Analysis Services Performance Guide

SQL Server White Paper

Writers: Thomas Kejser and Denny Lee

Editor: Beth Inghram

Contributors and Technical Reviewers:

Richard Tkachuk
T.K. Anand
Marius Dumitru
Greg Galloway
Siva Harinath
Edward Melomed
Akshai Mirchandani
Carl Rabeler
Elizabeth Vitt
Sedat Yogurtcuoglu
Anne Zorner
Sanjay Nayyar (IM-Group)
Greg Galloway (Artis Consulting)
Tomislav Piasevoli
Christopher Webb (Crossjoin Consulting)
Marco Russo (SQLBI)

Published: October 2011

Applies to: SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2

Summary: This white paper describes how business intelligence developers can apply query and processing performance-tuning techniques to their Microsoft SQL Server 2008 R2 Analysis Services OLAP solutions.

Copyright

Contents

1 Introduction

This guide contains information about building and tuning Analysis Services in SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2 cubes for the best possible performance. It 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 with techniques and design patterns that will help you achieve the best possible performance of even large cubes.

Cube performance can be divided into two types of workload: query performance and processing performance. Because these workloads are very different, this paper is organized into four main sections.

Design Patterns for Scalable Cubes – No amount of query tuning and optimization can beat 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 impacts 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 more careful attention to the cube design than others. At the end of the paper you will find a section that describes the special techniques you should apply when using these features.

2 Design Patterns for Scalable Cubes

Cubes present a unique challenge to the BI developer: they are ad-hoc databases that are expected to respond to most queries in short time. The freedom of the end user is limited only by the data model you implement. 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 well to value adding reporting – and a detailed treatment of optimal, industry specific data model is outside the scope of this document. However, there are a lot of common design patterns you can apply across all industries - this section deals with these patterns and how you can leverage them for increased scalability in your cube design.

2.1 Building Optimal Dimensions

A well-tuned dimension design is one of the most critical success factors of a high-performing Analysis Services solution. The dimensions of the cube are the first stop for 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 building optimized dimensions and properly using both attributes and hierarchies.

2.1.1 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 carry further information about the attribute – 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.

Using both **ValueColumn** and **NameColumn** to carry information eliminates the need for extraneous attributes. This reduces the total number of attributes in your design, making it more efficient.

It is a best practice to assign a numeric source field, if available, to the **KeyColumns** property rather than a string property. Furthermore, use a single column key instead of a composite, multi-column key. Not only do these practices this reduce processing time, they also reduce the size of the dimension and the likelihood of user errors. This is especially true for attributes that have a large number of members, that is, greater than one million members.

2.1.2 Hiding Attribute Hierarchies

For many dimensions, you will want the user to navigate hierarchies created for ease of access. For example, a customer dimension could 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 are 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 use user hierarchies instead.

2.1.2.1 Hiding the Surrogate Key

It is often a good idea to hide the surrogate key attribute in the dimension. If you expose the surrogate key to the client tools as a **ValueColumn**, those tools may refer to the key values in reports. The surrogate key in a Kimball star schema design holds no business information, and may even change if you remodel type2 history. After you create a dependency to the key in the client tools, you cannot change the key without breaking reports. Because of this, you don't want end-user reports referring to the surrogate key directly – and this is why we recommend hiding it.

The best design for a surrogate key is to hide it from users in the dimension design by setting the **AttributeHierarchyVisible** = **false** and by not including the attribute in any user hierarchies. This prevents end-user tools from referencing the surrogate key, leaving you free to change the key value if requirements change.

2.1.3 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. You should explicitly set the **OrderBy** or **OrderByAttribute** property of the attribute to explicitly control this ordering. Typically, this ordering is by attribute name or key, but it may also be 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 few cases where you don't care about the ordering of an attribute, yet the surrogate key is one such case. For such hidden attribute that you used only for implementation purposes, you can set **AttributeHierarchyOrdered** = **false** to save time during processing of the dimension.

2.1.4 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. But 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 may set a default date in the Adventure Works cube like this.

```
ALTER CUBE [Adventure Works]UPDATE
DIMENSION [Date], DEFAULT_MEMBER='[Date].[Date].&[2000]'
```

However, default members may cause issues in the client tool. For example, Microsoft Excel 2010 will not provide a visual indication that a default member is currently selected and hence implicitly influence the query result. This may confuse users who expect the **All** level to be the current member when no other members are implied by the query. Also, if you set a default member in a dimension with multiple hierarchies, you will typically get results that are hard for users to interpret.

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

2.1.5 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 is does not make sense to query at the **All** level. For example, you may have a currency dimension in the cube – and asking for "the sum of all currencies" is a meaningless question. It can even be expensive to ask for the **All** level of dimension if there is not 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.

2.1.6 Identifying Attribute Relationships

Attribute relationships 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 many 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 might or might not be manifested in the original dimension table that can be used by the Analysis Services engine to optimize performance. By default, all attributes are related to the key, and the attribute relationship diagram represents a "bush" where relationships all stem from the key attribute and end at each other's attribute.

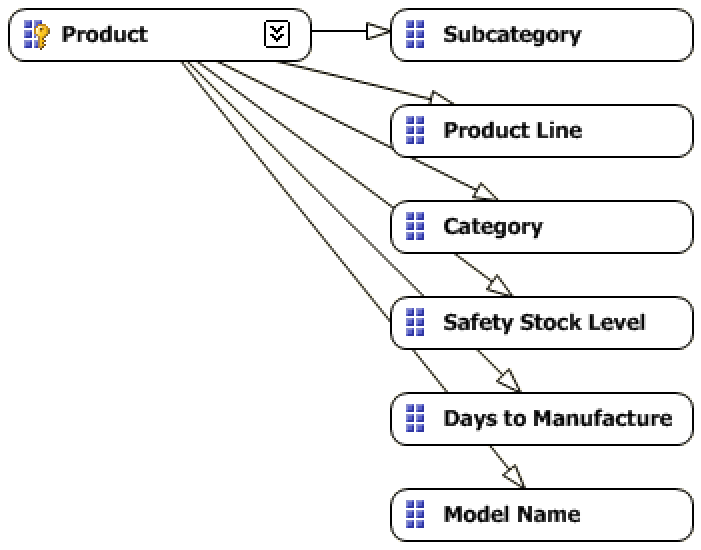


Figure : Bushy attribute relationships

You can optimize performance by defining hierarchical relationships supported by the data. In this case, a model name identifies the product line and subcategory, and the subcategory identifies a category. In other words, a single subcategory is not found in more than one category. If you redefine the relationships in the attribute relationship editor, the relationships are clearer.

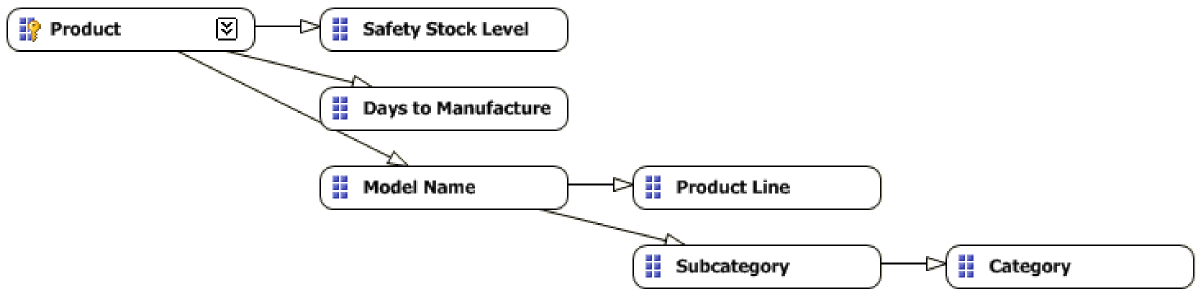


Figure : Redefined attribute relationships

Attribute relationships 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 for queries.
- Auto-Exist can more efficiently eliminate attribute combinations that do not exist in the data.

Consider the cross-product between **Subcategory** and **Category** in the two figures. In the first, where no attribute relationships have been explicitly defined, the engine must first find which products are in each subcategory and then determine which categories each of these products belongs to. For large dimensions, this can take a long time. If the attribute relationship is defined, the Analysis Services engine knows beforehand which category each subcategory belongs to via indexes built at process time.

2.1.6.1 Flexible vs. Rigid Relationships

When an attribute relationship is defined, the relation can either be flexible or rigid. A flexible attribute relationship is one where members can move around during dimension updates, and a rigid attribute relationship is one where 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. However, the relationship between customer and city may be flexible as customers move.

When a change is detected during process in a flexible relationship, all indexes for partitions referencing the affected dimension (including the indexes for attribute that are not affected) must be invalidated. This is an expensive operation and may cause **Process Update** operations to take a very long time. Indexes 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.

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

2.1.7 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 from the bottom of the hierarchy to the top of the hierarchy.

In an unnatural hierarchy, 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.

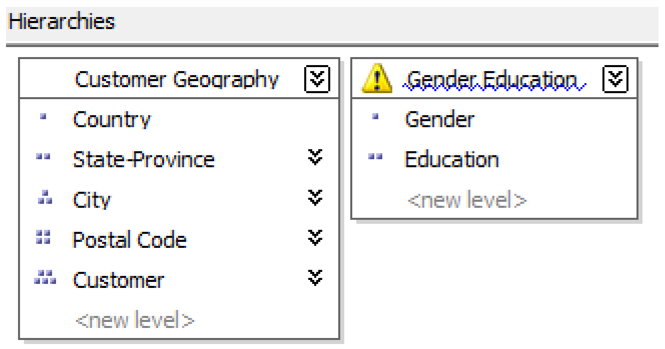


Figure : Natural and unnatural hierarchies

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.

2.1.8 Turning Off the Attribute Hierarchy

Member properties provide a different mechanism to expose dimension information. For a given attribute, member properties are automatically created for every direct attribute relationship. For the primary key attribute, this means that every attribute that is directly related to the primary key is available as a member property of the primary key attribute.

If you only want to access an attribute as member property, after you verify that the correct relationship is in place, 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. High-cardinality attributes such as phone numbers and addresses 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 both the querying and processing 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 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.

2.1.9 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. This may result in reading too much data 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 be fast ad-hoc structures, and putting the performance burden on the end user is rarely a good idea.

Analysis Services has the ability to materialize the references 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. However, be careful with materialized reference dimension – 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.

2.1.10 Fast-Changing Attributes

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

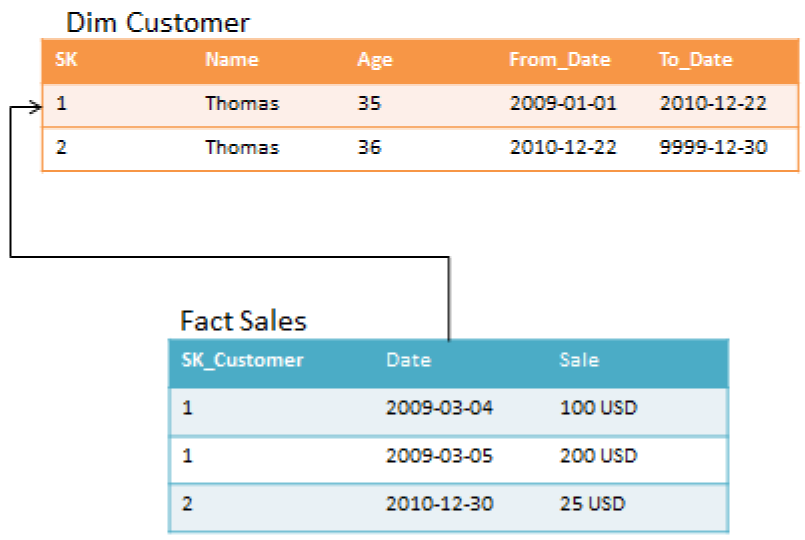
Type2 Fast-Changing Attributes - If you track every change to a fast-changing attribute, this may cause the dimension containing the attribute to grow very large. Type 2 attributes are typically added to a dimension with a **Process Add** command. At some point, running **Process Add** on a large dimension and running all the consistency checks will take a long time. Also, 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 of such a modeling challenge is the age of a customer – this will 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. An example of such a modeling challenge is to track the status attribute of a server in a hosting environment (“Running”, “Shut down”, “Overloaded” and so on). A status attribute like this may 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 with the locking implementation of Analysis Servicesin a production environment.

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

2.1.10.1 Type 2 Fast-Changing Attributes

If history tracking is a requirement of a fast-changing attribute, the best option is often to use the fact table to track history. This is best illustrated with an example. Consider again the customer dimension with the age attribute. Modeling the **Age** attribute directly in the customer dimension produces a design like this.



Dim Customer

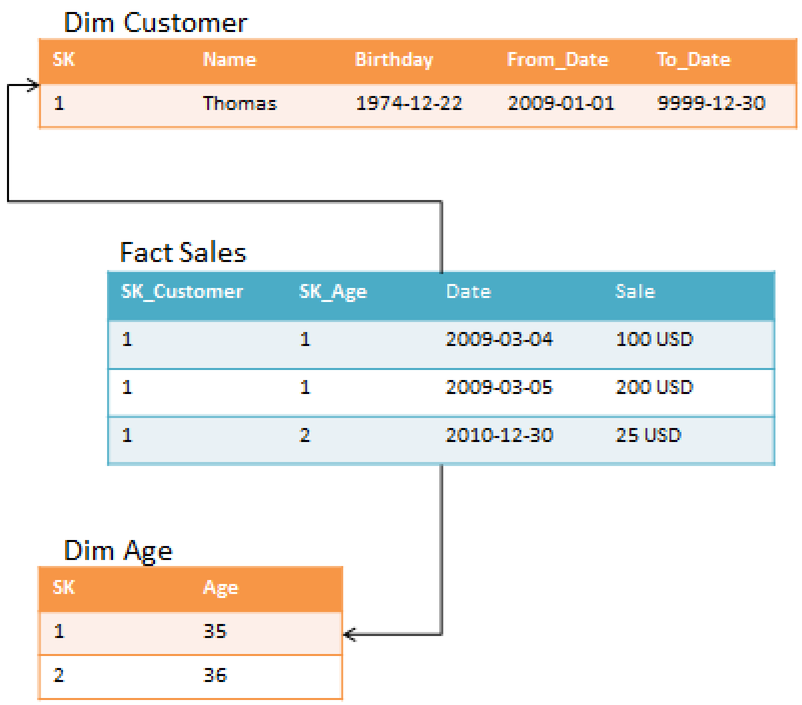
SK	Name	Age	From_Date	To_Date
1	Thomas	35	2009-01-01	2010-12-22
2	Thomas	36	2010-12-22	9999-12-30

Fact Sales

SK_Customer	Date	Sale
1	2009-03-04	100 USD
1	2009-03-05	200 USD
2	2010-12-30	25 USD

Figure : Age in customer dimension

Notice that every time Thomas has a birthday, a new row is added in the dimension table. The alternative design approach splits the customer dimension into two dimensions like this.



Dim Customer

SK	Name	Birthday	From_Date	To_Date
1	Thomas	1974-12-22	2009-01-01	9999-12-30

Fact Sales

SK_Customer	SK_Age	Date	Sale
1	1	2009-03-04	100 USD
1	1	2009-03-05	200 USD
1	2	2010-12-30	25 USD

Dim Age

SK	Age
1	35
2	36

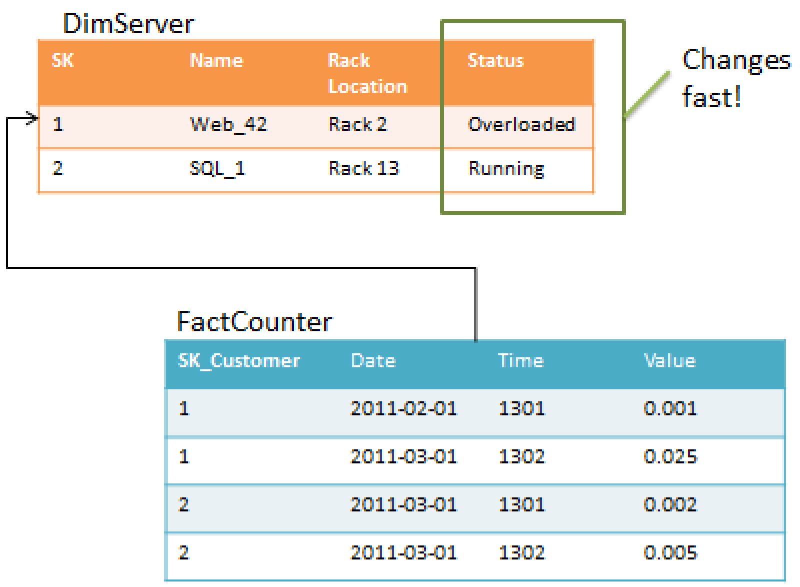
Figure : Age in its own dimension

Note that there are some restrictions on the situation where this design can be applied. It works best when the changing attribute takes on a small, distinct set of values. It also adds complexity to the design; by adding more dimensions to the model, it creates more work for the ETL developers when the fact table is loaded. Also, consider the storage impact on the fact table: With the alternative design, the fact table becomes wider, and more bytes have to be stored per row.

2.1.10.2 Type 1 Fast-Changing Attributes

Your business requirement may be updating an attribute of a dimension at high frequency, daily, or even hourly. For a small cube, running **Process Update** will help you address this issue. 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 (you can read more about tuning process update in the processing section).

Consider again the server hosting example: You may want to track the status, which changes frequently, of all servers. For the example, let us say that the server dimension is used by a fact table tracking performance counters. Assume you have modeled like this.

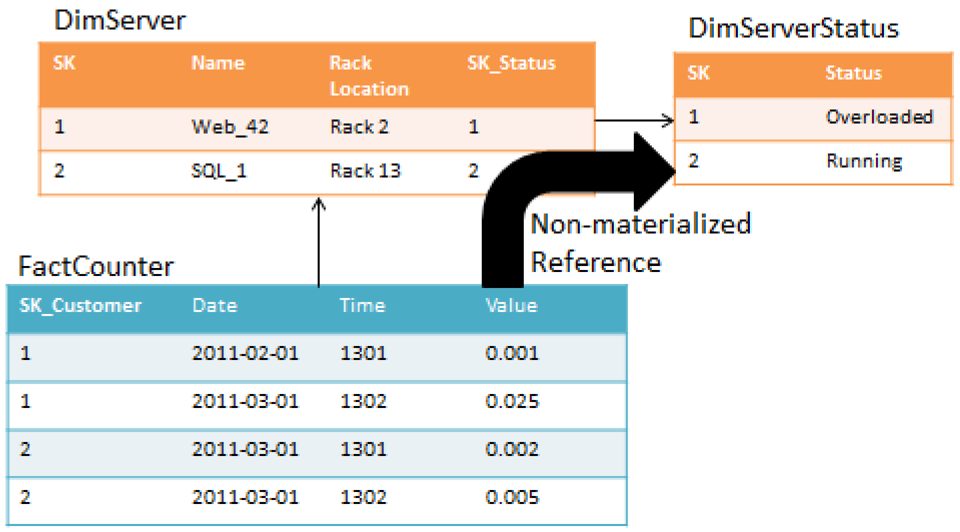


SK	Name	Rack Location	Status
1	Web_42	Rack 2	Overloaded
2	SQL_1	Rack 13	Running

SK_Customer	Date	Time	Value
1	2011-02-01	1301	0.001
1	2011-03-01	1302	0.025
2	2011-03-01	1301	0.002
2	2011-03-01	1302	0.005

Figure : Status column in server dimension

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



SK	Name	Rack Location	SK_Status
1	Web_42	Rack 2	1
2	SQL_1	Rack 13	2

SK	Status
1	Overloaded
2	Running

SK_Customer	Date	Time	Value
1	2011-02-01	1301	0.001
1	2011-03-01	1302	0.025
2	2011-03-01	1301	0.002
2	2011-03-01	1302	0.005

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**. But as described earlier, this means that the join to **DimServerStatus** will happen at run time, increasing CPU cost and query times. It also means that you cannot index attributes in **DimServer** because the intermediate dimension is not materialized. You have to carefully balance the tradeoff between processing time and query speeds.

2.1.11 Large Dimensions

In SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2, Analysis Services has some built-in limitations that limit the size of the dimensions you can create. First of all, it takes time to update a dimension – this is expensive because all indexes on fact tables have to be considered for invalidation 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. 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. The next version of SQL Server Analysis Services, code-named "Denali", is expected to remove this limitation.

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, even on SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2. But what does such a dimension mean to an ad-hoc user? How will the user navigate it? Which hierarchies will group the members of this dimension into reasonable sizes that can be rendered on a screen? 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 telco model of call detail records.

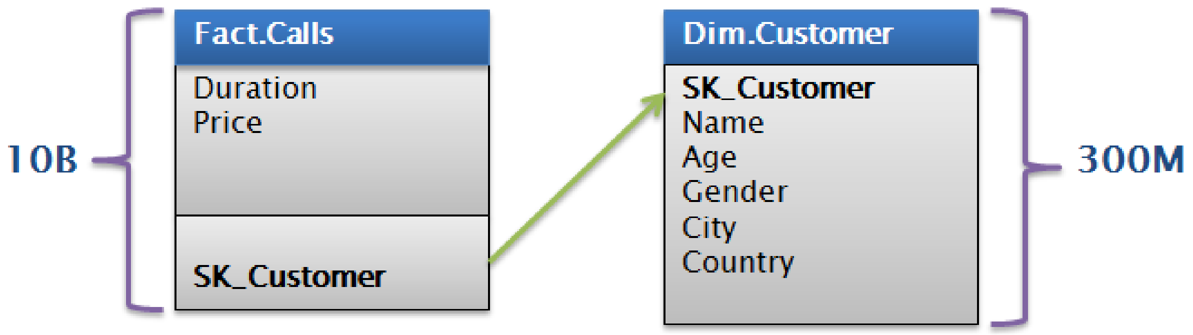


Figure : Call detail records (CDRs)

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 telco example, imagine a transformation like the following.

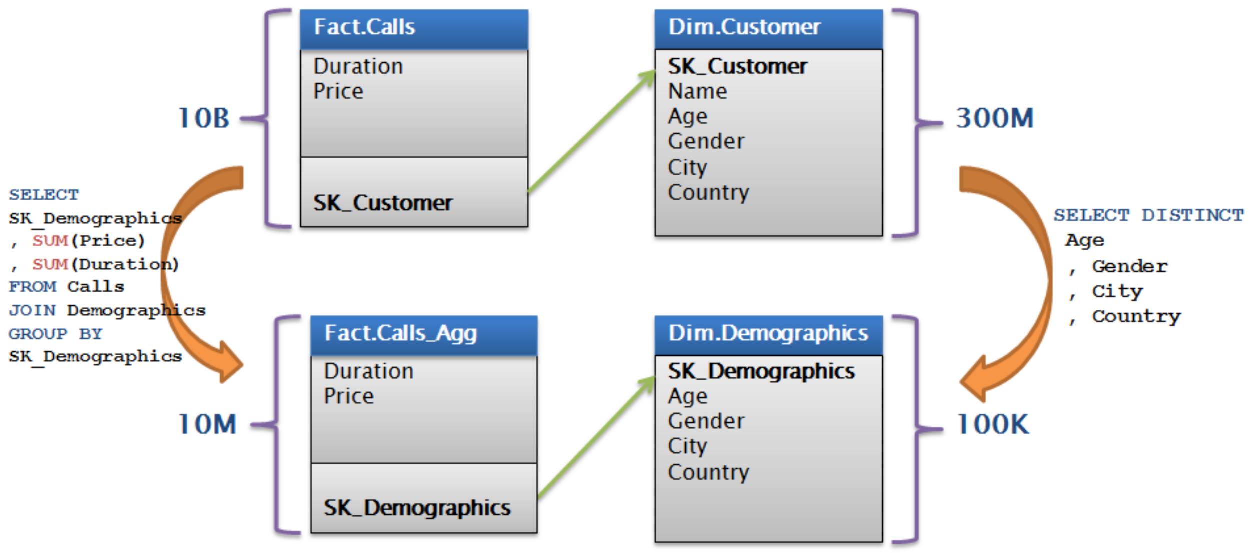


Figure : Cube built on aggregate

Using an aggregated fact table, this turns a 300-million-row dimension problem into 100,000-row dimension problem. You can 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.

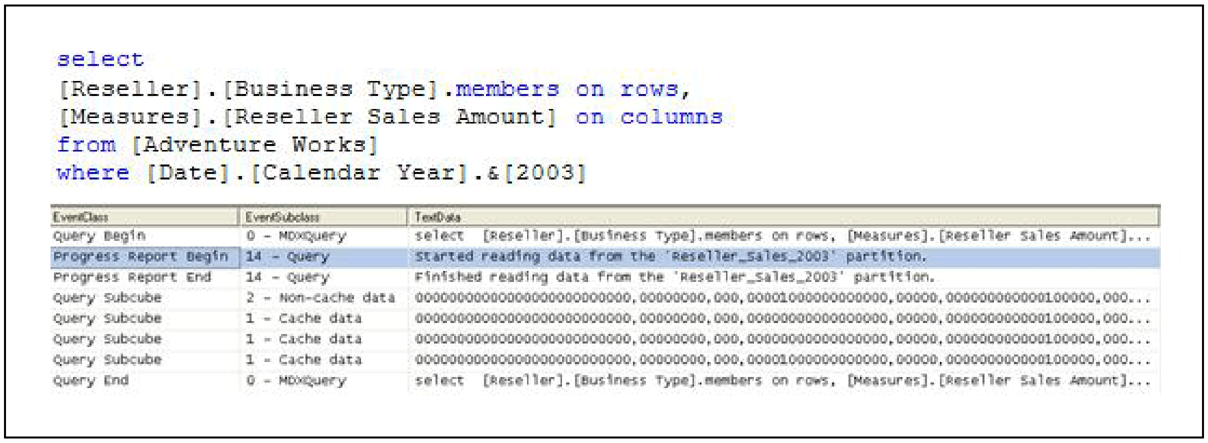
2.2 Partitioning a Cube

Partitions separate measure group data into physical storage units. Effective use of partitions can enhance query performance, improve processing performance, and facilitate data management. This section specifically addresses how you can use partitions to improve query performance. You must often make a tradeoff between query and processing performance in your partitioning strategy.

You can use multiple partitions to break up your measure group into separate physical components. The advantages of partitioning for improving query performance are partition elimination and aggregation design.

Partition elimination - Partitions that do not contain data in the subcube are not queried at all, thus avoiding the cost of reading the index (or scanning a table if the server is in ROLAP mode). While reading a partition index and finding no available rows is a cheap operation, as the number of concurrent users grows, these reads begin to put a strain in the threadpool. Also, for queries that do not have indexes to support them, Analysis Services will have to scan all potentially matching partitions for data.

Aggregation design - Each partition can have its own or shared aggregation design. Therefore, partitions queried more often or differently can have their own designs.



```
select
[Reseller].[Business Type].members on rows,
[Measures].[Reseller Sales Amount] on columns
from [Adventure Works]
where [Date].[Calendar Year].&[2003]
```

EventClass	EventSubclass	TextData
Query Begin	0 - MDXQuery	select [Reseller].[Business Type].members on rows, [Measures].[Reseller Sales Amount]...
Progress Report Begin	14 - Query	Started reading data from the 'Reseller_Sales_2003' partition.
Progress Report End	14 - Query	Finished reading data from the 'Reseller_Sales_2003' partition.
Query Subcube	2 - Non-cache data	0000000000000000000000000,00000000,000,00001000000000000,00000,000000000000100000,000...
Query Subcube	1 - Cache data	0000000000000000000000000,00000000,000,00000000000000000,00000,000000000000100000,000...
Query Subcube	1 - Cache data	0000000000000000000000000,00000000,000,00000000000000000,00000,000000000000100000,000...
Query Subcube	1 - Cache data	0000000000000000000000000,00000000,000,00001000000000000,00000,000000000000100000,000...
Query End	0 - MDXQuery	select [Reseller].[Business Type].members on rows, [Measures].[Reseller Sales Amount]...

Figure : Intelligent querying by partitions

Figure 10 displays the profiler trace of query requesting Reseller Sales Amount by Business Type from Adventure Works. 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.

2.2.1 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 slice definition is missing or if the information in the slice indicates that required data is present, 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 slicer** – There are cases where auto slice will not work – these are described in the next section. For those situations, you can manually set the slice. Manual slices are the only available slice option for ROLAP partitions and proactive caching partitions.

2.2.1.1 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 Min and Max DataIDs can specify a either a single member or a range of members. For example, partitioning by year results in the same Min and Max 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. Because you cannot control which DataID gets assigned to which members, you could be in a situation where the DataID for 2005 is 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).

2.2.1.2 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.

2.2.2 Partition Sizing

For nondistinct count measure groups, tests with partition sizes in the range of 200 MB to up to 3 GB indicate that partition size alone does not have a substantial impact on query speeds. In fact, we have successfully deployed good query performance on partitions larger than 3 GB.

The following graph shows four different query runs with different partition sizes (the vertical axis is total run time in hours). Performance is comparable between partition sizes and is only affected by the design of the security features in this particular customer cube.

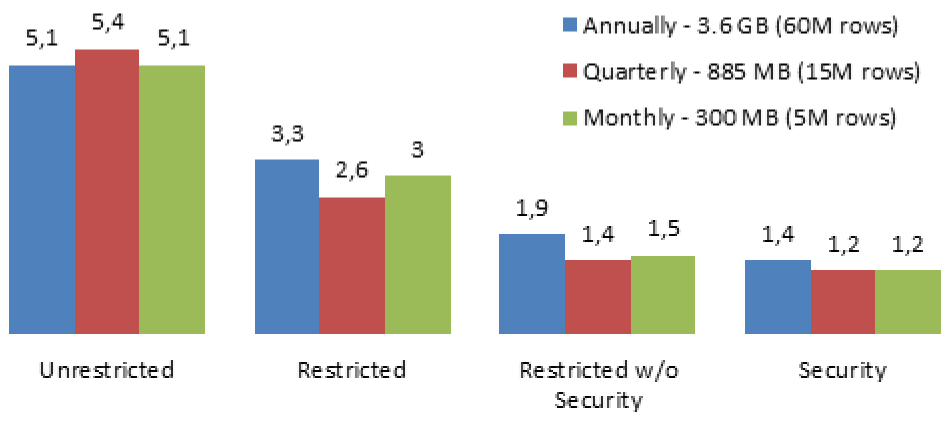


Figure : Throughput by partition size (higher is better)

The 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

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, you should therefore seek to keep the number of partitions in the cube in the low thousands – while at the same time balancing the requirements discussed here.

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.

2.2.3 Partition Strategy

From guidance on partition sizing, we can develop some common design patterns for partition strategies.

2.2.3.1 Partition by Date

Most cubes are built on at least one column containing a date. Because data often arrives in monthly, weekly, daily, or even hourly slices, it makes sense to partition the cube on date. Partitioning on date allows you to replace a full day in case you load faulty data. It allows you to selectively archive old data by moving the partition to cheap storage. And finally, it allows you to easily get rid of data, by removing an entire partition. Typically, a date partitioning scheme looks somewhat like this.

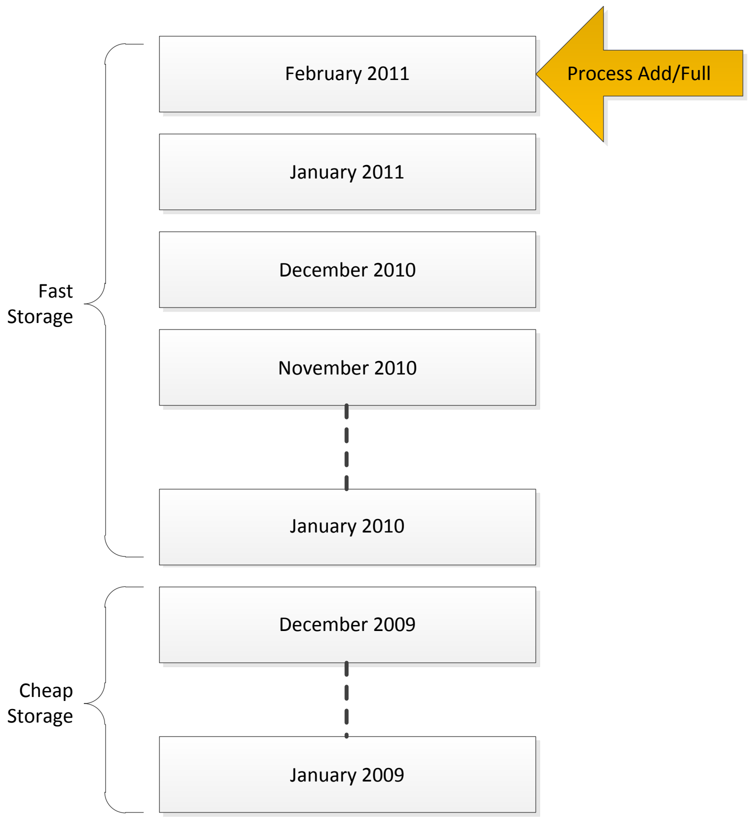


Figure : Partitioning by Date

Note that in order to move the partition to cheaper storage, you will have to change the data location and reprocesses the partition. 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. 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.

Ad 1) 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 may look like this.

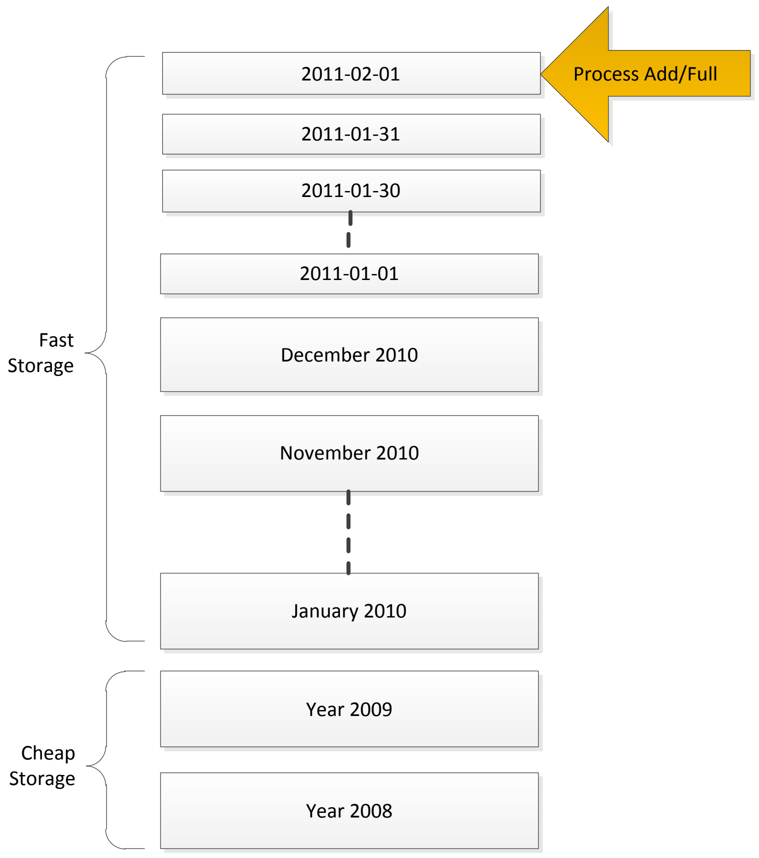


Figure : Modified Date Partitioning

This design addresses the metadata overhead of having too many partitions. But it is still bottlenecked by the maximum speed of the **Process Add** or **Process Full** 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.

2.2.3.2 Matrix Partitioning

For large cubes, it is often a good idea to implement a matrix partitioning scheme: partition on both date **and** some other key. The date partitioning is used to selectively delete or merge old partitions as described earlier. The other key can be used 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.

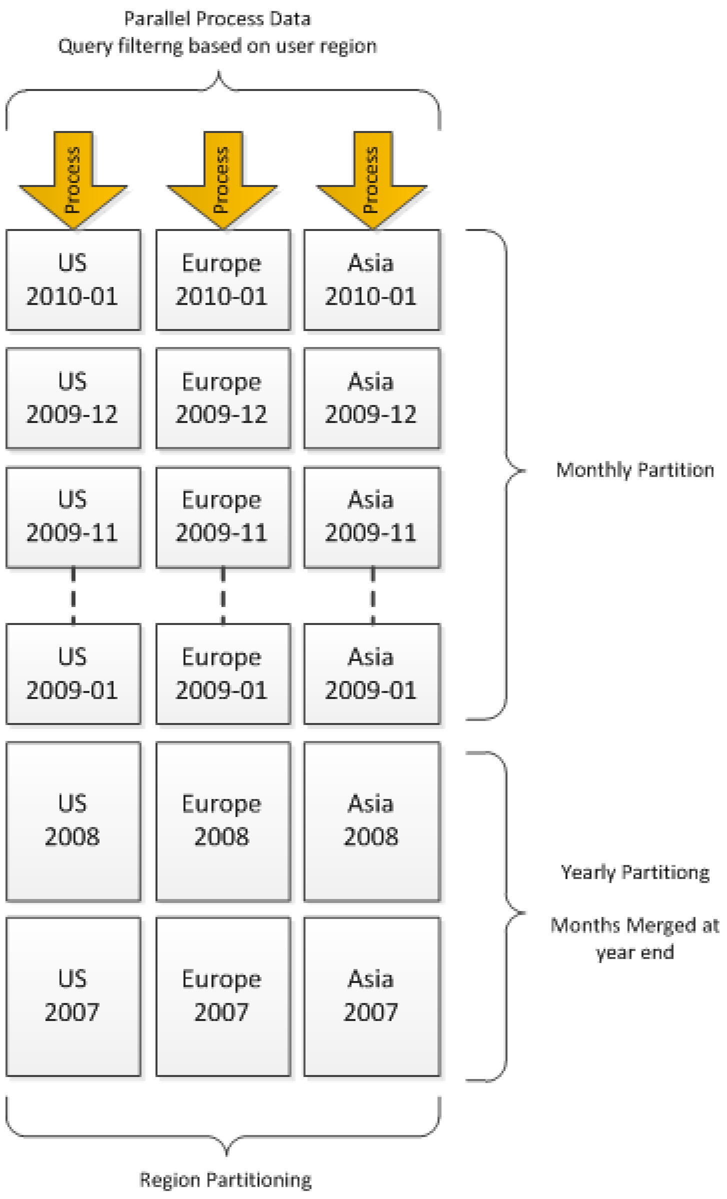


Figure : Example of matrix partitioning

If the retailer grows, they may choose to split the region partitions into smaller partitions to increase parallelism of load further and to limit the worst-case scans that a user can perform. For cubes that are expected to grow dramatically, it is a good idea to choose a partition key that grows with the business and gives you options for extending the matrix partitioning strategy appropriately. The following table contains 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 should pay special attention to user queries. Queries touching 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. In such a design, it is a good idea to group the stores into a number of buckets (that is, group the stores on each partition, rather than having individual partitions for each store). 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).

2.2.3.3 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 that fits the description in the previous section, 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 you expect every query to touch many partitions, it is important that you pay special attention to the **CoordinatorQueryBalancingFactor** and the **CoordinatorQueryMaxThread** settings, which are described in the SQL Server 2008 R2 Analysis Services Operations Guide.

2.3 Relational Data Source Design

Cubes are typically built on top of relational data sources to serve as data marts. Through the design surface, Analysis Services allows you to create powerful abstractions on top of the relational source. Computed columns and named queries are examples of this. This allows fast prototyping and also enabled you to correct poor relational design when you are not in control of the underlying data source. But the Analysis Services design surface is no panacea – a well-designed relational data source can make queries and processing of a cube faster. In this section, we explore 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.

2.3.1 Use a Star Schema for Best Performance

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

Note that the Analysis Services Unified Dimensional Model (UDM) is a dimensional model, with some additional features (reference dimensions) that support snowflakes and many-to-many dimensions. No matter which model you choose as the end-user reporting model, performance of the relational model boils down to one simple fact: joins are expensive! This is also partially true for the Analysis Services engine itself. 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. However, 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.

2.3.2 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 single solution here; 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.

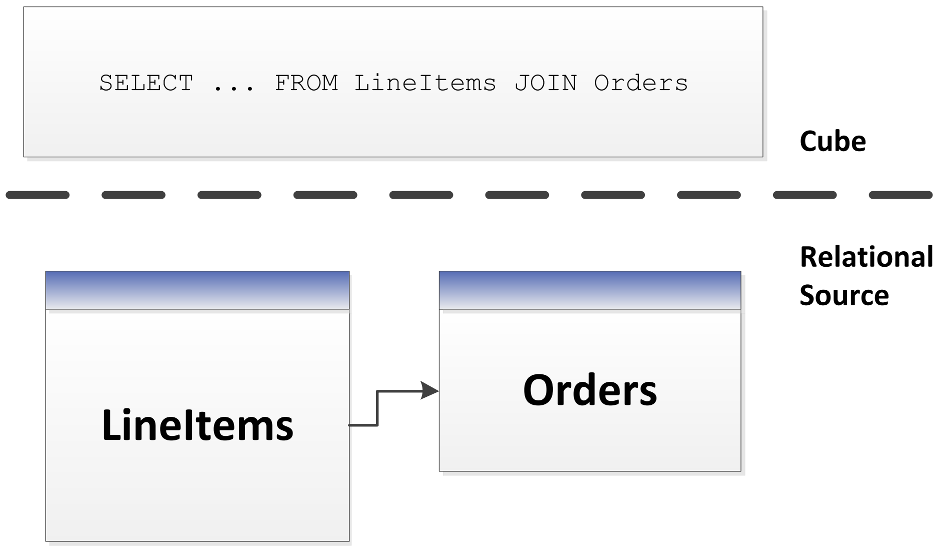
For example, instead of writing expressions like Sum(Customer.City.Members, cint(Customer.City.Currentmember.properties("Population"))), consider defining a separate measure group on the **City** table, with a sum measure on the **Population** column.

As a second example, you can compute the product of revenue * Products Sold at the leaves in the cube and aggregate with calculations. But computing this result in the source database instead can provide superior performance.

2.3.3 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.

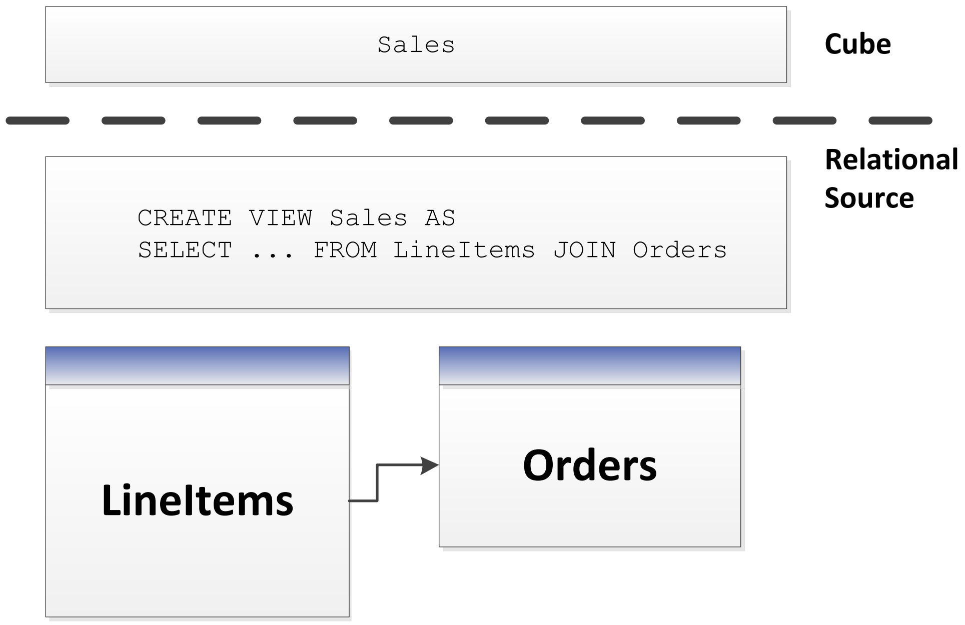
Consider a relational source that has chosen to normalize two tables you need to join to obtain a fact table – for example, a data model that splits a sales fact into order lines and orders. If you implement the fact table using query binding, your UDM will contain the following.



```
SELECT ... FROM LineItems JOIN Orders
```

Figure : 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.



```
CREATE VIEW Sales AS
SELECT ... FROM LineItems JOIN Orders
```

Figure : Implementing UDM on top of views

This 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 this change.

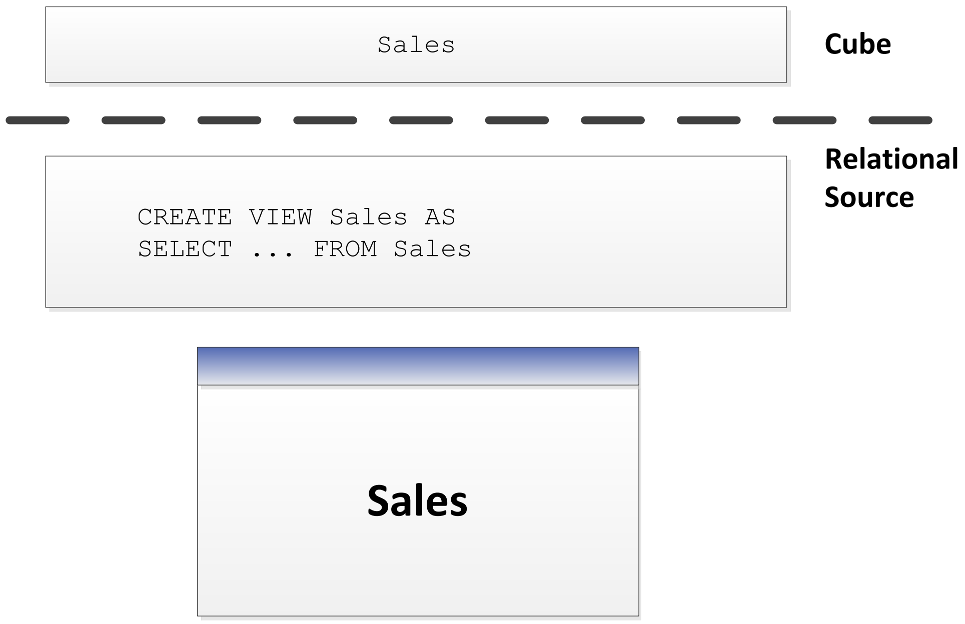


Figure : Implementing UDM on top of pre-joined tables

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

Views also provide easy of debugging. You can issue SQL queries directly on views to compare the relational data with the cube. Hence, views are good way to implement business logic that could you could 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.

2.3.3.1 Query Binding Dimensions

Query binding for dimensions does not exist in SQL Server 2008 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.

2.3.3.2 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, balancing this with the possibility of getting dirty reads. Views can also be used to preaggregate large fact tables using a GROUP BY statement; the relational database modeler can even choose to materialize views that use a lot of hardware resources.

2.4 Calculation Scripts

The calculation script in the cube allows you to express complex functionality of the cube, conferring the ability to directly manipulate the multidimensional space. In a few lines of code, you can elegantly build highly valuable business logic. But conversely, it takes only a few lines of poorly written calculation code to create a big performance impact on users. 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. The references section contains resources that will get you off to a good start.

The query tuning section of this guide provides high-level guidance on tuning individual queries. But even at design time, there are some best practices you should apply to the cube that avoid common performance mistakes. This section provides you with some basic rules; these are the bare minimum you should apply when building the cube script.

References:
MDX has a rich community of contributors on the web. Here are some links to get you started:

- 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

2.4.1 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 enable you to target aggregations to the subset. Attributes are also evaluated faster than sets by the formula engine. Using an attribute for this purpose also allows you to change the set by updating the dimension instead of deploying a new calculation scripts.

Example: Instead of this:

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

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

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.

2.4.2 Use SCOPE Instead of IIF When Addressing Cube Space

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

```
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;
```

2.4.3 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

You can reproduce each these features in MDX script (in fact, sometimes you must, because some are only supported in the Enterprise SKU), but doing so often hurts 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. For example, you might be able to improve performance of nondistributive unary operators (those involving *, /, or numeric values) with MDX. Furthermore, you may know some special characteristic of your data that allows you to take a shortcut that improves performance. 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 preaggregated values in the relational source. You can then target the hidden measure to the aggregate values with a SCOPE statement in the calculation script.

2.4.4 Comparing Objects and Values

When determining whether the current member or tuple is a specific object, use IS. For example, the following query is not only nonperformant, but incorrect. It forces unnecessary cell evaluation and compares values instead of members.

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

Furthermore, don’t perform extra steps when deducing whether **CurrentMember** is a particular member by involving **Intersect** and **Counting**.

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

Use IS instead.

```
[Customer].[Customer Geography].[Country].&[Australia] is [Customer].[Customer
Geography].currentmember
```

2.4.5 Evaluating Set Membership

Determining whether a member or tuple is in a set is best accomplished with **Intersect**. The **Rank** function does the additional operation of determining where in the set that object lies. If you don’t need it, don’t use it. For example, the following statement may do more work than you need it to do.

```
rank( [Customer].[Customer Geography].[Country].&[Australia],

<set expression> )>0
```

This statement uses **Intersect** to determine whether the specified information is in the set.

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

3 Tuning Query Performance

To improve query performance, you should understand the current situation, diagnose the bottleneck, and then apply one of several techniques including optimizing dimension design, designing and building aggregations, partitioning, and applying best practices. These should be the first stops for optimization, before digging into queries in general.

Much time can be expended pursuing dead ends – it is important to first understand the nature of the problem before applying specific techniques. To gain this understanding, it is often useful to have a mental model of how the query engine works. We will therefore start with a brief introduction to the Analysis Services query processor.

3.1 Query Processor Architecture

To make the querying experience as fast as possible for end users, the Analysis Services querying 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.

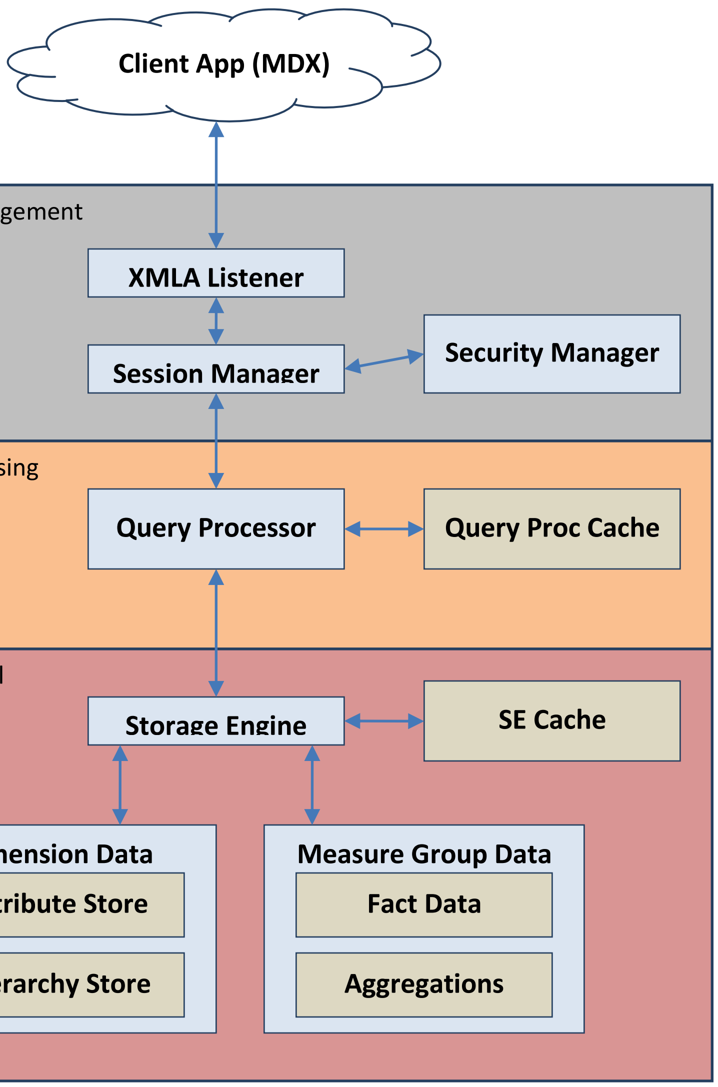


Figure : Analysis Services query processor architecture

3.1.1 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. Users authenticated by the Windows operating system and who have access to at least one database can connect to Analysis Services. 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. 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. The session provides the context under which client queries are executed by the query processor. A session exists until it is closed by the client application or the server.

3.1.2 Query Processing

The query processor executes MDX queries and generates a cellset or rowset in return. This section provides an overview of how the query processor executes queries. For more information about optimizing MDX, see Optimizing MDX.

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.

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.

3.1.2.1 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. At the top, the query context is can be reused only within the query. At the bottom, the 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.

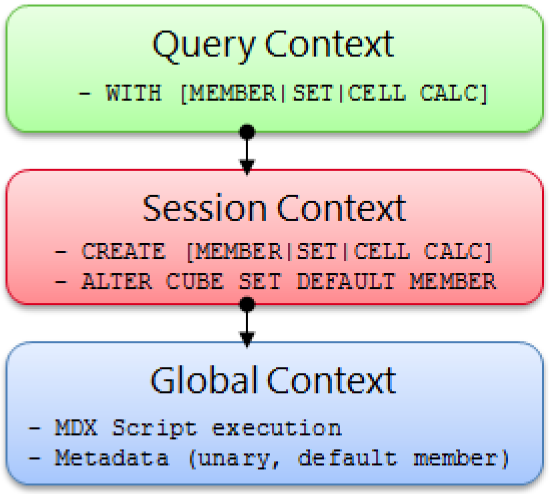

Query Context
- WITH [MEMBER|SET|CELL CALC]
Session Context
- CREATE [MEMBER|SET|CELL CALC]
- ALTER CUBE SET DEFAULT MEMBER
Global Context
- MDX Script execution
- Metadata (unary, default member)

Figure : Cache context layers

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 creates a query context to resolve the query calculated member, creates a session context to evaluate session calculations, and creates a global context to evaluate the MDX script and retrieve the security permissions of the user who submitted the query. Note that 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). If there are no query calculations, but there are session calculations, the query processor uses the session cache. The query processor selects the cache based on the presence of any calculation in the scope. This behavior is especially relevant to users with MDX-generating front-end tools. If the front-end tool creates any session calculations or query calculations, the global cache is not used, even if you do not specifically use the session or query calculations.

There are other calculation scenarios that impact 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 (meaning that there is no guarantee what the stored procedure will return). As a result, after 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. In addition, 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 cell security; functions such as **UserName**, **StrToSet, StrToMember,** and **StrToTuple**; or **LookupCube** functions in the MDX script or in the dimension or cell security definition disable the global cache. That is, 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 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, {(Food, USA), (Drink, Canada)} is an arbitrary set, as is {customer.geography.USA, customer.geography.[British Columbia]}. Note that an arbitrary shape on the query axis does not limit the use of any cache.

Based on this behavior, when your querying 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. By 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.

3.1.3 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.

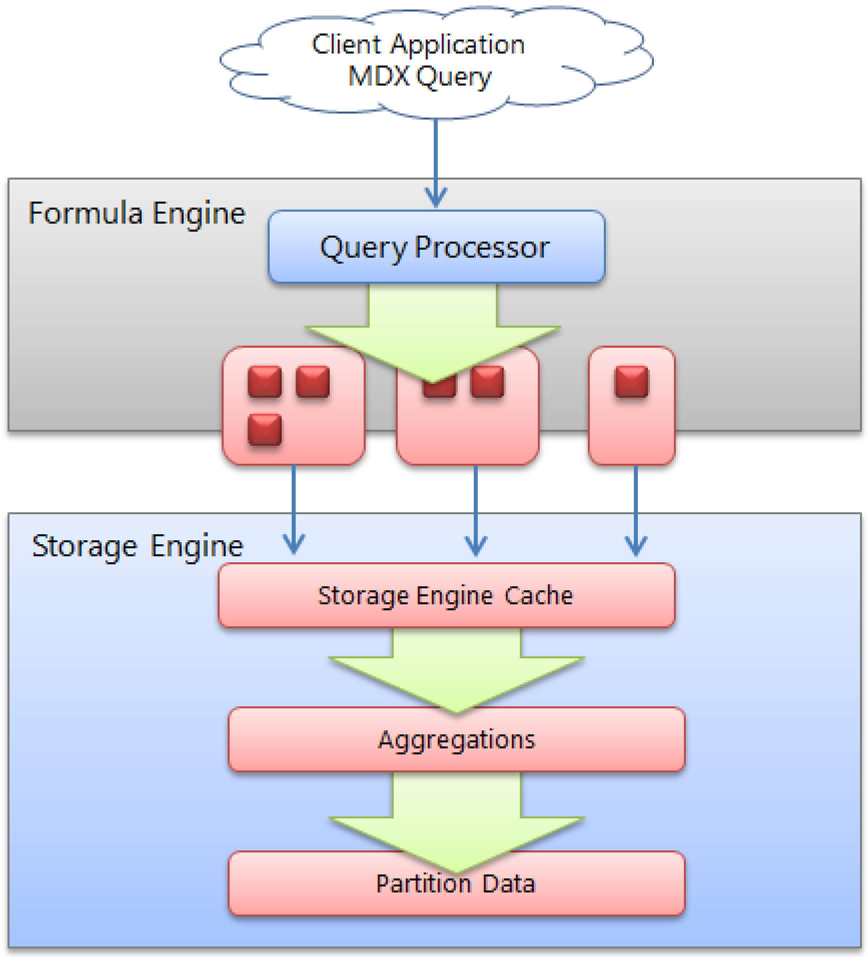


Figure : High-level overview of the data retrieval process

3.1.3.1 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:

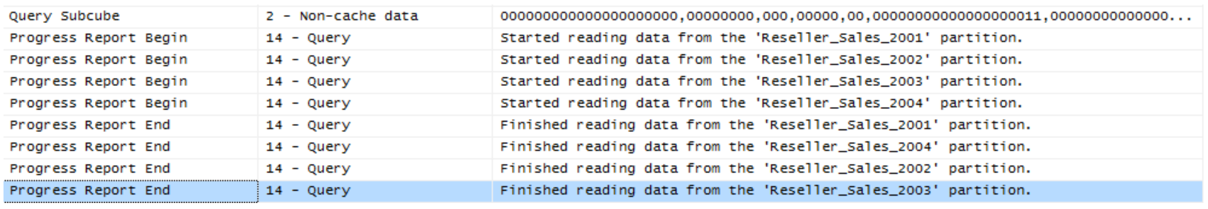
- If there is a matching subcube, the corresponding data cache is returned.
- If a subcube superset is found, a new data cache is generated and the results are filtered to fit the subcube request.
- If lower-grain data exists, the data cache registry can aggregate this data and make it available as well – 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.

3.1.3.2 Aggressive Data Scanning

Sometimes, in the evaluation of an expression, more data is requested than 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 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 figure demonstrates how a single query subcube event results in partition scans.



Query Subcube	2 - Non-cache data	00000000000000000000,00000000,000,00000,00,0000000000000000011,0000000000000...
Progress Report Begin	14 - Query	Started reading data from the 'Reseller_Sales_2001' partition.
Progress Report Begin	14 - Query	Started reading data from the 'Reseller_Sales_2002' partition.
Progress Report Begin	14 - Query	Started reading data from the 'Reseller_Sales_2003' partition.
Progress Report Begin	14 - Query	Started reading data from the 'Reseller_Sales_2004' partition.
Progress Report End	14 - Query	Finished reading data from the 'Reseller_Sales_2001' partition.
Progress Report End	14 - Query	Finished reading data from the 'Reseller_Sales_2004' partition.
Progress Report End	14 - Query	Finished reading data from the 'Reseller_Sales_2002' partition.
Progress Report End	14 - Query	Finished reading data from the 'Reseller_Sales_2003' partition.

Figure : Aggressive partition scanning

There are two potential solutions to this. If a calculation expression contains an arbitrary shape (this is defined in the section on the query processor cache), the query processor may not be able to determine that the data is limited to a single partition and 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.
- Disable Prefetch = 1 (this is done in the connection string): 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.

3.2 Query Processor Internals

There are several changes to query processor internals in SQL Server 2008 Analysis Services that are applicable today (compared to SQL Server 2005 Analysis Services). In this section, these changes are discussed before specific optimization techniques are introduced.

3.2.1 Subspace Computation

The key idea behind subspace computation is best introduced by contrasting it with a cell-by-cell evaluation of a calculation. (This is also known as a naïve calculation.) Consider a trivial calculation RollingSum that sums the sales for the previous year and the current year, and a query that requests the RollingSum for 2005 for all Products.

```
RollingSum = (Year.PrevMember, Sales) + Sales

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.

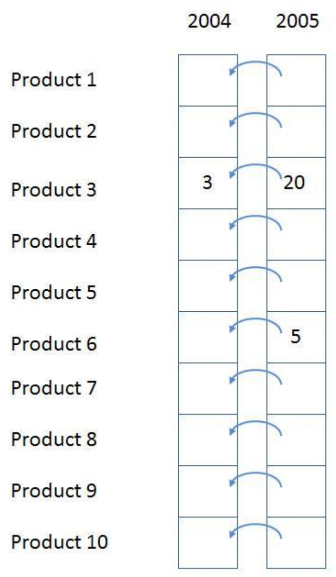


Figure : Cell-by-cell evaluation

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.

Firstly, 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.

Secondly, 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 applied to them to yields the following result.

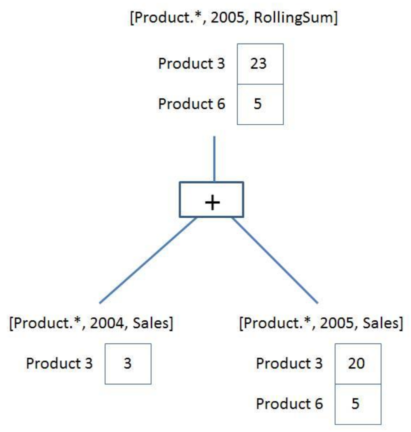


Figure : Execution plan

The + operator operates on *spaces*, not simply *scalar values.* It 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.

3.2.2 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.

3.2.3 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 other simple rules for 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**)

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).

3.2.4 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 across most of its space. Most of the time this is null because oftentimes (but not always) the result of an expression with null input values is null. The engine can then compute the null result once, and then it needs to compute only values for the much reduced nonnull space.

Another important use of the default values is in the condition in the **IIf** function. Knowing which branch is evaluated more often drives the execution plan. The default values of some popular expressions are listed in the following table.

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.

3.2.5 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 the expression doesn’t depend on can be eliminated, and then the expression can be resolved and projected back over the original subspace. The attributes an expression depends on are termed 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]
```

The 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.

3.3 Optimizing MDX

Debugging calculation 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.

3.3.1 Baselining Query Speeds

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.

3.3.1.1 Clearing the Analysis Services Caches

The Analysis Services formula engine and storage engine caches can be cleared with 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>
```

3.3.1.2 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.

3.3.1.3 Measure Query Speeds

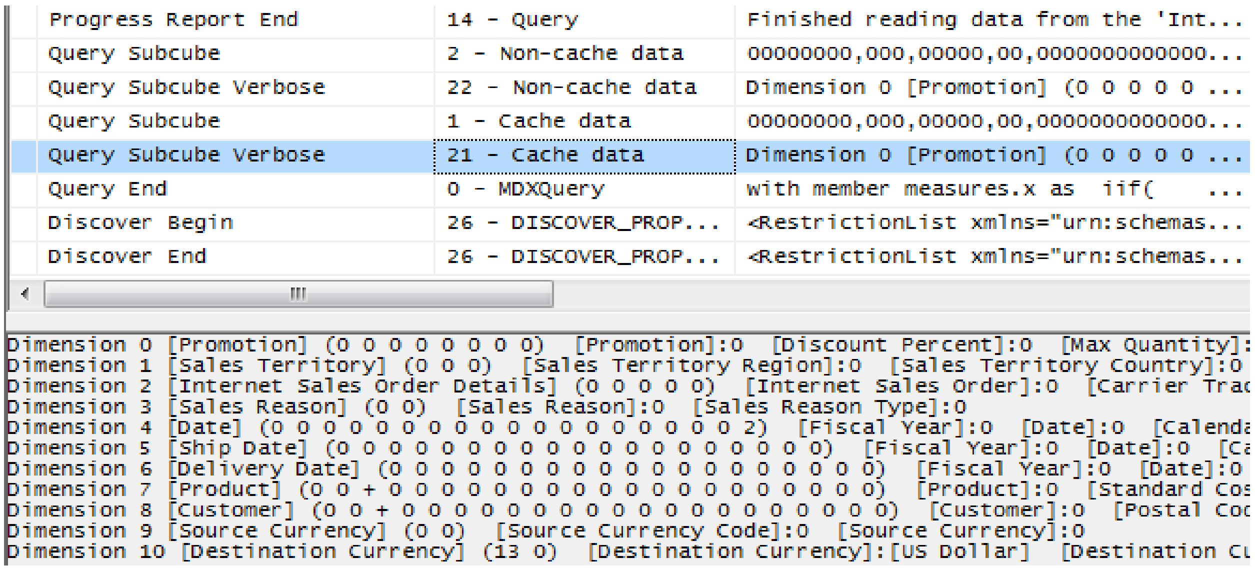
When all caches are clear, you should initialize the calculation script by executing a query that returns and caches nothing. Here is an example.

```
select {} on 0 from [Adventure Works]
```

Execute the query you want to optimize and then use SQL Server Profiler 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.



Progress Report End	14 - Query	Finished reading data from the 'Int...
Query Subcube	2 - Non-cache data	00000000,000,00000,00,000000000000...
Query Subcube Verbose	22 - Non-cache data	Dimension 0 [Promotion] (0 0 0 0 0 ...
Query Subcube	1 - Cache data	00000000,000,00000,00,000000000000...
Query Subcube Verbose	21 - Cache data	Dimension 0 [Promotion] (0 0 0 0 0 ...
Query End	0 - MDXQuery	with member measures.x as iif(...
Discover Begin	26 - DISCOVER_PROP...	<RestrictionList xmlns="urn:schemas...
Discover End	26 - DISCOVER_PROP...	<RestrictionList xmlns="urn:schemas...

```
Dimension 0 [Promotion] (0 0 0 0 0 0 0 0)  [Promotion]:0  [Discount Percent]:0  [Max Quantity]:
Dimension 1 [Sales Territory] (0 0 0)  [Sales Territory Region]:0  [Sales Territory Country]:0
Dimension 2 [Internet Sales Order Details] (0 0 0 0 0)  [Internet Sales Order]:0  [Carrier Trac
Dimension 3 [Sales Reason] (0 0)  [Sales Reason]:0  [Sales Reason Type]:0
Dimension 4 [Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2)  [Fiscal Year]:0  [Date]:0  [Calenda
Dimension 5 [Ship Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Fiscal Year]:0  [Date]:0  [Ca
Dimension 6 [Delivery Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Fiscal Year]:0  [Date]:0
Dimension 7 [Product] (0 0 + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Product]:0  [Standard Cos
Dimension 8 [Customer] (0 0 + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Customer]:0  [Postal Coc
Dimension 9 [Source Currency] (0 0)  [Source Currency Code]:0  [Source Currency]:0
Dimension 10 [Destination Currency] (13 0)  [Destination Currency]:[US Dollar]  [Destination Cu
```

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 query subcube verbose event textdata, see the following:

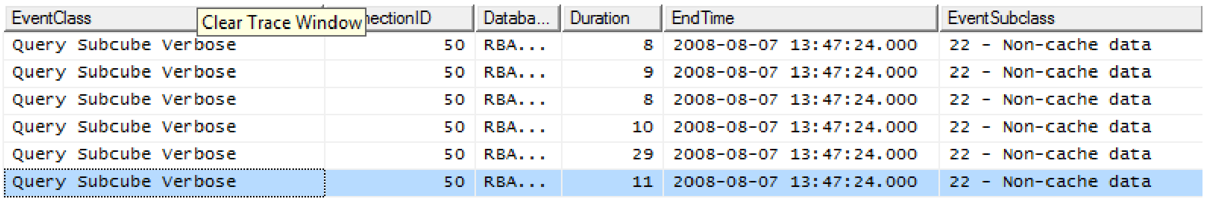
- Identifying and Resolving MDX Query Performance Bottlenecks in SQL Server 2005 Analysis Services (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/12/16/identifying-and-resolving-mdx-query-performance-bottlenecks-in-sql-server-2005-analysis-services.aspx)
- Configuring the Analysis Services Query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

SQL Server Management Studio displays the total query time. But be careful: This time is the amount of time taken to retrieve and display the cellset. For large results, the time to render the cellset on the client can rival the time it took the server to generate it. Instead of using SQL Server Management Studio, use the SQL Server Profiler Query End event to measure how long the query takes from the server's perspective and get the Analysis Services engine duration.

3.3.2 Isolating the Problem

Diagnosing the problem may be straightforward if a simple query calls out a specific calculation (in which case you should continue to the next section), but if there are chains of expressions or a complex query, it can be time-consuming to locate the problem. Try to 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, problems can arise when client applications generate queries that demand large data volumes, push down to unnecessarily low granularities, unnecessarily bypass aggregations, or contain query 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.

When you have removed calculations until the performance issue reproduces, the first 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. Limit the events 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.



EventClass	...nectionID	Databa...	Duration	EndTime	EventSubclass
Query Subcube Verbose	50	RBA...	8	2008-08-07 13:47:24.000	22 - Non-cache data
Query Subcube Verbose	50	RBA...	9	2008-08-07 13:47:24.000	22 - Non-cache data
Query Subcube Verbose	50	RBA...	8	2008-08-07 13:47:24.000	22 - Non-cache data
Query Subcube Verbose	50	RBA...	10	2008-08-07 13:47:24.000	22 - Non-cache data
Query Subcube Verbose	50	RBA...	29	2008-08-07 13:47:24.000	22 - Non-cache data
Query Subcube Verbose	50	RBA...	11	2008-08-07 13:47:24.000	22 - Non-cache data

Figure : Trace of query subcube events

If the majority of time is spent in the storage engine with 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 with SQL Server Profiler if you see many query subcube events generated by a single query. Each request may not take long, but the sum of them may. If this is the case, consider warming the cache to make sure subcubes and calculations are already cached. Also, consider rewriting the query to remove arbitrary shapes, because arbitrary subcubes cannot be cached. For more information, see Cache Warming later in this white paper.

If the cube and MDX query are already fully optimized, you may consider doing thread, memory, and configuration tuning of the cube. You may even want to look at larger hardware. Server-level tuning techniques are described in the SQL Server 2008 R2 Analysis Services Operations Guide.

References:

- The SQL Server 2008 R2 Analysis Services Operations Guide (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/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://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.

3.3.3 Cell-by-Cell Mode vs. Subspace Mode

Almost always, performance obtained by using subspace (or block computation) mode is superior to that obtained by using cell-by-cell (nor naïve) mode. 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) in SQL Server Books Online.

The following table lists the most common reasons for leaving subspace mode.

Feature or function	Comment

Set aliases	Replace with a set expression rather than an alias. For example, this query operates in subspace mode.
	
	`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, almost the same query ,where the set is replaced with an alias, operates in cell-by-cell mode:

	`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
	
	Note: This functionality has been fixed with the latest service pack of SQL Server 2008 R2 Analysis Services.

Late binding in functions:
 	
LinkMember, **StrToSet**, **StrToMember**, **StrToValue**
	Late-binding functions are functions that depend on query context and cannot be statically evaluated. For example, 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`
	
	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`

User-defined stored procedures	User-defined stored procedures are evaluated in cell-by-cell mode. Some popular Microsoft Visual Basic for Applications (VBA) functions are natively supported in MDX, but they are still 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.

3.3.4 Avoid Assigning Nonnull Values to Otherwise Empty Cells

The Analysis Services engine is very efficient at using sparsity of the data to improve performance. Adding calculations with nonempty values replacing empty values does not allow Analysis Services to 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
```

Note, **non empty** operates on cell values but not on formatted values. In rare cases you can instead 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.

```
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 can only be used in rare cases is that the query is not equivalent – the second query eliminates completely empty rows. More importantly, neither Excel nor SQL Server Reporting Services supports the fourth argument in the format_string.

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) 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).

3.3.5 Sparse/Dense Considerations with “expr1 * expr2” Expressions

When you write expressions as products of two other expressions, place the *sparser* one on the left-hand side. Recall, an expression is sparse if there are few non-null values compared to the total number of cells; for more information, see Expression Sparsity earlier in this section.

Consider the following two queries, which have the signature of a currency conversion calculation of applying the exchange rate at leaves of the date dimension in Adventure Works. The only difference is that the order of the expressions in the product of the cell calculation changes. The results are the same, but using the sparser internet sales amount first results in about a 10% savings. (That's not much in this case, but it could be substantially more in others. Savings depends on relative sparsity between the two expressions, and performance benefits may vary).

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])
```

3.3.6 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>)
```

Where the condition evaluates to true, the value from the then branch is used; otherwise the else branch expression is used. Note the term *used* – one or both branches may be evaluated even if the value is not used. It may be cheaper for the engine to evaluate the expression over the entire space and use it when needed - termed an *eager* plan – than it would be to chop up the space into a potentially enormous number of fragments and evaluate only where needed - 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.

IIf considerations:

1) The first consideration is whether the *query plan is expensive or inexpensive*.
   Most **IIf** condition query plans are inexpensive, but complex nested conditions with more **IIf** functions can go to cell by cell.
2) The next consideration the engine makes is *what value the condition takes most*. This is driven by the condition's default value. 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 helps 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.

For example, one of the most common uses of the **IIf** function is to check whether the denominator is nonzero:

```
iif([Measures].[Internet Sales Amount]=0
  , null
  , [Measures].[Internet Order Quantity]/[Measures].[Internet Sales Amount])
```

There is no calculation on Internet Sales Amount; therefore it 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 SQL Server 2008 Analysis Services, you can overrule the default behavior with query hints.

```
iif(
      [<condition>
      , <then branch> [hint [Eager | Strict]]
      , <else branch> [hint [Eager | Strict]]
)
```

Here are the most common scenarios where you might want to change the default behavior:

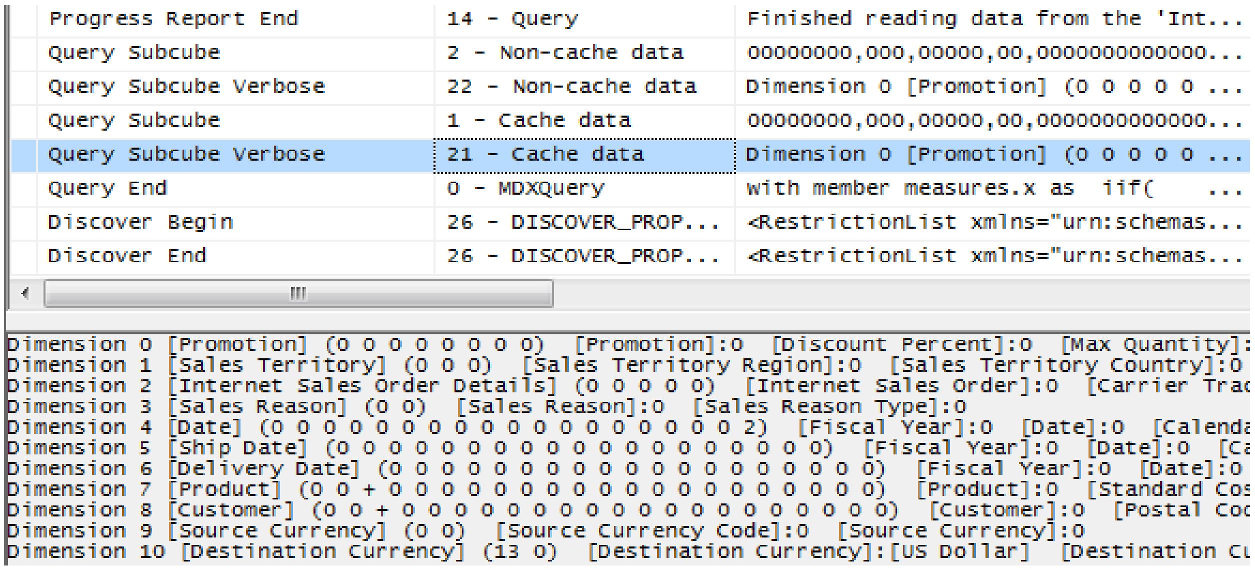
- The engine determines the query plan for the condition is expensive and evaluates each branch in strict mode.
- The condition is evaluated in cell-by-cell mode, and each branch is evaluated in eager mode.
- The branch expression is dense but easily evaluated.

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
```

The query plan is not expensive, the else branch is not the default branch, and the expression is dense, so it is evaluated in strict mode. This forces the engine to materialize the space over which it is evaluated. This can be seen in SQL Server Profiler with query subcube verbose events selected as displayed in Figure 26.



```
Progress Report End      14 - Query                 Finished reading data from the 'Int...
Query Subcube            2 - Non-cache data         00000000,000,00000,00,000000000000...
Query Subcube Verbose    22 - Non-cache data        Dimension 0 [Promotion] (0 0 0 0 0 ...
Query Subcube            1 - Cache data             00000000,000,00000,00,000000000000...
Query Subcube Verbose    21 - Cache data            Dimension 0 [Promotion] (0 0 0 0 0 ...
Query End                0 - MDXQuery               with member measures.x as  iif(    ...
Discover Begin           26 - DISCOVER_PROP...      <RestrictionList xmlns="urn:schemas...
Discover End             26 - DISCOVER_PROP...      <RestrictionList xmlns="urn:schemas...
```

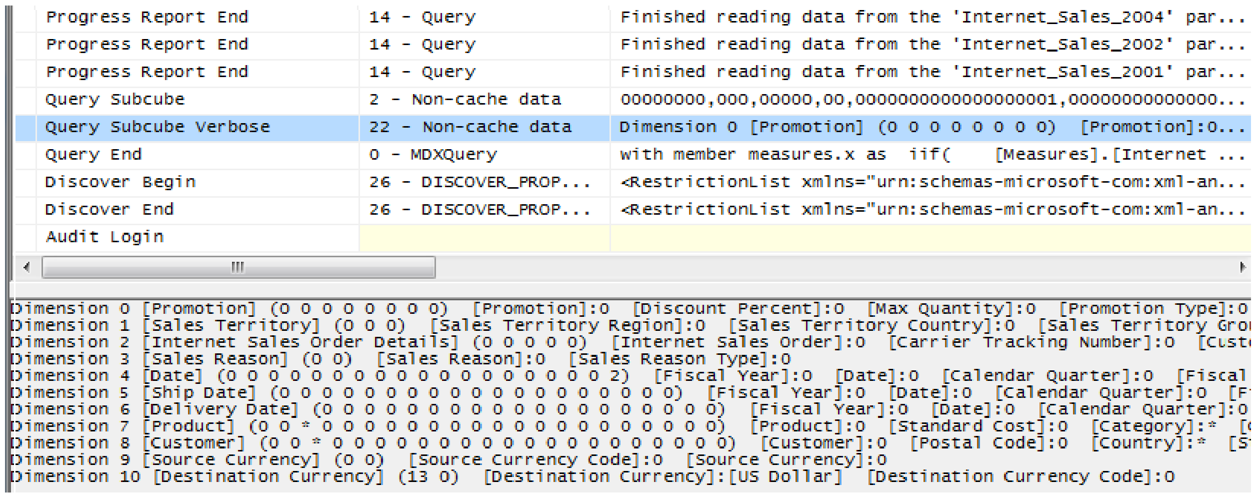
```
Dimension 0 [Promotion] (0 0 0 0 0 0 0 0)  [Promotion]:0  [Discount Percent]:0  [Max Quantity]:
Dimension 1 [Sales Territory] (0 0 0)  [Sales Territory Region]:0  [Sales Territory Country]:0
Dimension 2 [Internet Sales Order Details] (0 0 0 0 0)  [Internet Sales Order]:0  [Carrier Trac
Dimension 3 [Sales Reason] (0 0)  [Sales Reason]:0  [Sales Reason Type]:0
Dimension 4 [Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2)  [Fiscal Year]:0  [Date]:0  [Calenda
Dimension 5 [Ship Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Fiscal Year]:0  [Date]:0  [Ca
Dimension 6 [Delivery Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Fiscal Year]:0  [Date]:0
Dimension 7 [Product] (0 0 + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Product]:0  [Standard Cos
Dimension 8 [Customer] (0 0 + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Customer]:0  [Postal Coc
Dimension 9 [Source Currency] (0 0)  [Source Currency Code]:0  [Source Currency]:0
Dimension 10 [Destination Currency] (13 0)  [Destination Currency]:[US Dollar]  [Destination Cu
```

Figure : Default IIf query trace

Note 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 (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
```



```
Progress Report End       14 - Query              Finished reading data from the 'Internet_Sales_2004' par...
Progress Report End       14 - Query              Finished reading data from the 'Internet_Sales_2002' par...
Progress Report End       14 - Query              Finished reading data from the 'Internet_Sales_2001' par...
Query Subcube             2 - Non-cache data      00000000,000,00000,00,0000000000000000001,0000000000000...
Query Subcube Verbose     22 - Non-cache data     Dimension 0 [Promotion] (0 0 0 0 0 0 0 0)  [Promotion]:0...
Query End                 0 - MDXQuery            with member measures.x as  iif(    [Measures].[Internet ...
Discover Begin            26 - DISCOVER_PROP...   <RestrictionList xmlns="urn:schemas-microsoft-com:xml-an...
Discover End              26 - DISCOVER_PROP...   <RestrictionList xmlns="urn:schemas-microsoft-com:xml-an...
Audit Login

Dimension 0 [Promotion] (0 0 0 0 0 0 0 0)  [Promotion]:0  [Discount Percent]:0  [Max Quantity]:0  [Promotion Type]:0
Dimension 1 [Sales Territory] (0 0 0)  [Sales Territory Region]:0  [Sales Territory Country]:0  [Sales Territory Gro
Dimension 2 [Internet Sales Order Details] (0 0 0 0 0)  [Internet Sales Order]:0  [Carrier Tracking Number]:0  [Cust
Dimension 3 [Sales Reason] (0 0)  [Sales Reason]:0  [Sales Reason Type]:0
Dimension 4 [Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2)  [Fiscal Year]:0  [Date]:0  [Calendar Quarter]:0  [Fiscal
Dimension 5 [Ship Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Fiscal Year]:0  [Date]:0  [Calendar Quarter]:0  [F
Dimension 6 [Delivery Date] (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Fiscal Year]:0  [Date]:0  [Calendar Quarter]:0
Dimension 7 [Product] (0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Product]:0  [Standard Cost]:0  [Category]:*  [
Dimension 8 [Customer] (0 0 * 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)  [Customer]:0  [Postal Code]:0  [Country]:*  [S
Dimension 9 [Source Currency] (0 0)  [Source Currency Code]:0  [Source Currency]:0
Dimension 10 [Destination Currency] (13 0)  [Destination Currency]:[US Dollar]  [Destination Currency Code]:0
```

Figure : IIf trace with MDX query hints

Now the same attributes are marked with a '*' indicator, meaning that the expression is evaluated over the entire space instead of a partitioned space.

3.3.7 Cache Partial Expressions and Cell Properties

Partial expressions (those that are part of a calculated member or assignment) are not cached. So if an expensive subexpression is used more than once, consider creating a separate calculated member to allow the query processor to cache and reuse. For example, consider the following.

```
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);
```

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, this 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 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>;
```

3.3.8 Eliminate Varying Attributes in Set Expressions

Set expressions do not support *varying attributes*. This impacts 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 only including those exceeding $100 is computed.

```
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]
```

On a desktop box, this calculation takes approximately 2:29. However, the average of sales for all customers everywhere does not depend on the current city (this is just another way of saying that city is not a varying attribute). You can explicitly eliminate city as a varying attribute by overwriting it to 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 takes less than two seconds to complete. The following is a partial view aggregating the SQL Server Profiler traces of the two queries in the example by **EventClass** and **EventSubClass**.

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

The **Query Subcube** > **Non-Cache Data** durations are relatively small, denoting that most of the query calculation is done by the Analysis Services formula engine. This is apparent with the *AvgSales* calculation because most of the query durations correspond to the *Serial Results* event, which reports the status of serializing axes and cells. The use of `[All Customers]` ensures that the expression is evaluated only once for each Customer, improving performance.

3.3.9 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.

3.3.10 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 is 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.

This property oftentimes resulted in substantial performance improvements in past releases. However, starting with SQL Server 2008, the property is oftentimes ignored (because the engine automatically deals with nonempty cells in many cases) and can sometimes result in degraded performance. Eliminate it from the MDX script and add it back after performance testing demonstrates 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 has changed:

- It remains a guarantee that when NON_EMPTY_BEHAVIOR is null that the expression must also be null. (If this is not true, incorrect query results can still be returned.)
- However, 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 property is defined and is applied by the engine, it is semantically equivalent (not performance equivalent, however) to the following expression.

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

The NON_EMPTY_BEHAVIOR property is used if <e2> is sparse and <e1> is dense or <e1> is evaluated in the naïve *cell-by-cell* 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.

3.3.11 References

Below are links to some handy MDX optimization articles, books, and blog posts:

- Query calculated members invalidate formula engine cache (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/) by Chris Webb
- Measure datatypes (http://bidshelper.codeplex.com/wikipage?title=Measure%20Group%20Health%20Check&ProjectName=bidshelper)
- Currency datatype (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2008/09/25/the-many-benefits-of-money-data-type.aspx)

3.4 Aggregations

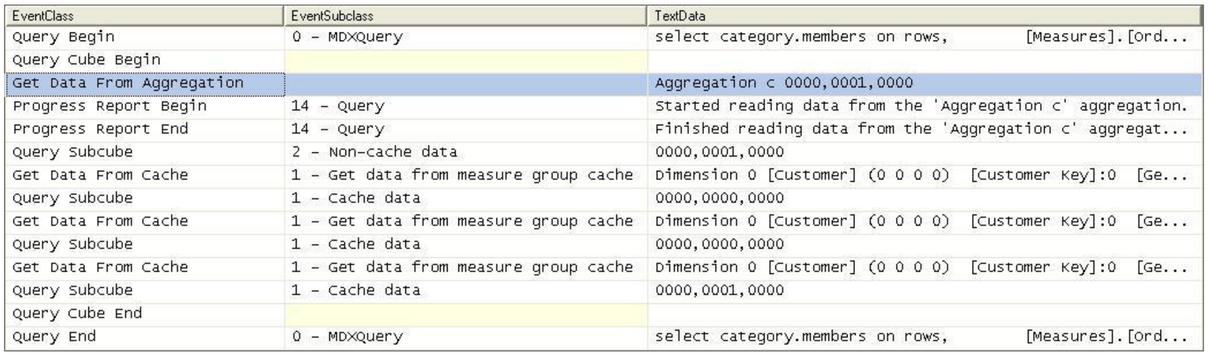
An *aggregation* is a data structure that stores precalculated data that Analysis Services uses to enhance query performance. You can define the aggregation design for each partition independently. Each partition can be thought of as being an aggregation at the lowest granularity of the measure group. 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.

3.4.1 Detecting Aggregation Hits

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.



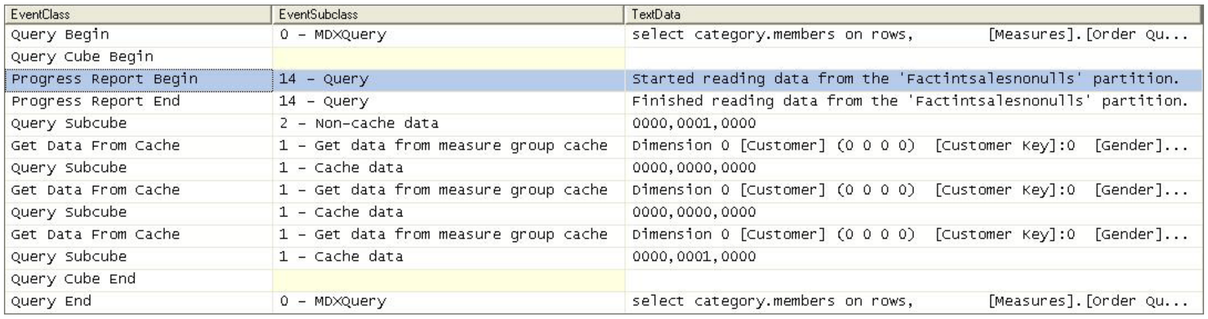
EventClass	EventSubclass	TextData
Query Begin	0 - MDXQuery	select category.members on rows, [Measures].[Ord...
Query Cube Begin		
Get Data From Aggregation		Aggregation c 0000,0001,0000
Progress Report Begin	14 - Query	Started reading data from the 'Aggregation c' aggregation.
Progress Report End	14 - Query	Finished reading data from the 'Aggregation c' aggregat...
Query Subcube	2 - Non-cache data	0000,0001,0000
Get Data From Cache	1 - Get data from measure group cache	Dimension 0 [Customer] (0 0 0 0) [Customer Key]:0 [Ge...
Query Subcube	1 - Cache data	0000,0000,0000
Get Data From Cache	1 - Get data from measure group cache	Dimension 0 [Customer] (0 0 0 0) [Customer Key]:0 [Ge...
Query Subcube	1 - Cache data	0000,0000,0000
Get Data From Cache	1 - Get data from measure group cache	Dimension 0 [Customer] (0 0 0 0) [Customer Key]:0 [Ge...
Query Subcube	1 - Cache data	0000,0001,0000
Query Cube End		
Query End	0 - MDXQuery	select category.members on rows, [Measures].[Ord...

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. 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.

When no aggregations can satisfy the query request, notice the missing **Get Data From Aggregation** event from the same cube with no aggregations as noted in the following figure.



EventClass	EventSubclass	TextData
Query Begin	0 - MDXQuery	select category.members on rows, [Measures].[Order Qu...
Query Cube Begin		
Progress Report Begin	14 - Query	Started reading data from the 'Factintsalesnonulls' partition.
Progress Report End	14 - Query	Finished reading data from the 'Factintsalesnonulls' partition.
Query Subcube	2 - Non-cache data	0000,0001,0000
Get Data From Cache	1 - Get data from measure group cache	Dimension 0 [Customer] (0 0 0 0) [Customer Key]:0 [Gender]...
Query Subcube	1 - Cache data	0000,0000,0000
Get Data From Cache	1 - Get data from measure group cache	Dimension 0 [Customer] (0 0 0 0) [Customer Key]:0 [Gender]...
Query Subcube	1 - Cache data	0000,0000,0000
Get Data From Cache	1 - Get data from measure group cache	Dimension 0 [Customer] (0 0 0 0) [Customer Key]:0 [Gender]...
Query Subcube	1 - Cache data	0000,0001,0000
Query Cube End		
Query End	0 - MDXQuery	select category.members on rows, [Measures].[Order Qu...

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.

3.4.2 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.

3.4.3 Aggregation Tradeoffs

Aggregations can improve query response time but they can increase processing time and disk storage space, use up memory that could 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.

3.4.4 Building Aggregations

Individual aggregations are organized into collections of aggregations called AggregationDesigns. You can apply an AggregationDesign to many partitions. As well, one measure group can 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 AggregationDesign. In this section we will discuss the following:

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

3.4.4.1 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. However, before you modify the **Aggregation Usage** property, you should consider whether you can take advantage of user hierarchies.

3.4.4.2 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; now it 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.

3.4.4.3 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.

3.4.4.4 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**— Every aggregation for the cube must include this attribute or a related attribute that is lower in the attribute chain. For example, 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 is 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 -** For all attributes (except All) in many-to-many, nonmaterialized reference dimensions, and data mining dimensions, default means **None**. 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 section.
- **Default Constraint 3**—**Unrestricted for Natural Hierarchies -** A natural hierarchy 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

In light of the behavior of the **Aggregation Usage** property, use the following guidelines:

- **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 the aggregation algorithm must consider. 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 five to ten **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 would only infrequently use it. Using this approach can help you reduce the aggregation space and get to the five to ten **Unrestricted** attributes per dimension. For example, you may have certain attributes that are only used by a few advanced users who are 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.

3.4.4.5 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. Use the Usage-Based Optimization Wizard with a 100% performance gain - this will 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 that when combined are redundant with one another. As well, aggregation designs have a costly metadata impact – don't overdesign but try to keep the number of aggregation designs per measure group to a minimum. Inspect the new aggregations compared to the old and ensure there are no near-duplicates. The aggregation design can be copied to other partitions in SQL Server Management Studio or Business Intelligence Design Studio.

References:

- Reintroducing Usage-Based Optimization in SQL Server 2008 Analysis Services (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/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/): This CodePlex project contains many useful Analysis Services CodePlex samples, including the Aggregation Manager

3.4.4.6 Large Cube Aggregation Considerations

It is important to note that small cubes may not need aggregations, because aggregations are not even 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 may not be the correct ones for large cubes.

However, 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.

It is important to note that 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. The result is a tradeoff in designing aggregations to be considered more carefully.

3.5 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.

3.5.1 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. This can be done by either pre-executing one or more queries or using the CREATE CACHE statement (which returns no cellsets and has the advantage of executing faster because it bypasses the query processor). This process is called *cache warming*.

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 there may be enough memory to keep a large portion of the data in the cache. In this case, aggregations are not needed and existing aggregations may never be used. In this scenario, cache warming can be used so that users will always have excellent performance.

But with larger cubes, there may be insufficient memory to keep enough of the data in cache. For that matter, cached results can be pushed out by other query results. Hence, cache warming will only help a portion of the queries—it is important to create well-designed aggregations to provide solid query performance. But because of the memory bottlenecks, it is important to note that too many aggregations may thrash the cache as different data resultsets and aggregations are requested and swapped from the cache.

3.5.2 Implementing a Cache Warming Strategy

While cache warming can improve the performance of a query, you should note that there is a significant difference between the performance of the query on a cold cache and a warm cache. As well, it is important to ensure there is enough memory available so that the cache is not being thrashed.

To warm the cache, it is important to remember that 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:

- 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.
- Finding many subcube requests to the same grain may indicate that the query processor is making many requests for slightly different data, resulting in the storage engine making many small but time-consuming I/O requests where it could more efficiently retrieve the data *en masse* and then return results from cache.
- To pre-execute queries, create an application (or use something like **ascmd**) that executes a set of generalized queries to simulate typical user activity in order to expedite the process of populating the cache. Execute these queries post-Analysis Services startup or post-processing to preload the cache prior to user queries.
  To determine how to generalize your queries, you can potentially refer to the Analysis Services query log to determine the dimension attributes typically queried. Be careful when you generalize because you may include attributes or subcubes that are not beneficial and unnecessarily take up cache.
- When testing the effectiveness of different cache-warming queries, you should 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, it may be necessary to schedule refreshes of the cache results. Also, limit cache warming to what can fit in memory, leaving enough for other queries to be cached.

References:

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

3.6 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, but the basic principle is that you have multiple query servers aimed at the same database (or the database is replicated) so there are multiple servers to address user queries. This can be beneficial in the cases like the following:

- In cases where 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, including (but not limited to):
    - Users executing many different unique queries thus filling up and thrashing available cache.
    - Complex or large queries requiring large subcubes thus requiring a large memory space.
    - Too many concurrent users accessing the same server.
- You have many long running queries against your Analysis Services cube, which will:
    - Block other queries.
    - 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/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/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/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/sqlcat/b/whitepapers/archive/2007/11/19/scale-out-querying-with-analysis-services-using-san-snapshots.aspx)

4 Tuning Processing Performance

In the following sections we will provide guidance on tuning cube 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 impacts 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 1) the object that you are processing, such as a dimension, cube, measure group, or partition, and 2) 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 the technical note Analysis Services 2005 Processing Architecture (http://msdn.microsoft.com/en-us/library/ms345142(SQL.90).aspx).

4.1 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 the baseline.

4.1.1 Performance Monitor Trace

Windows performance counters are the bread and butter of performance tuning Analysis Services. Use the tool **perfmon** to set up a trace with 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 measure these counters again 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.

4.1.2 Profiler Trace

To optimize the SQL queries that form part of processing, you should trace the relational database too. If the relational database is SQL Server, you use SQL Server Profiler for this. If you are not using SQL Server, consult your database vendor or DBA for help on tuning the database. In the following 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 also capture the 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**. Like the **perfmon** trace, configure the trace to save to a file for later analysis.

Configure your SQL Server Profiler trace to log to a table instead of a file. This makes it easier to correlate the traces later.

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

4.1.3 Determining Where You Spend Processing Time

To properly target the tuning of processing, you should first determine where you are spending your time: partition processing or dimension processing.

To assist with tuning and future monitoring, it is useful to split the dimension processing and partition processing into two different commands in the processing, to tune each 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 **perfmon** 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 operations included in a full cube processing.

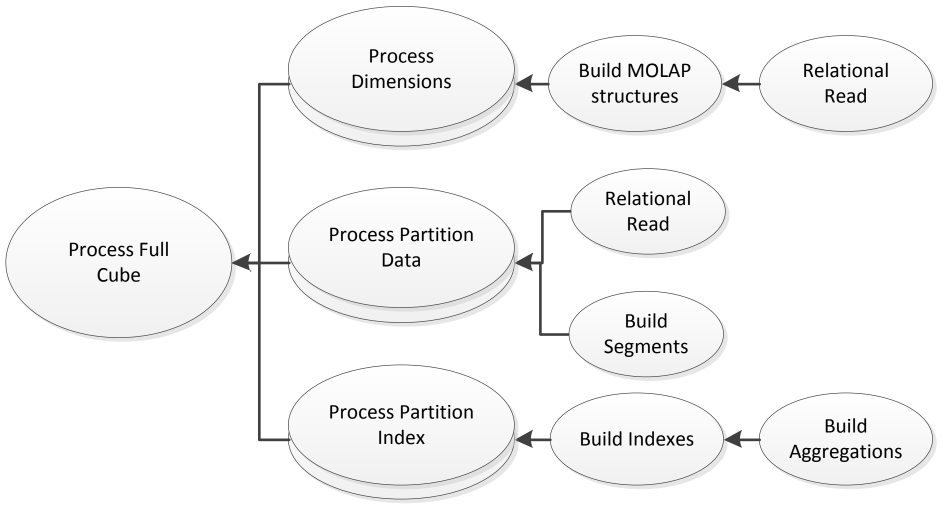


Figure : Full cube processing overview

4.2 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.

4.2.1 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.

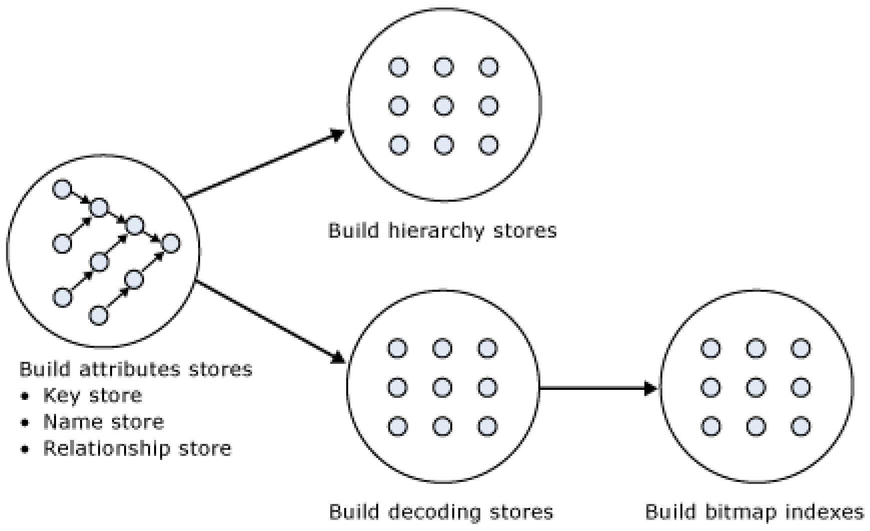


Figure : Dimension processing jobs

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 build 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, an ordering of the processing jobs is required. To provide the correct workflow, the storage engine analyzes the dependencies between attributes, and then it creates an execution tree with the correct ordering. The execution tree is then used to determine the best parallel execution of the dimension processing.

Figure : 20 displays an example execution tree for a Time dimension. The solid arrows represent the attribute relationships in the dimension. The dashed arrows represent the implicit relationship of each attribute to the All attribute.

Note: The dimension has been configured using cascading attribute relationships, which is a best practice for all dimension designs.

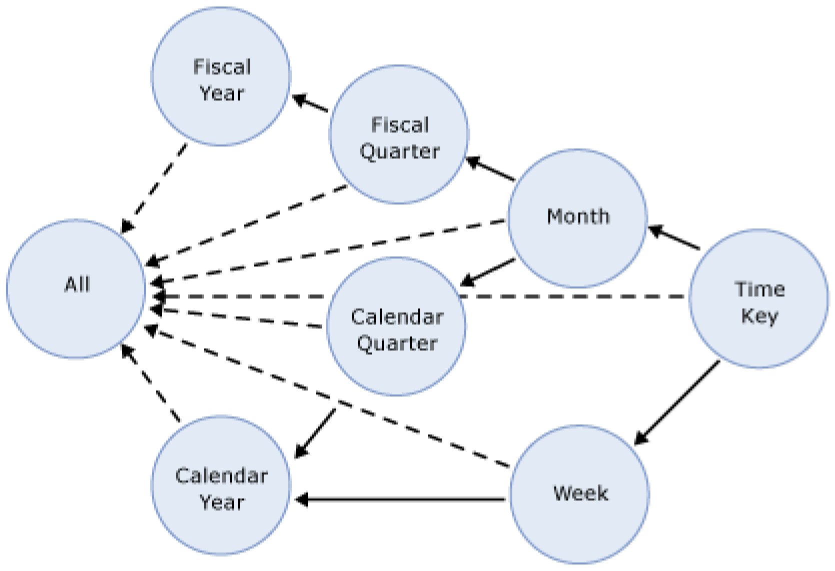


Figure : Execution tree example

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 1) the number of members and 2) 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. Attribute relationships lower 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 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.

4.2.2 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.

4.2.2.1 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.

4.2.2.2 ProcessAdd

Note that **ProcessAdd** is only available as an XMLA command and not from SQL Server Management Studio. **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** typically triggers a processing error when run, reporting duplicate key values. This 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.

Note that you cannot run a **ProcessAdd** on an empty dimension. The dimension must first be fully processed.

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 set the KeyDuplicate, see this forum thread: http://social.msdn.microsoft.com/Forums/en-US/sqlanalysisservices/thread/8e7f1304-56a1-467e-9cc6-68428bd92aa6?prof=required

4.3 Tuning Cube Dimension Processing

In section 2, we described how to create a good and high-performance dimension design. In SQL Server 2008 and SQL Server 2008 R2 Analysis Services, the Analysis Management Objects (AMO) warnings are provided by Business Intelligence Development Studio to assist you with following these best practices.

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.

4.3.1 Reduce Attribute Overhead

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.

4.3.1.1 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, you are creating a sales fact and users always query both date and store dimensions. Sometimes a filter is also applied by the store clerk dimension, but because you have already filtered down to stores, adding a bitmap on the store clerk may only yield a trivial benefit. In this case, you can consider disabling bitmap indexes on store clerk attributes.

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

4.3.1.2 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. Without 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.

4.3.2 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. Dimensions tend to be small, complex tables with very few changes, compared to facts that 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 be wasteful with the relational indexes.

To quickly tune the relational queries used for dimension processing you can use the Database Engine Tuning Advisor on a profiler trace of the dimension processing. For the small dimension tables, chances are that you can get away with adding every suggested index. For the 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.

4.3.2.1 Using ByTable Processing

By setting the **ProcessingGroup** property of the dimension to be **ByTable** you will change how Analysis Services behaves during dimension processing. Instead of sending multiple SELECT DISTINCT queries, the processing task instead requests the entire table with one query. If you have enough memory to hold all the new dimension data while processing is happening, this option can provide a fast way to optimize processing. However, you should be careful about this setting – if Analysis Services runs out of memory during processing, this 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.

4.4 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.

The following techniques for accomplishing this goal are discussed in this section: optimizing SQL source queries and using a partitioning strategy (both in the cube and the relational database) to optimize processing. For detailed guidance on server tuning, hardware optimization and relational indexing, see the SQL Server 2008 R2 Operations Guide.

4.4.1 Partition Processing Architecture

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

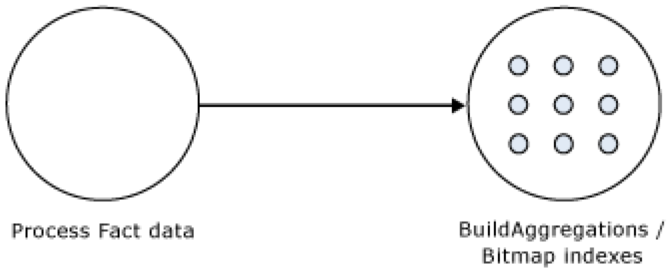


Figure : Partition processing jobs

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.

4.4.2 Partition-Processing Commands

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

The following partition processing commands are available:

- **ProcessFull**
- **ProcessData**
- **ProcessIndexes**
- **ProcessAdd**
- **ProcessClear**
- **ProcessClearIndexes**

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

ProcessData discards the storage contents of the object and rebuilds only the fact data.

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

ProcessAdd internally creates a temporary partition, processes it with the target fact data, and then merges it with the existing partition. Note that **ProcessAdd** is the name of the XMLA command, in Business Intelligence Development Studio and SQL Server Management Studio this is exposed as **ProcessIncremental.**

ProcessClear removes all data from the partition. Note the **ProcessClear** is the name of the XMLA command. In Business Intelligence Development Studio and SQL Server Management Studio, it is exposed 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 Business Intelligence Development Studio and SQL Server Management Studio.

4.4.3 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/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/sqlcat/b/whitepapers/archive/2007/11/15/analysis-services-processing-best-practices.aspx)

4.4.4 Optimizing Data Inserts, Updates, and Deletes

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

4.4.4.1 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. **ProcessAdd** internally creates a temporary partition and merges it. This results in data fragmentation over time and the need to periodically perform **ProcessFull**.

If your measure group contains multiple partitions, as described in the previous section, 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 impacting the existing partitions. When the new partition has completed processing, it is available for querying.

4.4.4.2 Updates

When you need to perform data updates, you can perform a **ProcessFull**. Of course it is useful if you can target the updates to a specific partition so 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 this scenario, 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.

4.4.4.3 Deletes

For deletions, multiple partitions provide a great mechanism for you to roll out expired data. Consider the following example. You currently have 13 months of data in a measure group, 1 month per partition. You want to roll out the oldest month from the cube. To do this, you can simply delete the partition without affecting any of the other partitions.

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

4.4.5 Picking Efficient Data Types in Fact Tables

During processing, data has to be moved out of SQL Server and into Analysis Services. The wider your rows are, the more bandwidth must be spent moving 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*)*

4.4.6 Tuning the Relational Partition Processing Query

During the **ProcessData** phase, rows are read from a relational source and 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 the query. 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.

4.4.6.1 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 send to the database. You can achieve this by denormalizing the joined columns to the fact table. If you are using a star schema design, you should already have done this.

References

- For background on relational star schemas and how to design and denormalize for optimal performance, refer to: Ralph Kimball, *The Data Warehouse Toolkit.*

4.4.6.2 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.

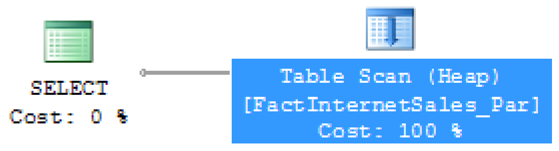


Figure : An optimal partition processing query

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

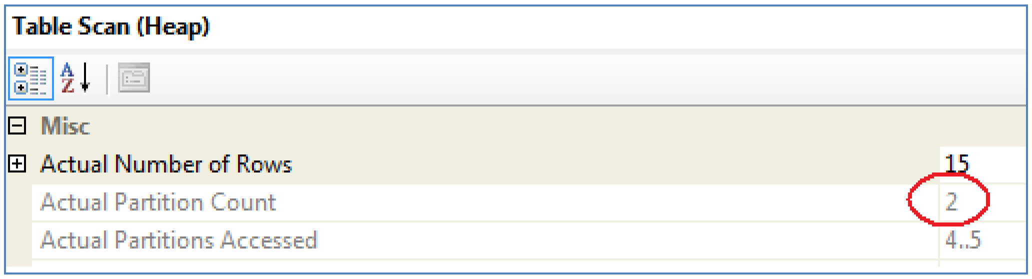


Table Scan (Heap)	
Misc	
Actual Number of Rows	15
Actual Partition Count	2
Actual Partitions Accessed	4..5

Figure : Too many partitions accessed

Both partition 4 and partition 5 are accessed. The value for **Actual Partition Count** should be 1. If this is not the case (as in the figure), you should consider repartitioning the relational source data so that each cube partition touches at most one relational partition.

4.4.7 Splitting Processing Index and Process Data

It is good practice to split partition processing into its components: **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 may just opt for running a **ProcessFull** on the measure group. For small cubes where performance is not a concern, this will work well.

4.4.8 Increasing Concurrency by Adding More Partitions

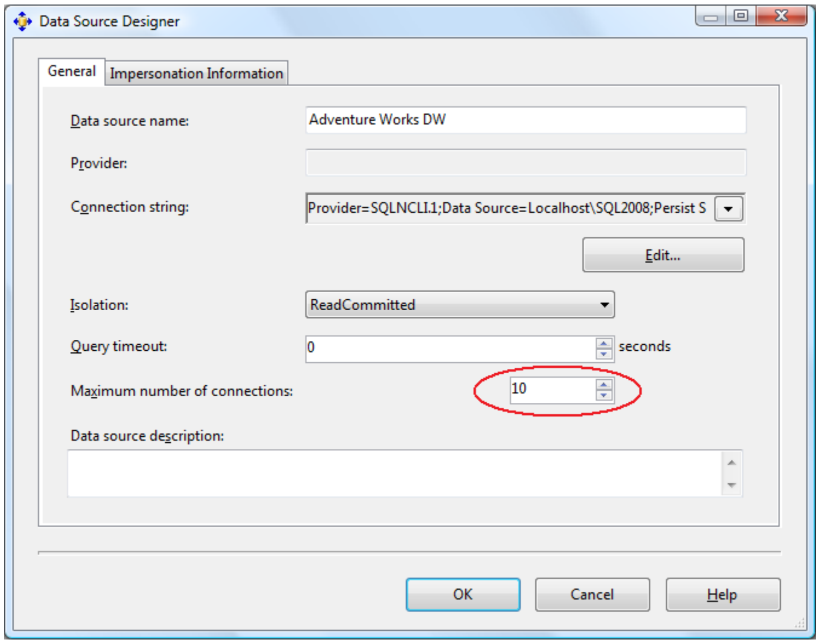
If your tuning is bound only by the amount of CPU power you have (as opposed to I/O, for example), you should optimize to make the best use of the CPU cores available to you. It is time to have a look at the **Processor:Total** counter from the baseline trace. If this counter is not 100%, you are not taking full advantage of your CPU power. As you continue the 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 perfmon 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. Keep in mind that 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.

4.4.9 Adjusting Maximum Number of Connections

When you increase parallelism of the processing above 10 concurrent partitions, you will need to adjust the maximum number of connections that Analysis Services keeps open on the database. This number can be changed in the properties of the data source (the **Maximum number of connections** box).



Data source name: Adventure Works DW
Provider:
Connection string: Provider=SQLNCLI.1;Data Source=Localhost\SQL2008;Persist S
Isolation: ReadCommitted
Query timeout: 0 seconds
Maximum number of connections: 10
Data source description:

Figure : Adding more database connections

Set this number to at least the number of partitions you want to process in parallel.

4.4.10 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 key figure you 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 important counter to look at is the temporary files counter – when an aggregation doesn't fit in memory, the aggregation data is spilled to temporary disk files. Building disk based aggregations is much more expensive, and if you notice this you may be able to find a way to either allow more memory to be available for the index building phase, or drop some of the larger aggregations to avoid this issue.

4.4.10.1 Add Partitions to Increase Parallelism

As was the case with **ProcessData**, processing more partitions in parallel can speed up **ProcessIndex**. The same tuning strategy applies: Keep increasing partition count until you no longer see an increase in processing speed.

4.4.11 Partitioning the Relational Source

The best partition strategy to implement in the relational source varies by database product capabilities, 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.

The following figures illustrate some examples of partition strategies you should consider.

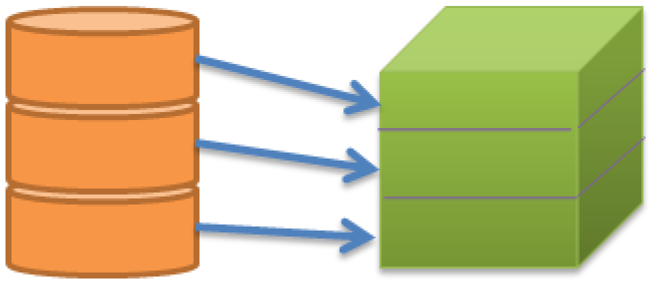


Figure : Matching partition strategies

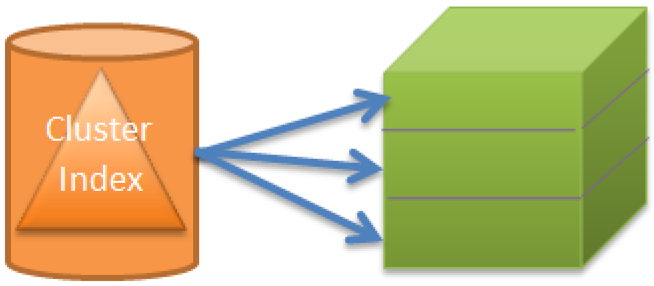


Figure : Clustering the relational table

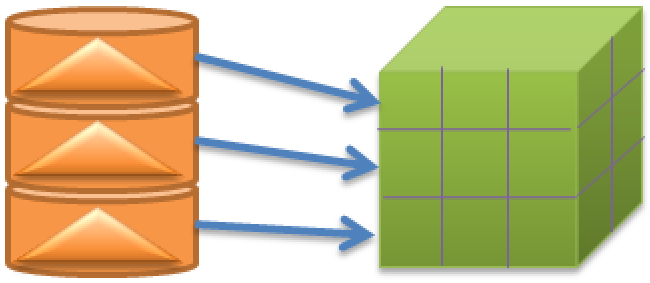


Figure : Supporting matrix partitioning with a combination of relational layouts

5 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.

5.1 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.

5.1.1 Partition Design

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

If each partition contains nonoverlapping range of values, this coordination between jobs is avoided and query performance can improve by orders of magnitude, depending on hardware! As well, there are a number of 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 and 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 your partitions with non-overlapping ranges, considering building a *hash of the distinct value*. A modulo function is simple and straightforward but it requires extra processing (for example, convert character key to integer values) and storage (for example, 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 (that is, 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 is against 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*.

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

5.1.2 Processing of Distinct Count

Distinct count measure groups have special requirements for partitioning. Normally, you use time and potentially some other dimension as the partitioning column (see the section on Partitioning earlier in this guide). 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.

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

Starting with SQL Server 2005 Analysis Services, it is possible to use noninteger columns for distinct count measure groups. However, for performance reasons (and the potential to hit the 4-GB limit) you should avoid this. The white paper Analysis Services Distinct Count Optimization (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2010/09/20/analysis-services-distinct-count-optimization-using-solid-state-devices.aspx) 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.

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.

5.1.3 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 also including the value that needs to be counted. 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.

5.1.4 Optimize the Disk Subsystem for Random I/O

As noted in the beginning of this section, distinct count queries have a heavy impact on the Analysis Services storage engine, which for large cubes means there is a large impact on the disk subsystem. For each query, Analysis Services generates potentially multiple processes—each one parsing the disk subsystem to perform a portion of the distinct count calculation. This 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 overall).

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/sqlcat/b/whitepapers/archive/2011/06/01/sql-server-2008r2-analysis-services-operations-guide.aspx)
- Analysis Services Distinct Count Optimization (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/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)

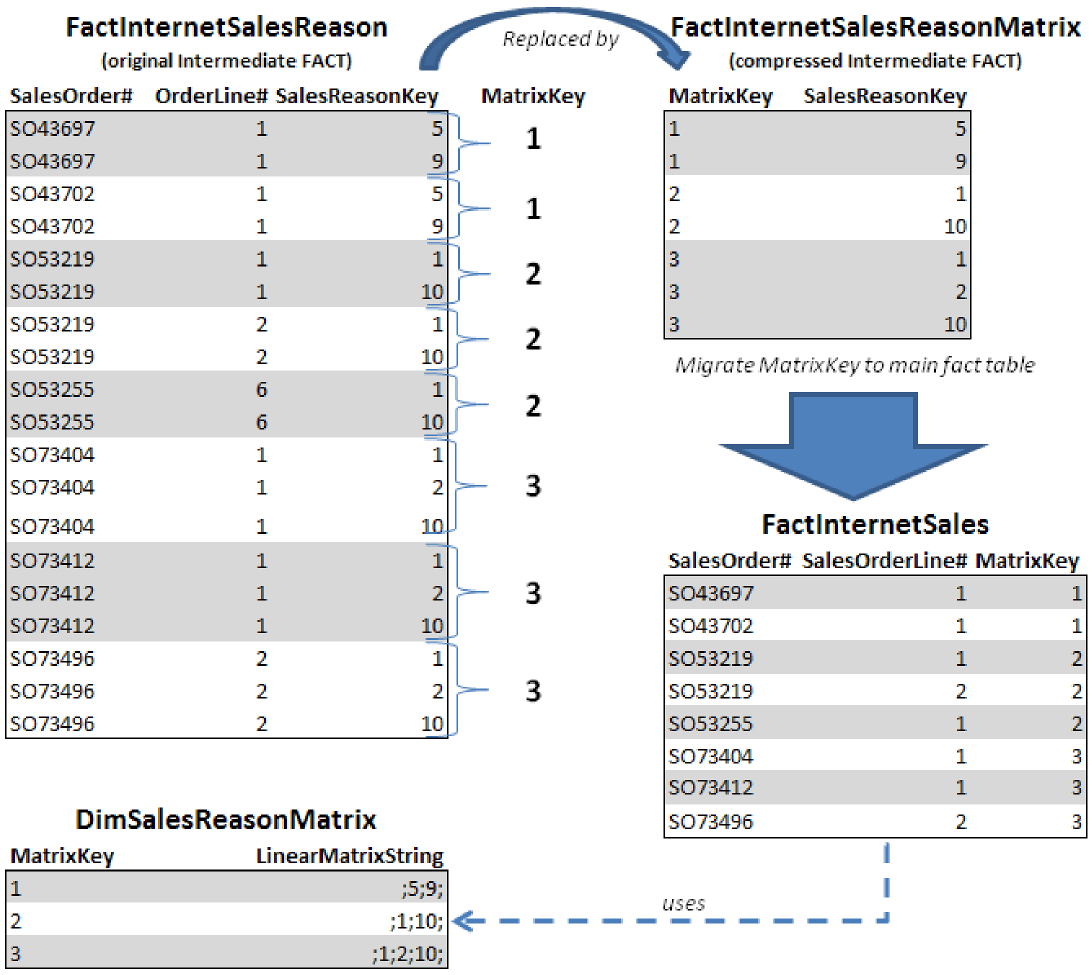
5.2 Large Many-to-Many Dimensions

Many-to-many relationships are used heavily in many business scenarios ranging from sales to accounting to healthcare. But at times there may be query performance issues when dealing with a large number of many-to-many relationships and perceived accuracy issues. One way to think about a many-to-many dimension is that it is a generalization of the distinct count measure. The use of many-to-many dimensions enables you to apply distinct count logic to other Analysis Services measures such as sum, count, max, min, and so on. To calculate these distinct count or aggregates, 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.

Performance and accuracy issues concerning many-to-many dimensions include the following:

- 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, reducing the size of your intermediate measure group (a general rule is less than 1 million rows) provides the best performance. For additional techniques, see the Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques white paper (http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=137).
- Many-to-many relationships cannot be aggregated (although it generally is not very easy to create general purpose aggregates for distinct counts as well). Therefore, queries involving many-to-many dimensions cannot use aggregations or aggregate caches—only a direct hit will work. There are specific situations where many-to-many relationships can be aggregated; you can find more information in the Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques white paper.
    - Because many-to-many cannot be aggregated, there are various MDX calculation issues with VisualTotals, subselects, and CREATE SUBCUBE.
- Similar to distinct count, 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, which removes repeated many-to-many relationships thus reducing the size of your intermediate measure group. As can be seen in the following figure, a *MatrixKey* is created based on each set of common dimension member combinations so that repeated combinations are eliminated.



FactInternetSalesReason
(original Intermediate FACT)

SalesOrder#	OrderLine#	SalesReasonKey	MatrixKey
SO43697	1	5	1
SO43697	1	9	
SO43702	1	5	1
SO43702	1	9	
SO53219	1	1	2
SO53219	1	10	
SO53219	2	1	2
SO53219	2	10	
SO53255	6	1	2
SO53255	6	10	
SO73404	1	1	3
SO73404	1	2	
SO73404	1	10	
SO73412	1	1	3
SO73412	1	2	
SO73412	1	10	
SO73496	2	1	3
SO73496	2	2	
SO73496	2	10	

FactInternetSalesReasonMatrix
(compressed Intermediate FACT)

MatrixKey	SalesReasonKey
1	5
1	9
2	1
2	10
3	1
3	2
3	10

FactInternetSales

SalesOrder#	SalesOrderLine#	MatrixKey
SO43697	1	1
SO43702	1	1
SO53219	1	2
SO53219	2	2
SO53255	1	2
SO73404	1	3
SO73412	1	3
SO73496	2	3

DimSalesReasonMatrix

MatrixKey	LinearMatrixString
1	;5;9;
2	;1;10;
3	;1;2;10;

Figure : Compressing the FactInternetSalesReason intermediate fact table (from Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques)

References:

- Many-to-Many Matrix Compression (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)
- Analysis Services: Should you use many-to-many dimensions? (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2008/02/11/analysis-services-should-you-use-many-to-many-dimensions.aspx)

5.3 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, the query performance tends to be suboptimal, especially for large parent-child dimensions, because aggregations are created only for the key attribute and the top attribute (that is, the **All** attribute) unless it is disabled. Therefore, a common best practice is to refrain from using parent-child hierarchies that contain a large number of members. (How big is large? There isn't a specific number because query performance at intermediate levels of the parent-child hierarchy degrades linearly with the number of members.) Additionally, limit the number of parent-child hierarchies in your cube.

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.

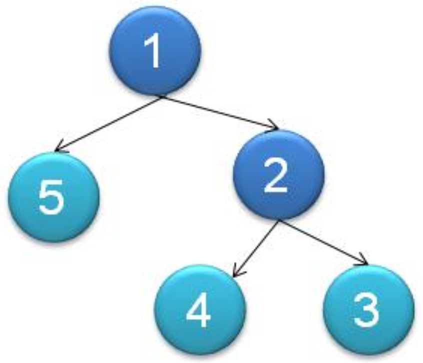


Figure : Sample parent-child hierarchy

The data from this parent-child hierarchy is represented in relational format as per 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 help convert your parent-child hierarchy into a regular hierarchy, refer to the Analysis Services Parent-Child Dimension Naturalizer tool in CodePlex (http://pcdimnaturalize.codeplex.com/wikipage?title=Home&version=12&ProjectName=pcdimnaturalize).

References:

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

5.4 Near Real Time and ROLAP

As your Analysis Services data becomes more valuable to the business, a common next requirement is to provide near real-time capabilities so users can have immediate access to their business intelligence system. Near real-time data has special requirements:

- 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.

It is important to note that due to the locking logic invoked by Analysis Services, long-running queries in Analysis Services can both prevent processing from committing and block other queries.

To provide near real-time results and avoid the Analysis Services query locking, start with using ROLAP so that the queries go directly to the relational database. Yet even relational databases have locking and/or concurrency issues that need to be dealt with. To minimize the impact of blocking queries within your relational database, place the real-time portion of the data into its own separate table but keep historical data within your partitioned table. After you have done this, you can apply other techniques. In this section we discuss the following:

- MOLAP switching
- ROLAP + MOLAP
- ROLAP partitioning

5.4.1 MOLAP Switching

The basic principle behind MOLAP switching is to create some partitions for historical data and another set of partitions for the latest data. The latencies associated with frequently processing the current

MOLAP partitions are in 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. 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 two copies of the same cube, alternating data processing between them (known as cube flipping).

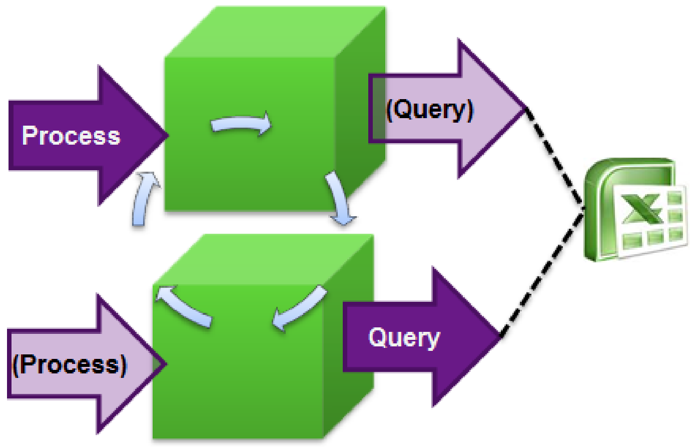


Figure : Cube-flipping concept

While one cube processes data, the other cube is available for querying. To flip between the cubes, you can use the Analysis Services Load Balancing Toolkit (http://sqlcat.com/sqlcat/b/toolbox/archive/2010/02/08/aslb.aspx) or create your own custom plug-in to your UI (you can use Excel to do this, for example) 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.

5.4.2 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 (RSCI); for more information, see Bulk Loading Data into a Table with Concurrent Queries (http://sqlcat.com/sqlcat/b/technicalnotes/archive/2009/04/06/bulk-loading-data-into-a-table-with-concurrent-queries.aspx).

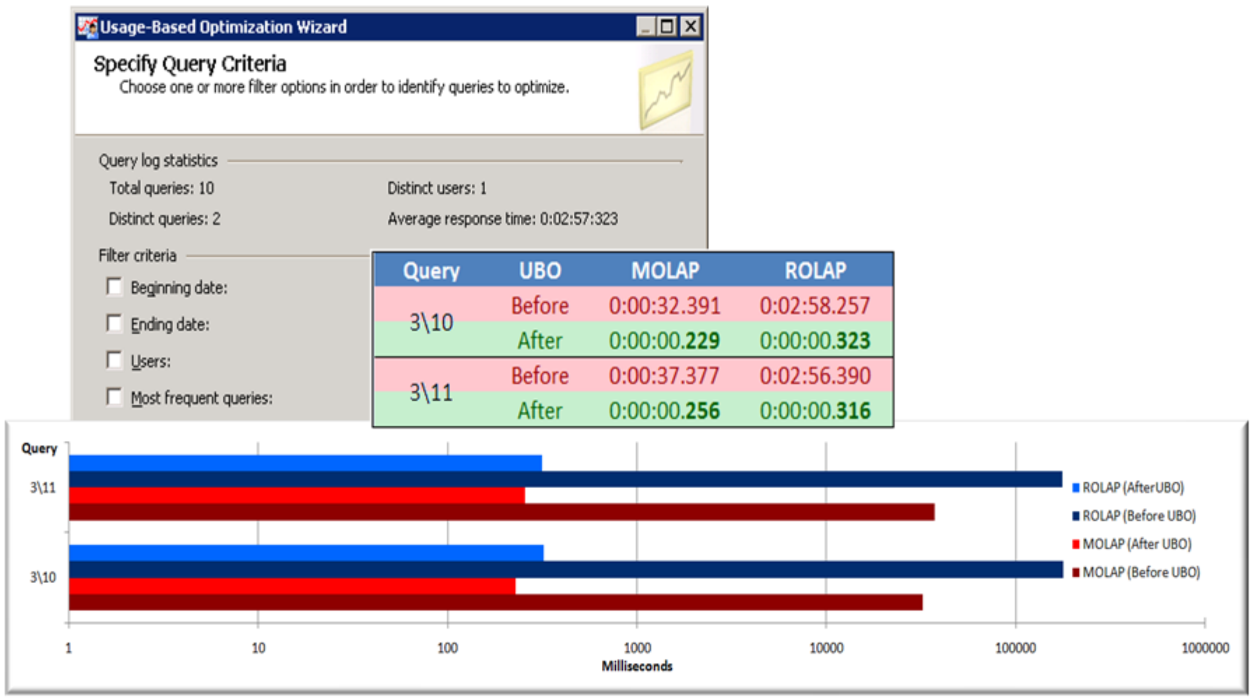
5.4.3 Comparing MOLAP Switching and ROLAP + MOLAP

The following table compares 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

5.4.4 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), it 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 the benefit of reducing or even removing the processing time of large cubes. As noted earlier, it is often a requirement if you need to have near real-time cubes. As can be seen in the following figure, the query performance of a ROLAP cube after usage-based optimization is applied can be comparable to MOLAP if the system is expertly tuned.



Usage-Based Optimization Wizard

Specify Query Criteria

Choose one or more filter options in order to identify queries to optimize.

Query log statistics

Total queries: 10 Distinct users: 1

Distinct queries: 2 Average response time: 0:02:57:323

Filter criteria

☐ Beginning date:

☐ Ending date:

☐ Users:

☐ Most frequent queries:

Query	UBO	MOLAP	ROLAP
3\10	Before	0:00:32.391	0:02:58.257
	After	0:00:00.229	0:00:00.323
3\11	Before	0:00:37.377	0:02:56.390
	After	0:00:00.256	0:00:00.316

Figure : Showcasing ROLAP vs. MOLAP performance before and after the application of usage-based optimization

5.4.4.1 ROLAP Design Recommendations

The recommendations for high performance querying of ROLAP cubes are listed here:

- 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 on).
- 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.

5.4.4.2 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. Often the most effective strategy is a combination of these two approaches as noted in the following table .

Aggregation Type	Advantages	Disadvantages
Cube-based	**Efficient query processing**: Analysis Services can use cube-based	**Processing overhead**: Analysis Services drops and re-creates indexed views

	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.	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.
Transparent	**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.	**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.

Here are some general rules:

- 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.

5.4.4.3 Limitations of ROLAP Aggregations

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

- You may have to design using table binding (and not query binding) to an actual table instead of a partition. The goal of this guidance is to ensure partition elimination.
    - This advice 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.
- Relational partition elimination will generally not work:
    - Normally, DW best practice is to use partitioned fact tables.
    - 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 generate bad SQL plans. This may vary depending on the relational engine you use.
- You cannot use:
    - A named query or a view in the DSV.
    - Any feature that will cause Analysis Services to generate a subquery. For example, you cannot use a Count of Rows measure, because a subquery is always generated when this type of measure is used.
- The measure group cannot have:
    - Any measures that use Max or Min aggregation.
    - Any 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 white paper Analysis Services ROLAP for SQL Server Data Warehouses (http://sqlcat.com/sqlcat/b/whitepapers/archive/2010/08/23/analysis-services-rolap-for-sql-server-data-warehouses.aspx).

6 Conclusion

This document provides the means to diagnose and address SQL Server 2008 Analysis Services processing and query performance issues.

For more information, see:

http://sqlcat.com/: SQL Customer Advisory Team

http://www.microsoft.com/sqlserver/: SQL Server Web site

http://technet.microsoft.com/en-us/sqlserver/: SQL Server TechCenter

http://msdn.microsoft.com/en-us/sqlserver/: SQL Server DevCenter

If you have any suggestions or comments, please do not hesitate to contact the authors. You can reach Thomas Kejser at tkejser@microsoft.com and Denny Lee at dennyl@microsoft.com.

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback.