the time) not null. They are usually evaluated in the cell-by-cell mode which is slow.

For performance reasons, it is always better to preserve the sparsity of expressions whenever possible by using nulls as the result of calculations, not 0 or some other scalar value.

*Related recipes:*

- ▶ *Setting special format for negative, zero, and null values* (chapter 1)
- ▶ *Finding the last date with data* (chapter 2)
- ▶ *Hiding calculation values on future dates* (chapter 2)

## Cache

The cache is an internal object where SSAS preserves the result of recent calculations. Cache can help speed up the query response time. We can also warm it up by firing a set of predetermined queries after the cube gets processed.

SSAS provides 3 types scopes of cache: global, session, and local cache. Only one cache scope can be used at a time.

*Related recipes:*

- ▶ *Calculating the difference between two times* (chapter 2)
- ▶ *Clearing Analysis Services cache* (chapter 9)

## Types of query

The last part of the Appendix lists the type of queries you can write against SQL Server Analysis Services and briefly explains each of them.

### MDX query

MDX query is a query written in MDX language and whose purpose is to return the result from an SSAS cube. It consists of the obligatory `SELECT` and `FROM` parts while the others are optional. If the query has axes, all axes up to the one with the highest ordinal must be used in the query. Providing an empty set (i.e. `{}`) is a way of bypassing one of the axes with a lower ordinal. In the `WITH` part of the query we can define calculated members, sets, and cells. In the `WHERE` part of the query we set the context to be used in query. That part is also known as the slicer. In the `FROM` part there can be a cube or another query, called the subquery or subselect. Cell properties and dimension properties are elements we can also include in the query. They bring additional information about objects used in the query.

Related recipes:

- ▶ [Skipping axis \(chapter 1\)](#)

## XMLA query

XMLA query is a type of language or protocol for issuing XMLA commands to an SSAS server. The commands are written in XML format, as is the result returned by the XMLA query. Commands range from processing the cube to modifying cube structure.

Related recipes:

- ▶ [Clearing Analysis Services cache \(chapter 9\)](#)

## Drillthrough

Drillthrough is a type of query and also a type of action available in SSAS cubes. This query returns detailed level data about a single cell. Data is returned directly from a single measure group in an SSAS cube, not its underlying fact table. Drillthrough works on regular members only.

Related recipes:

- ▶ [Performing custom drillthrough \(chapter 9\)](#)

## DMVs

Dynamic Management Views or DMVs are a special collection of system tables which can be queried using SQL-like syntax executed inside an MDX query. The server recognizes the SQL-like dialect and returns the data in tabular format.

There are several categories of DMVs; some are used to explore the cube structure, others to analyze the server's current state.

Related recipes:

- ▶ [Using SSAS Dynamic Management Views \(DMV\) to fast-document a cube \(chapter 9\)](#)
- ▶ [Using SSAS Dynamic Management Views \(DMVs\) to monitor activity and usage \(chapter 9\)](#)

## Stored procedures

Stored procedures are one or more functions implemented in a .NET assembly or COM DLL that is registered on the SSAS server. They extend the functionality of the server by providing new functions to be used in queries or to be called directly when required, outside of the query. Their biggest problem in their usage is that they run in cell-by-cell mode when used inside the definition of cells. They require expensive unmanaged marshalling for each call. Look for the summary of best practices for usage in this thread:

<http://tinyurl.com/DarrenStoredProcedures>

In order to write them, programming skills are required.

*Related recipes:*

- ▶ *Using Analysis Services stored procedures* (chapter 9)



# Index

## Symbols

[Data].[Date] attribute 86  
\$system schema 403  
.DataMember function 434  
.PrevMember function 379  
@offset parameter 73  
@span parameter 73  
CURRENT function 36

## A

ABC analysis  
implementing 241  
implementing, steps 241, 243  
implementing, working 244  
**Abs() function** 292  
**Accounts hierarchy** 339  
**AddCalculatedMembers() function** 44, 133  
**advanced MDX topics**  
about 337, 338  
complex sorts, performing 366  
cumulative value calculation, recursion used  
373  
leaves, displaying 338  
members with data in parent-child hierarchy,  
displaying 343  
random sample of hierarchy members,  
displaying 356  
random values, displaying 352  
Tally table utility dimension, displaying 347  
**Aggregate() function**  
about 66, 109, 132  
approaches 293  
**All member** 437, 441  
**ALLMEMBERS function** 365

**ALTER CUBE command** 21  
**ALTER To option** 386  
**Analysis Services**  
stored procedures, using 387  
**Analysis Services 2000**  
analysing, from 64-bit environment 400  
**Analysis Services cache**  
about 382  
ClearCache command, working 384, 385  
clearing 383-385  
cube object 385, 386  
post cache clearing step 387  
starting with 382  
tips 387  
**Ancestor() function** 187  
**arbitrary shaped sets, query optimization** 443  
**artificial members** 428  
**Ascendants() function** 167  
**Assemblies folder** 388  
**assignments, MDX script** 441  
**ASSP.InverseHierarchy() stored procedure**  
392  
**asymmetrical reporting** 438  
**AsymmetricSet() function** 30  
**attribute hierarchy**  
using, for today's date calculation 97, 98  
working 98  
Yes member 99  
**AttributeHierarchyVisible property** 251  
**attribute overwrites, MDX query** 435  
**attribute, query optimization**  
varying 443  
**attribute relationships, cube and dimension**  
design 437  
**auto-exists algorithm** 173

**average calculations**  
about 189  
Employee.Employees hierarchy 192  
empty rows, preserving 192, 193  
Siblings function, working 191  
specifics 193, 194  
starting with 189  
steps 190, 191

**Avg() function 190**

**axes, skipping**  
about 8  
concept 10  
one-dimensional query result, obtaining 9  
workaround 10  
working 9

**AXIS(1) 8**

**Axis() function 300, 377, 430**

**axis, MDX query 430, 431**

**axis, with measures**  
additional axes 304  
Axis() function, working 302, 303  
detecting 301  
Visible property 304

**axis, without measures**  
additional axes 307  
detecting 304, 305  
Visible property 306  
working 305, 306

**B**

**Back\_Color() functions 19**

**BACK\_COLOR property 151**

**best/worst value members**  
additional information, including 140, 142  
BACK\_COLOR property, working 145-149  
identifying 138-140  
multiple member's caption, displaying 142  
siblings, highlighting 143-145  
TopCount() function, working 140  
troubleshooting 149

**bit-string**  
about 319  
calculation visibility 324  
example 324  
for hierarchies on rows, calculating 320-323  
SSAS front-ends 324

string indicator, building 320

**block-computation, query optimization**  
versus cell-by-cell evaluation mode 443

**BottomCount() function 123, 125**

**BottomSum() function 125**

**bubble-up exceptions**  
about 149  
Descendants() function 151  
implementing 149, 151  
issues 153  
practical value 153  
working 151

**Business Intelligence Development Studio (BIDS) 165, 213, 348**

**C**

**cache, query optimization 445**

**CALCULATE command 84**

**calculated members, MDX query 428**

**calculate statement, MDX script 440**

**Cartesian product 178**

**cell-by-cell evaluation mode, query optimization**  
versus block-computation 443

**cell properties, MDX query 432**

**ClearCache command 384**

**CoalesceEmpty() function 193**

**column number, for Excel**  
adjusting 307-310

**column number, for OWC**  
adjusting 307-310

**complex sorts**  
about 366  
important points 371-373  
performing 367-369  
sorting 373  
working 370, 371

**conditional formatting**  
applying, on calculations 17, 18  
tips 19  
warning 19, 20  
working 18

**CONSTRAINED flag 82**

**content of axes**  
identifying 310-313  
potential problems 315

replacing 315  
 working 313, 314

**context-aware calculations**

- about 295, 296, 436
- building 297
- example 336
- issues, solving 336
- performance calculation 335
- pivot features 296
- utility dimension, implementing 327-333
- visibility calculation 336

**Count() function 193, 202**

**CREATE To option 386**

**cube and dimension design**

- about 437
- attribute relationships 437
- default member 437
- deployment versus processing 440
- dummy dimension 439
- granularity 439
- natural versus unnatural hierarchies 437
- parent-child hierarchies 438
- utility dimension 438, 439

**Cube Browser tab 22**

**CubelD tags 384**

**cumulative value calculation, recursion used**

- about 374
- algorithms, working 376, 377
- selecting 379
- solution, version 378, 379
- steps 374, 375

**current context, MDX query 434**

**CurrentMember function 114, 158, 162, 434**

**CurrentOrdinal function 323, 365**

**CURRENTORDINAL function 36**

**custom drillthrough**

- about 416, 417
- examples 421
- functions 420
- performing 418
- working 418-420

**Customer Count measure 236**

**customers fluctuation**

- analyzing 233
- analyzing, steps 234, 235
- analyzing, working 235, 236

**D**

**Database Properties window 383**

**data members, MDX query 434**

**Date.Date attribute 231**

**date differences**

- calculating 100
- DateDiff() function, working 101
- non-consecutive dates problem 102
- other date scenario 102

**DateDiff() function 102, 233**

**default member**

- defining 20
- environment, setting 21
- example 22, 23
- setting 21
- tips 23
- working 22

**default member, cube and dimension design 437**

**DefaultMember property 20**

**DELETE To option 386**

**dense expressions, query optimization**

- versus sparse expressions 444, 445

**departments**

- non-allocated company cycles, allocating to 219-227

**deployment, cube and dimension design**

- versus processing 440

**Descendants() function**

- about 36, 151, 152, 167, 340
- options 171

**dimension 178**

**dimensionality, MDX query 434**

**Dimension calculated measure 363**

**Dimension Designer. Dimension properties 432**

**DIMENSION PROPERTIES keyword 24**

**dimension properties, MDX query 432**

**DistinctCount 66**

**DMVs 382, 446**

**DOWNLOAD button 388**

**drillthrough 446**

**DRILLTHROUGH command 418, 421, 442**

**DRILLTHROUGH type 416**

**dummy dimension**

- DSV 265

**DW** 265  
total percentage calculation 265  
using, for non-existing histogram  
    implementation 257-262

working 262-264

**dummy dimension, cube and dimension design** 439

**dummy regular members** 428

**dynamic calculation, query optimization**  
    versus static calculation 444

**Dynamic Management Views.** *See DMVs*

**dynamic, MDX script**

versus static sets 442

**Dynamic parameter option** 395

## E

**empty rows**

multiple measure 326  
preserving 324, 325  
working 326

**error handling**

about 11  
division by zero errors 10  
division by zero errors, handling 11, 12  
division by zero errors, working 12  
requirements 11

**errors, MDX query** 436

**event**

Query Begin 415  
Query End 415

**Except() function** 25

**Existing function** 114

**Existing() function** 176

**EXISTING keyword**

about 79, 176  
tips 176  
using 175  
versus Filter() 177  
working 175

**EXISTING operator** 78, 232, 266

**Exists() function**

about 180, 240  
working 181

**explicit members, MDX query** 433

**Extract() function** 163

## F

**Filter() function**

about 37, 96, 133, 202  
versus Exists() 177

**flexible display units**

implementing, utility dimension used 275-  
    277

**forecasting**

linear regression used 206-212  
periodic cycles used 212, 213

**Fore\_Color() functions** 19

**FORE\_COLOR property** 18, 151

**Format() function** 82

**Format\_String() function** 15

**FORMAT\_STRING property** 13

**Format\_String() statement** 280

**FORMATTED\_VALUE** 13

**Formatted\_Value properties** 432

**formatted values**

troubleshooting 16

**Freeze() statement** 227, 441

**future dates calculation value**

about 83  
hiding 85, 86  
starting with 84  
working 86

## G

**Generate() function** 109, 110, 370

**GENERATE() function** 34

**GO statement** 387

**granularity, cube and dimension design** 439

## H

**Head() function** 335

**HideMemberIf option** 438

**Hierarchize() function** 392

**hierarchy**

combining, into one 134, 135  
limitation 137  
working 136, 137

**Hierarchy %** 186, 187

**highest result members**  
    extracting 116-118

result, verifying 121  
 TopCount() function, working 118-121

**histograms**  
 about 254  
 Attribute-based histograms, working 256  
 distinct count measure, working 256  
 implementing, distinct count measure used 254, 255  
 implementing, dummy dimension using 257-262  
 implementing, over existing hierarchies 254-256  
 implementing, over non-existing hierarchies 257-262  
 maximum frequency number, obtaining 262, 263

**I**

**I Agree button** 388  
**iif() function** 77, 140, 159, 230, 232  
**IIF() function** 12  
**IIF() statement** 12  
**implicit members, MDX query** 433  
**in calculations only** 430  
**Input Y** 208  
**InStr() function** 323  
**Intersect() function** 160, 244  
**IsAggregatable property** 29, 437  
**IsAncestor() function** 170  
**IsEmpty() function** 355, 359  
**IsError() function** 299, 302, 436  
**Is keyword** 159  
**IsLeaf()** 347  
**Item() function** 81, 163, 188, 217, 376  
**iteration, MDX query** 435

**L**

**last date with data**  
 calculated measure, creating 76, 77  
 finding 74, 75  
 working 77

**last date with data values**  
 format, using 82, 83  
 obtaining 79-81  
 time-non-sensitive calculation, optimizing 83  
 working 81

**LastPeriods() function** 73  
**LastSibling function** 291  
**late binding functions, query optimization** 444  
**leaves.** *See members without children*  
**LEAVES flag** 340, 343  
**Level %** 186  
**Levels() function** 365  
**linear regression**  
 used, for forecasting 206-212  
**LinRegIntercept() function** 212  
**LinRegPoint() function** 208, 210, 212  
**logical AND**  
 implementing, same member hierarchy, using 51-53  
 implementing, steps 53, 54  
 working 55, 56

**logical OR**  
 about 27  
 applying, with different hierarchy members 28, 29  
 complex scenario 30  
 requirements 27  
 working 29

**Lost Customers measure** 236

**lowest result members**  
 BottomCount() function, working 124, 125  
 BottomPercent() function 125  
 BottomSum() function 125  
 extracting 123  
 isolating 122-124

**M**

**Max() function** 78, 146, 232, 233  
**MDX**  
 calculations 248  
 FOR loop 37  
**MDX language** 423, 427  
**MDX query**  
 about 41, 445  
 additional information 44  
 debugging 41, 42  
 dissecting 41, 42  
 optimizing, NonEmpty() function used 48, 49  
 working 43, 44

**MDX query, in action**  
attribute overwrites 435  
context-aware calculations 436  
current context 434  
data members 434  
dimensionality 434  
errors 436  
explicit members 433  
implicit members 433  
iteration 435  
recursion 435, 436  
set-based operations 436  
visual totals 435

**MDX query, parts**  
about 427  
axis 430, 431  
calculated members 428  
cell properties 432  
dimension properties 432  
named sets 429, 430  
regular members 427, 428  
set alias 430  
slicer 431  
subquery 431, 432  
tuple 428, 429

**MDX script**  
about 440  
assignments 441  
calculate statement 440  
dynamic versus static sets 442  
scopes 440

**MDX Studio**  
about 424  
online versions 425

**member, separating from level**  
MDX, searching 253  
new attribute, using 249-251  
new attribute, working 252, 253  
scenarios 253

**members with data in parent-child hierarchy**  
alternatives 347  
displaying 343-346  
working 346

**MembersWithData property 434**

**members without children**  
Descendants() function 343  
displaying 338-340

optional arguments, avoiding 343  
ragged hierarchies, problems 343  
working 340-342

**MemberValue function**  
about 77, 231, 419  
using, for today's date calculation 94-96

**MemberValuefunction 76**

**MemberValue property 26, 229, 277, 278, 351, 366**

**MemberWithData property 347**

**Min() function 146**

**Model Name attribute 372**

**moving average**  
about 71  
calculating 72  
calculating, options 74  
future dates 74  
LastPeriods() function 73  
starting with 71  
working 73

**multidimensional cubes**  
about 156  
advantages 155

## N

**named sets, MDX query 429, 430**  
**natural hierarchies, cube and dimension**

**design**

versus unnatural hierarchies 437

**NEB property**

about 45  
adding, to calculation 46  
advantage 47  
using 45, 46, 47  
working 47

**negative format value**

FORMAT\_STRING Contents (MDX) option 16  
formatted values, troubleshooting 16  
LANGUAGE and FORMAT\_STRING on  
FORMATTED\_VALUE 16  
precaution 16  
setting 13, 14  
tips 15  
working 14

**nesting scopes, MDX script 441**

**new member set, creating**  
from initial one 34, 35  
Generate() function, working 36

**New Query button 110, 156, 214, 344**

**New Query icon 394**

**New XMLA Query button 382**

**non-allocated company expenses**  
allocating, to departments 219-227

**non-balanced hierarchies**  
ragged hierarchy 338

**NonEmpty() 238**

**NON\_EMPTY\_BEHAVIOR. See also NEB property**

**NON\_EMPTY\_BEHAVIOR property 8, 65**

**NonEmpty() function**  
about 8, 125, 217, 238, 298  
benefits 50  
errors 50  
NON EMPTY 50  
using, for MDX query optimization 48, 49

**NON EMPTY keyword 65, 355**

**NOT IN set logic**  
about 23  
implementing 24  
reversing 24, 25  
working 25, 26

**null format value**  
setting 13, 14  
working 14

**NullHandling property 16**

**NullProcessing property 269**

**Number of Days measures 230**

**O**

**OpenQuery command 398, 399**

**OpenRowset command 399**

**Order Count measure 232**

**Order() function 122, 232**

**Output X 208**

**OVER SQL clause 349**

**P**

**ParallelPeriod() function 69, 110, 291**

**parallel periods**  
calculating, for multiple dates in set 106-109

calculating, for multiple dates in slicer 110-114  
date set parameters 109  
design improving 110

**parent-child hierarchies 338**

**parent-child hierarchies, cube and dimension design 438**

**particular member**  
complex combination, detecting 160  
CurrentMember function, working 158  
detecting 156-158  
iif() function 159  
indicator measure, need for 159  
member comparison, versus value comparison 159  
permanent character 158  
Scope() statement 159

**percentage**  
about 183  
calculating 184  
calculating, steps 185, 186  
Hierarchy % 186  
leaf member values 189  
Level % 186  
non-existing root member 188, 189  
Parent %, working 186  
parent member, alternative member 188  
use cases 187

**periodic cycles**  
used, for forecasting 212-218

**PeriodsToDate() function 65**

**Permissions option 389**

**physical measure implementation, as placeholder**  
associated measure group 271  
for MDX assignments 266-269  
working 269, 270

**potential subcube 290**

**PowerShell**  
Yes member 99

**predefined structure only 439**

**Preserve option 269**

**processing, cube and dimension design**  
versus deployment 440

**Product Line attribute 372, 373**

**products**  
analyzing 271, 272

cube model, modifying 272, 273  
Price dimension 274  
prices, analyzing 271, 272  
Reseller Transaction Count measure 274

## Q

**query-based alternative, same branch**  
**members 169, 170, 171**  
**Query Begin event 415**  
**Query End event 415**  
**query optimization**  
about 442  
arbitrary shaped sets 443  
block-computation versus cell-by-cell evaluation mode 443  
cache 445  
dense expressions 444, 445  
late binding functions 444  
sparse expressions 444, 445  
static versus dynamic calculation 444  
varying attribute 443  
**query returning column count**  
Axis() function, working 299  
identifying 297-299  
SQL Server Management Studio 300  
**query returning row count**  
Axis() function, working 299  
calculation visibility 301  
identifying 297-299  
SQL Server Management Studio 300  
**query, types**  
about 445  
DMVs 446  
Drillthrough 446  
MDX query 445  
stored procedures 447

## R

**ragged hierarchy 183**  
**RandomSample function 359**  
**random sample of hierarchy members**  
displaying 356-358  
IsEmpty() function 359  
working 358  
**random values**  
displaying 352-354

Rnd() function, working 354, 355

### **Rank function 245**

#### **Rank() function**

about 212, 355, 444  
variant 197  
working 196

#### **ranks**

about 194  
calculating, steps 195, 196  
empty rows, preserving 197-201  
in multidimensional sets 201  
named sets, cons 202  
named sets, pros 202  
Rank() function, working 196

#### **recursion**

average of an average, calculating 37-39  
SSAS, versions 40  
using 40  
working 39, 40

#### **recursion, MDX query 435**

**recursive hierarchy.** *See parent-child hierarchies*

#### **regular members, MDX query 427, 428**

#### **related member search, in another dimension**

about 178  
alternative 181, 182  
leaf calculation 182  
non-leaf calculation 182  
starting with 179  
steps 180  
working 181

#### **related member search, in same dimension**

about 172  
EXISTING keyword, tips 176  
starting with 172, 173  
steps 174  
warning 177  
working 175

#### **reports**

concising 115, 116

#### **RETURN keyword 419**

#### **RGB() function 18**

#### **Rnd() function 354**

**rolling average.** *See moving average*

#### **Root() function 163**

#### **root member**

about 160

CurrentMember function, working 162, 163  
detecting 161  
scope-based solution 164

**row number, for Excel**  
adjusting 307-310

**row number, for OWC**  
adjusting 307-310

**ROW\_NUMBER operator** 350

**row numbers**  
calculating 316  
performance calculation 318  
related calculations 318  
SSAS front-ends 319  
Visible property 318  
working 317

**S**

**Sales YTD measure** 218

**same branch members**  
Descendants() function, options 171  
detecting 164-166  
query-based alternative 169, 170  
search, selecting 171  
working 166-169

**sample, displaying from random hierarchy**  
about 359  
Cardinality calculated measure 363  
Dimension calculated measure 362  
Hierarchy calculated measure 363  
Measure calculated measure 363  
N/A sign 366  
Ratio calculated measure 363  
Reseller Sales Amount measure, using 366  
steps 359-362  
Tally Table dimension 365  
value calculated measure 363

**Scope command** 308

**scopes, MDX script** 440

**Scope() statement** 159, 226

**SELF\_AND\_BEFORE flag** 341

**SELF flag** 343

**set alias, MDX query** 430

**set-based operations, MDX query** 436

**set iteration**  
about 30  
DESCENDANTS() function 32

FILTER() function, using 32  
initial set, reducing 31, 32  
performing 31  
query improvements, hints 33  
recursion, using 37

**Show all events option** 413

**Siblings function** 190

**slicer, MDX query** 431

**slow-moving goods**  
identifying 227-232

**sparse expressions, query optimization**  
versus dense expressions 444, 445

**SQL Server Management Studio** 382

**SSAS**  
about 8  
earlier versions 13  
Wiki site 423

**SSAS DMVs**  
about 400, 401, 406  
restrictions 409-411  
using, for activity monitoring 406-408  
using, for activity usage 406-408  
working 408, 409

**SSAS front-ends generated MDX queries**  
about 411  
capturing 411-415  
solution 415  
tips 416  
working 415

**SSAS-Info** 423

**SSMS** 8

**static calculation, query optimization**  
versus dynamic calculation 444

**static sets, MDX script**  
versus dynamic sets 442

**Stop Selected Trace button** 415

**stored procedures** 447

**stored procedures, Analysis Services**  
about 387, 388  
existing VBA functions 393  
GO statement 391, 392  
tips 392  
using 389, 390  
working 390, 391

**string functions**  
IsError()) 44  
MemberToStr 44

**SetToStr** 44  
 using, for today's date calculation 87-89  
 working 90-92  
**StrToMember() function** **82, 89, 444**  
**subcube** **166**  
**subquery, MDX query** **431, 432**  
**Subtotal OK measure** **345**  
**Sum() function** **112, 202**  
**SUM() function** **62**  
**SUM-IF combination** **32**

**T**

**Tail() function** **81, 188**  
**Tally table utility dimension**  
 about 347, 348  
 implementing 348-350  
 working 350-352  
**time-based calculation**  
 approaches 294  
 implementing, utility dimension used 281-292  
**time differences**  
 calculating 103  
 duration formatting 105  
 duration formatting, web examples 105  
 working 104  
**Time Intelligence Wizard** **281**  
**today's date**  
 Adventure Works cube 93  
 calculating, attribute hierarchy used 97, 98  
 calculating, MemberValue function used 94, 95  
 calculating, string function used 87-89  
 many-to-many relationships 99  
 MemberValue function, working 96  
 Now() function 93  
 problems 94  
 relative periods 93  
 string function, working 90-92  
 ValueColumn property, using, in Date dimension 96  
**TopCount() calculation relative, to another hierarchy**  
 creating 125-127  
 Generate() function, working 128-130

relative context support, in SSAS front-ends 130  
**TopCount() function** **36, 116, 118, 119, 131, 140, 233, 274, 358**  
**TopPercent() function** **125, 244**  
**TopSum() function** **116, 122**  
**Total of all members**  
 displaying 130, 132  
 Reseller.Reseller hierarchy 133  
 top N members, isolating 132  
**true dynamic evaluation** **430**  
**T-SQL environment**  
 linked server, troubleshooting 400  
 MDX queries, executing 394-397  
 SSAS server, working 397-399  
**tuple, MDX query**  
 about 428, 429  
 dimensionality 429  
**Type property** **349**

**U**

**UnaryOperator function** **419**  
**unbalanced hierarchy.** *See parent-child hierarchies*  
**UNION() function** **29**  
**unnatural hierarchies, cube and dimension design**  
 versus natural hierarchies 437  
**utility dimension**  
 calculation visibility 336  
 filtered set approach, format string 279, 280  
 fine-tuning the calculations 294  
 implementing, context-aware calculations used 327-333  
 issues, solving 336  
 MemberValue property 277  
 performance calculation 335  
 set-based approach 278, 279  
 using, for flexible display units display 275-277  
 using, for time-based calculation implementation 281-293  
 working 277, 333, 334  
**utility dimension, cube and dimension design**  
**438, 439**

## V

**ValueColumn** property **77, 101**  
**Value properties** **432**  
**VBA functions, in MDX** **44**  
**Visible property** **301**  
**VisualTotals() function** **435**  
**visual totals, MDX query** **435**

## X

**XMLA query** **446**

## Y

**Year-over-Year.** *See* **YoY growth**  
**Year-To-Date.** *See* **YTD value**  
**YoY growth**  
calculating **66, 67, 68**

ParallelPeriod() function, working **68-71**  
**YTD() function** **64, 293**

### **YTD value**

Aggregate() function **66**  
calculating **61**  
example **63**  
Inception-To-Date calculation **63**  
issues **64, 65**  
precaution **63**  
query **60**  
solutions **64, 65**  
starting with **60**  
working **62**

## Z

**zero format value**  
setting **13, 14**  
working **14**





## Thank you for buying **MDX with Microsoft SQL Server 2008 R2 Analysis Services: Cookbook**

### About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.PacktPub.com](http://www.PacktPub.com).

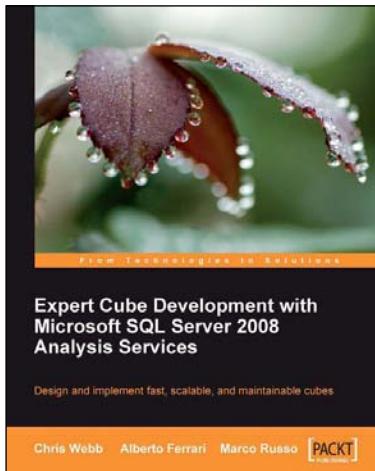
### About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

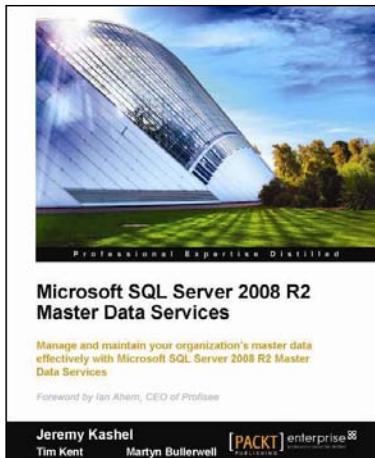


## Expert Cube Development with Microsoft SQL Server 2008 Analysis Services

ISBN: 978-1-847197-22-1      Paperback: 360 pages

Design and implement fast, scalable and maintainable cubes with Microsoft SQL Server 2008 Analysis Services with this book and eBook

1. A real-world guide to designing cubes with Analysis Services 2008
2. Model dimensions and measure groups in BI Development Studio
3. Implement security, drill-through, and MDX calculations



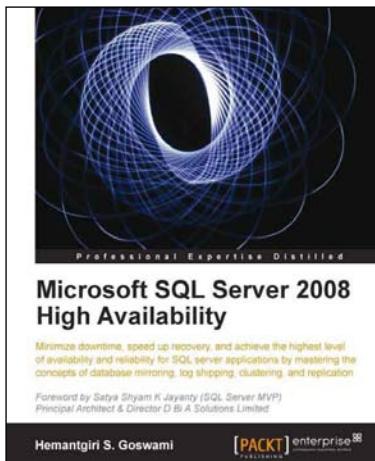
## Microsoft SQL Server 2008 R2 Master Data Services

ISBN: 978-1-849680-50-9      Paperback: 300 pages

Manage and maintain your organization's master data effectively with Microsoft SQL Server 2008 R2 Master Data Services with this book and eBook

1. Gain a comprehensive guide to Microsoft SQL Server R2 Master Data Services (MDS) with this book and eBook
2. Explains the background to the practice of Master Data Management and how it can help organizations
3. Introduces Master Data Services, and provides a step-by-step installation guide

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

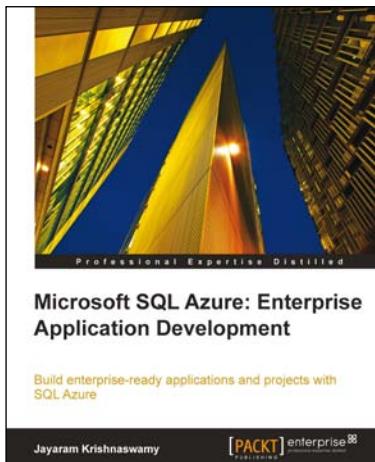


## Microsoft SQL Server 2008 High Availability

ISBN: 978-1-849681-22-3      Paperback: 308 pages

Minimize downtime, speed up recovery, and achieve the top level of high availability and reliability for Microsoft SQL Server applications with this book and eBook

1. Install various SQL Server High Availability options in a step-by-step manner
2. A guide to SQL Server High Availability for DBA aspirants, proficient developers and system administrators
3. Learn the pre and post installation concepts and common issues you come across while working on SQL Server High Availability



## Microsoft SQL Azure Enterprise Application Development

ISBN: 978-1-849680-80-6      Paperback: 420 pages

Build enterprise-ready applications and projects with Microsoft SQL Azure using this book and eBook

1. Develop large scale enterprise applications using Microsoft SQL Azure
2. Understand how to use the various third party programs such as DB Artisan, RedGate, ToadSoft etc developed for SQL Azure
3. Master the exhaustive Data migration and Data Synchronization aspects of SQL Azure.
4. Includes SQL Azure projects in incubation and more recent developments including all 2010 updates

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles