

SQL Server Technical Article

Analysis Services Performance Guide for SQL Server 2012 and SQL Server 2014

**Summary:** This white paper describes how business intelligence developers can apply query and processing performance-tuning techniques to their OLAP solutions running on Microsoft SQL Server Analysis Services.

This paper is based on the performance guide for 2008 R2 and has been reviewed and updated to validate performance on SQL Server 2012 and SQL Server 2014.

**Writers:** Karan Gulati and Jon Burchel

**Editor:** Jeannine Takaki

**Contributors and Technical Reviewers:** Akshai Mirchandani, Siva Harinath, Lisa Liu

**Published:** May 2014

**Applies to:** SQL Server 2012 (including 2012 SP1), SQL Server 2014

Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2014 Microsoft Corporation. All rights reserved.

Microsoft, Excel, SQL Server, Visual Basic, Visual Studio, and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

[1 Introduction 7](#_Toc387860792)

[1.1 Overview of Performance Goals 7](#_Toc387860793)

[2 Design Patterns for Scalable Cubes 7](#_Toc387860794)

[2.1 Building Optimal Dimensions 8](#_Toc387860795)

[2.1.1 Using the KeyColumns, ValueColumn, and NameColumn Properties Effectively 8](#_Toc387860796)

[2.1.2 Hiding Attribute Hierarchies 9](#_Toc387860797)

[2.1.3 Setting or Disabling Ordering of Attributes 9](#_Toc387860798)

[2.1.4 Setting Default Attribute Members 10](#_Toc387860799)

[2.1.5 Removing the All Level 10](#_Toc387860800)

[2.1.6 Identifying Attribute Relationships 10](#_Toc387860801)

[2.1.7 Using Hierarchies Effectively 12](#_Toc387860802)

[2.1.8 Turning Off the Attribute Hierarchy 13](#_Toc387860803)

[2.1.9 Reference Dimensions 14](#_Toc387860804)

[2.1.10 Fast-Changing Attributes 15](#_Toc387860805)

[2.1.11 Large Dimensions 18](#_Toc387860806)

[2.2 Partitioning a Cube 20](#_Toc387860807)

[2.2.1 Partition Slicing 21](#_Toc387860808)

[2.2.2 Partition Sizing 23](#_Toc387860809)

[2.2.3 Partition Strategy 24](#_Toc387860810)

[2.3 Relational Data Source Design 28](#_Toc387860811)

[2.3.1 Use a Star Schema for Best Performance 29](#_Toc387860812)

[2.3.2 Consider Moving Calculations to the Relational Engine 29](#_Toc387860813)

[2.3.3 Use Views 30](#_Toc387860814)

[2.4 Calculation Scripts 32](#_Toc387860815)

[2.4.1 Learn MDX Basics 32](#_Toc387860816)

[2.4.2 Use Attributes Instead of Sets 32](#_Toc387860817)

[2.4.3 Use SCOPE Instead of IIF When Addressing Cube Space 33](#_Toc387860818)

[2.4.4 Avoid Mimicking Engine Features with Expressions 34](#_Toc387860819)

[2.4.5 Comparing Objects and Values 35](#_Toc387860820)

[2.4.6 Evaluating Set Membership 35](#_Toc387860821)

[3 Tuning Query Performance 36](#_Toc387860822)

[3.1 Query Processor Architecture 36](#_Toc387860823)

[3.1.1 Session Management 37](#_Toc387860824)

[3.1.2 Query Processing 38](#_Toc387860825)

[3.1.3 Data Retrieval 41](#_Toc387860826)

[3.2 Query Processor Internals 43](#_Toc387860827)

[3.2.1 Subspace Computation 43](#_Toc387860828)

[3.2.2 Expensive vs. Inexpensive Query Plans 45](#_Toc387860829)

[3.2.3 Expression Sparsity 45](#_Toc387860830)

[3.2.4 Default Values 46](#_Toc387860831)

[3.2.5 Varying Attributes 47](#_Toc387860832)

[3.3 Optimizing MDX 47](#_Toc387860833)

[3.3.1 Creating a Query Speed Baseline 48](#_Toc387860834)

[3.3.2 Isolating the Problem 50](#_Toc387860835)

[3.3.3 Cell-by-Cell Mode vs. Subspace Mode 52](#_Toc387860836)

[3.3.4 Avoid Assigning Non Null Values to Otherwise Empty Cells 54](#_Toc387860837)

[3.3.5 Sparse/Dense Considerations with “expr1 \* expr2” Expressions 55](#_Toc387860838)

[3.3.6 IIf Function in SQL Server 2008 Analysis Services 56](#_Toc387860839)

[3.3.7 Cache Partial Expressions and Cell Properties 60](#_Toc387860840)

[3.3.8 Eliminate Varying Attributes in Set Expressions 61](#_Toc387860841)

[3.3.9 Eliminate Cost of Computing Formatted Values 62](#_Toc387860842)

[3.3.10 NON\_EMPTY\_BEHAVIOR 63](#_Toc387860843)

[3.4 Aggregations 64](#_Toc387860844)

[3.4.1 Detecting Aggregation Hits 65](#_Toc387860845)

[3.4.2 How to Interpret Aggregations 66](#_Toc387860846)

[3.4.3 Aggregation Tradeoffs 67](#_Toc387860847)

[3.4.4 Building Aggregations 67](#_Toc387860848)

[3.5 Cache Warming 71](#_Toc387860849)

[3.5.1 Cache Warming Guidelines 71](#_Toc387860850)

[3.5.2 Implementing a Cache Warming Strategy 72](#_Toc387860851)

[3.6 Scale-Out 73](#_Toc387860852)

[4 Tuning Processing Performance 74](#_Toc387860853)

[4.1 Baselining Processing 74](#_Toc387860854)

[4.1.1 Performance Monitor Trace 74](#_Toc387860855)

[4.1.2 Profiler Trace 75](#_Toc387860856)

[4.1.3 Determining Where You Spend Processing Time 76](#_Toc387860857)

[4.2 Tuning Dimension Processing 77](#_Toc387860858)

[4.2.1 Dimension Processing Architecture 77](#_Toc387860859)

[4.2.2 Dimension Processing Commands 80](#_Toc387860860)

[4.3 Tuning Cube Dimension Processing 81](#_Toc387860861)

[4.3.1 Reduce Attribute Overhead 82](#_Toc387860862)

[4.3.2 Tuning the Relational Dimension Processing Queries 83](#_Toc387860863)

[4.4 Tuning Partition Processing 83](#_Toc387860864)

[4.4.1 Partition Processing Architecture 84](#_Toc387860865)

[4.4.2 Partition Processing Commands 84](#_Toc387860866)

[4.4.3 Partition Processing Performance Best Practices 85](#_Toc387860867)

[4.4.4 Optimizing Data Inserts, Updates, and Deletes 85](#_Toc387860868)

[4.4.5 Picking Efficient Data Types in Fact Tables 87](#_Toc387860869)

[4.4.6 Tuning the Relational Partition Processing Query 87](#_Toc387860870)

[4.4.7 Splitting Processing Index and Process Data 88](#_Toc387860871)

[4.4.8 Increasing Concurrency by Adding More Partitions 89](#_Toc387860872)

[4.4.9 Adjusting Maximum Number of Connections 89](#_Toc387860873)

[4.4.10 Tuning the Process Index Phase 90](#_Toc387860874)

[4.4.11 Partitioning the Relational Source 91](#_Toc387860875)

[5 Special Considerations 91](#_Toc387860876)

[5.1 Distinct Count 92](#_Toc387860877)

[5.1.1 Partition Design 92](#_Toc387860878)

[5.1.2 Processing of Distinct Count 92](#_Toc387860879)

[5.1.3 Distinct Count Partition Aggregation Considerations 93](#_Toc387860880)

[5.1.4 Optimize the Disk Subsystem for Random I/O 94](#_Toc387860881)

[5.2 Large Many-to-Many Dimensions 94](#_Toc387860882)

[5.3 Parent-Child Dimensions 96](#_Toc387860883)

[5.4 Near Real Time and ROLAP 97](#_Toc387860884)

[5.4.1 MOLAP Switching 98](#_Toc387860885)

[5.4.2 ROLAP + MOLAP 99](#_Toc387860886)

[5.4.3 Comparing MOLAP Switching and ROLAP + MOLAP 99](#_Toc387860887)

[5.4.4 ROLAP 100](#_Toc387860888)

[5.5 NUMA 103](#_Toc387860889)

[5.5.1 NUMA Optimizations in SSAS 104](#_Toc387860890)

[5.5.2 General NUMA Tips 104](#_Toc387860891)

[5.5.3 Specific NUMA Configurations for SSAS 105](#_Toc387860892)

[5.5.4 Addenda: NUMA with Tabular Models 110](#_Toc387860893)

[6 Conclusion 110](#_Toc387860894)

[Send feedback 110](#_Toc387860895)

[7 Resources 110](#_Toc387860896)

# Introduction

This guide contains a collection of tips and design strategies to help you build and tune Analysis Services cubes for the best possible performance. This guide is primarily aimed at business intelligence (BI) developers who are building a new cube from scratch or optimizing an existing cube for better performance.

The goal of this guide is to provide you with the necessary background to understand design tradeoffs, and to suggest techniques and design patterns that can help you achieve the best possible performance of even large cubes.

The guide was previously published in 2008, and has been updated to cover changes in SQL Server 2012 and SQL Server 2014.

## Overview of Performance Goals

Cube performance can be assessed by looking at two types of workloads: query performance, and processing performance. Because these workloads are very different, this paper is organized into the following four sections, divided equally among query and processing performance advice.

**Design Patterns for Scalable Cubes** – No amount of query tuning and optimization can match the benefits of a well-designed data model. This section contains guidance to help you get the design right the first time. In general, good cube design follows Kimball modeling techniques, and if you avoid some typical design mistakes, you are in very good shape.

**Tuning Query Performance** - Query performance directly affects the quality of the end-user experience. As such, it is the primary benchmark used to evaluate the success of an online analytical processing (OLAP) implementation. Analysis Services provides a variety of mechanisms to accelerate query performance, including aggregations, caching, and indexed data retrieval. This section also provides guidance on writing efficient Multidimensional Expressions (MDX) calculation scripts.

**Tuning Processing Performance** - Processing is the operation that refreshes data in an Analysis Services database. The faster the processing performance, the sooner users can access refreshed data. Analysis Services provides a variety of mechanisms that you can use to influence processing performance, including parallelized processing designs, relational tuning, and an economical processing strategy (for example, incremental versus full refresh versus proactive caching).

**Special Considerations** – Some features of Analysis Services such as distinct count measures and many-to-many dimensions require careful attention to cube design. At the end of the paper you will find a section that describes the special techniques you should apply when using these features.

# Design Patterns for Scalable Cubes

Cubes present a unique challenge to the BI developer: they are databases that are expected to respond quickly to most queries. Depending on the data model you implement, the end user might have considerable freedom to create ad hoc queries. Achieving a balance between user freedom and scalable design will determine the success of a cube.

Each industry has specific design patterns that lend themselves to value-added reporting. A detailed treatment of optimal, industry specific data model is outside the scope of this document, but there are many common design patterns that you can apply across all industries. This section explains these patterns and how you can leverage them for increased scalability in your cube design.

## Building Optimal Dimensions

A well-tuned **dimension design** is a critical success factor for a high-performing Analysis Services solution. The dimensions in the cube are the first place users look when performing data analysis and their design has a deep impact on the performance of all measures in the cube.

Dimensions are composed of attributes, which are related to each other through hierarchies. Efficient use of attributes is a key design skill to master, and studying and implementing the attribute relationships available in the business model can help improve cube performance.

In this section, you will find guidance on how to optimize dimensions and properly use attributes and hierarchies.

### Using the KeyColumns, ValueColumn, and NameColumn Properties Effectively

When you add a new attribute to a dimension, three properties are used to define the attribute.

* The **KeyColumns** property specifies one or more source fields that uniquely identify each instance of the attribute.
* The **NameColumn** property specifies the source field that will be displayed to end users. If you do not specify a value for the **NameColumn** property, it is automatically set to the value of the **KeyColumns** property.
* **ValueColumn** allows you to store additional information about the attribute, which is typically used for calculations. Unlike member properties, this property of an attribute is strongly typed, providing increased performance when it is used in calculations. The contents of this property can be accessed through the **MemberValue** MDX function.

We recommend that you use both **ValueColumn** and **NameColumn** in your dimension design, because smart use of these properties can eliminate the need for extraneous attributes. Reducing the total number of attributes used in your design also makes it more efficient.

Additionally, use these practices to reduce processing time, reduce the size of the dimension, and minimize the likelihood of user errors:

* Assign a numeric source field, if available, to the **KeyColumns** property, rather than a string property.
* Use a single column key instead of a composite, multi-column key. This is especially true for attributes that have a large number of members, that is, greater than one million members.

### Hiding Attribute Hierarchies

For many dimensions, you will want the user to navigate hierarchies that are created for ease of access. For example, a Customer dimension might be navigated by drilling into Country and City before reaching the customer name, or by drilling through age groups or income levels. Such hierarchies, covered in more detail later, make navigation of the cube easier, and make queries more efficient.

In addition to user hierarchies, Analysis Services by default creates a flat hierarchy for every attribute in a dimension. These automatically generated hierarchies are called **attribute hierarchies**. Hiding attribute hierarchies is often a good idea, because a lot of hierarchies in a single dimension will typically confuse users and make client queries less efficient. Consider setting **AttributeHierarchyVisible** = **false** for most attribute hierarchies and expose user hierarchies instead.

#### Hiding the Surrogate Key

It is often a good idea to hide the surrogate key attribute in the dimension. The reason is that if you expose the surrogate key to the client tools as a **ValueColumn**, client tools might use the key values in reports, meaning you cannot change the key without breaking reports. For example, the surrogate key in a Kimball star schema design holds no business information, but you might have to change key values if you remodel type 2 history. However, this change would not be possible if you had created a dependency to the key in the client tools

To avoid end-user reports referring to the surrogate key directly, we recommend that you hide it. The best way to hide a surrogate key from users is to set the **AttributeHierarchyVisible** = **false** in the dimension design process, and then remove the attribute from any user hierarchies. This prevents end-user tools from referencing the surrogate key, leaving you free to change the key value if requirements change.

### Setting or Disabling Ordering of Attributes

In most cases, you want an attribute to have an explicit ordering. For example, you will want a City attribute to be sorted alphabetically. To explicitly control the ordering, you can set the **OrderBy** or **OrderByAttribute** property of the attribute. Typically, you will specify that the attribute be ordered by attribute name or key, but you can use another attribute. If you include an attribute only for the purpose of ordering another attribute, make sure you set **AttributeHierarchyEnabled** = **false** and **AttributeHierarchyOptimizedState** = **NotOptimized** to save on processing operations.

There are a few cases in which you don’t care about the ordering of an attribute, and the surrogate key is one such case. For hidden attributes such as these, which you use only for implementation purposes, you can set **AttributeHierarchyOrdered** = **false** to save time during processing of the dimension.

### Setting Default Attribute Members

Any query that does not explicitly reference a hierarchy will use the **current** member of that hierarchy. The default behavior of Analysis Services is to assign the **All** member of a dimension as the **default member**, which is normally the desired behavior.

However, for some attributes, such as the current day in a date dimension, it sometimes makes sense to explicitly assign a default member. For example, you might set a default date in the Adventure Works cube like this.

ALTER CUBE [Adventure Works]UPDATE

DIMENSION [Date], DEFAULT\_MEMBER='[Date].[Date].&[2000]'

Some client tools might not handle default members correctly, though. For example, Microsoft Excel 2010 will not provide a visual indication that a default member is currently selected, but the default member will nonetheless influence the query result. This is often confusing to users who expect the **All** level to be the current member, given that no other members are referenced in the query.

Also, when a default member is set in a dimension with multiple hierarchies, the results can be hard for users to interpret.

In general, you should explicitly set default members only on dimensions with single hierarchies or in hierarchies that do not have an **All** level.

### Removing the All Level

Most dimensions roll up to a common **All** level, which is the aggregation of all descendants. But there are some exceptions where it does not make sense to query at the **All** level. For example, you may have a currency dimension in the cube, but asking for “the sum of all currencies” is a meaningless question.

It can also be expensive to ask for the **All** level of a dimension if there is no good aggregate to respond to the query. For example, if you have a cube partitioned by currency, asking for the **All** level of currency will cause a scan of all partitions, which could be expensive and lead to a useless result.

In order to prevent users from querying meaningless **All** levels, you can disable the **All** member in a hierarchy. You do this by setting the **IsAggregateable** = **false** on the attribute at the top of the hierarchy. Note that if you disable the **All** level, you should also set a default member as described in the previous section. If you don’t, Analysis Services will choose one for you.

### Identifying Attribute Relationships

[Attribute relationships](http://msdn2.microsoft.com/en-us/library/ms176124.aspx) define hierarchical dependencies between attributes. In other words, if ***A*** has a related attribute ***B***, written ***A***  ***B***, there is one member in ***B*** for every member in ***A***, and potentially multiple members in ***A*** for a given member in ***B***. For example, given an attribute relationship City  State, if the current city is Seattle, we know the State must be Washington.

Often, there are relationships between attributes that can be used by the Analysis Services engine to optimize performance, and these relationships might not necessarily be manifested in the original dimension table.

When you begin designing attribute hierarchies, by default, all attributes are related to the key, and the attribute relationship diagram resembles a “bush” in which relationships all stem from the key attribute and end at each other’s attribute.

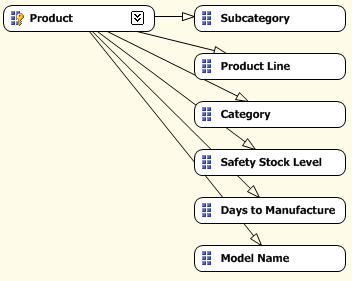


Figure . Sub optimal “bushy” attribute relationships

You can optimize performance by changing the bush to more of a tree: that is, by defining hierarchical relationships supported by the data.

In the example shown in figures 1 and 2, a model name identifies the product line and subcategory, and the subcategory identifies a category. No subcategory is found in more than one category. You can redefine the relationships in the attribute relationship editor to make these relationships clearer.

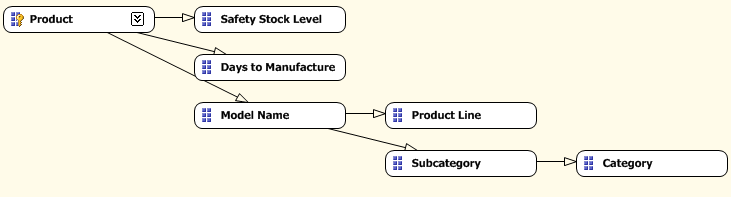


Figure . Optimized tree-like attribute relationships

Attribute relationships can help performance in three significant ways:

* Cross-products between levels in the hierarchy do not need to go through the key attribute. This saves CPU time during queries.
* Aggregations built on attributes can be reused for queries on related attributes. This saves resources during processing and queries.
* Auto-Exist can more efficiently eliminate attribute combinations that do not exist in the data.

Consider the cross-product between Subcategory and Category, given the two designs shown in Figures 1 and 2. In the design in Figure 1, no attribute relationships have been explicitly defined, and therefore the engine must first find which products belong to each subcategory, and then determine which categories each of these products belongs to. For large dimensions, this can take a long time. However, if the attribute relationship is defined as depicted in Figure 2, the Analysis Services engine can use indexes, which are built at processing time, to determine the correct category for each subcategory, making queries much faster.

#### Flexible vs. Rigid Relationships

When an attribute relationship is defined, the relation can either be flexible or rigid.

* In a **flexible attribute relationship**, members can move around during dimension updates. For example, the relationship between customer and city should perhaps be flexible, as customers might move.
* In a **rigid attribute relationship**, the member relationships are guaranteed to be fixed. For example, the relationship between month and year is fixed because a particular month isn’t going to change its year when the dimension is reprocessed.

The choice of whether a relationship is flexible or rigid is not merely one of semantics; it affects processing. When a change in a flexible relationship is detected during processing, all indexes for partitions referencing the affected dimension must be invalidated (including the indexes for attributes that are not affected). This is an expensive operation and may cause **Process Update** operations to take a very long time. Indexes that have been invalidated by changes in flexible relationships must be rebuilt after a **Process Update** operation with a **Process Index** on the affected partitions; this adds even more time to cube processing. For more information on how **Process Update** works, see this [blog article](http://blogs.msdn.com/b/karang/archive/2012/05/03/processupdate_2d00_insight.aspx) ( http://blogs.msdn.com/b/karang/archive/2012/05/03/processupdate\_2d00\_insight.aspx).

Flexible relationships are the default setting. Carefully consider the advantages of rigid relationships and change the default where the design allows it.

### Using Hierarchies Effectively

Analysis Services enables you to build two types of user hierarchies: natural and unnatural hierarchies. Each type has different design and performance characteristics.

* In a **natural** **hierarchy**, all attributes participating as levels in the hierarchy have direct or indirect attribute relationships extending from the bottom of the hierarchy to the top of the hierarchy.
* In an **unnaturalhierarchy***,* the hierarchy consists of at least two consecutive levels that have no attribute relationships. Typically these hierarchies are used to create drill-down paths of commonly viewed attributes that do not follow any natural hierarchy. For example, users may want to view a hierarchy of Gender and Education.

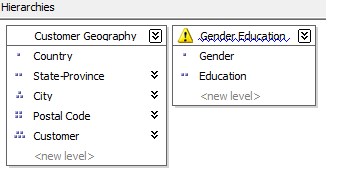


Figure . Examples of natural and unnatural hierarchies

From a performance perspective, natural hierarchies behave very differently than unnatural hierarchies do. In natural hierarchies, the hierarchy tree is materialized on disk in hierarchy stores. In addition, all attributes participating in natural hierarchies are automatically considered to be aggregation candidates.

Unnatural hierarchies are not materialized on disk, and the attributes participating in unnatural hierarchies are not automatically considered as aggregation candidates. Rather, they simply provide users with easy-to-use drill-down paths for commonly viewed attributes that do not have natural relationships. By assembling these attributes into hierarchies, you can also use a variety of MDX navigation functions to easily perform calculations like percent of parent.

To take advantage of natural hierarchies, define cascading attribute relationships for all attributes that participate in the hierarchy.

### Turning Off the Attribute Hierarchy

[Member properties](http://msdn.microsoft.com/library/ms145528.aspx) (http://msdn.microsoft.com/library/ms145528.aspx) provide a mechanism for exposing dimension information. For a given attribute, member properties are automatically created for every direct attribute relationship. Therefore, if the attribute in question is the also primary key, every attribute that is directly related to the primary key is also available as a member property of the primary key attribute.

If you only want to access an attribute as a member property, and not expose the attribute directly, you can disable the attribute’s hierarchy by setting the **AttributeHierarchyEnabled** property to **False**. From a processing perspective, disabling the attribute hierarchy can improve performance and decrease cube size because the attribute will no longer be indexed or aggregated. This can be especially useful for high-cardinality attributes that have a one-to-one relationship with the primary key (such as phone numbers and addresses), and attributes that typically do not require slice-and-dice analysis. By disabling the hierarchies for these attributes and accessing them via member properties, you can save processing time and reduce cube size.

Deciding whether to disable the attribute’s hierarchy requires that you consider the following impacts of using member properties.

* Member properties cannot be placed on a query axis in an MDX query in the same manner as attribute hierarchies and user hierarchies. To query a member property, you must query the attribute that contains that member property.

For example, if you require the work phone number for a customer, you must query the properties of the customer and then request the phone number property. As a convenience, most front-end tools easily display member properties in their user interfaces.

* In general, filtering measures using member properties is slower than filtering using attribute hierarchies, because member properties are not indexed and do not participate in aggregations. The actual impact to query performance depends on how you use the attribute.

For example, if your users want to slice and dice data by both account number and account description, from a querying perspective you may be better off having the attribute hierarchies in place and removing the bitmap indexes if processing performance is an issue.

### Reference Dimensions

Reference dimensions allow you to build a dimensional model on top of a snow flake relational design. While this is a powerful feature, you should understand the implications of using it.

By default, a reference dimension is **non-materialized**. This means that queries have to perform the join between the reference and the outer dimension table at query time. Also, filters defined on attributes in the outer dimension table are not driven into the measure group when the bitmaps there are scanned. When this happens, too much data is read from disk to answer user queries. Leaving a dimension as non-materialized prioritizes modeling flexibility over query performance. Consider carefully whether you can afford this tradeoff. Cubes are typically intended to support fast ad-hoc queries, and subjecting the end user to bad query performance is rarely a good idea. Users are not in a position to understand that you are using reference dimensions, and users can’t do anything to avoid them.

Analysis Services does provide an option that lets you materialize the reference dimension. When you enable this option, memory and disk structures are created that make the dimension behave just like a denormalized star schema. This means that you will retain all the performance benefits of a regular, non-reference dimension.

Be careful when using materialized reference dimensions though– if you run a process update on the intermediate dimension, any changes in the relationships between the outer dimension and the reference will *not* be reflected in the cube. Instead, the original relationship between the outer dimension and the measure group is retained – which is most likely not the desired result. In a way, you can consider the reference table to be a rigid relationship to attributes in the outer attributes. The only way to reflect changes in the reference table is to fully process the dimension.

### Fast-Changing Attributes

Some data models contain attributes that change very often. Depending on which type of history tracking you need, you may face different challenges.

**Type2 Fast-Changing Attributes** - If you try to track every change in a fast-changing attribute, the dimension containing the attribute might grow very large. Type 2 attributes are typically added to a dimension with a **Process Add** command, but at some point, running **Process Add** on a large dimension and running all the consistency checks will take a long time.

In general, having a huge dimension is unwieldy because users will have trouble querying it and the server will have trouble keeping it in memory. A good example is the customer age– the values would change every year and cause the customer dimension to grow dramatically.

**Type 1 Fast-Changing Attributes** – Even if you do not track every change to the attribute, you may still run into issues with fast-changing attributes. To reflect a change in the data source to the cube, you have to run **Process Update** on the changed dimension. As the cube and dimension grows larger, running **Process Update** becomes expensive.

For example, you might be tracking the status of a server in a hosting environment, using values like “Running”, “Shut down”, and “Overloaded”. A status attribute like this might change several times per day or even per hour. Running frequent **Process Updates** on such a dimension to reflect changes can be an expensive operation, and it may not be feasible in a production environment because Analysis Services locks some files while processing.[[1]](#footnote-1)

In the following sections, we will look at some modeling options you can use to address these problems.

#### Type 2 Fast-Changing Attributes

If history tracking of a fast-changing attribute is a requirement in your solution, the best option is often to use the fact table to track history. This approach is best illustrated with an example.

Consider the customer dimension with an age attribute, as discussed ablove. Modeling the Age attribute directly in the Customer dimension produces a design like the following diagram. Notice that every time Thomas has a birthday, a new row is added to the dimension table.

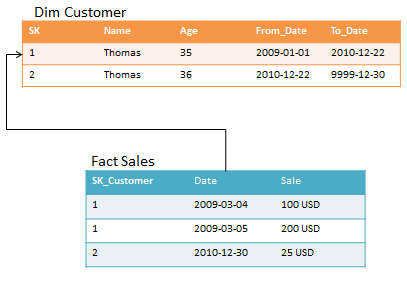


Figure . Age included in customer dimension

An alternative design approach splits the customer dimension into two dimensions, like this:

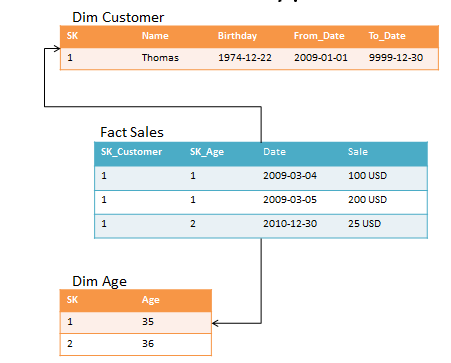


Figure . Age split into a separate dimension

There are, however, some restrictions on where this approach can be applied. Creating a separate dimension works best when the changing attribute takes on a small, distinct set of values. This design also adds complexity; when you add more dimensions to the model, you create more work for the ETL developers when the fact table is loaded. Also, consider the storage impact on the fact table: With the second design, the fact table becomes wider, and more bytes have to be stored per row.

#### Type 1 Fast-Changing Attributes

Your business requirement might include updating some attribute of a dimension daily, or even hourly. For a small cube, running **Process Update** will help you do this. But as the cube grows larger, the run time of **Process Update** can become too long for the batch window or the real-time requirements of the cube.

Consider, for example, the cube that tracks server hosting: You might want to track the status of all servers, which changes frequently. Assume for the example that the server dimension is used by a fact table that captures performance counters and that you have designed the data model like this:

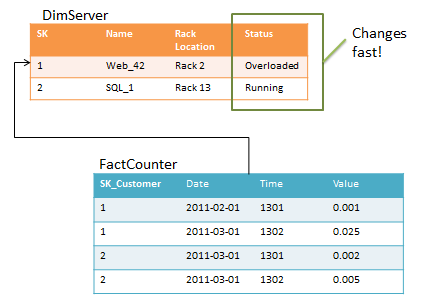


Figure . Status column in server dimension

The problem with this model is the **Status** column. If the **Fact Counter** table is large and status changes a lot, **Process Update** will take a very long time to run. To optimize, consider the following design instead.

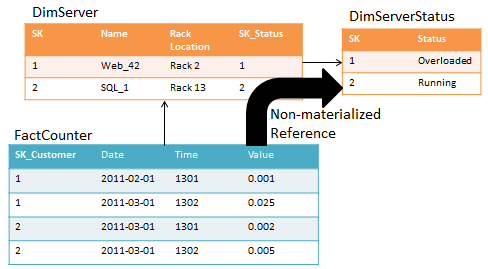


Figure : Status column in its own dimension

If you implement **DimServer** as the intermediate reference table to **DimServerStatus**, Analysis Services no longer has to keep track of the metadata in the **FactCounter** when you run **Process Update** on **DimServerStatus**.

However, this also means that the join to **DimServerStatus** will happen at run time, increasing CPU cost and query times. Plus you cannot index attributes in **DimServer** becausethe intermediate dimension is not materialized.

In summary, you have to carefully balance the tradeoff between processing time and query speeds.

### Large Dimensions

In SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2, Analysis Services had some built-in limitations that limit the size of the dimensions you can create. First of all, it took a long time to update a dimension . The operation was expensive for these reasons:

* First, all indexes on the fact tables must be considered potentially invalid when an attribute changes.
* Second, string values in dimension attributes are stored on a disk structure called the string store. This structure has a size limitation of 4 GB. Thus if a dimension contains attributes where the total size of the string values (this includes translations) exceeds 4 GB, you will get an error during processing.

Consider for a moment a dimension with tens or even hundreds of millions of members. Such a dimension can be built and added to a cube, no matter whether you are running on SQL Server 2005, 2008, or 2008 R2. But what does such a dimension mean to an ad-hoc user? How will the user navigate it? Are there hierarchies that can group the members of this dimension into reasonable sizes for rendering in a client? While it may make sense for some reporting purposes to search for individual members in such a dimension, it may not be the right problem to solve with a cube.

When you build cubes, ask yourself: is this a cube problem? For example, think of this typical telecommunications data set, which models detailed records of phone calls.

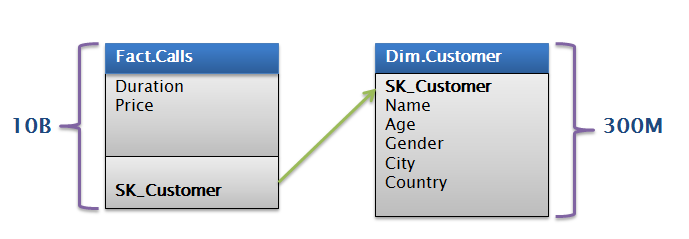


Figure : Call detail records (CDRs)

In this particular example, there are 300 million customers in the data model. There is no good way to group these customers and allow ad-hoc access to the cube at reasonable speeds. Even if you manage to optimize the space used to fit in the 4-GB string store, how would users browse a customer dimension like this?

If you find yourself in a situation where a dimension becomes too large and unwieldy, consider building the cube on top of an aggregate. For the call records data set, imagine a transformation like the following:

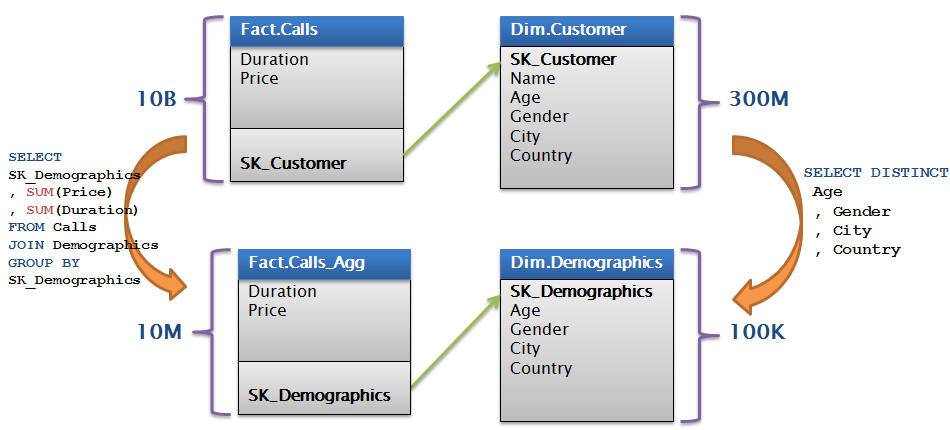


Figure : Cube built on aggregate

When you substitute an aggregated fact table, the problem dimension with 300 million rows turns into a much smaller dimension with 100,000 rows. You might consider aggregating the facts to save storage too – alternatively, you can add a demographics key directly to the original fact table, process on top of this data source, and rely on MOLAP compression to reduce data sizes.

Beginning in SQL Server 2012 Analysis Services, you can reconfigure string storage to accommodate very large strings in dimension attributes or partitions, and exceed the previous 4 GB file size limit for string stores. If your dimensions or partitions include string stores of this size, you can work around the file size constraint by changing the **StringStoresCompatibilityLevel** property. However, note these limitations:

* String storage configuration is optional, which means that even new databases that you create in SQL Server 2012 will continue to use the default string store architecture, which is subject to the 4 GB maximum file size.
* Using the larger string storage architecture has a small but noticeable impact on performance. You should use it only if your string storage files are close to or at the maximum 4 GB limit.
* Changing the string storage settings of an object requires that you reprocess the object itself and any dependent object. Processing is required to complete the procedure

To configure storage for larger strings, set the **StringStoresCompatibilityLevel** property on a dimension or partition. Valid values for this property include the following:

|  |  |
| --- | --- |
| **Value** | **Description** |
| 1050 | Specifies the default string storage architecture, subject to a 4 GB maximum file size per store. |
| 1100 | Specifies larger string storage, supports up to 4 billion unique strings per store. |

For additional information, see the following article:

[Configure String Storage for Dimensions and Partitions](http://technet.microsoft.com/en-us/library/gg471589.aspx) (http://technet.microsoft.com/en-us/library/gg471589.aspx)

## Partitioning a Cube

Effective use of partitions can enhance query performance, improve processing performance, and facilitate data management. Partitions work by separating the data in measure groups into physical storage units. You can use multiple partitions to break up your measure group into separate physical components.

This section specifically addresses how you can use partitions to improve query performance. The advantages of partitioning for query performance are two-fold: you can eliminate partitions that aren’t needed in a query and optimize aggregation design. However, in your partitioning strategy you must often make a tradeoff between query and processing performance.

**Partition elimination -** Partitions that do not contain data in the requested subcube are not queried at all, thus avoiding the cost of reading the index, or of scanning a table if the server is in ROLAP mode. While reading a partition index and finding no available rows is a cheap operation, these reads begin to put a strain on the thread pool as the number of concurrent users grows. Moreover, when Anaysis Services encounters queries that do not have indexes to support them, all potentially matching partitions must be scanned for data.

**Aggregation design -** Each partition can have its own unique aggregation design, or it can use a shared aggregation design. Partitions queried more often or differently might have their own designs.

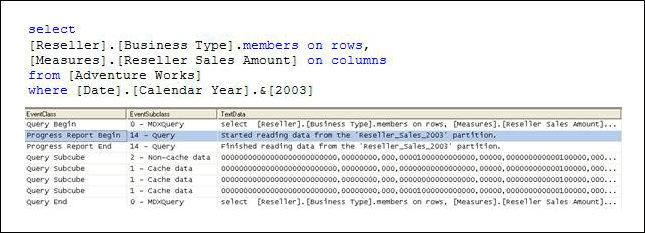


Figure : Intelligent querying by partitions

Figure 10 shows an example of a Profiler trace on a query that requests the Reseller Sales Amount from Adventure Works for the year 2003, grouped by Business Type. The Reseller Sales measure group of the Adventure Works cube contains four partitions: one for each year. Because the query slices on 2003, the storage engine can go directly to the 2003 Reseller Sales partition and ignore other partitions.

### Partition Slicing

Partitions are bound to a source table, view, or source query. When the formula engine requests a subcube, the storage engine looks at the metadata of partition for the relevant measure group. Each partition may contain a slice definition, a high level description of the minimum and maximum attribute DataIDs that exist in that dimension. If it can be determined from the slice definition that the requested subcube data is not present in the partition, that partition is ignored. If the required data is present, or if the slice definition is missing, the partition is accessed by first looking at the indexes (if any) and then scanning the partition segments.

The slice of a partition can be set in two ways:

* **Auto slice** – When Analysis Services reads the data during processing, it keeps track of the minimum and maximum attribute DataID reads. These values are used to set the slice when the indexes are built on the partition.
* **Manual slice**– when Analysis Services reads the data during processing

#### Manual slicer

There are cases where auto slice (described in the next section) will not work. In such cases, you can manually set the slice. Manual slices are also the only available slice option for ROLAP partitions and proactive caching partitions.

#### Auto Slice

During processing of MOLAP partitions, Analysis Services internally identifies the range of data that is contained in each partition by using the Min and Max DataIDs of each attribute, to calculate the range of data that is contained in the partition. The data range for each attribute is then combined to create the slice definition for the partition.

The minimum and maximum DataIDs can specify either a single member or a range of members. For example, partitioning by Year results in the same minimum and maximum DataID slice for the Year attribute, and queries to a specific moment in time only result in partition queries to that year’s partition.

It is important to remember that the partition slice is maintained as a range of DataIDs that you have no explicit control over. DataIDs are assigned during dimension processing as new members are encountered. Because Analysis Services just looks at the minimum and maximum value of the DataID, you can end up reading partitions that don’t contain relevant data.

For example: if you have a partition, **P2003\_4**, that contains both 2003 and 2004 data, you are not guaranteed that the minimum and maximum DataID in the slide contain values next to each other (even though the years are adjacent). In our example, let us say the DataID for 2003 is 42 and the DataID for 2004 is 45. You can’t specify which DataID gets assigned to which members, so the DataID for 2005 might be 44. When a user requests data for 2005, Analysis Services looks at the slice for **P2003\_4**, sees that it contains data in the interval 42 to 45 and therefore concludes that this partition has to be scanned to make sure it does not contain the values for DataID 44 (because 44 is between 42 and 45).

Because of this behavior, auto slice typically works best if the data contained in the partition maps to a single attribute value. When that is the case, the maximum and minimum DataID contained in the slice will be equal and the slice will work efficiently.

Note that the auto slice is not defined and indexes are not built for partitions with fewer rows than **IndexBuildThreshold** (which has a default value of 4096).

#### Manually Setting Slices

No metadata is available to Analysis Services about the content of ROLAP and proactive caching partitions. Because of this, you must manually identify the slice in the properties of the partition. It is a best practice to manually set slices in ROLAP and proactive caching partitions.

However, as shown in the previous section, there are cases where auto slice will not give you the desired partition elimination behavior. In these cases you can benefit from defining the slice yourself for MOLAP partitions. For example, if you partition by year with some partitions containing a range of years, defining the slice explicitly avoids the problem of overlapping DataIDs. This can only be done with knowledge of the data – which is where you can add some optimization as a BI developer.

It is generally not a best practice to create partitions before you are ready to fill them with data. But for real-time cubes, it is sometimes a good idea to create partitions in advance to avoid locking issues. When you take this approach, it is also a good idea to set a manual slice on MOLAP partitions to make sure the storage engine does not spend time scanning empty partitions.

**References**

Mosha Pasumasky has written on this topic: <http://sqlblog.com/blogs/mosha/archive/2008/10/14/get-most-out-of-partition-slices.aspx>,

Jesse Orosz has also written a useful [blog entry](http://jesseorosz.spaces.live.com/blog/cns!E322FD91218E57CF!304.entry) (<http://jesseorosz.spaces.live.com/blog/cns!E322FD91218E57CF!304.entry>) stating the four reasons why setting partition slice in MOLAP is important. Note number three in his list: “automatic slice detection is not perfect.”

### Partition Sizing

Testing using nondistinct count measure groups on partitions ranging from 200 MB up to 3 GB suggests that partition size alone does not have a substantial impact on query speeds. In fact, good query performance has been achieved on partitions larger than 3 GB.

The following graph shows four different query runs, using the same Customer cube but with different partition sizes. Notice that performance is comparable between cubes with different partition sizes, and that throughput is affected more by the design of the security features.

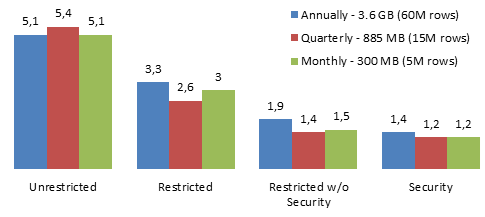


Figure : Throughput by partition size (higher is better)

Thus, your partitioning strategy should be based on these factors:

* Increasing processing speed and flexibility
* Increasing manageability of bringing in new data
* Increasing query performance from partition elimination as described earlier
* Support for different aggregation designs

Of course, as you add more partitions, the metadata overhead of managing the cube grows exponentially. This affects **ProcessUpdate** and **ProcessAdd** operations on dimensions, which have to traverse the metadata dependencies to update the cube when dimensions change. As a rule of thumb, follow these guidelines, while balancing the requirements discussed above.

* Keep the number of partitions in the cube in the low thousands.
* For large cubes, prefer larger partitions over creating too many partitions. This also means that you can safely ignore the Analysis Management Objects (AMO) warning in Microsoft Visual Studio that partition sizes should not exceed 20 million rows.

### Partition Strategy

Based on this guidance for partition sizing, we can also present some common design patterns for partitioning strategies:

* partition by date.
* use a partition matrix.
* use a hash to partition.

These strateges will be described in more detail.

#### Partition by Date

Most cubes include at least one column containing a date. Also, because the source data often arrives in monthly, weekly, daily, or even hourly slices, it makes sense to partition the cube on date. Partitioning by date has many advantages:

* Easily replace a full day in case you load faulty data.
* Selectively archive old data by moving the partition to cheap storage.
* Easily get rid of data, by removing an entire partition.

Typically, a date partitioning scheme looks somewhat like this. This design works very well for small to medium-sized cubes. It is reasonably simple to implement and the number of partitions is kept low. To move the partition to cheaper storage, you simply change the data location and reprocess the partition.



Figure 12: Partitioning by Date

However, it does suffer from a few drawbacks:

1. If the granularity of the partitioning is small enough (for example, hourly), the number of partitions can quickly become unmanageable.
2. Assuming data is added only to the latest partition, partition processing is limited to one TCP/IP connection reading from the data source. If you have a lot of data, this can be a scalability limit.

If you have a lot of date-based partitions, it is often a good idea to merge the older ones into large partitions. You can do this either by using the Analysis Services merge functionality or by dropping the old partitions, creating a new, larger partition, and then reprocessing it. Reprocessing will typically take longer than merging, but we have found that compression of the partition can often increase if you reprocess.

A modified date partitioning scheme might look like this. This design addresses the metadata overhead of having too many partitions.



Figure 13: Modified Date Partitioning

However, this design too is bottlenecked by the maximum speed of the **Process Add** or **Process Full** job for the latest partition. If your data source is SQL Server, the speed of a single database connection can be hundreds of thousands of rows every second – which works well for most scenarios. But if the cube requires even faster processing speeds, consider matrix partitioning.

#### Matrix Partitioning

For large cubes, it is often a good idea to implement a **matrix partitioning** scheme. This means partitioning the data on both date **and** some other key.

* Use the date partitioning to selectively delete or merge old partitions.
* Use the second key to achieve parallelism during partition processing and to restrict certain users to a subset of the partitions.

For example, consider a retailer that operates in US, Europe, and Asia. You might decide to partition like this.

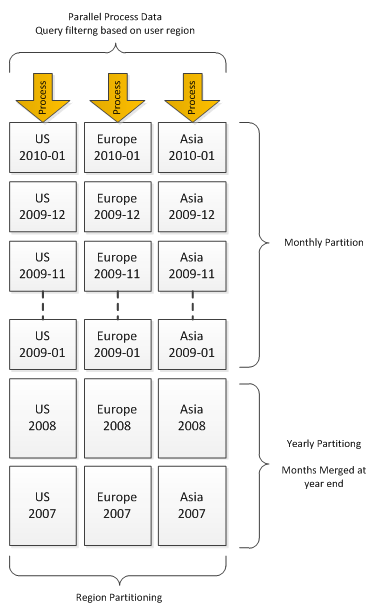


Figure : Example of matrix partitioning

As the retailer grows, they might need to split the regional partitions into smaller partitions, to further increase parallelism during load and to limit the worst-case scans that a user can perform. Therefore, in cubes that are expected to grow dramatically, choose a partition key that will grow with the business and give you options for extending the matrix partitioning strategy.

The following table contains some examples of such partitioning keys.

|  |  |  |
| --- | --- | --- |
| Industry | Example partition key | Source of data proliferation |
| Web retail | Customer key | Adding customers and transactions |
| Store retail | Store key | Adding new stores |
| Data hosting | Host ID or rack location | Adding a new server |
| Telecommunications | Switch ID, country code, or area code | Expanding into new geographical regions or adding new services |
| Computerized manufacturing | Production line ID or machine ID | Adding production lines or (for machines) sensors |
| Investment banking | Stock exchange or financial instrument | Adding new financial instruments, products, or markets |
| Retail banking | Credit card number or customer key | Increasing customer transactions |
| Online gaming | Game key or player key | Adding new games or players |

If you implement a matrix partitioning scheme, you must pay special attention to user queries. Queries that touch several partitions for every subcube request, such as a query that asks for a high-level aggregate of the partition business key, result in a high thread usage in the storage engine. Because of this, we recommend that you partition the business key so that single queries touch no more than the number of cores available on the target server. For example, if you partition by Store Key and you have 1,000 stores, queries touching the aggregation of all stores will have to touch 1,000 partitions. To avoid this, you might try grouping the stores into a number of buckets – that is, rather than having individual partitions for each store, assign stores to each partition. For example, if you run on a 16-core server, you can group the store into buckets of around 62 stores for each partition (1,000 stores divided into 16 buckets).

#### Hash Partitioning

Sometimes it is not possible to come up with a good distribution of business keys for partitioning the cube. Perhaps you just don’t have a good key candidate, or perhaps the distribution of the key is unknown at design time. In such cases, a brute-force approach can be used:

* Partition on the hash value of a key that has a high enough cardinality and where there is little skew.
* If every query will touch many partitions, pay special attention to the **CoordinatorQueryBalancingFactor** and the **CoordinatorQueryMaxThread** settings, which are described in the [SQL Server 2008 R2 Analysis Services Operations Guide](http://msdn.microsoft.com/en-us/library/hh226085.aspx) (http://msdn.microsoft.com/en-us/library/hh226085.aspx).

## Relational Data Source Design

Cubes are typically built on top of relational data sources that serve as data marts. Analysis Services lets you design powerful abstractions on top of this relational source. Abstractions allow for fast prototyping and enable you to compensate for poor relational design when you are not in control of the underlying data source. However, the Analysis Services design surface cannot conquer all ills, and a well-designed relational data source can make queries and processing of a cube much faster.

In this section, we will describe some of the options that you should consider when designing a relational data source. A full treatment of relational data warehousing is out of scope for this document, but we will provide references where appropriate.

### Use a Star Schema for Best Performance

It is widely debated what the most efficient modeling technique is to support ad-hoc reporting and analysis: the star schema, snowflake schema, third to fifth normal form, or data vault models (in order of increased normalization). All are considered good candidates for reporting by professional warehouse designers.

The Unified Dimensional Model (UDM) used by Analysis Services is itself a dimensional model, with some additional features (such as reference dimensions) that support snowflakes and many-to-many dimensions. Thus, no matter which model you use for end-user reporting, performance boils down to one simple fact: joins are expensive!

This is also partially true for the Analysis Services engine. For example: If a snowflake is implemented as a non-materialized reference dimension, users will wait longer for queries, because the join is done at run time inside the Analysis Services engine.

The largest impact of snowflakes occurs during processing of the partition data. For example, if you implement a fact table as a join of two big tables (for example, separating order lines and order headers instead of storing them as pre-joined values), processing of facts will take longer, because the relational engine has to compute the join.

* It is possible to build an Analysis Services cube on top of a highly normalized model, but be prepared to pay the price of joins when accessing the relational model. In most cases, that price is paid at processing time.
* In MOLAP data models, materialized reference dimensions help you store the result of the joined tables on disk and give you high speed queries even on normalized data.
* If you are running ROLAP partitions, queries will pay the price of the join at query time, and your user response times or your hardware budget will suffer if you are unable to resist normalization.

### Consider Moving Calculations to the Relational Engine

Sometimes calculations can be moved to the relational engine and be processed as simple aggregates with much better performance. There is no one-size-fits-all solution we can propose, but if you’re encountering performance issues, consider whether the calculation can be resolved in the source database or data source view (DSV) and prepopulated, rather than evaluated at query time.

For example, consider expressions like this one:

Sum(Customer.City.Members, cint(Customer.City.Currentmember.properties(“Population”))), …

Instead, you could define a separate measure group on the **City** table, with a SUM measure on the **Population** column.

As a second example, consider these two solutions. Which do you think will provide superior performance?

* Compute the product of Revenue \* Products Sold at the leaves in the cube and aggregate with calculations.
* Compute this result in the source database.

### Use Views

It is generally a good idea to build your UDM on top of database views. A major advantage of views is that they provide an abstraction layer on top of the physical relational model. If the cube is built on top of views, the relational database can, to some degree, be remodeled without breaking the cube.

Consider a relational source that has chosen to normalize two tables you need to join to obtain a fact table – for example, a data model that splits a sales fact into order lines and orders. If you implement the fact table using query binding, your UDM will contain the following.



Figure 15: Using named queries in UDM

In this model, the UDM now has a dependency on the structure of the **LineItems** and **Orders** tables – along with the join between them. If you instead implement a **Sales** view in the database, you can model like this.



Figure 16: Implementing UDM on top of views

This revised, view-centric model gives the relational database the freedom to optimize the joined results of **LineItems** and **Order** (for example by storing it denormalized), without any impact on the cube. It would be transparent for the cube developer if the DBA of the relational database implemented a change like the following one.



Figure 17: Implementing UDM on top of pre-joined tables

If the relational data modelers insist on normalization, give them a chance to change their minds and denormalize without breaking the cube model. Views provide encapsulation, and it is good practice to use them:

* Views make debugging easier. You can issue SQL queries directly on views to compare the relational data with the cube.
* Views are a good way to implement business logic that you can mimic with query binding in the UDM.
* While the UDM syntax is similar to the SQL view syntax, you cannot issue SQL statements against the UDM.

#### Query Binding Dimensions

Query binding for dimensions is not supported directly in SQL Server Analysis Services, but you can implement it by using a view (instead of tables) for your underlying dimension data source. That way, you can use hints, indexed views, or other relational database tuning techniques to optimize the SQL statement that accesses the dimension tables through your view. This also allows you to turn a snowflake design in the relational source into a UDM that is a pure star schema.

#### Processing Through Views

Depending on the relational source, views can often provide means to optimize the behavior of the relational database. For example, in SQL Server you can use the NOLOCK hint in the view definition to remove the overhead of locking rows as the view is scanned, if you accept the possibility of some reads being dirty.

Views can also be used to pre-aggregate large fact tables using a GROUP BY statement. The relational database modeler might even choose to materialize views that use a lot of hardware resources.

## Calculation Scripts

The calculation script in the cube allows you to define complex functionality in the cube, and directly manipulate the multidimensional space. In a few lines of code, you can build elegant and valuable business logic. By the same token, it takes only a few lines of poorly written calculation code to create a big performance impact on users.

This section describes some best practices you can apply to the cube to avoid common performance mistakes. Consider these basic rules the bare minimum that you should understand and apply when building the cube script.

### Learn MDX Basics

If you plan to design a cube with a large calculation script, we highly recommend that you learn the basics of writing good MDX code – the language used for calculations. Fortunately MDX has a rich community of contributors. Start with these resources to learn MDX coding and design practices, which will get you off to a good start:

* Pearson, Bill: “Stairway to MDX”
  + <http://www.sqlservercentral.com/stairway/72404/>
* Piasevoli, Tomislav: *MDX with Microsoft SQL Server 2008 R2 Analysis Services Cookbook*
  + <http://www.packtpub.com/mdx-with-microsoft-sql-server-2008-r2-analysis-services/book>
* Russo, Marco: MDX Blog:
  + <http://sqlblog.com/blogs/marco_russo/archive/tags/MDX/default.aspx>
* Pasumansky, Mosha: Blog
  + <http://sqlblog.com/blogs/mosha/>
* Piasevoli, Tomislav: Blog
  + <http://tomislav.piasevoli.com>
* Webb, Christopher: Blog
  + <http://cwebbbi.wordpress.com/category/mdx/>
* Spofford, George, Sivakumar Harinath, Christopher Webb, Dylan Hai Huang, and Francesco Civardi,: *MDX Solutions: With Microsoft SQL Server Analysis Services 2005 and Hyperion Essbase*, ISBN: 978-0471748083
  + <http://www.amazon.com/MDX-Solutions-Microsoft-Analysis-Services/dp/0471748080>

### Use Attributes Instead of Sets

When you need to refer to a fixed subset of dimension members in a calculation, use an attribute instead of a set. Attributes let you target aggregations to the subset. Attributes are also evaluated faster than sets by the formula engine. Using an attribute for this purpose also let you change the set by updating the dimension instead of deploying a new calculation script.

For example, the following expression defines a range as a set:

CREATE SET [Current Day] AS TAIL([Date].[Calendar].members, 1)

CREATE SET [Previous Day] AS HEAD(TAIL(Date].[Calendar].members),2),1)

Instead, do this (assuming today is 2011-06-16):

|  |  |
| --- | --- |
| Calendar Key Attribute | Day Type Attribute  (Flexible relationship to key) |
| 2011-06-13 | Old Dates |
| 2011-06-14 | Old Dates |
| 2011-06-15 | Previous Day |
| 2011-06-16 | Current Day |

**Process Update** the dimension when the day changes. Users can now refer to the current day by addressing the **Day Type** attribute instead of the set.

### Use SCOPE Instead of IIF When Addressing Cube Space

Sometimes, you want a calculation to apply to only a specific subset of cube space. In that case, SCOPE is a better choice than IIF.

Here is an example of what *not* to do.

CREATE MEMBER CurrentCube.[Measures].[SixMonthRollingAverage] AS

IIF ([Date].[Calendar].CurrentMember.Level

IS [Date].[Calendar].[Month]

, SUM ([Date].[Calendar].CurrentMember.LAG(5)

:[Date].[Calendar].CurrentMember

,[Measures].[Internet Sales Amount]) / 6

, NULL)

Instead, use the Analysis Services SCOPE function as shown below.

CREATE MEMBER CurrentCube.[Measures].[SixMonthRollingAverage] AS NULL ,FORMAT\_STRING = "Currency", VISIBLE = 1;

SCOPE ([Measures].[SixMonthRollingAverage], [Date].[Calendar].[Month].Members);

THIS = SUM( [Date].[Calendar].CurrentMember.LAG(5)

:[Date].[Calendar].CurrentMember

, [Measures].[Internet Sales Amount]) / 6;

END SCOPE;

### Avoid Mimicking Engine Features with Expressions

Several native features can be mimicked with MDX:

* Unary operators
* Calculated columns in the data source view (DSV)
* Measure expressions
* Semiadditive measures

If you must emulate these these features by using MDX script (for example, some features are only available in the Enterprise SKU), be aware that doing so can hurt performance.

For example, using **distributive unary operators** (that is, those whose member order does not matter, such as +, -, and ~) is generally twice as fast as trying to mimic their capabilities with assignments.

There are rare exceptions.

* You might be able to improve performance of **nondistributive unary operators** (those involving \*, /, or numeric values) by using MDX.
* You might know some special characteristic of your data that allows you to take a shortcut that improves performance.

However, such optimizations require expert-level tuning, and in general, you can rely on the Analysis Services engine features to do the best job.

Measure expressions also provide a unique challenge, because they disable the use of aggregates (data has to be rolled up from the leaf level). One way to work around this is to use a hidden measure that contains pre-aggregated values in the relational source. You can then target the hidden measure to the aggregate values with a SCOPE statement in the calculation script.

### Comparing Objects and Values

When determining whether the current member or tuple is a specific object, you should use the MDX **IS** function to get better performance.

For example, the following query not only performs badly, but returns incorrect results. It forces unnecessary cell evaluation and compares values instead of members.

[Customer].[Customer Geography].[Country].&[Australia] = [Customer].[Customer Geography].currentmember

The following example is even worse, because it performs extra steps to deduce whether **CurrentMember** is a particular member by using **Intersect** and **Counting**.

INTERSECT({[Customer].[Customer Geography].[Country].&[Australia]}, [Customer].[Customer Geography].currentmember).count > 0

The following example demonstrates how you can use **IS** instead:

[Customer].[Customer Geography].[Country].&[Australia] IS [Customer].[Customer Geography].currentmember

### Evaluating Set Membership

Determining whether a member or tuple is in a set is best accomplished by using **INTERSECT**. The **RANK** function is less performant because it does the additional operation of determining where in the set that object lies. If you don’t need this additional operation, don’t use **RANK**.

For example, the following statement might do more work than you need.

RANK( [Customer].[Customer Geography].[Country].&[Australia],

<set expression> )>0

The following example demonstrates how to use **INTERSECT** instead, to determine whether the specified information is in the set.

INTERSECT({[Customer].[Customer Geography].[Country].&[Australia]}, <set> ).count > 0

# Tuning Query Performance

To improve query performance, you should understand the current situation, diagnose the bottleneck, and then apply one of these techniques:

* Optimize dimension design.
* Design and build aggregations.
* Create appropriate partitions.
* Apply best practices.

These should be the first stops for optimization, before digging into queries in general. The problem with jumping into queries too quickly is that much time can be expended pursuing dead ends.

Also it is important to first understand the nature of the problem before applying specific techniques. To gain this understanding, you should have a mental model of how the query engine works. We will therefore start with a brief introduction to the Analysis Services query processor.

## Query Processor Architecture

To make the querying experience as fast as possible for end users, the Analysis Services query architecture provides several components that work together to efficiently retrieve and evaluate data. The following figure identifies the three major operations that occur during querying—session management, MDX query execution, and data retrieval—as well as the server components that participate in each operation.

Session Management

XMLA Listener

Session Manager

Security Manager

Query Processing

Query Processor

Query Proc Cache

Data Retrieval

Storage Engine

SE Cache

Dimension Data

Attribute Store

Hierarchy Store

Measure Group Data

Fact Data

Aggregations

Client App (MDX)

Figure 18: Analysis Services query processor architecture

### Session Management

Client applications communicate with Analysis Services using XML for Analysis (XMLA) over TCP/IP or HTTP. Analysis Services provides an XMLA listener component that handles all XMLA communications between Analysis Services and its clients. The Analysis Services Session Manager controls how clients connect to an Analysis Services instance.

1. Users authenticated by the Windows operating system and who have access to at least one database can connect to Analysis Services.
2. After a user connects to Analysis Services, the Security Manager determines user permissions based on the combination of Analysis Services roles that apply to the user.
3. Depending on the client application architecture and the security privileges of the connection, the client creates a session when the application starts, and then it reuses the session for all of the user’s requests.
4. The session provides the context under which client queries are executed by the query processor.
5. A session exists until it is closed by the client application or the server.

### Query Processing

The query processor executes MDX queries[[2]](#footnote-2) and generates a cellset or rowset in return. This section provides an overview of how the query processor executes queries.

To retrieve the data requested by a query, the query processor builds an execution plan to generate the requested results from the cube data and calculations. There are two major different types of query execution plans: **cell-by-cell** (**naïve**) evaluation or **block mode** (**subspace**) computation. Which one is chosen by the engine can have a significant impact on performance. For more information, see [Subspace Computation](#_Subspace_Computation).

To communicate with the storage engine, the query processor uses the execution plan to translate the data request into one or more subcube requests that the storage engine can understand. A subcube is a logical unit of querying, caching, and data retrieval—it is a subset of cube data defined by the crossjoin of one or more members from a single level of each attribute hierarchy. An MDX query can be resolved into multiple subcube requests, depending the attribute granularities involved and calculation complexity.

For example, a query involving every member of the Country attribute hierarchy (assuming it’s not a parent-child hierarchy) would be split into two subcube requests: one for the **All** member and another for the countries.

As the query processor evaluates cells, it uses the query processor cache to store calculation results. The primary benefits of the cache are to optimize the evaluation of calculations and to support the reuse of calculation results across users (with the same security roles). To optimize cache reuse, the query processor manages three cache layers that determine the level of cache reusability: global, session, and query.

#### Query Processor Cache

During the execution of an MDX query, the query processor stores calculation results in the query processor cache. The primary benefits of the cache are to optimize the evaluation of calculations and to support reuse of calculation results across users. To understand how the query processor uses caching during query execution, consider the following example: You have a calculated member called Profit Margin. When an MDX query requests Profit Margin by Sales Territory, the query processor caches the nonnull Profit Margin values for each Sales Territory. To manage the reuse of the cached results across users, the query processor distinguishes different contexts in the cache:

* **Query Context**—contains the result of calculations created by using the WITH keyword within a query. The query context is created on demand and terminates when the query is over. Therefore, the cache of the query context is not shared across queries in a session.
* **Session Context** —contains the result of calculations created by using the CREATE statement within a given session. The cache of the session context is reused from request to request in the same session, but it is not shared across sessions.
* **Global Context** —contains the result of calculations that are shared among users. The cache of the global context can be shared across sessions if the sessions share the same security roles.

The contexts are tiered in terms of their level of reuse. The following diagram shows the different types of context, in the order of increasing scope for re-use.

* **Query context** can be reused only within the query.
* **Global context** has the greatest potential for reuse across multiple sessions and users because the session context will derive from the global context and the query context will derive itself from the session context.

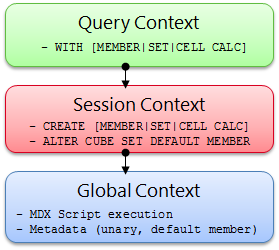


Figure 19: Reuse of the cache by context

During execution, every MDX query must reference all three contexts to identify all of the potential calculations and security conditions that can impact the evaluation of the query. For example, to resolve a query that contains a query calculated member, the query processor requires three contexts:

* A query context to resolve the query calculated member.
* A session context to evaluate session calculations.
* A global context to evaluate the MDX script and retrieve the security permissions of the user who submitted the query.

These contexts are created only if they aren’t already built. After they are built, they are reused where possible.

Even though a query references all three contexts, it will typically use the cache of a **single** context. This means that on a per-query basis, the query processor must select which cache to use. The query processor always attempts to use the broadly applicable cache depending on whether or not it detects the presence of calculations at a narrower context.

If the query processor encounters calculations created at query time, it always uses the **query context**, even if a query also references calculations from the global context. There is an exception to this – queries with query calculated members of the form Aggregate(<set>) do share the session cache. However, if there are no query calculations, but there are session calculations, the query processor uses the session cache.

The query processor selects which cache to use, based on the presence of any calculation in the scope. This behavior is especially relevant to users with client tools that generate their own MDX. If the front-end tool **creates** any session calculations or query calculations, the global cache is not used, even if the query does not specifically **use** those session or query calculations.

There are other calculation scenarios that affect how the query processor caches calculations. When you call a stored procedure from an MDX calculation, the engine always uses the **query cache**. This is because stored procedures are nondeterministic, which simply means that there is no guarantee what the stored procedure will return. As a result, when a nondeterministic calculation is encountered during the query, nothing is cached globally or in the session cache. Instead, the remaining calculations are stored in the query cache.

The following scenarios determine how the query processor caches calculation results:

* The use of MDX functions that are locale-dependent (such as **CAPTION** or **.Properties**) prevents the use of the global cache, because different sessions may be connected with different locales and cached results for one locale may not be correct for another locale.
* The use of any of these functions disables the global cache: cell security; functions such as **UserName**, **StrToSet, StrToMember,** and **StrToTuple**; **LookupCube** functions in the MDX script or in the dimension or cell security definition. In other words, just one expression that uses any of these functions or features disables global caching for the entire cube.
* If visual totals are enabled for the session (by setting the default MDX Visual Mode property in the Analysis Services connection string to 1), the query processor uses the query cache for all queries issued in that session.
* If you enable visual totals for a query by using the MDX **VisualTotals** function, the query processor uses the query cache.
* Queries that use the subselect syntax (SELECT FROM SELECT) or queries that are based on a session subcube (CREATE SUBCUBE) result in the query or, respectively, session cache to be used.
* Arbitrary shapes can only use the query cache if they are used in a subselect, in the WHERE clause, or in a calculated member. An arbitrary shape is any set that cannot be expressed as a crossjoin of members from the same level of an attribute hierarchy. For example, these expressions both are arbitrary sets:

{(Food, USA), (Drink, Canada)}

{customer.geography.USA, customer.geography.[British Columbia]}.

* An arbitrary shape on the query axis does not limit the use of any cache.

Based on this behavior, when your query workload can benefit from reusing data across users, it is a good practice to define calculations in the **global scope**. An example of this scenario is a structured reporting workload where you have few security roles.

In contrast, if you have a workload that requires individual data sets for each user, such as in an HR cube where you have many security roles or you are using dynamic security, the opportunity to reuse calculation results across users is lessened or eliminated. As a result, the performance benefits associated with reusing the query processor cache are not as high.

### Data Retrieval

When you query a cube, the query processor breaks the query into subcube requests for the storage engine. For each subcube request, the storage engine first attempts to retrieve data from the storage engine cache. If no data is available in the cache, it attempts to retrieve data from an aggregation. If no aggregation is present, it must retrieve the data from the fact data from a measure group’s partition data.

Retrieving data from a partition requires I/O activity. This I/O can either be served from the file system cache or from disk. Additional details of the I/O subsystem of Analysis Services can be found in the [SQL Server 2008 R2 Analysis Services Operations Guide](http://msdn.microsoft.com/en-us/library/hh226085.aspx) ( http://msdn.microsoft.com/en-us/library/hh226085.aspx).

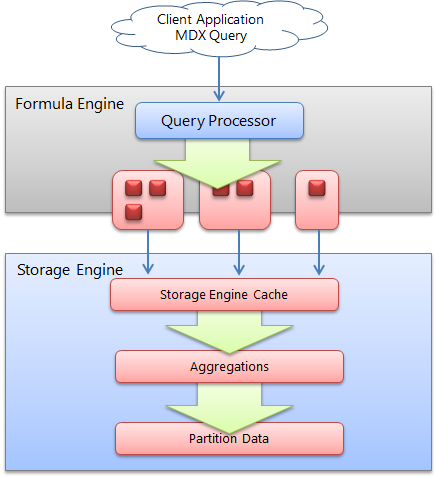


Figure 20: High-level overview of the data retrieval process

#### Storage Engine Cache

The storage engine cache is also known as the data cache registry because it is composed of the dimension and measure group caches that are the same structurally. When a request is made from the Analysis Services formula engine to the storage engine, it sends a request in the form of a subcube describing the structure of the data request and a data cache structure that will contain the results of the request. Using the data cache registry indexes, it attempts to find a corresponding subcube:

* If there is a matching subcube, the corresponding data cache is returned.
* If a subcube superset is found, a new data cache is generated and the results are filtered to fit the subcube request.
* If lower-grain data exists, the data cache registry can aggregate this data and make it available as well – and the new subcube and data cache are also registered in the cache registry.
* If data does not exist, the request goes to the storage engine and the results are cached in the cache registry for future queries.

Analysis Services allocates memory via memory holders that contain statistical information about the amount of memory being used. Memory holders are in the form of **nonshrinkable** and **shrinkable** memory; each combination of a subcube and data cache forms a single shrinkable memory holder. When Analysis Services is under heavy memory pressure, cleaner threads remove shrinkable memory. Therefore, ensure your system has enough memory; if it does not, your data cache registry will be cleared out (resulting in slower query performance) when it is placed under memory pressure.

#### Aggressive Data Scanning

Sometimes, in the evaluation of an expression, more data is requested than is actually required to determine the result.

If you suspect more data is being retrieved than is required, you can use SQL Server Profiler to diagnose how a query decomposes into subcube query events and partition scans. For subcube scans, check the verbose subcube event and whether more members than required are retrieved from the storage engine. For small cubes, this likely isn’t a problem. For larger cubes with multiple partitions, it can greatly reduce query performance.

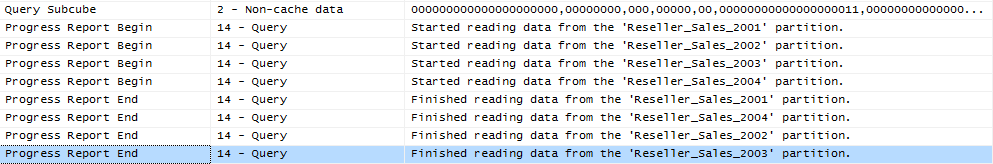
The following example demonstrates how a single query subcube event results in partition scans.

Figure . Aggressive partition scanning

There are two potential solutions to the problem of overly aggressive partition scanning:

* If a calculation expression contains an arbitrary shape, the query processor might not be able to determine that the data is limited to a single partition, and thus will request data from all partitions. Try to eliminate the arbitrary shape.
* Other times, the query processor is simply overly aggressive in asking for data. For small cubes, this doesn’t matter, but for very large cubes, it does. If you observe this behavior, potential solutions include the following:
* Contact Microsoft Customer Service and Support for further advice.
* Sometimes Analysis Services requests additional data from the source to prepopulate the cache; it may help to turn it off so that Analysis Services does not request too much data. To do this, edit the connection string and set **Disable Prefetch = 1**.

## Query Processor Internals

There were changes to query processor internals in SQL Server 2008 that are still important to consider in designing solutions using SQL Server 2012 and SQL Server 2014 Analysis Services. In this section, we will provide some background on these changes and introduce specific optimization techniques.

### Subspace Computation

The significance of **subspace computation** is best understood by contrasting it with **cell-by-cell** evaluation of a calculation, also known as **naïve calculation**.

Consider a trivial calculation of a rolling sum, summing the sales for the previous and current years.

RollingSum = (Year.PrevMember, Sales) + Sales

Now, consider a query that requests the rolling sum for 2005 for all products.

SELECT 2005 on columns, Product.Members on rows WHERE RollingSum

A cell-by-cell evaluation of this calculation proceeds as represented in the following figure.

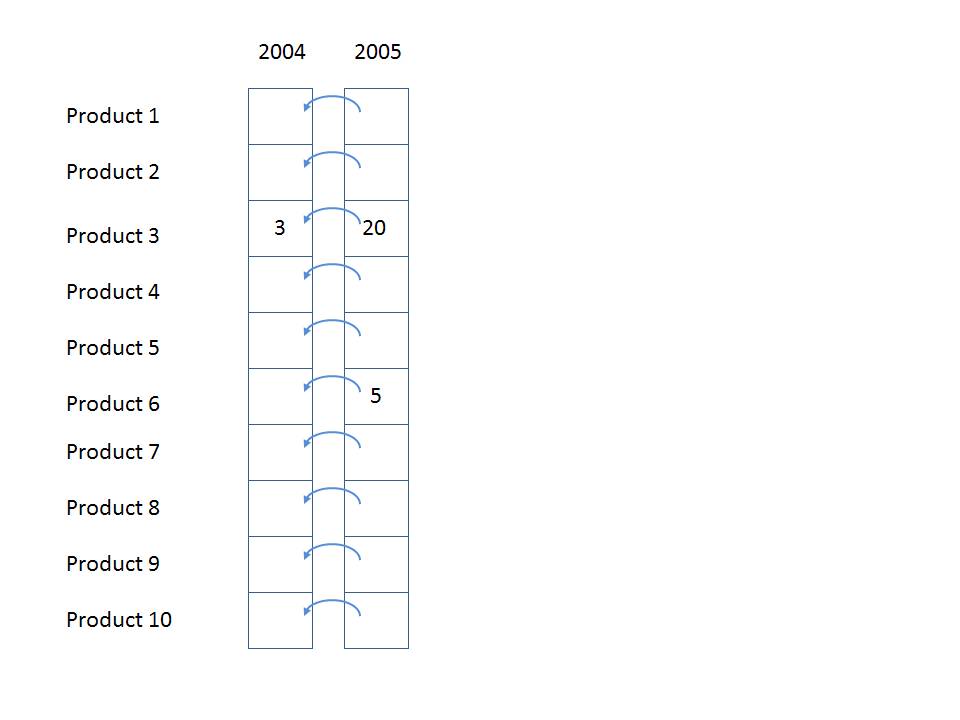


Figure 22: Cell-by-cell evaluation

The 10 cells for **[2005, All Products]** are each evaluated in turn. For each, the previous year is located, and then the sales value is obtained and then added to the sales for the current year. There are two significant performance issues with this approach.

1. If the data is **sparse** (that is, thinly populated), cells are calculated even though they are bound to return a null value. In the previous example, calculating the cells for anything but Product 3 and Product 6 is a waste of effort. The impact of this can be extreme—in a sparsely populated cube, the difference can be several orders of magnitude in the numbers of cells evaluated.
2. Even if the data is totally **dense**, meaning that every cell has a value and there is no wasted effort visiting empty cells, there is much repeated effort. The same work (for example, getting the previous Year member, setting up the new context for the previous Year cell, checking for recursion) is redone for each Product. It would be much more efficient to move this work out of the inner loop of evaluating each cell.

Now consider the same example performed using subspace computation. In subspace computation, the engine works its way down an execution tree determining what spaces need to be filled. Given the query, the following space needs to be computed, where \* means every member of the attribute hierarchy.

[Product.\*, 2005, RollingSum]

Given the calculation, this means that the following space needs to be computed first.

[Product.\*, 2004, Sales]

Next, the following space must be computed.

[Product.\*, 2005, Sales]

Finally, the **+** operator needs to be added to those two spaces.

If Sales were itself covered by calculations, the spaces necessary to calculate Sales would be determined and the tree would be expanded. In this case, Sales is a base measure, so the storage engine data is used to fill the two spaces at the leaves, and then, working up the tree, the operator is applied to fill the space at the root. Hence the one row (Product3, 2004, 3) and the two rows { (Product3, 2005, 20), (Product6, 2005, 5)} are retrieved, and the + operator is applied to them, yielding the following result.

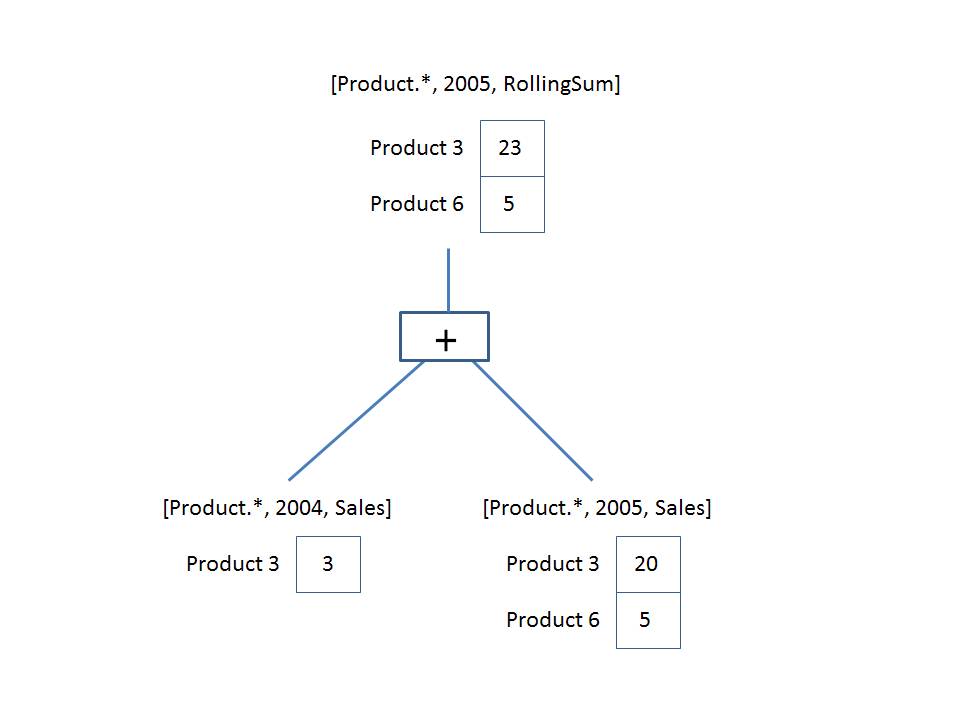


Figure 23: Execution plan

The + operator operates on **spaces**, not just on scalar values*.* The operator is responsible for combining the two given spaces to produce a space that contains each product that appears in either space with the summed value. This is the **query execution plan**. Note that it operates only on data that could contribute to the result. There is no notion of the theoretical space over which the calculation must be performed.

A query execution plan is not one or the other but can contain both subspace and cell-by-cell nodes. Some functions are not supported in subspace mode, causing the engine to fall back to cell-by-cell mode. But even when evaluating an expression in cell-by-cell mode, the engine can return to subspace mode.

### Expensive vs. Inexpensive Query Plans

It can be costly to build a query plan. In fact, the cost of building an execution plan can exceed the cost of query execution. The Analysis Services engine has a coarse classification scheme—expensive versus inexpensive. A plan is deemed **expensive** if cell-by-cell mode is used or if cube data must be read to build the plan. Otherwise the execution plan is deemed **inexpensive**.

Cube data is used in query plans in several scenarios. Some query plans result in the mapping of one member to another because of MDX functions such as **PrevMember** and **Parent**. The mappings are built from cube data and materialized during the construction of the query plans. The **IIf**, CASE, and IF functions can generate expensive query plans as well, should it be necessary to read cube data in order to partition cube space for evaluation of one of the branches. For more information, see [IIf Function in SQL Server 2008 Analysis Services](#_IIf_Function_in).

### Expression Sparsity

An expression’s **sparsity** refers to the number of cells with nonnull values compared to the total number of cells in the result of the evaluation of the expression. If there are relatively few nonnull values, the expression is termed sparse. If there are many, the expression is dense. As we shall see later, whether an expression is sparse or dense can influence the query plan.

But how can you tell whether an expression is dense or sparse? Consider a simple noncalculated measure – is it dense or sparse? In OLAP, base fact measures are considered sparse by the Analysis Services engine. This means that the typical measure does not have values for every attribute member. For example, a customer does not purchase most products on most days from most stores. In fact it’s the quite the opposite. A typical customer purchases a small percentage of all products from a small number of stores on a few days.

The following table lists some simple rules for assessing the sparsity or denseness of some popular expressions.

|  |  |
| --- | --- |
| Expression | Sparse/dense |
| Regular measure | Sparse |
| Constant Value | Dense (excluding constant null values, true/false values) |
| Scalar expression; for example, count, .properties | Dense |
| <exp1>+<exp2>  <exp1>-<exp2> | Sparse if both exp1 and exp2 are sparse; otherwise dense |
| <exp1>\*<exp2> | Sparse if either exp1 or exp2 is sparse; otherwise dense |
| <exp1> / <exp2> | Sparse if <exp1> is sparse; otherwise dense |
| Sum(<set>, <exp>)  Aggregate(<set>, <exp>) | Inherited from <exp> |
| IIf(<cond>, <exp1>, <exp2>) | Determined by sparsity of default branch (refer to [**IIf function**](#_IIf_Function_in)) |

For more information about sparsity and density, see [Gross margin - dense vs. sparse block evaluation mode in MDX](http://sqlblog.com/blogs/mosha/archive/2008/11/01/gross-margin-dense-vs-sparse-block-evaluation-mode-in-mdx.aspx) (http://sqlblog.com/blogs/mosha/archive/2008/11/01/gross-margin-dense-vs-sparse-block-evaluation-mode-in-mdx.aspx).

### Default Values

Every expression has a default value—the value the expression assumes most of the time. The query processor calculates an expression’s default value and reuses that value across most of its space. Most of the time this value is null, because often (but not always) the result of an expression with null input values is null. The value of reuse is that the engine can compute the null result once, and then compute other values only for the much reduced nonnull space.

Another important use of the default value is in the condition in the **IIF** function. The engine must know which branch is evaluated more often to drive the execution plan.

The following table lists the default values of some popular expressions.

|  |  |  |
| --- | --- | --- |
| Expression | Default value | Comment |
| Regular measure | Null | None. |
| IsEmpty(<regular measure>) | True | The majority of theoretical space is occupied by null values. Therefore, **IsEmpty** will return True most often. |
| <regular measure A> = <regular measure B> | True | Values for both measures are principally null, so this evaluates to True most of the time. |
| <member A> IS <member B> | False | This is different than comparing values – the engine assumes that different members are compared most of the time. |

### Varying Attributes

Cell values mostly depend on attribute coordinates. But some calculations do not depend on every attribute. For example, the following expression depends only on the Customer attribute in the customer dimension.

[Customer].[Customer Geography].properties("Postal Code")

When this expression is evaluated over a subspace involving other attributes, any attributes that the expression doesn’t require can be eliminated, and then the expression can be resolved and projected back over the original subspace. We call the attributes that an expression depends its **varying attributes**. For example, consider the following query.

with member measures.Zip as

[Customer].[Customer Geography].currentmember.properties("Postal Code")

SELECT measures.zip on 0,

[Product].[Category].members ON 1

FROM [Adventure Works]

WHERE [Customer].[Customer Geography].[Customer].&[25818]

This expression depends on the Customer attribute and not the Category attribute; therefore, Customer is a varying attribute and Category is not. In this case the expression is evaluated only once for the customer and not as many times as there are product categories.

## Optimizing MDX

Debugging performance issues across a cube can be difficult if there are many calculations. The first step is to try to narrow down where the problem expression is and then apply best practices to the MDX. In order to narrow down a problem, you will first need a baseline.

### Creating a Query Speed Baseline

Before beginning optimization, you need reproducible cold-cache baseline measurements.

To do this, you should be aware of the following three Analysis Services caches:

* The formula engine cache
* The storage engine cache
* The file system cache

Both the Analysis Services and the operating system caches need to be cleared before you start taking measurements.

#### Clearing the Analysis Services Caches

The Analysis Services formula engine and storage engine caches can be cleared by using the XMLA **ClearCache** command. You can use SQL Server Management Studio to run **ClearCache**.

<ClearCache

xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">

<Object>

<DatabaseID><database name></DatabaseID>

</Object>

</ClearCache>

#### Clearing the Operating System Caches

The file system cache is a bit harder to get rid of because it resides inside Windows itself. You can use any of the following tools to perform this task:

* **Fsutil.exe: Windows File System Utility**

If you have created a separate Windows volume for the cube database, you can dismount the volume itself using the following command:

**fsutil.exe volume dismount** < Drive Letter | Mount Point >

This clears the file system cache for this drive letter or mount point. If the cube database resides only on this location, running this command results in a clean file system cache.

* **RAMMap: Sysinternals tool**

Alternatively, you can use **RAMMap** from Sysinternals (as of this writing, RAMMap v1.11 is available at: <http://technet.microsoft.com/en-us/sysinternals/ff700229.aspx>). RAMMap can help you understand how Windows manages memory. This tool not only allows you to read the file system cache content, it also allows you to purge it. On the **empty** menu, click **Empty System Working Set**, and then click **Empty Standby List**. This clears the file system cache for the entire system. Note that when **RAMMap** starts up, it temporarily freezes the system while it reads the memory content – this can take some time on a large machine. Hence, **RAMMap** should be used with care.

* **Analysis Services Stored Procedure Project (CodePlex): FileSystemCache class**

There is currently a CodePlex project called the Analysis Services Stored Procedure Project found at: <http://asstoredprocedures.codeplex.com/wikipage?title=FileSystemCache>. This project contains code for a utility that enables you to clear the file system cache using a stored procedure that you can run directly on Analysis Services.

Note that neither **FSUTIL** nor **RAMMap** should be used in production cubes –both cause disruption to service. Also note that neither **RAMMap** nor the Analysis Services Stored Procedures Project is supported by Microsoft.

**References**

For additional information, see this article by Greg Galloway that discusses usage of RAMMP <http://www.artisconsulting.com/blogs/greggalloway/Lists/Posts/Post.aspx?ID=19>

#### Measure Query Speeds

When all caches are clear, you should initialize the calculation script by executing a query that returns and caches nothing. Here is an example.

SELECT {} ON 0 FROM [Adventure Works]

Execute the query you want to optimize and then use [SQL Server Profiler](http://msdn.microsoft.com/en-us/library/ms174779.aspx) with the **Standard (default)** trace and these additional events enabled:

* Query Processing\Query Subcube Verbose
* Query Processing\Get Data From Aggregation

Save the profiler trace, because it contains important information that you can use to diagnose slow query times.

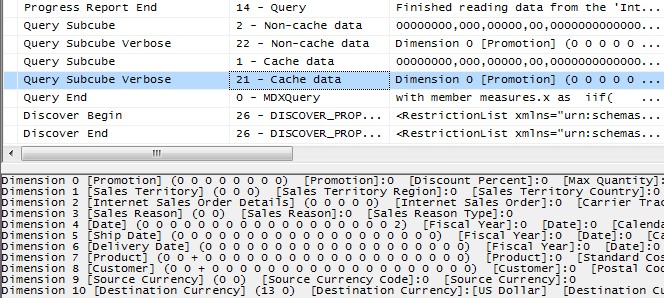


Figure : Sample trace

The text for the query subcube verbose event deserves some explanation. It contains information for each attribute in every dimension:

* **0**: Indicates that the attribute is not included in query (the **All** member is hit).
* **\*** : Indicates that every member of the attribute was requested.
* **+** : Indicates that two or more members of the attribute were requested.
* -: Indicates that a slice below granularity is requested.
* **<integer value>** : Indicates that a single member of the attribute was hit. The integer represents the member’s data ID (an internal identifier generated by the engine).

For more information about the **textdata** field in the Query Subcube Verbose event, see the following resources:

* [Identifying and Resolving MDX Query Performance Bottlenecks in SQL Server 2005 Analysis Services](http://sqlcat.com/whitepapers/archive/2007/12/16/identifying-and-resolving-mdx-query-performance-bottlenecks-in-sql-server-2005-analysis-services.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/12/16/identifying-and-resolving-mdx-query-performance-bottlenecks-in-sql-server-2005-analysis-services.aspx)
* [Configuring the Analysis Services Query Log](http://msdn.microsoft.com/en-us/library/cc917676.aspx) (http://msdn.microsoft.com/en-us/library/cc917676.aspx): Refer to the section, *The Dataset Column in the Query Log Table*.

SQL Server Management Studio displays the total query time; however, it displays only the time taken to retrieve and display the cellset. For large results, the time to render the cellset on the client can actually rival the time it took the server to generate the cellset.

Therefore, instead of using SQL Server Management Studio to measure query time, capture the Query End event (using SQL Server Profiler or other tools) to measure how long the query takes from the server’s perspective and get the Analysis Services engine duration.

### Isolating the Problem

Diagnosing the problem may be straightforward if a simple query calls out a specific calculation. In that case, continue to the next section for remedies.

However, if there are chains of expressions or a complex query, it can be time-consuming to locate the problem. There are a couple ways to scope the problem space:

* Reduce the query to the simplest expression possible that continues to reproduce the performance issue.
* If possible, remove expressions such as MDX scripts, unary operators, measure expressions, custom member formulas, semi-additive measures, and custom rollup properties.

With some client applications, the query generated by the client itself, not the cube, can be the problem. For example, the following problems can arise when client applications generate queries:

* Queries demand large data volumes.
* Queries push down to unnecessarily low granularities.
* Queries unnecessarily bypass aggregations.
* Queries contain calculations that bypass the global and session query processor caches.

If you can confirm that the issue is in the cube itself, comment out calculated members in the cube or query until you have narrowed down the offending calculation. Using a binary chop method is useful to quickly reduce the query to the simplest form that reproduces the issue. Experienced tuners will be able to quickly narrow in on typical calculation issues.

After you have removed calculations to the simplest form in which the performance issue reproduces, the next step is to determine whether the problem lies in the query processor (the formula engine) or the storage engine.

To determine the amount of time the engine spends scanning data, use the SQL Server Profiler trace created earlier. We recommend that you limit the events you capture to noncached storage engine retrievals, by selecting only the query subcube verbose event and filtering on event subclass = 22. The result will be similar to the following:

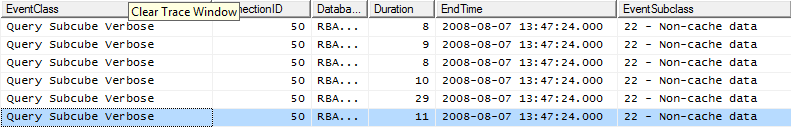


Figure : Trace of query subcube events

If the majority of time is spent in the storage engine (one sign of this is long-running **query subcube** events), the problem is likely with the storage engine. In this case, consider optimizing dimension design, designing aggregations, or using partitions to improve query performance. In addition, you may want to consider optimizing the disk subsystem.

If the majority of time is not spent in the storage engine but in the query processor, focus on optimizing the MDX script or the query itself. Note, the problem can involve both the formula and storage engines.

A **fragmented query space** can be diagnosed when you see many query subcube events generated by a single query. Each request might not take long, but the sum of them can have a significant impact on performance. When you find this behavior, consider warming the cache to make sure that the necessary subcubes and calculations are already cached. You should also consider rewriting the query to remove arbitrary shapes, because arbitrary subcubes cannot be cached. For more information, see [Cache Warming](#_Cache_Warming) later in this white paper.

If the cube and MDX query are already fully optimized, you might consider tuning the cube for thread usage and memory usage. Additional server-level tuning techniques are described in the SQL Server 2008 R2 Analysis Services Operations Guide.

Finally, you might want to consider hardware improvements.

**References**

* [The SQL Server 2008 R2 Analysis Services Operations Guide](http://sqlcat.com/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx)
* [Predeployment I/O Best Practices](http://sqlcat.com/whitepapers/archive/2007/11/21/predeployment-i-o-best-practices.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/11/21/predeployment-i-o-best-practices.aspx): The concepts in this document provide an overview of disk I/O and its impact query performance; focus on the random I/O context.
* [Scalable Shared Databases Part 5](http://sqlvelocity.typepad.com/blog/2011/02/scalable-shared-database-part-5.html) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx): Review to better understand on query performance in context of random I/O vs. sequential I/O.

### Cell-by-Cell Mode vs. Subspace Mode

Almost always, performance obtained by using subspace (block) computation mode is superior to that obtained by using cell-by-cell (naïve) mode.

The following table lists the most common reasons for using cell-by-cell mode, and the steps you can take to use subspace mode instead:

| **Feature or function** | **Comment** |
| --- | --- |
| Use of aliases rather than set expressions | Replace aliases with set expressions where possible.  For example, this query operates in cell-by-cell mode, because an alias has been used instead of a set expression:  with  set y as [Product].[Category].[Category].members  member measures.Naive as  sum(  y,  [Measures].[Internet Sales Amount]  )  select  {measures.Naive,[Measures].[Internet Sales Amount]} on 0 ,  [Customer].[Customer Geography].[Country].members on 1  from [Adventure Works]  cell properties value  In contrast, this very similar query operates in subspace mode because the set expression is explicitly defined.  with  member measures.SubspaceMode as  sum(  [Product].[Category].[Category].members,  [Measures].[Internet Sales Amount]  )  select  {measures.SubspaceMode,[Measures].[Internet Sales Amount]} on 0 ,  [Customer].[Customer Geography].[Country].members on 1  from [Adventure Works]  cell properties value  However, Note: This functionality has been fixed with the latest service pack of SQL Server 2008 R2 Analysis Services. |
| Late binding functions in queries | Late-binding functions are functions that depend on query context and cannot be statically evaluated. These typically include **LinkMember**, **StrToSet**, **StrToMember**, **StrToValue.**  A query is late-bound if an argument can be evaluated only in context.  with member measures.x as (strtomember([Customer].[Customer Geography].currentmember.uniquename),[Measures].[Internet Sales Amount])  select measures.x on 0,  [Customer].[Customer Geography].[Country].members on 1 from [Adventure Works]  cell properties value  In contrast, the following code is statically bound.  with member measures.x as (strtomember("[Customer].[Customer Geography].[Country].&[Australia]"),[Measures].[Internet Sales Amount])  select measures.x on 0,  [Customer].[Customer Geography].[Country].members on 1 from [Adventure Works]  cell properties value |
| User-defined stored procedures | User-defined stored procedures are evaluated in cell-by-cell mode. |
| VBA functions | Some popular Microsoft Visual Basic for Applications (VBA) functions are natively supported in MDX, but they are not optimized to work in subspace mode. |
| **LookupCube** | Linked measure groups are often a viable alternative. |
| Application of cell level security | By definition, cell level security requires cell-by-cell evaluation to ensure the correct security context is applied; therefore performance improvements of block computation cannot be applied. |

For more information, including the list of functions supported in subspace mode, see “[Performance Improvements for MDX in SQL Server 2008 Analysis Services](http://msdn.microsoft.com/en-us/library/bb934106(v=SQL.105).aspx) (http://msdn.microsoft.com/en-us/library/bb934106(v=SQL.105).aspx) in SQL Server Books Online.

### Avoid Assigning Non Null Values to Otherwise Empty Cells

The Analysis Services engine is very efficient at using sparsity of the data to improve performance. However, if you use calculations in which nonempty values replace empty values, Analysis Services cannot eliminate these rows.

For example, the following query replaces empty values with the dash; therefore the **NON EMPTY** keyword does not eliminate them.

WITH member measures.x AS

IIF( NOT ISEMPTY([Measures].[Internet Sales Amount]),[Measures].[Internet Sales Amount],"-")

SELECT descendants([Date].[Calendar].[Calendar Year].&[2004] ) ON 0,

NON EMPTY [Customer].[Customer Geography].[Customer].members ON 1

FROM [Adventure Works]

WHERE measures.x

The **NON EMPTY** keyword also operates on cell values but not on formatted values. Therefore, in rare cases you can use the format string to replace null values with the same character while still eliminating empty rows and columns in roughly half the execution time, as shown in this example:.

WITH member measures.x AS

[Measures].[Internet Sales Amount], FORMAT\_STRING = "#.00;(#.00);#.00;-"

SELECT descendants([Date].[Calendar].[Calendar Year].&[2004] ) ON 0,

NON EMPTY [Customer].[Customer Geography].[Customer].members ON 1

FROM [Adventure Works]

WHERE measures.x

The reason this workaround can only be used in rare cases is that the queries are not really equivalent – the second query eliminates completely empty rows. More importantly, neither Excel nor SQL Server Reporting Services supports the fourth argument of the **FORMAT\_STRING** expression.

**References**

* For more information about using the **FORMAT\_STRING** calculation property, see [FORMAT\_STRING Contents (MDX)](http://msdn.microsoft.com/en-us/library/ms146084.aspx) (http://msdn.microsoft.com/en-us/library/ms146084.aspx) in SQL Server Books Online.
* For more information about how Excel uses the **FORMAT\_STRING** property, see [Create or delete a custom number format](http://office.microsoft.com/en-us/excel-help/create-or-delete-a-custom-number-format-HP010342372.aspx) (http://office.microsoft.com/en-us/excel-help/create-or-delete-a-custom-number-format-HP010342372.aspx).

### Sparse/Dense Considerations with “expr1 \* expr2” Expressions

When you write expressions as products of two other expressions, you can get better performance by placing the [**sparser**](file:///C:/Users/Beth/Documents/Vend/Docs/SQL%20White%20papers/SSASPerfGuide2008_v1_(2).docx#_Expression_Sparsity) expression on the left-hand side. An expression is considered sparse if there are few non-null values compared to the total number of cells. For more information, see [Expression Sparsity](#_Expression_Sparsity) earlier in this section.

Consider the following two queries on Adventure Works, which perform a currency conversion calculation, applying the exchange rate at leaves of the Date dimension. The only difference between the two queries is the order of the expressions in the product of the cell calculation.

**Sparse First**

WITH cell CALCULATION x for '({[Measures].[Internet Sales Amount]},leaves([Date]))'

AS [Measures].[Internet Sales Amount] \*

([Measures].[Average Rate],[Destination Currency].[Destination Currency].&[EURO])

SELECT

NON EMPTY [Date].[Calendar].members ON 0,

NON EMPTY [Product].[Product Categories].members ON 1

FROM [Adventure Works]

WHERE ([Measures].[Internet Sales Amount], [Customer].[Customer Geography].[State-Province].&[BC]&[CA])

**Dense First**

WITH cell CALCULATION x FOR '({[Measures].[Internet Sales Amount]},leaves([Date]))'

AS

([Measures].[Average Rate],[Destination Currency].[Destination Currency].&[EURO])\*

[Measures].[Internet Sales Amount]

SELECT

NON EMPTY [Date].[Calendar].members ON 0,

NON EMPTY [Product].[Product Categories].members ON 1

FROM [Adventure Works]

WHERE ([Measures].[Internet Sales Amount], [Customer].[Customer Geography].[State-Province].&[BC]&[CA])

The same results are returned, but using the sparser [Internet Sales Amount] first results in about a 10% savings in speed. This savings could be substantially greater in other calculations. The amount of improvement depends on the relative sparsity between the two expressions, so performance benefits may vary.

### IIf Function in SQL Server 2008 Analysis Services

The **IIF** MDX function is a commonly used expression that can be costly to evaluate. The engine optimizes performance based on a few simple criteria. The **IIF** function takes three arguments:

iif(<condition>, <then branch>, <else branch>)

In other words:

* Where the condition evaluates to TRUE, the value from the THEN branch is used.
* Otherwise the ELSE branch expression is used.

Note that we say “used” rather than “evaluated”. In fact, one or both branches may be evaluated even if theire values are not used. It might be cheaper for the engine to evaluate the expression over the entire space and use it when needed – that’s what we call an **eager** **plan** – than it would be to chop up the space into a potentially enormous number of fragments and evaluate only where needed – whatwe call a **strict** **plan**.

**Note:** One of the most common errors in MDX scripting is using **IIF** when the condition depends on cell coordinates instead of values. If the condition depends on cell coordinates, use scopes and assignments as described in section 2. When this is done, the condition is not evaluated over the space and the engine does not evaluate one or both branches over the entire space. Admittedly, in some cases, using assignments forces some unwieldy scoping and repetition of assignments, but it is always worthwhile comparing the two approaches.

Thus, when using **IIF**, consider these conditions:

1. Is the query plan is expensive orinexpensive?

Most **IIF** condition query plans are inexpensive, but complex nested conditions with more IIF functions can go to cell by cell computation.

1. What value does the condition take most often?

The engine examines the condition’s [default value](#_Default_Values). If the condition’s default value is true, the THEN branch is the default branch – the branch that is evaluated over most of the subspace.

Knowing a few simple rules on how the condition is evaluated can help you to determine the default branch:

* In sparse expressions, most cells are empty. The default value of the **ISEMPTY** function on a sparse expression is **true**.
* Comparison to zero of a sparse expression is **true**.
* The default value of the **IS** operator is **false**.
* If the condition cannot be evaluated in subspace mode, there is no default branch.

An example might help illustrate how to use these rules. One of the most common uses of the **IIF** function is to check whether the denominator is nonzero, as shown here:

IIF([Measures].[Internet Sales Amount]=0

, null

, [Measures].[Internet Order Quantity]/[Measures].[Internet Sales Amount])

Note that there is no calculation on Internet Sales Amount.

* Therefore the expression is a **regular measure expression** and it is sparse.
* Therefore the default value of the condition is **true**.
* Thus the default branch is the **THEN** branch with the null expression.

The following table shows how each branch of an **IIF** function is evaluated.

|  |  |  |  |
| --- | --- | --- | --- |
| Branch query plan | Branch is default branch | Branch expression sparsity | Evaluation |
| Expensive | Not applicable | Not applicable | Strict |
| Inexpensive | True | Not applicable | Eager |
| Inexpensive | False | Dense | Strict |
| Inexpensive | False | Sparse | Eager |

In Analysis Services, you can overrule the default behavior by using query hints.

IIF(

[<condition>

, <then branch> [hint [Eager | Strict]]

, <else branch> [hint [Eager | Strict]]

)

You might want to change the default behavior in the following common scenarios:

* The engine determines the query plan for the condition is expensive and evaluates each branch in strict mode.
* The condition is evaluated in cell-by-cell mode, and each branch is evaluated in eager mode.
* The branch expression is dense but easily evaluated.

For example, consider the following simple expression, which takes the inverse of a measure.

WITH member

measures.x AS

IIF(

[Measures].[Internet Sales Amount]=0

, null

, (1/[Measures].[Internet Sales Amount]) )

SELECT {[Measures].x} ON 0,

[Customer].[Customer Geography].[Country].members \*

[Product].[Product Categories].[Category].members ON 1

FROM [Adventure Works]

cell properties value

This query meets the conditions described above:

* The query plan is not expensive.
* The **ELSE** branch is not the default branch.
* The expression is dense.

Therefore, this expression is evaluated in strict mode. Using strict mode forces the engine to materialize the space over which it is evaluated.

You can diagnose this behavior in SQL Server Profiler by looking at the Query Subcube Verbose events:.

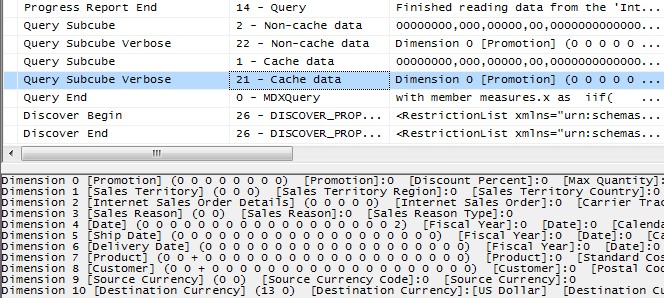


Figure : Default IIf query trace

Pay attention to the subcube definition for the Product and Customer dimensions (dimensions 7 and 8 respectively) with the ‘+’ indicator on the Country and Category attributes. This means that more than one but not all members are included. The query processor has determined which tuples meet the condition and partitioned the space, and it is evaluating the fraction over that space.

To prevent the query plan from partitioning the space, the query can be modified as follows. Note the query hint, in bold.

WITH member

measures.x AS

IIF(

[Measures].[Internet Sales Amount]=0

, null

, (1/[Measures].[Internet Sales Amount]) **hint eager**)

SELECT {[Measures].x} on 0,

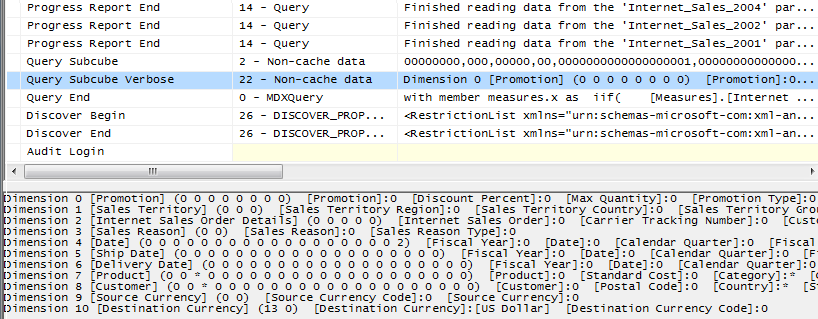
[Customer].[Customer Geography].[Country].members \*

[Product].[Product Categories].[Category].members ON 1

FROM [Adventure Works]

cell properties value

Figure : IIf trace with MDX query hints

Now in the Profiler trace, in the Query Subcube Verbose event, the same attributes are marked with ***‘\*’***, meaning that the expression is evaluated over the entire space instead of a partitioned space. 

### Cache Partial Expressions and Cell Properties

**Partial expressions**, or expressions that are part of a calculated member or assignment, are not cached. Therefore, if an expensive subexpression is used more than once, consider creating a separate calculated member to allow the query processor to cache and reuse the expression.

For example, consider the following expression:

this = IIF(<expensive expression >= 0, 1/<expensive expression>, null);

The repeated partial expressions can be extracted and replaced with a hidden calculated member as follows.

CREATE member currentcube.measures.MyPartialExpression AS <expensive expression> , visible=0;

this = IIF(measures.MyPartialExpression >= 0, 1/ measures.MyPartialExpression, null);

When rewritten, only the value cell property is cached.

If you have complex cell properties to support such things as bubble-up exception coloring, consider creating a separate calculated measure. For example, the following expression includes color in the definition, which creates extra work every time the expression is used.

CREATE member currentcube.measures.[Value] AS <exp> , backgroundColor=<complex expression>;

The following expression is more efficient because it creates a calculated measure to handle the color effect.

CREATE member currentcube.measures.MyCellProperty AS <complex expression> , visible=0;

CREATE member currentcube.measures.[Value] AS <exp> , backgroundColor=<MyCellProperty>;

### Eliminate Varying Attributes in Set Expressions

Set expressions do not support [**varying attributes**](#_Varying_Attributes). This restriction applies to all set functions including **Filter**, **Aggregate**, **Avg**, and others. You can work around this problem by explicitly overwriting invariant attributes to a single member.

For example, in this calculation, the average of sales is computed, but only for sales exceeding $100. The query also runs very slowly – ina recent desktop test on a fast server, it took about 55 seconds.

WITH member measures.AvgSales AS

AVG(

FILTER(

descendants([Customer].[Customer Geography].[All Customers],,leaves)

, [Measures].[Internet Sales Amount]>100

)

,[Measures].[Internet Sales Amount]

)

SELECT measures.AvgSales ON 0,

[Customer].[Customer Geography].[City].members ON 1

FROM [Adventure Works]

However, the average of sales for all customers everywhere should not depend on the current city. In other words, City should not be a **varying attribute**.

You can eliminate City as a varying attribute and make the expression more efficient by using the **All** member as follows:

WITH member measures.AvgSales AS

AVG(

FILTER(

descendants([Customer].[Customer Geography].[All Customers],,leaves)

, [Measures].[Internet Sales Amount]>100

)

,[Measures].[Internet Sales Amount]

)

member measures.AvgSalesWithOverWrite AS (measures.AvgSales, [All Customers])

SELECT measures.AvgSalesWithOverWrite ON 0,

[Customer].[Customer Geography].[City].members ON 1

FROM [Adventure Works]

With the modification, this query ran much faster, in under a second.

To see the difference, you can compare the SQL Server Profiler traces of the two queries:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EventClass > EventSubClass | AvgSalesWithOverwrite | | AvgSales | |
| Events | Duration | Events | Duration |
| Query Cube End | 1 | 515 | 1 | 161526 |
| Serial Results End | 1 | 499 | 1 | 161526 |
| Query Dimension | 586 |  |  |  |
| Get Data From Cache > Get Data from Flat Cache | 586 |  |  |  |
| Query Subcube > Non-Cache Data | 5 | 64 | 5 | 218 |

Figure . Effect of removing varying attributes on query duration

Look at the duration of **Query Subcube** > **Non-Cache Data**. From the fact that this value is relatively small, you can deduce that most of the query calculation is done by the formula engine.

Now look at the **AvgSales** calculation, and notice that most of the query durations correspond to the values for the **Serial Results** event. The latter event reports the status of serializing axes and cells.

From this analysis you can see that using [All Customers] ensures that the expression is evaluated only once for each Customer, improving performance.

### Eliminate Cost of Computing Formatted Values

In some circumstances, the cost of determining the format string for an expression outweighs the cost of the value itself. To determine whether this applies to a slow-running query, compare execution times with and without the formatted value cell property, as in the following query.

SELECT [Measures].[Internet Average Sales Amount] ON 0

FROM[Adventure Works] cell properties value

If the result is noticeable faster without the formatting, apply the formatting directly in the script as follows.

SCOPE([Measures].[Internet Average Sales Amount]);

FORMAT\_STRING(this) = "currency";

END SCOPE;

Execute the query (with formatting applied) to determine the extent of any performance benefit.

### NON\_EMPTY\_BEHAVIOR

In some situations, it is expensive to compute the result of an expression, even if you know it will be null beforehand based on the value of some indicator tuple. In earlier versions of SQL Server Analysis Services, the **NON\_EMPTY\_BEHAVIOR** property was sometimes helpful for these kinds of calculations. When this property evaluates to null, the expression is guaranteed to be null and (most of the time) vice versa.

In past releases, changing this property often resulted in substantial performance improvements. However, starting with SQL Server 2008, the property can often be ignored, because the engine automatically deals with nonempty cells in many cases, and manually setting the property can result in degraded performance.

To determine whether you should use this property or not, we recommend that you eliminate it from the MDX script and do some performance testing, and add it back only if using it leads to improvement.

For assignments, the property is used as follows.

this = <e1>;

Non\_Empty\_Behavior(this) = <e2>;

For calculated members in the MDX script, the property is used this way.

create member currentcube.measures.x as <e1>, non\_empty\_behavior = <e2>

In SQL Server 2005 Analysis Services, there were complex rules on how the property could be defined, when the engine used it or ignored it, and how the engine would use it.

In SQL Server 2008 Analysis Services, the behavior of this property changed, and remains as described here for SQL Server 2012 and SQL Server 2014:

* When **NON\_EMPTY\_BEHAVIOR** is null, the expression must also be null. If this is not true, incorrect query results can be returned.
* The reverse is not necessarily true. That is, the **NON\_EMPTY\_BEHAVIOR** expression can return non null when the original expression is null.
* The engine more often than not ignores this property and deduces the nonempty behavior of the expression on its own.

If the **NON\_EMPTY\_BEHAVIOR** property is defined and is applied by the engine, it is semantically equivalent to the following expression, even if performance is not necessarily equivalent:

this = <e1> \* iif(isempty(<e2>), null, 1)

The **NON\_EMPTY\_BEHAVIOR** property is used if <e2> is [sparse](file:///C:/Users/Beth/Documents/Vend/Docs/SQL%20White%20papers/SSASPerfGuide2008_v1_(2).docx#_Sparsity) and <e1> is [dense](file:///C:/Users/Beth/Documents/Vend/Docs/SQL%20White%20papers/SSASPerfGuide2008_v1_(2).docx#_Sparsity) or <e1> is evaluated in **naïve** or [**cell-by-cell**](file:///C:/Users/Beth/Documents/Vend/Docs/SQL%20White%20papers/SSASPerfGuide2008_v1_(2).docx#_Subspace_computation) mode.

If these conditions are not met and both <e1> and <e2> are sparse (that is, if <e2> is much sparser than <e1>), you may be able to achieve improved performance by forcing the behavior as follows.

this = iif(isempty(<e2>), null, <e1>);

The **NON\_EMPTY\_BEHAVIOR** property can be expressed as a simple tuple expression including simple member navigation functions such as .**PREVMEMBER** or **.PARENT** or an enumerated set. An enumerated set is equivalent to **NON\_EMPTY\_BEHAVIOR** of the resultant sum.

**References**

The following links provide help on how to optimize MDX:

* [Query calculated members invalidate formula engine cache](http://cwebbbi.wordpress.com/2009/01/30/formula-caching-and-query-scope/) (http://cwebbbi.wordpress.com/2009/01/30/formula-caching-and-query-scope/) by Chris Webb
* [Subselect preventing caching](http://cwebbbi.wordpress.com/2008/10/28/reporting-services-generated-mdx-subselects-and-formula-caching/) (http://cwebbbi.wordpress.com/2008/10/28/reporting-services-generated-mdx-subselects-and-formula-caching/) by Chris Webb
* [Measure datatypes](http://www.codeplex.com/bidshelper/Wiki/View.aspx?title=Measure%20Group%20Health%20Check) (http://bidshelper.codeplex.com/wikipage?title=Measure%20Group%20Health%20Check&ProjectName=bidshelper)
* [Currency datatype](http://sqlcat.com/technicalnotes/archive/2008/09/25/the-many-benefits-of-money-data-type.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2008/09/25/the-many-benefits-of-money-data-type.aspx)

## Aggregations

An **aggregation** is a data structure that stores precalculated data that Analysis Services uses to enhance query performance. You can define the aggregation design for each partition independently. Each partition can be thought of as being an aggregation at the lowest granularity of the measure group. Aggregations that are defined for a partition are processed out of the leaf level partition data by aggregating it to a higher granularity.

When a query requests data at higher levels, the aggregation structure can deliver the data more quickly because the data is already aggregated in fewer rows. As you design aggregations, you must consider the querying benefits that aggregations provide compared with the time it takes to create and refresh the aggregations. In fact, adding unnecessary aggregations can worsen query performance because the rare hits move the aggregation into the file cache at the cost of moving something else out.

While aggregations are physically designed per measure group partition, the optimization techniques for maximizing aggregation design apply whether you have one or many partitions. In this section, unless otherwise stated, aggregations are discussed in the fundamental context of a cube with a single measure group and single partition. For more information about how you can improve query performance using multiple partitions, see [Partition Strategy](#_Partition_Strategy).

### Detecting Aggregation Hits

You can use SQL Server Profiler to view how and when aggregations are used to satisfy queries.

Within SQL Server Profiler, there are several events that describe how a query is fulfilled. The event that specifically pertains to aggregation hits is the **Get Data From Aggregation** event.

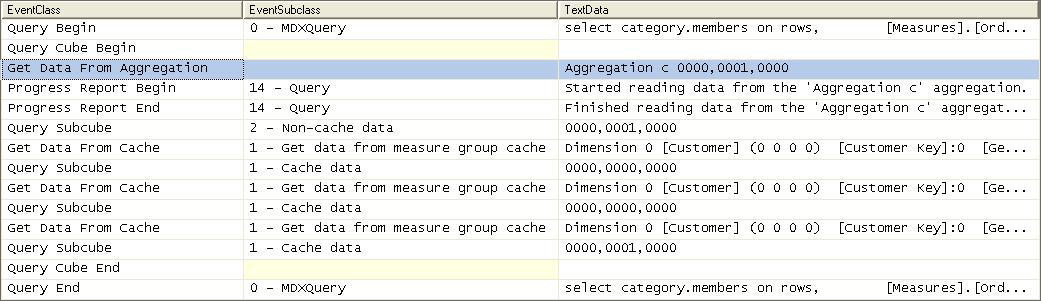


Figure : Scenario 1: SQL Server Profiler trace for cube with an aggregation hit

This figure displays a SQL Server Profiler trace of the query’s resolution against a cube with aggregations. In the SQL Server Profiler trace, the operations that the storage engine performs to produce the result set are revealed.

The storage engine gets data from Aggregation C 0000, 0001, 0000 as indicated by the **Get Data From Aggregation** event. In addition to the aggregation name, Aggregation C, Figure 10 displays a vector, **000, 0001, 0000**, that describes the content of the aggregation. More information on what this vector actually means is described in the next section, [How to Interpret Aggregations](#_How_to_Interpret). The aggregation data is loaded into the storage engine measure group cache from where the query processor retrieves it and returns the result set to the client.

Suppose no aggregations can satisfy the query request? In that case, the **Get Data From Aggregation** event will be missing, as you can see from the following example, which shows the same cube with no aggregations.

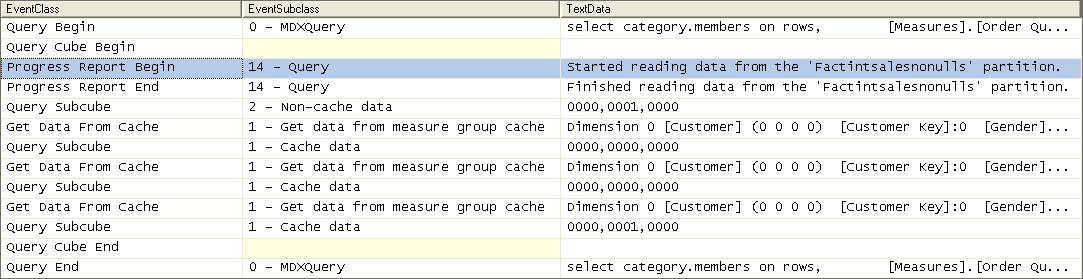


Figure : Scenario 2: SQL Server Profiler trace for cube with no aggregation hit

After the query is submitted, rather than retrieving data from an aggregation, the storage engine goes to the detail data in the partition. From this point, the process is the same. The data is loaded into the storage engine measure group cache.

### How to Interpret Aggregations

When Analysis Services creates an aggregation, each dimension is named by a vector, indicating whether the attribute points to the attribute or to the **All** level. The attribute level is represented by ***1*** and the **All** level is represented by ***0***.

For example, consider the following examples of aggregation vectors for the product dimension:

* **Aggregation By ProductKey Attribute** = [Product Key]:1 [Color]:0 [Subcategory]:0 [Category]:0 or **1000**
* **Aggregation By Category Attribute** = [Product Key]:0 [Color]:0 [Subcategory]:0 [Category]:1 or **0001**
* **Aggregation By ProductKey.All** and **Color.All** and **Subcategory.All** and **Category.All** = [Product Key]:0 [Color]:0 [Subcategory]:0 [Category]:0 or **0000**

To identify each aggregation, Analysis Services combines the dimension vectors into one long vector path, also called a **subcube**, with each dimension vector separated by commas.

The order of the **dimensions in the vector** is determined by the order of the dimensions in the measure group. To find the order of dimensions in the measure group, use one of the following two techniques:

1. With the cube opened in SQL Server Business Intelligence Development Studio, review the order of dimensions in a measure group on the **Cube Structure** tab. The order of dimensions in the cube is displayed in the **Dimensions** pane.
2. As an alternative, review the order of dimensions listed in the cube’s XMLA definition.

The order of **attributes in the vector for each dimension** is determined by the order of attributes in the dimension. You can identify the order of attributes in each dimension by reviewing the dimension XML file. For example, the subcube definition (0000, 0001, 0001) describes an aggregation for the following:

* Product – All, All, All, All
* Customer – All, All, All, State/Province
* Order Date – All, All, All, Year

Understanding how to read these vectors is helpful when you review aggregation hits in SQL Server Profiler. In SQL Server Profiler, you can view how the vector maps to specific dimension attributes by enabling the **Query Subcube Verbose** event. In some cases (such as when attributes are disabled), it may be easier to view the **Aggregation Design** tab and use the **Advanced** view of the aggregations.

### Aggregation Tradeoffs

Aggregations can improve query response time but they can increase processing time and disk storage space, consume memory that could otherwise be allocated to cache, and potentially slow the speed of other queries. The latter may occur because there is a direct correlation between the number of aggregations and the duration for the Analysis Services storage engine to parse them. As well, aggregations may cause thrashing due to their potential impact to the file system cache. A general rule of thumb is that aggregations should be less than 1/3 the size of the fact table.

### Building Aggregations

Individual aggregations are organized into collections of aggregations called **AggregationDesigns**. Once created, an **AggregationDesign** can be applied to many partitions. A single measure group can also have multiple **AggregationDesigns**, so that you can choose different sets of aggregations for different partitions.

To help Analysis Services successfully apply the **AggregationDesign** algorithm, you can perform the following optimization techniques to influence and enhance the aggregation design. In this section we will discuss the following:

* The importance of attribute hierarchies for aggregations
* Aggregation design and partitions
* Specifying statistics about cube data
* Suggesting aggregation candidates
* Usage-based optimization
* Large cube aggregations
* Distinct count partition aggregation considerations

#### Importance of Attribute Hierarchies

Aggregations work better when the cube is based on a multidimensional data model that includes natural hierarchies. While it is common in relational databases to have attributes independent of each other, multidimensional star schemas have attributes related to each other to create natural hierarchies. This is important because it allows aggregations built at a lower level of a natural hierarchy to be used when querying at a higher level.

Note that attributes that are exposed only in attribute hierarchies are not automatically considered for aggregation by the Aggregation Design Wizard. Therefore, queries involving these attributes are satisfied by summarizing data from the primary key. Without the benefit of aggregations, query performance against these attributes hierarchies can be slow. To enhance performance, it is possible to flag an attribute as an aggregation candidate by using the **Aggregation Usage** property. For more information about this technique, see [Suggesting Aggregation Candidates](#_Suggesting_aggregation_candidates_1). However, before you modify the **Aggregation Usage** property, you should consider whether you can take advantage of user hierarchies.

#### Aggregation Design and Partitions

When you define your partitions, they do not necessarily have to contain uniform datasets or aggregation designs. For example, for a given measure group, you may have 3 yearly partitions, 11 monthly partitions, 3 weekly partitions, and 1–7 daily partitions. Heterogeneous partitions with different levels of detail allows you to more easily manage the loading of new data without disturbing existing, larger, and stale partitions (more on this in the processing section) and you can design aggregations for groups of partitions that share the same access pattern. For each partition, you can use a different aggregation design. By taking advantage of this flexibility, you can identify those data sets that require higher aggregation design.

Consider the following example. In a cube with multiple monthly partitions, new data may flow into the single partition corresponding to the latest month. Generally that is also the partition most frequently queried. A common aggregation strategy in this case is to perform usage-based optimization to the most recent partition, leaving older, less frequently queried partitions as they are.

If you automate partition creation, it is easy to simply set the **AggregationDesignID** for the new partition at creation time and specify the slice for the partition. After that the partition is ready to be processed. At a later stage, you may choose to update the aggregation design for a partition when its usage pattern changes – again, you can just update the **AggregationDesignID**, but you will also need to invoke **ProcessIndexes** so that the new aggregation design takes effect for the processed partition.

#### Specifying Statistics About Cube Data

To make intelligent assessments of aggregation costs, the design algorithm analyzes statistics about the cube for each aggregation candidate. Examples of this metadata include member counts and fact table counts. Ensuring that your metadata is up-to-date can improve the effectiveness of your aggregation design.

Whenever you use multiple partitions for a given measure group, ensure that you update the data statistics for each partition. More specifically, it is important to ensure that the partition data and member counts (such as **EstimatedRows** and **EstimatedCount** properties) accurately reflect the specific data in the partition and not the data across the entire measure group.

#### Suggesting Aggregation Candidates

When Analysis Services designs aggregations, the aggregation design algorithm does not automatically consider every attribute for aggregation. Consequently, in your cube design, verify the attributes that are considered for aggregation and determine whether you need to suggest additional aggregation candidates. To streamline this process, Analysis Services uses the **Aggregation Usage** property to determine which attributes it should consider. For every measure group, verify the attributes that are automatically considered for aggregation and then determine whether you need to suggest additional aggregation candidates.

**Aggregation Usage Rules**

An **aggregation candidate** is an attribute that Analysis Services considers for potential aggregation. To determine whether or not a specific attribute is an aggregation candidate, the storage engine relies on the value of the **Aggregation Usage** property. The **Aggregation Usage** property is assigned a per-cube attribute, so it globally applies across all measure groups and partitions in the cube. For each attribute in a cube, the **Aggregation Usage** property can have one of four potential values: **Full**, **None**, **Unrestricted**, and **Default**.

* **Full**— Each aggregation for the cube must include either this attribute, or a related attribute that is lower in the attribute chain. For example, suppose you have a product dimension with the following chain of related attributes: [Product], [Product Subcategory], and [Product Category]. If you specify the **Aggregation Usage** for [Product Category] to be **Full**, Analysis Services may create an aggregation that includes [Product Subcategory] as opposed to [Product Category], given that [Product Subcategory] is related to [Category] and can be used to derive [Category] totals.
* **None**—No aggregation for the cube can include this attribute.
* **Unrestricted**—No restrictions are placed on the aggregation designer; however, the attribute must still be evaluated to determine whether it is a valuable aggregation candidate.
* **Default**—The designer applies a **default rule** based on the type of attribute and dimension. This defines the default value of the **Aggregation Usage** property.

The default rule is highly conservative about which attributes are considered for aggregation. The default rule is broken down into four constraints.

* **Default Constraint 1—Unrestricted** - For a dimension’s measure group granularity attribute, default means **Unrestricted**. The granularity attribute is the same as the dimension’s key attribute as long as the measure group joins to a dimension using the primary key attribute.
* **Default Constraint 2**—**None for Special Dimension Types –**The default means **None** in these cases: all attributes in a many-to-many solution (except **All**); nonmaterialized reference dimensions; data mining dimensions. This means you can sometimes benefit from creating leaf level projections for many-to-many dimensions. Note, these defaults do not apply for parent-child dimensions; for more information, see the [Special Considerations > Parent-Child dimensions](#_Parent-Child_Dimensions) section.
* **Default Constraint 3**—**Unrestricted for Natural Hierarchies -** A naturalhierarchy is a user hierarchy where all attributes participating in the hierarchy contain attribute relationships to the attribute sourcing the next level. For such attributes, default means **Unrestricted,** except for nonaggregatable attributes, which are set to **Full** (even if they are not in a user hierarchy).
* **Default Constraint 4**—**None For Everything Else**. For all other dimension attributes, default means **None**.

**Aggregation Usage Guidelines**

Given these behaviors of the **Aggregation Usage** property, apply the following guidelines when designing or using aggregations:

* **Attributes exposed solely as attribute hierarchies**- If a given attribute is only exposed as an attribute hierarchy such as color, you may want to change its **Aggregation Usage** property as follows.
  + First, change the value of the **Aggregation Usage** property from **Default** to **Unrestricted** if the attribute is a commonly used attribute or if there are special considerations for improving the performance in a particular pivot or drilldown. For example, if you have highly summarized scorecard style reports, you want to ensure that the users experience good initial query response time before drilling around into more detail.
  + While setting the **Aggregation Usage** property of a particular attribute hierarchy to **Unrestricted** is appropriate is some scenarios, do not set all attribute hierarchies to **Unrestricted**. Increasing the number of attributes to be considered increases the problem space for the aggregation algorithm. The wizard can take at least an hour to complete the design and considerably much more time to process. Set the property to **Unrestricted** only for the commonly queried attribute hierarchies. The general rule is to allow 5-10 **Unrestricted** attributes per dimension.
  + Next, change the value of the **Aggregation Usage** property from **Default** to **Full** in the unusual case that it is used in virtually every query you want to optimize. This is a rare case, and this change should be made only for attributes that have a relatively small number of members.
* **Infrequently used attributes**—For attributes participating in natural hierarchies, you may want to change the **Aggregation Usage** property from **Default** to **None** if users use it infrequently. Using this approach can help you reduce the aggregation space and limit the design to the desired level of 5-10 **Unrestricted** attributes per dimension. For example, certain attributes might be used only by a few advanced users, and they might be willing to accept slightly slower performance. In this scenario, you are essentially forcing the aggregation design algorithm to spend time building only the aggregations that provide the most benefit to the majority of users.

#### Usage-Based Optimization

The Usage-Based Optimization Wizard reviews the queries in the query log (which you must set up beforehand) and designs aggregations that cover up to the top 100 slowest queries. We recommend that you specify a 100% performance gain when you use the Usage-Based Optimization Wizard, to design aggregations to avoid hitting the partition directly.

After the aggregations are designed, you can add them to the existing design or completely replace the design. Be careful adding them to the existing design – the two designs may contain aggregations that serve almost identical purposes, but which when combined are redundant. Always inspect the new aggregations compared to the old and ensure there are no near-duplicates.

Note that aggregation designs have a costly metadata impact. Do not overdesign, but try to keep the number of aggregation designs per measure group to a minimum.

When you are satisfied with your aggregations, you can copy the aggregation design to other partitions, using either SQL Server Management Studio or the design tools in SQL Server Data Tools.

**References**

* [Reintroducing Usage-Based Optimization in SQL Server 2008 Analysis Services](http://sqlcat.com/technicalnotes/archive/2008/11/18/reintroducing-usage-based-optimization-in-sql-server-2008-analysis-services.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2008/11/18/reintroducing-usage-based-optimization-in-sql-server-2008-analysis-services.aspx)
* [Analysis Services 2005 Aggregation Design Strategy](http://sqlcat.com/technicalnotes/archive/2007/09/11/analysis-services-2005-aggregation-design-strategy.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2007/09/11/analysis-services-2005-aggregation-design-strategy.aspx)
* [Microsoft SQL Server Community Samples: Analysis Services](http://sqlsrvanalysissrvcs.codeplex.com/) (http://sqlsrvanalysissrvcs.codeplex.com/): This CodePlex project contains many useful Analysis Services CodePlex samples, including the Aggregation Manager

#### Large Cube Aggregation Considerations

Small cubes might not even need aggregations, because aggregations are not built for partitions with fewer records than the **IndexBuildThreshold** (which has a default value of 4096). Even if the cube partitions exceed the **IndexBuildThreshold**, aggregations that are correctly designed for smaller cubes might not be the correct ones for large cubes.

As cubes become larger, it becomes more important to design aggregations and to do so correctly. As a general rule of thumb, MOLAP performance is approximately between 10 and 40 million rows per second per core, plus the I/O for aggregating data.

Larger cubes have more constraints such as small processing windows and/or not enough disk space. Therefore it may be difficult to create all of your desired aggregations. You will need to weigh the tradeoffs carefully when designing aggregations.

## Cache Warming

Cache warming can be a last-ditch effort for improving the performance of a query. The following sections describe guidelines and implementation strategies for cache warming.

### Cache Warming Guidelines

During querying, memory is primarily used to store cached results in the storage engine and query processor caches. To optimize the benefits of caching, you can often increase query responsiveness by preloading data into one or both of these caches. To do this, perform [**cache warming**](http://sqlcat.com/technicalnotes/archive/2007/09/11/how-to-warm-up-the-analysis-services-data-cache-using-create-cache-statement.aspx) with either of these methods:

* Pre-executing one or more queries.
* Using the CREATE CACHE statement. This statement returns no cellsets and has the advantage of executing faster because it bypasses the query processor.

When possible, Analysis Services returns results from the Analysis Services data cache without using aggregations, because it is the fastest way to get data. With smaller cubes, the server might have enough memory to keep a large portion of the data in the cache. In such cases, aggregations are not needed and existing aggregations may never be used. Cache warming can be used to ensure that users will always have excellent performance.

With larger cubes, however, the server might not have sufficient memory to keep query data in the cache. Additionally, cached results can be pushed out by other query results. Hence, cache warming might help only some of the queries.

Therefore, in larger cubes, it is important to create well-designed aggregations to provide solid query performance. Note that too many aggregations can thrash the cache, as different data result sets and aggregations are requested and swapped from the cache.

### Implementing a Cache Warming Strategy

While cache warming can improve the performance of a query, there is a significant difference between performance on a cold cache and on a warm cache. You need to ensure that sufficient memory is available, so that the cache is not being thrashed.

#### What to cache

To discover what needs to be cached (which can be difficult at times), use SQL Server Profiler to trace the query execution and examine the subcube events.

* If you find many subcube requests to the same grain, the query processor might be making many requests for slightly different data, resulting in the storage engine making many small but time-consuming I/O requests. It would be better to retrieve the data *en masse* and then return results from the cache.
* To pre-execute queries, you can create an application (or use something like **ascmd**) that executes a set of generalized queries to simulate typical user activity. This expedites the process of populating the cache. Execute these queries right after you start Analysis Services or after processing, to preload the cache prior to user queries.
* To determine how to generalize your queries, you can sometimes refer to the Analysis Services query log to determine the dimension attributes typically queried. However, be careful when you generalize based on this information, because you might include attributes or subcubes that are not beneficial and unnecessarily take up cache.
* When testing the effectiveness of different cache-warming queries, be sure to empty the query results cache between each test to ensure the validity of your testing.
* Because cached results can be pushed out by other query results, you should consider scheduling refresh of the cache results.
* Limit cache warming to what can fit in memory, leaving enough for other queries to be cached.

#### How to warm the cache

The Analysis Services formula engine can only be warmed by MDX queries. To warm the storage engine caches, you can use the **WITH CACHE** or **CREATE CACHE** statements, as described in this article:

* [How to warm up the Analysis Services data cache using Create Cache statement?](http://sqlcat.com/technicalnotes/archive/2007/09/11/how-to-warm-up-the-analysis-services-data-cache-using-create-cache-statement.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2007/09/11/how-to-warm-up-the-analysis-services-data-cache-using-create-cache-statement.aspx)

## Scale-Out

If you have many concurrent users querying your Analysis Services cubes, a potential query performance solution is to scale out your Analysis Services query servers. There are different forms of scale-out, which are discussed in the [Analysis Services 2008 R2 Operations Guide](http://sqlcat.com/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx) (<http://sqlcat.com/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx>), but the basic principle is that ) so there are multiple servers to address user queries. You can do this by pointing multiple query servers at the same database, or by replicating the database.

Scale-out is beneficial in cases such as the following:

* Your server is under memory pressure due to concurrency. Scaling out allows you to distribute the query load to multiple servers, thus alleviating memory bottlenecks on a single server. Memory pressure can be caused by many issues. For example:
  + Users execute many different unique queries, filling up and thrashing available cache.
  + Complex or large queries require large subcubes and a large memory space.
  + Too many concurrent users access the same server.
* You have many long running queries against your Analysis Services cube, which will block other queries, or block processing commits.In this case, scaling out the long-running queries to separate servers can help alleviate contention problems.

References

* [SQL Server 2008 R2 Analysis Services Operations Guide](http://sqlcat.com/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx)
* [Scale-Out Querying for Analysis Services with Read-Only Databases](http://sqlcat.com/whitepapers/archive/2010/06/08/scale-out-querying-for-analysis-services-with-read-only-databases.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2010/06/08/scale-out-querying-for-analysis-services-with-read-only-databases.aspx)
* [Scale-Out Querying with Analysis Services](http://sqlcat.com/whitepapers/archive/2007/12/16/scale-out-querying-with-analysis-services.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/12/16/scale-out-querying-with-analysis-services.aspx)
* [Scale-Out Querying with Analysis Services Using SAN Snapshots](http://sqlcat.com/whitepapers/archive/2007/11/19/scale-out-querying-with-analysis-services-using-san-snapshots.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/11/19/scale-out-querying-with-analysis-services-using-san-snapshots.aspx)

# Tuning Processing Performance

In the following sections we will provide guidance on tuning **processing**. Processing is the operation that loads data from one or more data sources into one or more Analysis Services objects. Although OLAP systems are not generally judged by how fast they process data, processing performance affects how quickly new data is available for querying. Every application has different data refresh requirements, ranging from monthly updates to near real-time data refreshes; however, in all cases, the faster the processing performance, the sooner users can query refreshed data.

Analysis Services provides several processing commands, allowing granular control over the data loading and refresh frequency of cubes.

To manage processing operations, Analysis Services uses centrally controlled jobs. A processing job is a generic unit of work generated by a processing request.

From an architectural perspective, a job can be broken down into parent jobs and child jobs. For a given object, you can have multiple levels of nested jobs depending on where the object is located in the OLAP database hierarchy. The number and type of parent and child jobs depend on these factors:

* The object that you are processing, such as a dimension, cube, measure group, or partition.
* The processing operation that you are requesting, such as **ProcessFull**, **ProcessUpdate**, or **ProcessIndexes**.

For example, when you issue a **ProcessFull** operation for a measure group, a parent job is created for the measure group with child jobs created for each partition. For each partition, a series of child jobs are spawned to carry out the **ProcessFull** operation of the fact data and aggregations. In addition, Analysis Services implements dependencies between jobs. For example, cube jobs are dependent on dimension jobs.

The most significant opportunities to tune performance involve the processing jobs for the core processing objects: dimensions and partitions. Each of these has its own section in this guide.

References

Additional background information on processing can be found in this technical article: [Analysis Services 2005 Processing Architecture](http://msdn.microsoft.com/en-us/library/ms345142(SQL.90).aspx) (http://msdn.microsoft.com/en-us/library/ms345142(SQL.90).aspx).

## Baselining Processing

To quantify the effects of your tuning and diagnose problems, you should first create a baseline. The baseline allows you to analyze root causes and to target optimization effort.

This section describes how to set up a baseline.

### Performance Monitor Trace

Windows performance counters are the bread and butter of performance tuning Analysis Services. Use the free **Performance Monitor** tool (Perfmon) to set up a trace, using these counters:

* **MSOLAP: Processing** 
  + **Rows read/sec**
* **MSOLAP: Proc Aggregations** 
  + **Temp File Bytes Writes/sec**
  + **Rows created/Sec**
  + **Current Partitions**
* **MSOLAP: Threads**
  + **Processing pool idle threads**
  + **Processing pool job queue length**
  + **Processing pool busy threads**
* **MSSQL: Memory Manager**
  + **Total Server Memory**
  + **Target Server Memory**
* **Process** 
  + **Virtual Bytes – msmdsrv.exe**
  + **Working Set – msmdsrv.exe**
  + **Private Bytes – msmdsrv.exe**
  + **% Processor Time – msmdsrv.exe and sqlservr.exe**
* **MSOLAP: Memory**
  + **Quote Blocked**
* **Logical Disk:** 
  + **Avg. Disk sec/Transfer – All Instances**
* **Processor:** 
  + **% Processor Time – Total**
* **System:**
  + **Context Switches / sec**

Configure the trace to save data to a file. Measuring every 15 seconds will be sufficient for tuning processing. As you tune processing, you should re-measure these counters after each change to see whether you are getting closer to your performance goal. Also note the total time used by processing.

The following sections explain how to use and interpret the individual counters.

### Profiler Trace

To optimize the SQL queries that retrieve data used during processing, you should put a trace on the relational database too. If the relational database is SQL Server, you can use free tools such as SQL Server Profiler or Extended Events, or you can use third-party tools. For databases other than SQL Server, consult your database vendor or DBA for help on tuning the database.

In the following section we will assume that you use SQL Server as the relational foundation for Analysis Services. (For users of other databases, the knowledge here will most likely transfer cleanly to your platform.)

In your SQL Server Profiler trace you should capture these events:

* **Performance**/**Showplan XML Statistics Profile**
* **TSQL/SQL:BatchCompleted**

Include these event columns:

* **TextData**
* **Reads**
* **DatabaseName**
* **SPID**
* **Duration**

You can use the **Tuning** template and just add the **Reads** column and **Showplan XML Statistics Profiles**. We suggest that rather than running the tool in real-time, you configure the trace to save for later analysis. You can save to a file, or log to a table. With the latter option the data is available in tabular format, and you can more easily correlate the traces from different tools.

The performance data gathered by these traces will be used in the following section to help you tune processing.

### Determining Where You Spend Processing Time

To properly target the tuning of processing, first determine where the most time is being spent: in partition processing, or in dimension processing.

To make tuning and future monitoring easier, you might want to split the dimension processing and partition processing into two different commands. That way you can tune each step individually.

For partition processing, you should distinguish between **ProcessData** and **ProcessIndex**—the tuning techniques for each are very different. If you follow our recommended best practice of doing **ProcessData** followed by **ProcessIndex** instead of **ProcessFull**, the time spent in each should be easy to read.

If you use **ProcessFull** instead of splitting into **ProcessData** and **ProcessIndex**, you can get an idea of when each phase ends by observing the following performance counters:

* During **ProcessData** the counter **MSOLAP:Processing – Rows read/Sec** is greater than zero.
* During **ProcessIndex** the counter **MSOLAP:Proc Aggregations – Row created/Sec** is greater than zero.

**ProcessData** can be further split into the time spent by the SQL Server process and the time spent by the Analysis Services process. You can use the **Process** counters collected to see where most of the CPU time is spent. The following diagram provides an overview of the many different operations included in a full process of a cube.



Figure : Full cube processing overview

## Tuning Dimension Processing

The performance goal of dimension processing is to refresh dimension data in an efficient manner that does not negatively impact the query performance of dependent partitions. The following techniques for accomplishing this goal are discussed in this section:

* Reducing attribute overhead.
* Optimizing SQL source queries.

To provide a mental model of the workload, we will first introduce the dimension processing architecture.

### Dimension Processing Architecture

During the processing of MOLAP dimensions, jobs are used to extract, index, and persist data in a series of dimension stores.

To create these dimension stores, the storage engine uses the series of jobs displayed in the following diagram.

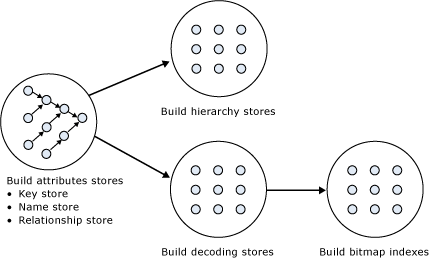


Figure 32: Dimension processing jobs

**Build Attribute Stores -** For each attribute in a dimension, a job is instantiated to extract and persist the attribute members into an attribute store. The attribute store consists of the key store, name store, and relationship store. The data structures created during dimension processing are saved to disk with the following extensions:

* Hierarchy stores: **\*.ostore**, **\*.sstore** and **\*.lstore**
* Key store: **\*.kstore**, **\*.khstore** and **\*.ksstore**
* Name Store: **\*.asstore**, **\*.ahstore** and **\*.hstore**
* Relationship store: **\*.data** and **\*.data.hdr**
* Decoding Stores: **\*.dstore**
* Bitmap indexes: **\*.map** and **\*.map.hdr**

Because the relationship stores contain information about dependent attributes, the processing jobs must be ordered to provide the correct workflow. The storage engine analyzes the dependencies between attributes, and then creates an execution tree with the correct ordering. The execution tree is then used to determine the best parallel execution of the dimension processing.

The following figure displays an example execution tree for a Time dimension. Note that the dimension has been configured using cascading attribute relationships, which is a best practice for all dimension designs.

* The solid arrows represent the attribute relationships in the dimension.
* The dashed arrows represent the implicit relationship of each attribute to the **All** attribute.

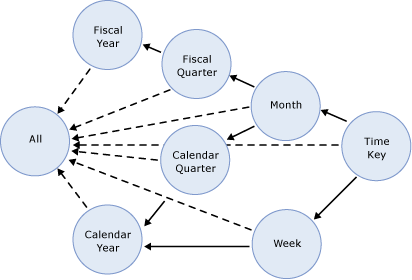


Figure : Execution tree example

In this example, the **All** attribute proceeds first, given that it has no dependencies to another attribute, followed by the **Fiscal Year** and **Calendar Year** attributes, which can be processed in parallel. The other attributes proceed according to the dependencies in the execution tree, with the key attribute always being processed last, because it always has at least one attribute relationship, except when it is the only attribute in the dimension.

The time taken to process an attribute is generally dependent on these factors:

* The number of members
* The number of attribute relationships.

While you cannot control the number of members for a given attribute, you **can** improve processing performance by using cascading attribute relationships. This is especially critical for the key attribute, because it has the most members and all other jobs (hierarchy, decoding, bitmap indexes) are waiting for it to complete.

In general, using attribute relationships lowers the memory requirement during processing. When an attribute is processed, all dependent attributes must be kept in memory. If you have no attribute relationships, all attributes must be kept in memory while the key attribute is processed. This may cause out-of-memory conditions.

**Build Decoding Stores -** Decoding stores are used extensively by the storage engine. During querying, they are used to retrieve data from the dimension. During processing, they are used to build the dimension’s bitmap indexes.

**Build Hierarchy Stores -** A **hierarchy store** is a persistent representation of the tree structure. For each natural hierarchy in the dimension, a job is instantiated to create the hierarchy stores.

**Build Bitmap Indexes -** To efficiently locate attribute data in the relationship store at querying time, the storage engine creates bitmap indexes at processing time. For attributes with a very large number of members, the bitmap indexes can take some time to process. In most scenarios, the bitmap indexes provide significant querying benefits; however, when you have high-cardinality attributes, the querying benefit that the bitmap index provides may not outweigh the processing cost of creating the bitmap index.

### Dimension Processing Commands

When you need to perform a process operation on a dimension, you issue dimension processing commands. Each processing command creates one or more jobs to perform the necessary operations.

From a performance perspective, the following dimension processing commands are the most important:

* **ProcessData**
* **ProcessFull**
* **ProcessUpdate**
* **ProcessAdd**

The **ProcessFull** and **ProcessData** commands discard all storage contents of the dimension and rebuild them. Behind the scenes, **ProcessFull** executes all dimension processing jobs and performs an implicit **ProcessClear** on all dependent partitions. This means that whenever you perform a **ProcessFull** operation of a dimension, you need to perform a **ProcessFull** operation on dependent partitions to bring the cube back online. **ProcessFull** also builds indexes on the dimension data itself (note that indexes on the partitions are built separately). If you do **ProcessData** on a dimension, you should do **ProcessIndexes** subsequently so that dimension queries are able to use these indexes.

Unlike **ProcessFull**, **ProcessUpdate** does not discard the dimension storage contents. Instead, it applies updates intelligently in order to preserve dependent partitions. More specifically, **ProcessUpdate** sends SQL queries to read the entire dimension table and then applies changes to the dimension stores.

**ProcessAdd** optimizes **ProcessUpdate** in scenarios where you only need to insert new members. **ProcessAdd** does not delete or update existing members. The performance benefit of **ProcessAdd** is that you can use a different source table or data source view named query that restrict the rows of the source dimension table to only return the new rows. This eliminates the need to read all of the source data. In addition, **ProcessAdd** also retains all indexes and aggregations (flexible and rigid).

**ProcessUpdate** and **ProcessAdd** have some special behaviors that you should be aware of. These behaviors are discussed in the following sections.

#### ProcessUpdate

A **ProcessUpdate** can handle inserts, updates, and deletions, depending on the type of attribute relationships (rigid versus flexible) in the dimension. Note that **ProcessUpdate** drops invalid aggregations and indexes, requiring you to take action to rebuild the aggregations in order to maintain query performance. However, flexible aggregations are dropped only if a change is detected.

When **ProcessUpdate** runs, it must walk through the partitions that depend on the dimension. For each partition, all indexes and aggregation must be checked to see whether they require updating. On a cube with many partitions, indexes, and aggregates, this can take a very long time. Because this dependency walk is expensive, **ProcessUpdate** is often the most expensive of all processing operations on a well-tuned system, dwarfing even large partition processing commands.

#### ProcessAdd

**ProcessAdd** is the preferred way of managing Type 2 changing dimensions. Because Analysis Services knows that existing indexes do not need to be checked for invalidation, **ProcessAdd** typically runs much faster than **ProcessUpdate**.

In the default configuration of Analysis Services, **ProcessAdd** often triggers a processing error when run, and reports duplicate key values. This error is caused by the “addition” of non-key properties that already exist in the dimension. For example, consider the addition of a new customer to a dimension. If the customer lives in a country that is already present in the dimension, this country cannot be added (it is already there) and Analysis Services throws an error. The solution in this case is to set the <**KeyDuplicate>** to **IgnoreError** on the dimension processing command.

**Notes**

* You cannot run a **ProcessAdd** on an empty dimension. The dimension must first be fully processed.
* Formerly, **ProcessAdd** was available only as an XMLA command; however, in SQL Server 2012, you can configure incremental updates using the **Process Add** option in SQL Server Management Studio.

References

* For detailed information about automating **ProcessAdd,** see Greg Galloway’s blog entry: <http://www.artisconsulting.com/blogs/greggalloway/Lists/Posts/Post.aspx?ID=4>
* For information about how to avoid setting duplicate keys, see this forum thread: <http://social.msdn.microsoft.com/Forums/en-US/sqlanalysisservices/thread/8e7f1304-56a1-467e-9cc6-68428bd92aa6?prof=required>

## Tuning Cube Dimension Processing

In section 2, we described how to create a high-performance dimension design.SQL Server Data Tools and Analysis Management Objects (AMO) provides many warnings to assist you in remembering and following these best practices.

### Reduce Attribute Overhead

When it comes to dimension processing, you must pay a price for having many attributes. If the processing time for the dimension is restrictive, you most likely have to change the attribute to design in order to improve performance.

Every attribute that you include in a dimension impacts the cube size, the dimension size, the aggregation design, and processing performance. Whenever you identify an attribute that will not be used by end users, delete the attribute entirely from your dimension. After you have removed extraneous attributes, you can apply a series of techniques to optimize the processing of remaining attributes.

#### Remove Bitmap Indexes

During processing of the primary key attribute, bitmap indexes are created for every related attribute. Building the bitmap indexes for the primary key can take time if it has one or more related attributes with high cardinality. At query time, the bitmap indexes for these attributes are not useful in speeding up retrieval, because the storage engine still must sift through a large number of distinct values. This may have a negative impact on query response times.

For example, the primary key of the customer dimension uniquely identifies each customer by account number; however, users also want to slice and dice data by the customer’s social security number. Each customer account number has a one-to-one relationship with a customer social security number. You can consider removing the creation of bitmaps for the social security number.

You can also consider removing bitmap indexes from attributes that are always queried together with other attributes that already have bitmap indexes that are highly selective. If the other attributes have sufficient selectivity, adding another bitmap index to filter the segments will not yield a great benefit.

For example, assume you are creating a Sales fact table, and user queries always use the Date and Store dimensions. Sometimes a filter is also applied by using Store Clerk. However, because you have already filtered on Stores, adding a bitmap on Store Clerk might yield only trivial benefit. In this case, you might consider disabling bitmap indexes on the Store Clerk attributes.

You can disable the creation of bitmap indexes for an attribute by setting the **AttributeHierarchyOptimizedState** property to **Not Optimized**.

#### Optimize Attribute Processing Across Multiple Data Sources

When a dimension comes from multiple data sources, using cascading attribute relationships allows the system to segment attributes during processing according to data source. If an attribute’s key, name, and attribute relationships come from the same database, the system can optimize the SQL query for that attribute by querying only one database.

If you do not use cascading attribute relationships, the SQL Server OPENROWSET function, which provides a method for accessing data from multiple sources, is used to merge the data streams. In this situation, the processing for the attribute is extremely slow, because it must access multiple OPENROWSET derived tables.

If you have the option, consider performing ETL to bring all data needed for the dimension into the same SQL Server database. This allows you to utilize the relational engine to tune the query.

### Tuning the Relational Dimension Processing Queries

Unlike fact partitions, which only send one query to the server per partition, dimension process operations send multiple queries. The reason is that dimension tables and fact tables tend to be rather different:

* Dimensions tend to be small, complex tables with very few changes.
* Facts are typically simpler tables, but with many changes.

Tables that have the characteristics of dimensions can often be heavily indexed with little insert/update performance overhead to the system. You can use this to your advantage during processing and make liberal use of relational indexes.

To quickly tune the relational queries used for dimension processing, capture a Profiler trace of the dimension processing and use the Database Engine Tuning Advisor to generate recommendations based on the trace. For small dimension tables, chances are that you can get away with adding every suggested index. For larger tables, target the indexes towards the longest-running queries. For detailed tuning advice on large dimension tables, see the SQL Server 2008 R2 Analysis Services Operations Guide.

#### Using ByTable Processing

You can change howAnalysis Services behaves during dimension processing by setting the **ProcessingGroup** property of the dimension to the value, **ByTable**. When you make this change, instead of sending multiple SELECT DISTINCT queries, the processing task will request the entire table using one query. If you have enough memory to hold all the new dimension data during processing, this option can provide a fast way to optimize processing.

However, you should be careful when using this setting – if Analysis Services runs out of memory during processing, it will have a large impact on both query and processing performance. Experiment with this setting carefully before putting it into production.

Note also that **ByTable** processing will cause duplicate key (KeyDuplicate) errors because SELECT DISTINCT is not executed for each attribute, and the same members will be encountered repeatedly during processing. Therefore, you will need to specify a custom error configuration and disable the KeyDuplicate errors.

## Tuning Partition Processing

The performance goal of partition processing is to refresh fact data and aggregations in an efficient manner that satisfies your overall data refresh requirements.

In this section, we discuss the following techniques for efficient data refresh[[3]](#footnote-3):

* Optimizing SQL source queries.
* Using a partitioning strategy (both in the cube and the relational database).

### Partition Processing Architecture

During partition processing, source data is extracted and stored on disk using the series of jobs displayed In Figure 33.

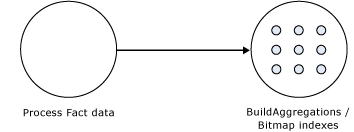


Figure : Partition processing jobs

**Process Fact Data -** Fact data is processed using three concurrent threads that perform the following tasks:

* Send SQL statements to extract data from data sources.
* Look up dimension keys in dimension stores and populate the processing buffer.
* When the processing buffer is full, write out the buffer to disk.

**Build Aggregations and Bitmap Indexes -** Aggregations are built in memory during processing. Although too few aggregations may have little impact on query performance, excessive aggregations can increase processing time without much added value on query performance.

If aggregations do not fit in memory, chunks are written to temp files and merged at the end of the process. Bitmap indexes are also built during this phase and written to disk on a segment-by-segment basis.

### Partition Processing Commands

When you need to perform a process operation on a partition, you issue one of the following partition processing commands, which creates one or more jobs to perform the necessary operations:

* **ProcessFull**

Discards the storage contents of the partition and then rebuilds them. Behind the scenes, **ProcessFull** executes jobs for **ProcessData** and **ProcessIndexes**.

* **ProcessData**

Discards the storage contents of the object and rebuilds only the fact data.

* **ProcessIndexes**

Preserves the data created during **ProcessData** and creates new aggregations and bitmap indexes based on it. **ProcessIndexes** requires a partition to have built its data already.

* **ProcessAdd**

Internally creates a temporary partition, processes it with the target fact data, and then merges it with the existing partition. Previously, this option was available in SQL Server Management Studio as **ProcessIncremental.**

* **ProcessClear**

Removes all data from the partition.Previously,this option was available in Business Intelligence Development Studio as **UnProcess.**

* **ProcessClearIndexes**

Removes all indexes and aggregates from the partition. This brings the partitions in the same state as if **ProcessClear** followed by **ProcessData** had just been run.Note that **ProcessClearIndexes** is the name of the XMLA command. This command is not available in SQL Server Data Tools or SQL Server Management Studio.

### Partition Processing Performance Best Practices

When designing your fact tables, use the guidance in the following technical notes:

* [Top 10 Best Practices for Building a Large Scale Relational Data Warehouse](http://sqlcat.com/top10lists/archive/2008/02/06/top-10-best-practices-for-building-a-large-scale-relational-data-warehouse.aspx) (http://sqlcat.com/sqlcat/b/top10lists/archive/2008/02/06/top-10-best-practices-for-building-a-large-scale-relational-data-warehouse.aspx)
* [Analysis Services Processing Best Practices](http://sqlcat.com/whitepapers/archive/2007/11/15/analysis-services-processing-best-practices.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/11/15/analysis-services-processing-best-practices.aspx)

### Optimizing Data Inserts, Updates, and Deletes

This section provides guidance on how to efficiently refresh partition data to handle inserts, updates, and deletes.

#### Inserts

If you have a browsable, processed cube and you need to add new data to an existing measure group partition, you can apply one of the following techniques:

* **ProcessFull**—Perform a **ProcessFull** operation for the existing partition. During the **ProcessFull** operation, the cube remains available for browsing with the existing data while a separate set of data files are created to contain the new data. When the processing is complete, the new partition data is available for browsing. Note that **ProcessFull** is technically not necessary, given that you are only doing inserts. To optimize processing for insert operations, you can use **ProcessAdd**.
* **ProcessAdd**—Use this operation to append data to the existing partition files. If you frequently perform **ProcessAdd**, we recommend that you periodically perform **ProcessFull** in order to rebuild and recompress the partition data files. The reason for doing so is that, internally, **ProcessAdd** creates a temporary partition and merges it, which can result in data fragmentation over time, hence the need to periodically perform **ProcessFull**.

If your measure group contains multiple partitions, a more effective approach is to create a new partition that contains the new data and then perform **ProcessFull** on that partition. This technique allows you to add new data without affecting the existing partitions. When the new partition has completed processing, it is available for querying.

#### Updates

When you need to perform data updates, you can perform a **ProcessFull**. Ideally you will target the updates to a specific partition, so that you only have to process a single partition.

Rather than directly updating fact data, a better practice is to use a **journaling** **mechanism** to implement data changes. In journaling, you turn an update into an insertion that corrects that existing data. With this approach, you can simply continue to add new data to the partition by using a **ProcessAdd**. By using journaling, you also have an audit trail of the changes that have been made to the fact table.

#### Deletes

For deletions, consider using multiple partitions, to remove expired data.

Consider the following example. You currently have 13 months of data in a measure group, 1 month per partition. You want to remove the oldest month of data from the cube. If you have partitioned the data correctly, you can simply delete the partition without affecting any of the other partitions.

If there are any old dimension members that appeared only in the expired month, you can remove these using a **ProcessUpdate** operation on the dimension, but only if it contains flexible relationships.

If you need to delete members from the key/granularity attribute of a dimension, you must set the dimension’s **UnknownMember** property to **Hidden**. This is because the server does not know if there is a fact record assigned to the deleted member. After this property has been set appropriately, the member will be hidden at query time.

Another option is to remove the data from the underlying table and perform a **ProcessFull** operation. However, this may take longer than **ProcessUpdate**.

As your dimension grows larger, you may want to perform a **ProcessFull** operation on the dimension to completely remove deleted keys. However, if you do this, all related partitions must also be reprocessed. This may require a large batch window and is not viable for all scenarios.

### Picking Efficient Data Types in Fact Tables

During processing, data has to be read from SQL Server and copied into Analysis Services. The wider your rows are, the more bandwidth must be spent copying the rows.

Some data types are, by the nature of their design, faster to use than others. For fastest performance, consider using only these data types in fact tables.

|  |  |
| --- | --- |
| Fact column type | Fastest SQL Server data types |
| Surrogate keys | **tinyint**, **smallint**, **int**, **bigint** |
| Date key | **int** in the format yyyyMMdd |
| Integer measures | **tinyint**, **smallint**, **int**, **bigint** |
| Numeric measures | **smallmoney**, **money**, **real**, **float**  *(*Note that **decimal** and **vardecimal** require more CPU power to process than **money** and **float** types) |
| Distinct count columns | **tinyint**, **smallint**, **int**, **bigint**  *(*If your count column is **char**, consider either hashing or replacing with surrogate key*)* |

### Tuning the Relational Partition Processing Query

During the **ProcessData** phase, rows are read from a relational source and written into Analysis Services. Analysis Services can consume rows at a very high rate during this phase. To achieve these high speeds, you need to tune the relational database to provide a proper throughput.

In the following subsection, we assume that your relational source is SQL Server. If you are using another relational source, some of the advice still applies – consult your database specialist for platform specific guidance.

Analysis Services uses the partition information to generate its queries. Unless you have done any query binding in the UDM, the SELECT statement issues to the relational source is very simple. It consists of:

* A SELECT of the columns required to process. This will be the dimension columns and the measures.
* Optionally, a WHERE criterion if you use partitions. You can control this WHERE criterion by changing the query binding of the partition.

#### Getting Rid of Joins

If you are using a database view or a UDM named query as the basis of partitions, you should seek to eliminate joins in the query sent to the relational database. You can eliminate joins by denormalizing the joined columns to the fact table. If you are using a star schema design, you should already have done this.[[4]](#footnote-4)

#### Getting Relational Partitioning Right

If you use partitioning on the relational side, you should ensure that each cube partition touches at most one relational partition. To check this, use the **XML Showplan** event from your SQL Server Profiler trace.

If you got rid of all joins, your query plan should look something like the following figure.

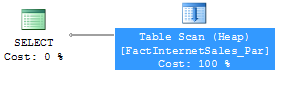


Figure : An optimal partition processing query

Click on the table scan (it may also be a range scan or index seek in your case) and bring up the Properties pane.

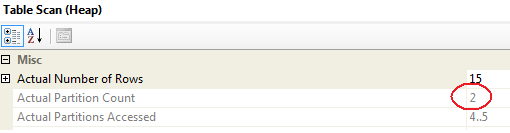


Figure : Too many partitions accessed

In this example, both partition 4 and partition 5 are accessed, and the partition count is 2. In general, you want the value for **Actual Partition Count** to be 1. If this is not the case (as in the example), you should consider repartitioning the relational source data so that each cube partition touches at most one relational partition.

### Splitting Processing Index and Process Data

It is good practice to split partition processing into two phases: **ProcessData** and **ProcessIndex**. This has several advantages.

First, it allows you to restart a failed processing at the last valid state. For example, if you fail processing during **ProcessIndex**, you can restart this phase instead of reverting to running **ProcessData** again.

Second, **ProcessData** and **ProcessIndex** have different performance characteristics. Typically, you want to have more parallel commands executing during **ProcessData** than you want during **ProcessIndex**. By splitting them into two different commands, you can override parallelism on the individual commands.

Of course, if you don’t want to micromanage partition processing, you might opt for running a **ProcessFull** on the measure group. This works well on small cubes where performance is not a concern.

### Increasing Concurrency by Adding More Partitions

If your tuning is bound only by the amount of CPU power you have (as opposed to I/O, for example), you should make the best use of the CPU cores available to you.

1. Take a baseline trace and examine the **Processor:Total** counter.
2. If this counter is not 100%, you are not taking full advantage of your CPU power.

As you continue tuning, keep comparing the baselines to measure improvement, and watch out for bottlenecks to appear again as you push more data through the system.

Using multiple partitions can enhance processing performance. Partitions allow you to work on many, smaller parts of the fact table in parallel. Because a single connection to SQL Server can only transfer a limited amount of rows per second, adding more partitions (and hence more connections) can increase throughput. How many partitions you can process in parallel depends on your CPU and machine architecture. As a rule of thumb, keep increasing parallelism until you no longer see an increase in **MSOLAP:Processing – Rows read/Sec**. You can measure the amount of concurrent partitions you are processing by looking at the performance counter **MSOLAP: Proc Aggregations - Current Partitions**.

Being able to process multiple partitions in parallel is useful in a variety of scenarios; however, there are a few guidelines that you must follow.

* Whenever you process a measure group that has no processed partitions, Analysis Services must initialize the cube structure for that measure group. To do this, it takes an exclusive lock that prevents parallel processing of partitions. You should eliminate this lock before you start the full parallel process on the system. To remove the initialization lock, ensure that you have at least one processed partition per measure group before you begin the parallel operation.
* If you do not have a processed partition, you can perform a **ProcessStructure** on the cube to build its initial structure and then proceed to process measure group partitions in parallel. You will not encounter this limitation if you process partitions in the same client session and use the **MaxParallel** XMLA element to control the level of parallelism.

### Adjusting Maximum Number of Connections

When you increase parallelism of processing above 10 concurrent partitions, you will need to adjust the maximum number of connections that Analysis Services keeps open on the database.

To change this number:

1. Edit the data source
2. In the **Properties** pane, review the **Maximum number of connections** box.
3. Set this number to at least the number of partitions you want to process in parallel.

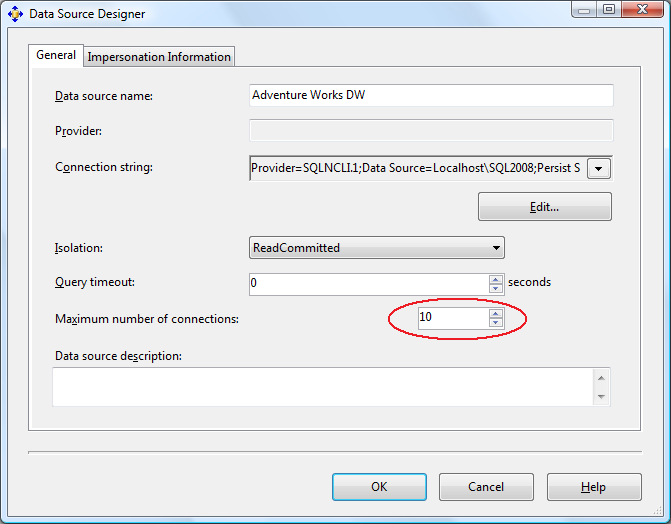


Figure : Adding more database connections

### Tuning the Process Index Phase

During the **ProcessIndex** phase the aggregations in the cube are built. At this point, no more activity happens in the Relational Engine, and if Analysis Services and the Relational Engine are sharing the same box, you can dedicate all your CPU cores to Analysis Services.

The number you want to optimize during **ProcessIndex** is the performance counter **MSOLAP:Proc Aggregations – Row created/Sec.** As the counter increases, the **ProcessIndex** time decreases. You can use this counter to check if your tuning efforts improve the speed.

An additional counter to examine is the temporary files counter. When an aggregation doesn’t fit in memory, the aggregation data is spilled to temporary disk files, making it more expensive to build disk based aggregations. Therefore, if you notice this counter increasing, consider making more memory available for the index building phase. Alternatively, consider dropping some of the larger aggregations, to avoid this issue.

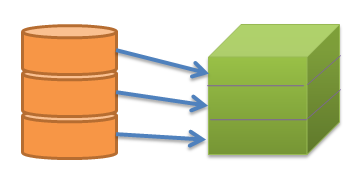
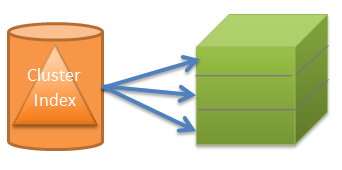
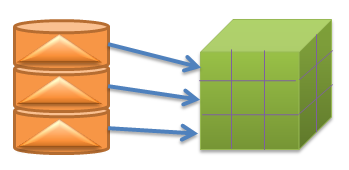
#### Add Partitions to Increase Parallelism

Processing more partitions in parallel can also speed up **ProcessIndex**. Therefore, the same tuning strategy applies to indexing as to processing data:

* Keep increasing partition count until you no longer see an increase in processing speed.

### Partitioning the Relational Source

The best partition strategy to implement in the relational source depends on the capabilities of the database product, but some general guidance applies.

* It is often a good idea to **reflect the cube partition strategy in the relation design**. Partitions in the relational source serve as “coarse indexes,” and matching relational partitions with the cube allows you to get the best possible table scan speeds by touching only the records you need. 
* Another way to achieve that effect is to **use a SQL Server clustered index** (or the equivalent in your preferred database engine) to support fast scan queries during partition processing. 
* If you have used a **matrix partition schema** as described earlier, you may even want to **combine the partition and cluster index strategy**, using partitioning to support one of the partitioned dimension and cluster indexes to support the other. 

# Special Considerations

There are certain features of Analysis Services that provide a lot of business intelligence value, but that require special attention to succeed. This section describes these scenarios and the tuning you can apply when you encounter them.

## Distinct Count

Distinct count measures are architecturally very different from other Analysis Services measures because they are not additive in nature. This means that more data must be kept on disk and in general, most distinct count queries have a heavy impact on the storage engine.

### Partition Design

When distinct count partitions are queried, each partition’s segment jobs must coordinate with each other to avoid counting duplicates. For example, if you count distinct customers based on the customer ID and the same customer ID is in multiple partitions, the jobs must recognize the match so that they do not count the same customer more than once.

If each partition contains a nonoverlapping range of values, this coordination between jobs is avoided and query performance can improve by orders of magnitude, depending on hardware. You can also perform additional optimizations to help improve distinct count performance:

* The key to improving distinct count query performance is to have a partitioning strategy that involves a time period together with your distinct count value. Start by partitioning by time and x number of distinct value partitions of equal size with non-overlapping ranges, where x is the number of cores. Refine x by testing with different partitioning schemes.
* To distribute your distinct value across partitions with non-overlapping ranges, considering building a hash of the distinct value. A modulo function is simple and straightforward; for example, convert character key to integer values. However, it requires extra processing and storage, since you might have to maintain an IDENTITY table. A hash function such as the SQL **HashBytes** function will avoid the latter issues but may introduce **hash key collisions**, when the hash value is repeated based on different source values.
* The distinct count measure must be directly contained in the query. If you partition your cube by the hash of the distinct value, it is important that your query be written to use the hash of the distinct value , versus the distinct value itself. Even if the distinct value and the hash of the distinct value have the same distribution of data, and even if you partition data by the latter, the header files contain only the range of values associated with the hash of the distinct value. This ultimately means that the Analysis Services storage engine must query all of the partitions to perform the distinct on the distinct value.
* The distinct count values need to be continuous.

References

For more information, see [Analysis Services Distinct Count Optimization Using Solid State Devices](http://sqlcat.com/technicalnotes/archive/2010/09/20/analysis-services-distinct-count-optimization-using-solid-state-devices.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2010/09/20/analysis-services-distinct-count-optimization-using-solid-state-devices.aspx).

### Processing of Distinct Count

Distinct count measure groups have special requirements for partitioning. Normally, you will use time and potentially some other dimension as the partitioning column. However, if you partition a distinct count measure group, you should partition on the value of the distinct count measure column instead of a dimension.

To do this, group the distinct count measure column into separate, nonoverlapping intervals. Each interval should contain approximately the same amount of rows from the source. These intervals then form the source of your Analysis Services partitions.

Because the parallelism of the **ProcessData** phase is limited by the amount of partitions you have, for optimal processing performance, you should split the distinct count measure into as many equal-sized nonoverlapping intervals as you have CPU cores on the Analysis Services computer.

It is possible to use noninteger columns for distinct count measure groups. However, for performance reasons, and to avoid hitting the 4-GB limit, you should avoid this.

You should also investigate the possibility of optimizing the relational database for the particular SQL queries that are generated during processing of distinct count partitions. The processing query will send an ORDER BY clause in the SQL, and there may be techniques that you can follow to build indexes in the relational database that will produce better performance for this query pattern.

References

[Analysis Services Distinct Count Optimization](http://sqlcat.com/whitepapers/archive/2008/04/17/analysis-services-distinct-count-optimization.aspx) (<http://sqlcat.com/sqlcat/b/technicalnotes/archive/2010/09/20/analysis-services-distinct-count-optimization-using-solid-state-devices.aspx>)

The white paper describes how you can use hash functions to transform noninteger columns into integers for distinct count. It also provides examples of the nonoverlapping interval-partitioning strategy.

### Distinct Count Partition Aggregation Considerations

Aggregations created for distinct count partitions are different because distinct count values cannot be naturally aggregated at different levels. Analysis Services creates aggregations at the different granularities by including the value that needs to be counted.

In other words, if you think of an aggregation as a GROUP BY on the aggregation granularities, a distinct count aggregation is a GROUP BY on the aggregation granularities and the distinct count column. Having the distinct count column in the aggregation data allows the aggregation to be used when querying a higher granularity—but unfortunately it also makes the aggregations much larger.

To get the most value out of aggregations for distinct count partitions, design aggregations at a commonly viewed higher level attribute related to the distinct count attribute. For example, a report about customers is typically viewed at the Customer Group level; hence, build aggregations at that level. A common approach is run the typical queries against your distinct count partition and use usage-based optimization to build the appropriate aggregations.

### Optimize the Disk Subsystem for Random I/O

Distinct count queries have a huge impact on the Analysis Services storage engine, which means that in large cubes affects the disk subsystem as well. For each such query, Analysis Services generates potentially multiple processes, each one parsing the disk subsystem to perform a portion of the distinct count calculation. This activity results in heavy random I/O on the disk subsystem, which can significantly reduce the query performance of your distinct counts, and all of your Analysis Services queries.

The disk optimization techniques described in the SQL Server 2008 R2 Analysis Services Operations Guide are especially important for distinct count measure groups.

References

* [SQL Server 2008 R2 Analysis Services Operations Guide](http://sqlcat.com/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx)
* [Analysis Services Distinct Count Optimization](http://sqlcat.com/whitepapers/archive/2008/04/17/analysis-services-distinct-count-optimization.aspx) (http://sqlcat.com/sqlcat/b/whitepapers/archive/2008/04/17/analysis-services-distinct-count-optimization.aspx)
* [Analysis Services Distinct Count Optimization Using Solid State Devices](http://sqlcat.com/technicalnotes/archive/2010/09/20/analysis-services-distinct-count-optimization-using-solid-state-devices.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2010/09/20/analysis-services-distinct-count-optimization-using-solid-state-devices.aspx)
* [SQLBI Many-to-Many Project](http://www.sqlbi.com/Projects/Manytomanydimensionalmodeling/tabid/80/Default.aspx) (http://www.sqlbi.com/Projects/Manytomanydimensionalmodeling/tabid/80/Default.aspx)

## Large Many-to-Many Dimensions

Many-to-many relationships are used heavily in many business scenarios ranging from sales to accounting to healthcare. However, you can experience query performance issues and perceived accuracy issues when the solution includes many-to-many relationships.

One way to think about the problem is to regard a many-to-many dimension as a generalization of the distinct count measure. Using a many-to-many dimension lets you apply distinct count logic to other Analysis Services measures such as SUM, COUNT, MAX, MIN, and so on. However, to calculate these values, the Analysis Services storage engine must parse through the lowest level of granularity of data. This is because when a query includes a many-to-many dimension, the query calculation is performed at query-time between the measure group and intermediate measure group at the attribute level. The result is a processor- and memory-intensive process to return the result.

When many-to-many dimensions are used, you might experience the following performance and accuracy issues:

* The join between the measure group and intermediate measure group is a hash join strategy; hence it is very memory-intensive to perform this operation.
* Because queries involving many-to-many dimensions result in a join between the measure group and an intermediate measure group, best performance is achieved by reducing the size of your intermediate measure group. A general rule is less than 1 million rows.
* Many-to-many relationships cannot be aggregated. Therefore, queries involving many-to-many dimensions cannot use aggregations or aggregate caches—only a direct hit will work. There are various MDX calculation issues with VisualTotals, subselects, and CREATE SUBCUBE [[5]](#footnote-5).
* There may be perceived double counting issues because it is difficult to identify which members of the dimension are involved with the many-to-many relationship.

To help improve the performance of many-to-many dimensions, one can make use of the [Many-to-Many matrix compression](http://bidshelper.codeplex.com/wikipage?title=Many-to-Many%20Matrix%20Compression) (http://bidshelper.codeplex.com/wikipage?title=Many-to-Many%20Matrix%20Compression), which removes repeated many-to-many relationships thus reducing the size of your intermediate measure group.

The following figure shows how a **MatrixKey** can be created to eliminate repeated combinations. The MatrixKey is based on combinations of common dimension members.

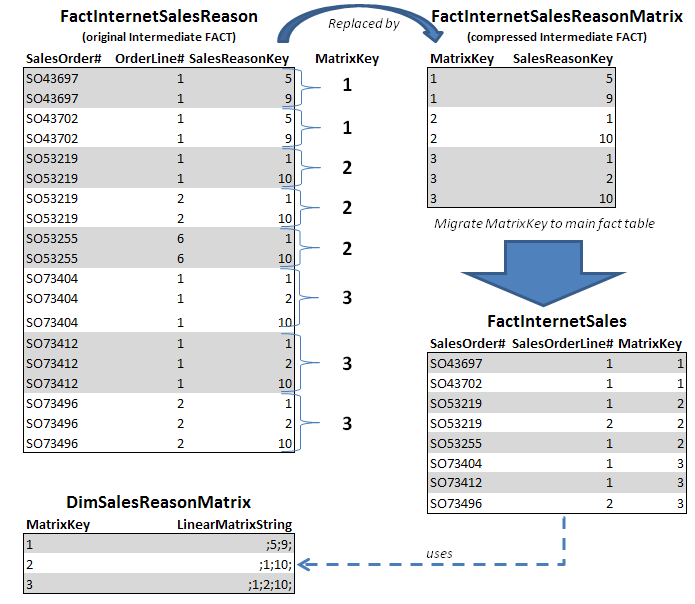


Figure : Compressing the FactInternetSalesReason intermediate fact table

References

* [Many-to-Many Matrix Compression](http://bidshelper.codeplex.com/wikipage?title=Many-to-Many%20Matrix%20Compression) (http://bidshelper.codeplex.com/wikipage?title=Many-to-Many%20Matrix%20Compression)
* [SQLBI Many-to-Many Project](http://www.sqlbi.com/Projects/Manytomanydimensionalmodeling/tabid/80/Default.aspx) (http://www.sqlbi.com/Projects/Manytomanydimensionalmodeling/tabid/80/Default.aspx)
* [Analysis Services: Should you use many-to-many dimensions?](http://sqlcat.com/technicalnotes/archive/2008/02/11/analysis-services-should-you-use-many-to-many-dimensions.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2008/02/11/analysis-services-should-you-use-many-to-many-dimensions.aspx)

## Parent-Child Dimensions

The parent-child dimension is a compact and powerful way to represent hierarchies in a relational database – especially ragged and unbalanced hierarchies. Yet within Analysis Services, query performance on parent-child dimensions tends to be suboptimal. This is especially true for large parent-child dimensions, because aggregations are created only for the key attribute and the top attribute. The top attribute is generally the **All** attribute, unless it has been disabled.

Therefore, a common best practice is to refrain from using parent-child hierarchies that contain a large number of members. You might ask, how big is large? Unfortunately, there isn’t a single answer or specific number, because query performance at intermediate levels of the parent-child hierarchy degrades linearly with the number of members. If you are in a design scenario with a large parent-child hierarchy, consider altering the source schema to reorganize part or all of the hierarchy into a regular hierarchy with a fixed number of levels.

For example, say you have a parent-child hierarchy such as the one shown here.

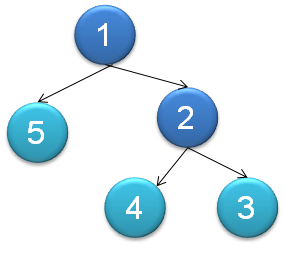


Figure : Sample parent-child hierarchy

The data from this parent-child hierarchy is represented in relational format as in the following table.

|  |  |
| --- | --- |
| SK | Parent\_SK |
| 1 | NULL |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 1 |

Converting this table to a regularly hierarchy results in a relational table with the following format.

|  |  |  |  |
| --- | --- | --- | --- |
| SK | Level0\_SK | Level1\_SK | Level2\_SK |
| 1 | 1 | NULL | NULL |
| 2 | 1 | 2 | NULL |
| 3 | 1 | 2 | 3 |
| 4 | 1 | 2 | 4 |
| 5 | 1 | 5 | NULL |

After the data has been reorganized into the user hierarchy, you can use the **Hide Member If** property of each level to hide the redundant or missing members.

To convert your parent-child hierarchy into a regular hierarchy, refer to the [Analysis Services Parent-Child Dimension Naturalizer](http://pcdimnaturalize.codeplex.com/wikipage?title=Home&version=12&ProjectName=pcdimnaturalize) tool in CodePlex:

<http://pcdimnaturalize.codeplex.com/wikipage?title=Home&version=12&ProjectName=pcdimnaturalize>

Another optimization that you might consider is to limit the total number of parent-child hierarchies in your cube.

References

* [Analysis Services Parent-Child Dimension Naturalizer](http://pcdimnaturalize.codeplex.com/wikipage?title=Home&version=12&ProjectName=pcdimnaturalize) (http://pcdimnaturalize.codeplex.com/wikipage?title=Home&version=12&ProjectName=pcdimnaturalize)
* [Including Child Members Multiple Places in a Parent-Child Hierarchy](http://sqlcat.com/technicalnotes/archive/2008/03/17/including-child-members-multiple-places-in-a-parent-child-hierarchy.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2008/03/17/including-child-members-multiple-places-in-a-parent-child-hierarchy.aspx)

## Near Real Time and ROLAP

As your Analysis Services data becomes more valuable to the business, you’ll be asked to provide near real-time capabilities so that users can have immediate access to their business intelligence system. Providing analytics for cube data in near real-time brings about some special problems:

* Typically the data must reside in memory for low latency access.
* Often, you do not have time to maintain indexes on the data.
* You will typically run into locking and/or concurrency issues that must be dealt with.
* Due to the locking logic invoked by Analysis Services, long-running queries in Analysis Services can prevent processing from committing **and** block other queries.

To provide near real-time results and avoid Analysis Services query locking, start with the relational source:

* Use ROLAP, so that the queries go directly to the relational database.
* Proactively deal with locking and/or concurrency issues in the relational sources.
* Place the real-time portion of the data into its own separate table but keep historical data within your partitioned table. This can minimize the impact of blocking queries within your relational database.

After you have optimized the relational source, go on to apply these techniques, discussed in this section:

* MOLAP switching
* ROLAP + MOLAP
* ROLAP partitioning

### MOLAP Switching

The basic principle behind MOLAP switching is that you create some partitions for historical data and another set of partitions for the latest data. When you do this, the latency associated with frequent process of the current MOLAP partitions is reduced to mere minutes.

This methodology is well suited for something like a time-zone scenario in which you have active partitions throughout the day. For example, say you have active partitions for different regions such as New York, London, Mumbai, and Tokyo. In this scenario, you would create partitions by both time and the specific region. This provides you with the following benefits:

* You can fully process (as often as needed) the active region / time partition (for example, Tokyo / Day 1) without interfering with other partitions (for example, New York / Day 1).
* You can “roll with the daylight” and process New York, London, Mumbai, and Tokyo with minimal overlap.

However, long-running queries for a region can block the processing for that region. For example, a processing commit of current New York data might be blocked by an existing long running query for New York data.

To alleviate this problem, use **cube flipping**, by creating two copies of the same cube. While one cube processes data, the other cube is available for querying.

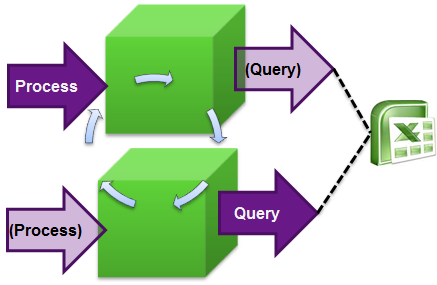


Figure : Cube-flipping concept

To flip between the cubes, you can use the [Analysis Services Load Balancing](http://sqlcat.com/toolbox/archive/2010/02/08/aslb.aspx) Toolkit (<http://sqlcat.com/sqlcat/b/toolbox/archive/2010/02/08/aslb.aspx>) or create your own custom plug-in to your UI that can detect which cube it should query against. It will be important for the plug-in to hold session state so that user queries use the query cache. Session state should automatically refresh when the connection string is changed. Excel, for example, can do this.

### ROLAP + MOLAP

The basic principle behind ROLAP + MOLAP is to create two sets of partitions: a ROLAP partition for frequently updated current data and MOLAP partitions for historical data. In this scenario, you typically can achieve latencies in terms of seconds. If you use this technique, be sure to follow these guidelines:

* Maintain a coherent ROLAP cache. For example, if you query the relational data, the results are placed into the storage engine cache. By default, the next query uses that storage engine cache entry, but the cache entry may not reflect any new changes to the underlying relational database. It is even possible to have aggregate values stored in the data cache that when aggregated up do not add up correctly to the parent.
* Use **Real Time OLAP = true** within the connection string.
* Assume that the MOLAP partitions are write-once / read-sometimes. If you need to make changes to the MOLAP partitions, ensure the changes do not have an impact on users querying the system.
* For the ROLAP partition, ensure that the underlying SQL data source can handle concurrent queries and load. A potential solution is to use [Read Committed Snapshot Isolation](http://msdn.microsoft.com/en-us/library/ms188277.aspx) (http://msdn.microsoft.com/en-us/library/ms188277.aspx ) (RSCI)

References

[Bulk Loading Data into a Table with Concurrent Queries](http://sqlcat.com/technicalnotes/archive/2009/04/06/bulk-loading-data-into-a-table-with-concurrent-queries.aspx) (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2009/04/06/bulk-loading-data-into-a-table-with-concurrent-queries.aspx).

### Comparing MOLAP Switching and ROLAP + MOLAP

The following table compares the benefits of the MOLAP switching and ROLAP + MOLAP methodologies.

|  |  |  |
| --- | --- | --- |
| Component | MOLAP Switching | ROLAP + MOLAP |
| Relational Tuning | Low | Must get right |
| AS locking | Need to handle | Minimal |
| Cache Usage | Good | Poor |
| Relational Concurrency | N/A | RSCI |
| Data Storage | Best Compression | ROLAP sizes typically 2x MOLAP |
| Aggregation Management | SQL Server Profiler + UBO | Manual |
| Latency | Minutes | Seconds |

### ROLAP

In general, MOLAP is the preferred storage choice for Analysis Services; because MOLAP typically provides faster access to the data, especially if your disk subsystem is optimized for random I/O. This design can handle attributes more efficiently and it is easier to manage.

However, ROLAP against SQL Server can be a solid choice for very large cubes with excellent performance, and provides the additional benefit of reducing processing time of large cubes, or eliminating processing entirely. This might be a requirement if you need to implement near real-time cubes.

The following figure shows the query performance of a ROLAP cube after usage-based optimization has been applied. Performance is comparable to MOLAP if the system is expertly tuned.

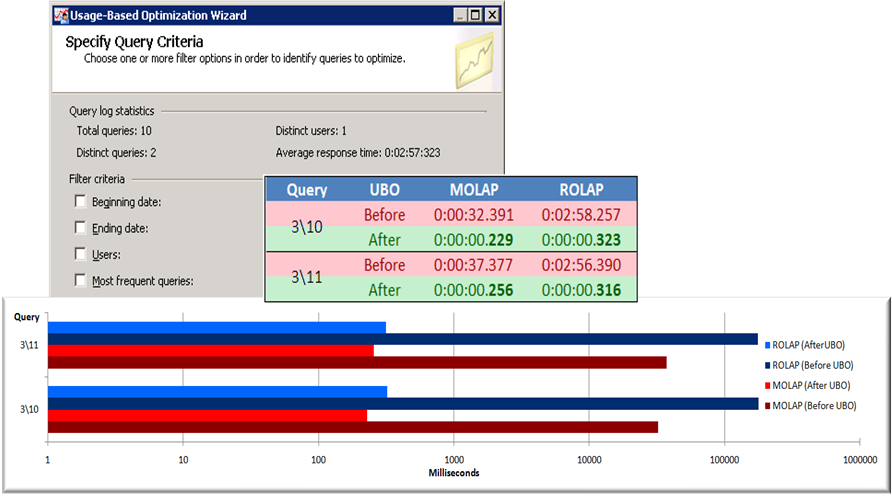


Figure : ROLAP and MOLAP performance with usage-based optimization

#### ROLAP Design Recommendations

Follow these recommendations to improve for high performance querying of ROLAP cubes:

* Simplify the data structure of your underlying SQL data source to minimize page reads. For example, remove unused columns, try to use **int** columns, and so forth.
* Use a star schema without snowflaking, because joins can be expensive.
* Avoid scenarios such as many-to-many dimensions, parent-child dimensions, distinct count, and ROLAP dimensions.

#### ROLAP Aggregation Design Recommendations

When working with ROLAP partitions, you can create aggregations in two ways:

* Create cube-based aggregations by using the Analysis Services aggregations tools.
* Create your own transparent aggregations directly against the SQL Server database.

Both approaches rely on the creation of indexed views within SQL Server but offer different advantages and disadvantages. In general:

* Transparent aggregations have greater value in an environment where multiple cubes are referencing the same fact table.
* Transparent aggregations and cube-based aggregations could be used together to get the most efficient design:
  + Start with a set of transparent aggregations that will work for the most commonly run queries.
  + Add cube-based aggregations using usage-based optimization for important queries that are taking a long time to run.

To design the most effective strategy, you might consider a combination of these two approaches, which have their respective advantages and disadvantages:

Table . Cube-based aggregations

|  |  |
| --- | --- |
| Cube-based aggregations | |
| Advantages | **Efficient query processing**: Analysis Services can use cube-based aggregations even if the query and aggregation granularities do not exactly match.  For example, a query on [Month] can use an aggregation on [Day], which requires only the summarization of up to 31 numbers. |
| **Aggregation design efficiency:** Analysis Services includes the Aggregation Design Wizard and the Usage-Based Optimization Wizard to create aggregation designs based on storage and percentage constraints or queries submitted by client applications. |
| Disadvantages | **Processing overhead**: Analysis Services drops and re-creates indexed views associated with cube-based aggregations during cube partition processing. Dropping and re-creating the indexes can take an excessive amount of time in a large-scale data warehouse. |

Table . Transparent aggregations

|  |  |
| --- | --- |
| Transparent aggregation using the relational source | |
| Advantages | **Reuse of existing indexes across cubes:** While aggregate views can also be created by queries that do not know of their existence, the issue is that Analysis Services may unexpectedly drop the indexed views |
| **Less overhead during cube processing**: Analysis Services is unaware of the aggregations and does not drop the indexed views during partition processing. There is no need to drop indexed views because the relational engine maintains the indexes continuously, such as during INSERT, UPDATE, and DELETE operations against the base tables. |
| Disadvantages | **No sophisticated aggregation algorithms**: Indexed views must match query granularity. The query optimizer doesn’t consider dimension hierarchies or aggregation granularities in the query execution plan.  For example, an SQL query with GROUP BY on [Month] can’t use an index on [Day]. |
| **Maintenance overhead**: Database administrators must maintain aggregations by using SQL Server Management Studio or other tools. It is difficult to keep track of the relationships between indexed views and ROLAP cubes. |
| **Design complexity**: Database Engine Tuning Advisor can help to facilitate aggregation design tasks by analyzing SQL Server Profiler traces, but it can’t identify all possible candidates. Moreover, data warehouse (DW) architects must manually study SQL Server Profiler traces to determine effective aggregations. |

#### Limitations of ROLAP Aggregations

While ROLAP is very powerful, there are some strict limitations that must be first considered before using this approach:

1. You may have to design using **table binding** (not **query binding**) to an actual table instead of a partition.

* The goal of this design is to ensure partition elimination.
* Advice on ROLAP aggregations is specific to SQL Server as a data source. For other data sources, carefully evaluate the behavior of ROLAP queries when accessing a partitioned table.
* It is not possible to create an indexed view on a view containing a subselect statement. This will prevent Analysis Services from creating index view aggregations.

1. Relational partition elimination will generally not work.

* Normally, data warehousing best practice is to use partitioned fact tables. However, if you need to use ROLAP aggregations, you must use separate tables in the relational database for each cube partition.
* Partitions require named queries, and those tend to generate bad SQL plans. This may vary depending on the relational engine you use.

1. You cannot use some features.

* You cannot use named queries or views in the DSV.
* Any feature that will cause Analysis Services to generate a subquery cannot be used for ROLAP aggregations. For example, you cannot use a Count of Rows measure, because a subquery is always generated when this type of measure is used.

1. There are limitations on measure groups.

* You cannot aggregate any measures that uses MAX or MIN.
* You cannot use measures that are based on nullable fields in the relational data source.

References

For more information about how to optimize your ROLAP design, see the following white paper:

[Analysis Services ROLAP for SQL Server Data Warehouses](http://sqlcat.com/whitepapers/archive/2010/08/23/analysis-services-rolap-for-sql-server-data-warehouses.aspx) (<http://sqlcat.com/sqlcat/b/whitepapers/archive/2010/08/23/analysis-services-rolap-for-sql-server-data-warehouses.aspx>).

<http://blogs.msdn.com/b/sqlcat/archive/2013/11/05/forcing-numa-node-affinity-for-analysis-services-tabular-databases.aspx>

## NUMA

Non Uniform Memory Access (NUMA) is a computer memory design, applied at the chip level by the manufacturer. NUMA architectures address the problem of multiple processors needing to use the same memory by providing separate memory for each processor.

In general, the point of NUMA architectures is to alleviate performance bottlenecks associated with lots of logical processors actively accessing a shared memory system. A shared physical memory bus can suffer from contention, which stalls memory access operations. NUMA solves this problem in part by providing more independence for each group of logical processors. However, multi-processor systems without NUMA can make the problem of memory bottlenecks worse.

In NUMA architectures, applications that are **NUMA-aware** implement optimizations to take advantage of the locality of memory. Without those optimizations, the applications can behave inconsistently due to slowdowns when accessing memory that is not local to the NUMA nodes on which the threads are executing.

The SQL Server database engine became NUMA-aware in 2005. To get the best performance in a NUMA environment, be sure to apply the latest hotfixes.

**Note**: Microsoft Windows 7, Windows Server 2008 R2, and Windows Server 2012 all support NUMA architecture over up to 64 logical cores. Significant improvements were made in Windows Server 2012 to support NUMA integration with Hyper-V (http://technet.microsoft.com/en-us/library/hh831410.aspx).

### NUMA Optimizations in SSAS

Analysis Services has implemented various NUMA optimizations to take advantage of NUMA locality, and in each release continues to add more optimizations. For example, optimizations in SQL Server 2012 and in 2012 SP1 allow you to optimize the way that memory is allocated among NUMA nodes in systems that have many NUMA nodes.

### General NUMA Tips

In general, we recommend that SSAS developers or administrators who are considering tuning for NUMA architecture review their overall optimization strategies before using the advanced techniques described in this section.

* For MOLAP solutions, memory bus bottlenecks occur only when there are a large number of active logical processors. Typically if you have less than 8 cores, you will not observe major bottlenecks and you can conclude that NUMA is not the source of performance problems.
* To estimate the overhead associated with referencing memory on a separate NUMA node, you can use the tool CoreInfo, from SysInternals. The metric **Approximate Cross-NUMA Node Access Cost** from this tool indicates how much performance might be affected when memory is accessed across NUMA nodes. However, the SQL CAT team found that the calculations from this metric are relative and can change on subsequent executions on the same machine. We recommend that you conduct multiple runs and compare performance on different models and at different times.
* NUMA characteristics have the greatest effect on serial execution of individual queries. Therefore concurrency of queries is somewhat orthogonal to the performance of a NUMA system. However, if you have not performed these optimizations, be sure to do so before working with NUMA settings:
  + Examine the degree of concurrency on your server. If lots of user queries are being performed concurrently, it is highly recommended that you experiment with the [multi-user configuration settings](http://support.microsoft.com/kb/2135031) (http://support.microsoft.com/kb/2135031). This configuration might slow down individual queries, but it should give you a better balance of responsiveness among the different concurrent queries; that is, no single long-running query should prevent other shorter queries from executing.
  + To manage query blocking issues, use techniques described previously, in the [2008 R2 Operations Guide](http://msdn.microsoft.com/en-us/library/hh226085.aspx) (http://msdn.microsoft.com/en-us/library/hh226085.aspx).
* Hyper-V provides settings that make it relatively easy to affinitize SSAS models to a specific NUMA node within virtual machines.
* If you remove all of the CPUs in a NUMA node, it becomes more difficult to use the RAM in that node. Memory can still be allocated from that physical node’s memory, but Windows will avoid using it and will only give pages from those nodes when the affinitized nodes are out of physical pages.
* There is physical memory local to each NUMA node, and there are on-chip caches (L1/L2/L3) associated with each physical CPU. You can physically adjust the amount of memory chips associated with a NUMA node if you want, but you can’t adjust the amount of caches on each CPU.

### Specific NUMA Configurations for SSAS

As part of the SQL Server 2012 and 2012 SP1 releases, the following changes were made to Analysis Services to optimize NUMA in many-node machines.

#### Eliminate cross NUMA-node file access by using separate thread pools

A new thread pool, **IOProcess**, was added to support separation of reads from other activities. The new **IOProcess** thread pool handles read jobs. By separating out the segment scan operations into a separate thread pool, it is possible to make the scans localized to a NUMA node and therefore improve the memory access performance of the read operations. The processing threads are not specifically NUMA aware.

On machines with 4 or more NUMA nodes, the **IOProcess** thread pool is actually a collection of thread pools, with each NUMA node having its own pool of IOProcess threads. These can be allocated to different cores or different processors as described below.

To ensure that file IO operations are consistently assigned to the same **IOProcess** thread pool, an algorithm was implemented that spreads partition reads across all IO thread pools. Currently the algorithm bases the distribution on the ordinal position of the partition in the partitions collection for a measure group. This means that whenever a partition is scanned, it will use threads only from a single NUMA node. Even if other cores are available on other NUMA nodes, those cores will not be used. In a typical environment, multiple partitions will be queried and therefore the load will be evenly distributed across all the thread pools and NUMA nodes.

(This design is subject to change in future, but it is important to understand its effect in existing implementations.)

Dimension read jobs are always assigned to the **IOProcess** thread pool for NUMA node 0. Therefore, NUMA node 0 will typically be assigned a larger percentage of work. However, since most dimension operations operate on cached data, this additional load should have no noticeable impact. The **Process** thread pool will continue to handle ROLAP and processing related jobs, including writing new files.

Note, however, that processing of aggregations and indexes will use the IO thread pool to scan the partition data. Testing has shown that use of this new thread pool improves the performance of that stage of processing.

If you wish to control the use of **IOProcess** thread pools in a NUMA-aware environment, there are three basic approaches:

1. Affinitize each thread pool to all processors from a NUMA node.
2. Affinitize thread pools per core[[6]](#footnote-6).
3. Allow the threads to run on any NUMA node.

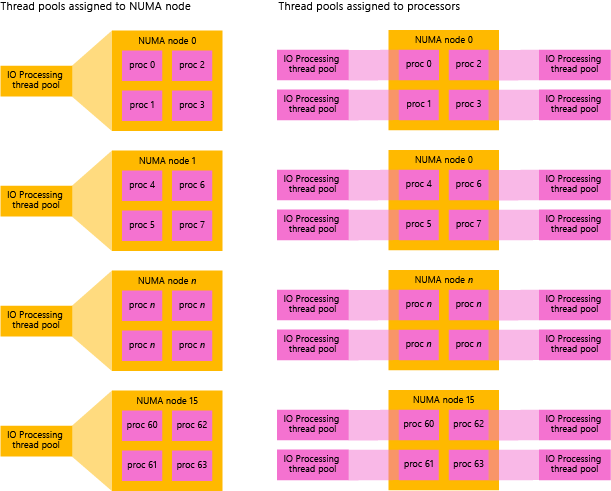


Figure . Comparing methods 1 and 2 for NUMA affinitization

In general, performance is optimized when each IO thread is on its own core. In general, performance can be improved when the active threads are not blocked on shared data structures. One of the most important bottlenecks for high end NUMA systems is the queue of segment scan jobs that need to be executed concurrently. The act of inserting and removing these jobs was found to be a severe bottleneck in the Process thread pool. By separating out the segment scan jobs into their own thread pool and by then increasing the number of **IOProcess** thread pools to be based on either the number of NUMA nodes or the number of cores, we can reduce one of the important bottlenecks in the system because each thread pool has its own queue of jobs.

However, one small issue that you should keep in mind is that the **IOProcess** threads do not register themselves as cancellable objects, because the expectation is that these threads will be performing only short duration read operations on files. As a result, if you cancel a query that requested an IO operation, any jobs on the IOProcess thread might continue to run for a short period of time after the query was cancelled, but new IO jobs would not be created for the canceled query.

#### Modify the PerNumaNode setting

As with other SSAS features, the default behavior of the **IOProcess** thread pool is intended to cover the most common scenarios. By default, there will be only one **IOProcess** thread pool if there are less than four NUMA nodes available. When there are four or more NUMA nodes, the default is to create one **IOProcess** thread pool per NUMA node. The threads in each thread pool will be affinitized to the cores in the corresponding NUMA nodes.

To give you more control over the NUMA thread pools, modify the Analysis Services configuration file (msmdsrv.ini), and change the value of the **PerNumaNode** setting (ThreadPool\IOProcess). The default value of this setting is ***-1***, which indicates that the server should use the default 4 NUMA node threshold.

* Change this value to ***0*** to disable the per NUMA node thread pool behavior. In effect, this setting reverts the server to use of a single **IOProcess** thread pool.
* Change this value to ***1*** to create one **IOProcess** thread pool per NUMA node. This setting overrides the default behavior where servers with less than 4 NUMA nodes would only have one **IOProcess** thread pool.
* Change the value to ***2*** to instruct the engine to create one **IOProcess** thread pool per core.

#### Use the GroupAffinity mask to optimize resource usage

Analysis Services now supports more than 64 CPUs, by using **processor groups**. A processor group in Windows can contain a maximum of 64 CPUs. Systems with more than 64 CPUs will contain multiple processor groups.

To support multiple processor groups, the following changes were made in SSAS 2012:

* The AS engine was modified to understand processor groups. By default, threads in the thread pools might be able to use all available logical processors automatically – including those from different processor groups.
* A new configuration property, **GroupAffinity** (in the section, **<ThreadPool> <Process>**), was added for each thread pool in the server. This property lets the SSAS administrator control which CPUs are used for each thread pool.
* For diagnostic purposes, the msmdsrv.log file in SSAS 2012 and 2014 contains the following entries at service start that reflect the size of each of the five thread pools:
  + Query
  + ParsingShort
  + ParsingLong
  + Processing
  + IOProcessing
* The msmdsrv.log file outputs the affinity information for each NUMA node.

Although the **GroupAffinity** setting was added to support affinitizing thread pools, this property can also be used to control the CPUs that are used for specific operations. That is, by defining a **GroupAffinity** mask, administrators can allocate threads for IO, processing, query, or parsing to specific CPUs. This optimization can improve resource usage, and better enable resource sharing across multiple processes on the same server.

To use the **GroupAffinity** setting, you must define a bitmask that specifies affinity for each processor in a processor group as follows:

* The **GroupAffinity** property can have as many comma-separated hex values as there are defined CPU groups on a server.
* If the mask contains fewer bits than the number of CPUs for the processor group, then non-specified bits are set to zero.

For example, the following entry in the msmdsrv.ini file would affinitize threads to 16 logical processors in the first two processor groups on the server:

<GroupAffinity>0xFFFF,0xFFFF</GroupAffinity>

In contrast, the following entry would affinitize threads to CPUs 4-7 in the first processor group, and the first 32 CPUs in the second processor group:

<GroupAffinity>0x00F0,0xFFFFFFFF</GroupAffinity>

If no **GroupAffinity** value is specified for a thread pool (default) then that thread pool is allowed to spread work across available processor groups and CPUs.

**Note**: Although VertiPaq can use more than 64 CPUs, setting the **GroupAffinity** property is currently ***not*** supported for the VertiPaq thread pool, even though an entry exists in the msmdsrv.ini file.

For additional examples of how to use this property, see [this topic in Books Online](http://msdn.microsoft.com/library/ms175657.aspx) (http://msdn.microsoft.com/library/ms175657.aspx).

**Note:** Support for processor groups was added around the same time as enhancements specific to NUMA, but the two are not directly related or dependent on each other.

#### Turn off SSAS resource monitoring

If your business needs allow it, you may also choose to lower the overhead of the system by turning off the Resource Monitoring feature. You can do this by modifying the configuration file and changing the value of **ResourceMonitoringEnabled** to ***0***.

#### Enable multiuser settings

When multiple users access SQL Server Analysis Services concurrently, expensive queries – such as those that scan many partitions – can monopolize the system and block other users. To force these longer-running queries to cede priority to small queries and achieve high concurrency, you can alter the following settings in msmdsrv.ini:

* Set **CoordinatorQueryBalancingFactor** to ***1***
* Set **CoordinatorQueryBoostPriorityLevel** to ***0***

If you are unsure of the appropriate values for these properties, we recommend that you run the Best Practice Analyzer, which includes a rule that checks the current values for both properties.

If you get the message "Server not configured for optimal concurrent query throughput", you can edit the msmdsrv.ini configuration file and change **CoordinatorQueryBalancingFactor** and **CoordinatorQueryBoostPriorityLevel** to use the values recommended by the BPA tool.

These settings are described in more detail in the 2008 Operations Guide.

#### Modify Windows file cache behavior

Another tuning option that was added in SSAS 2012 SP1 was the ability to specify that files should be scanned in Random mode (see FILE\_FLAG\_RANDOM\_ACCESS). There are known negative consequences of using this approach (see <http://support.microsoft.com/kb/2549369>) but this option has been demonstrated to improve performance of the Windows file system cache on high end hardware with lots of memory.

To modify this setting, edit the msmdsrv.ini file, set the **RandomFileAccessMode** property to a value of **1**, and optionally restart the service. Changes to this server property do not *require* a service restart to take effect, but if the server is not restarted Analysis Services will not release open files or change the way it accesses open files, and the setting will affect only newly opened or created files.

**Note**: You should make this change only if the computer has sufficient memory. Using **Random** mode will cause some pages to stay in memory longer, which might cause new bottlenecks.

References

* Forcing NUMA Node Affinity for tabular models databases (John Sirmon, SQLCAT) <http://blogs.msdn.com/b/sqlcat/archive/2013/11/05/forcing-numa-node-affinity-for-analysis-services-tabular-databases.aspx>
* Analysis Services thread pool changes in SQL Server 2012 (Wayne Robertson, CSS SQL Escalation Services <http://blogs.msdn.com/b/psssql/archive/2012/01/31/analysis-services-thread-pool-changes-in-sql-server-2012.aspx>
* The impact of NUMA on SQL Server workloads (Linchi Shea) <http://sqlblog.com/blogs/linchi_shea/archive/2012/01/30/performance-impact-the-cost-of-numa-remote-memory-access.aspx>

For general information about NUMA and Hyper-V, we recommend these articles:

* Processor Groups (MSDN) <http://msdn.microsoft.com/library/dd405503%28VS.85%29.aspx>
* NUMA Properties <http://msdn.microsoft.com/library/windows/desktop/aa363804(v=vs.85).aspx>
* A detailed look at NUMA and processor groups <http://blogs.msdn.com/b/saponsqlserver/archive/2010/09/28/windows-2008-r2-groups-processors-sockets-cores-threads-numa-nodes-what-is-all-this.aspx>
* Hyper-V in Windows Server 2012 and NUMA <http://blogs.technet.com/b/windowsserver/archive/2012/04/05/windows-server-8-beta-hyper-v-amp-scale-up-virtual-machines-part-1.aspx>

<http://blogs.technet.com/b/windowsserver/archive/2012/04/06/windows-server-8-beta-hyper-v-amp-scale-up-virtual-machines-part-2.aspx>

### Addenda: NUMA with Tabular Models

The focus of this section was on NUMA optimizations and their effect on MOLAP models. For **tabular models** running in memory, you should conduct tests to understand the effects of NUMA when running a large production query workload. If you do not account for NUMA you run the risk of incurring performance costs if the model references memory running on a remote NUMA node.

For more information about NUMA issues and working with tabular models, see the [2014 white paper by Alberto Ferrari](http://go.microsoft.com/fwlink/?LinkId=398938) (http://go.microsoft.com/fwlink/?LinkId=398938).

# Conclusion

This document provides recommendations for diagnosing and resolving processing and query performance issues in SQL Server 2012 and SQL Server 2014 Analysis Services. Based on the workload, your performance gains might be different. We recommend that you do performance testing with an appropriate number of users to determine the appropriate tuning steps.

Additionally, in your testing, you might consider using solid state drives (SSD) and assess the benefits of performance in conjunction with specific workloads. Some customers have reported benefits such as improved read access times with SSDs.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:Analysis%20Services%20MOLAP%20Performance%20Guide%20for%20SQL%20Server%202012%20and%202014)

# Resources

For more information, see:

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server Tech Center

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server Dev Center

1. For an example, see http://geekswithblogs.net/darrengosbell/archive/2007/04/24/SSAS-Processing-ForceCommitTimeout-and-quotthe-operation-has-been-cancelledquot.aspx. [↑](#footnote-ref-1)
2. For more information about optimizing MDX, see [Optimizing MDX](#_Optimizing_MDX_1). [↑](#footnote-ref-2)
3. For detailed guidance on server tuning, hardware optimization and relational indexing, see the SQL Server 2008 R2 Operations Guide. [↑](#footnote-ref-3)
4. For background on relational star schemas and how to design and denormalize for optimal performance, refer to: Ralph Kimball, *The Data Warehouse Toolkit.* [↑](#footnote-ref-4)
5. You can find more information in the [Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques](http://www.microsoft.com/downloads/details.aspx?FamilyID=3494E712-C90B-4A4E-AD45-01009C15C665&displaylang=en) white paper. [↑](#footnote-ref-5)
6. With this option, separate thread pools are instantiated per core. The threads in each thread pool are affinitized to *all* cores in that NUMA node but one core serves as the *ideal* processor. For more information, see [SetThreadIdealProcessor](http://msdn.microsoft.com/library/windows/desktop/ms686253(v=vs.85).aspx) (<http://msdn.microsoft.com/library/windows/desktop/ms686253(v=vs.85).aspx)> in the Windows API documentation. [↑](#footnote-ref-6)