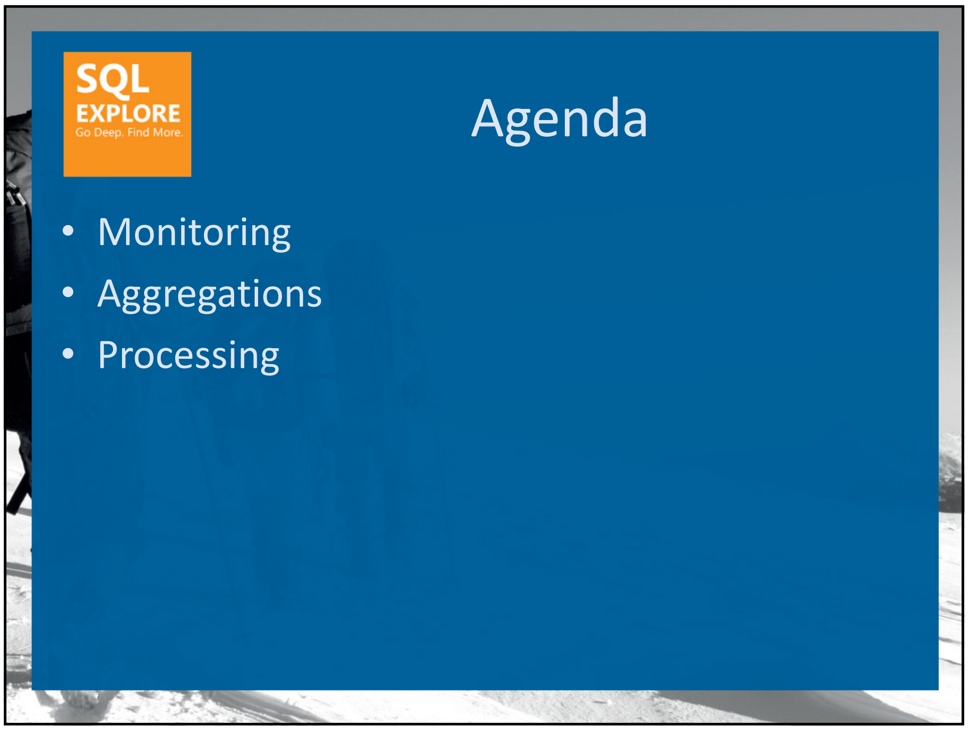
Extreme Analysis Service

SQL EXPLORE
Go Deep. Find More.

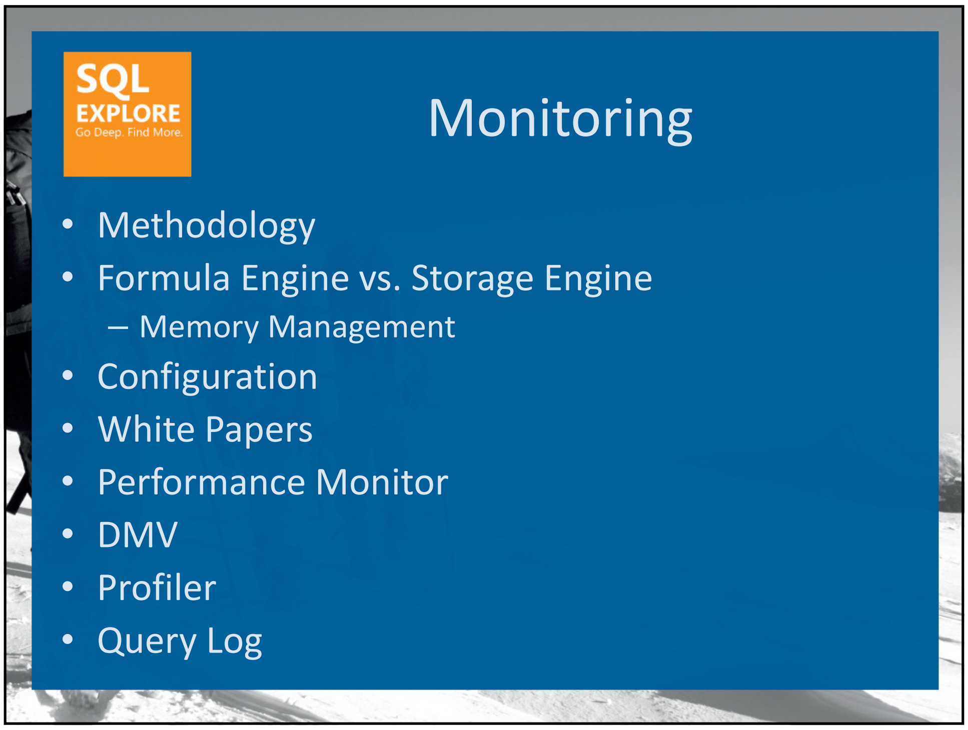
Itay Braun
CTO Twingo



SQL EXPLORE
Go Deep. Find More.

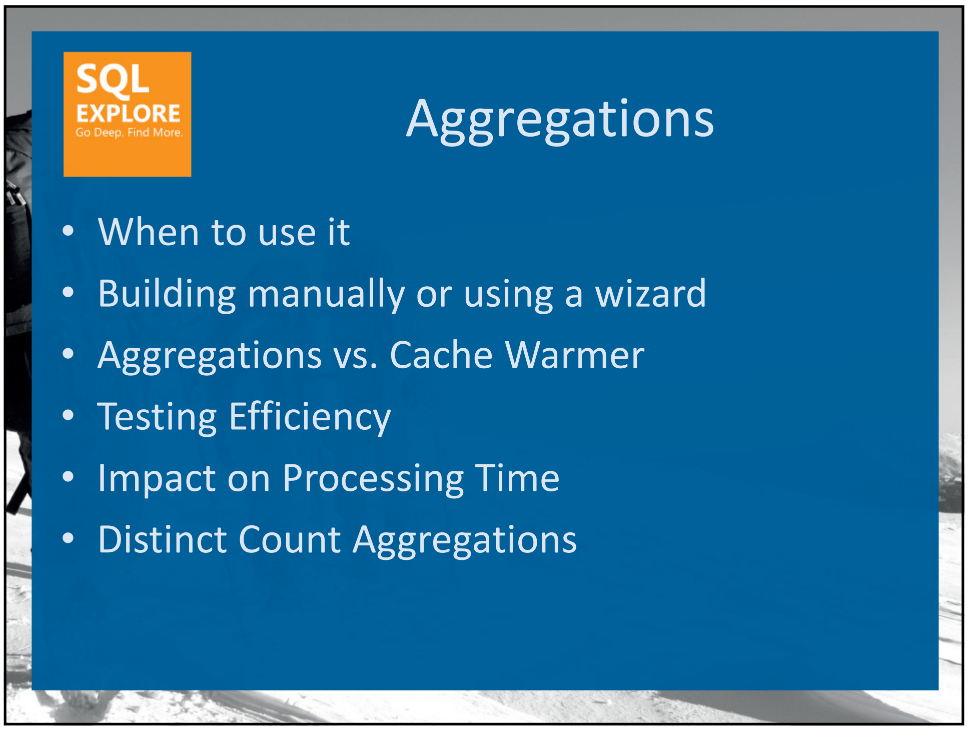
Agenda

- Monitoring
- Aggregations
- Processing



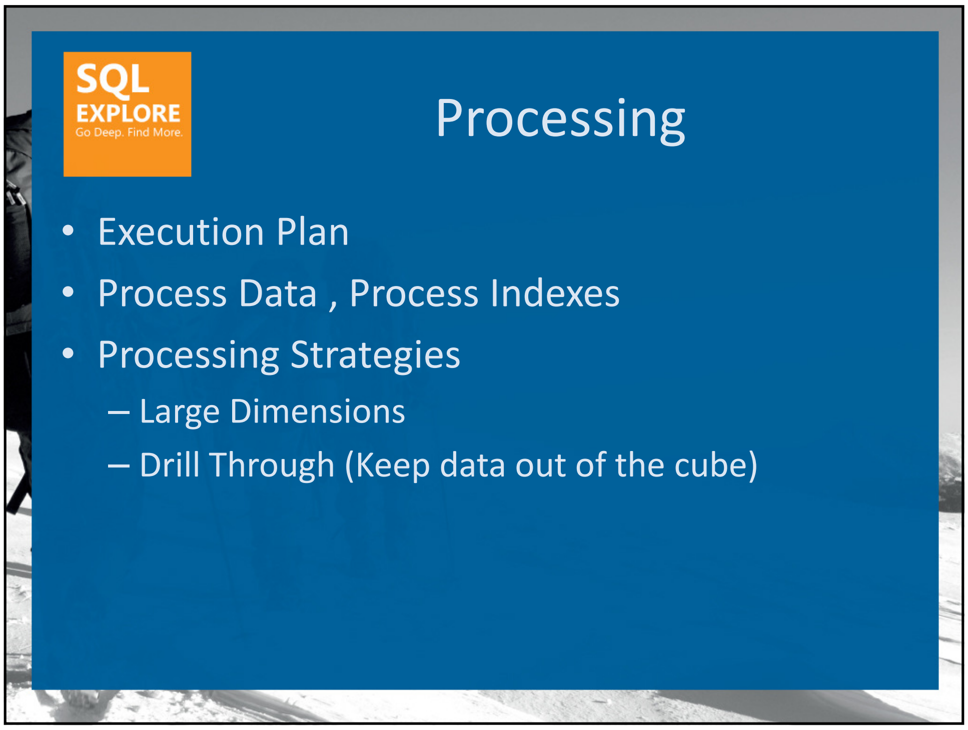
Monitoring

- Methodology
- Formula Engine vs. Storage Engine
  - Memory Management
- Configuration
- White Papers
- Performance Monitor
- DMV
- Profiler
- Query Log



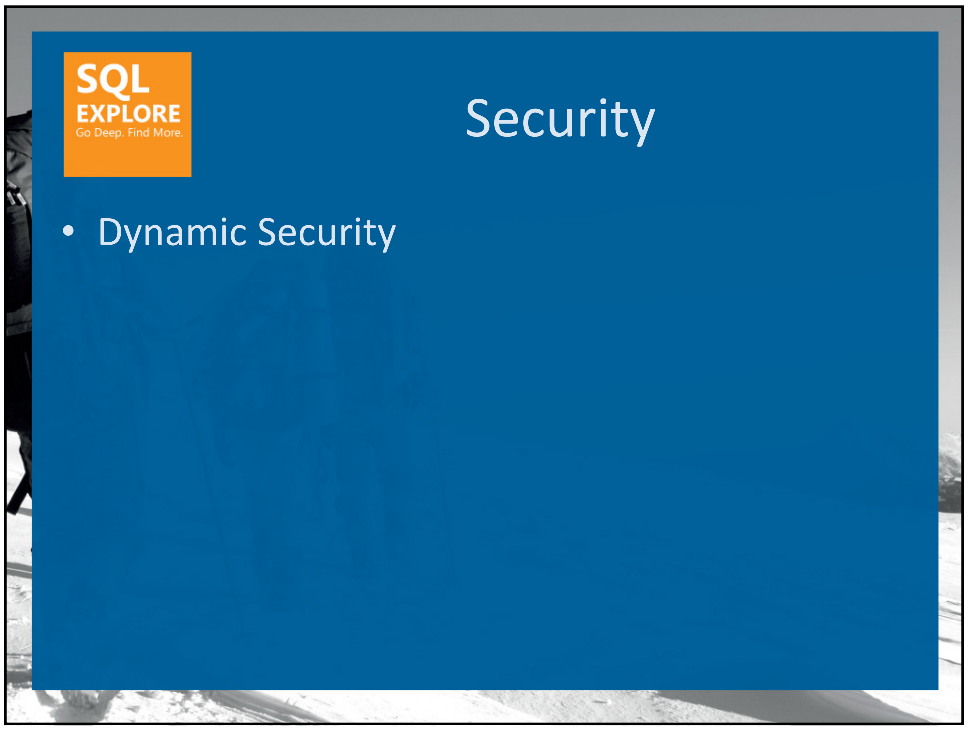
Aggregations

- When to use it
- Building manually or using a wizard
- Aggregations vs. Cache Warmer
- Testing Efficiency
- Impact on Processing Time
- Distinct Count Aggregations



Processing

- Execution Plan
- Process Data , Process Indexes
- Processing Strategies
  - Large Dimensions
  - Drill Through (Keep data out of the cube)



Security

- Dynamic Security

SQL
EXPLORE
Go Deep. Find More.

see you
Next Year
at SQL Explore

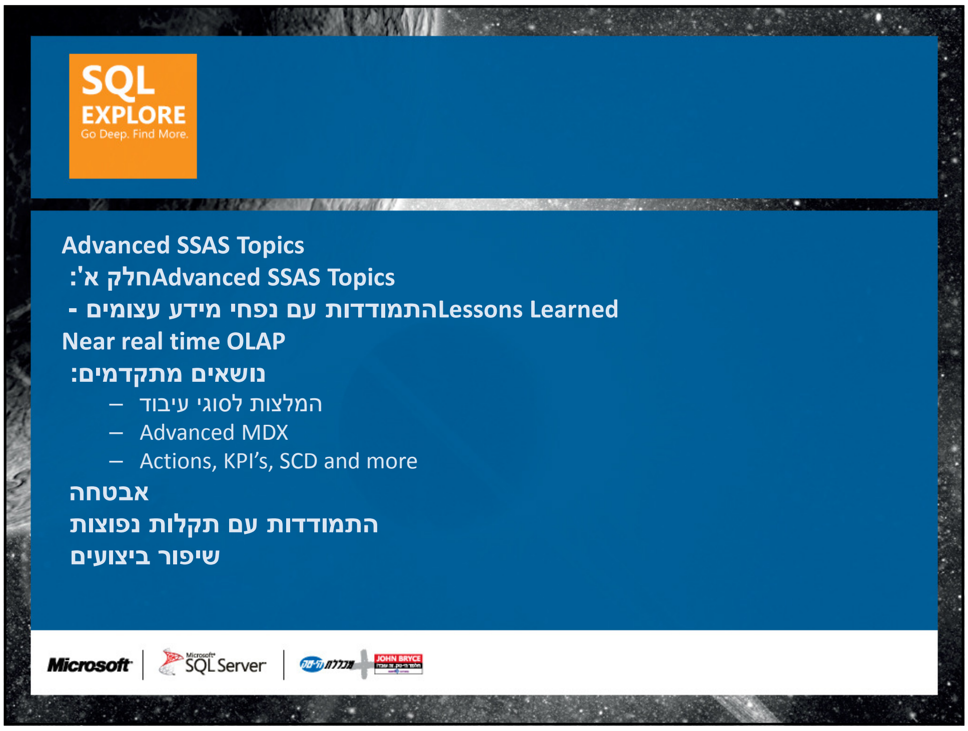
Microsoft | Microsoft SQL Server | JOHN BRYCE

SQL EXPLORE
Go Deep. Find More.

Extreme Analysis Service

Itay Braun
CTO Twingo

Microsoft | SQL Server | John Bryce



SQL EXPLORE
Go Deep. Find More.

Advanced SSAS Topics

חלק א': Advanced SSAS Topics

Lessons Learned - התמודדות עם נפחי מידע עצומים

Near real time OLAP

נושאים מתקדמים:

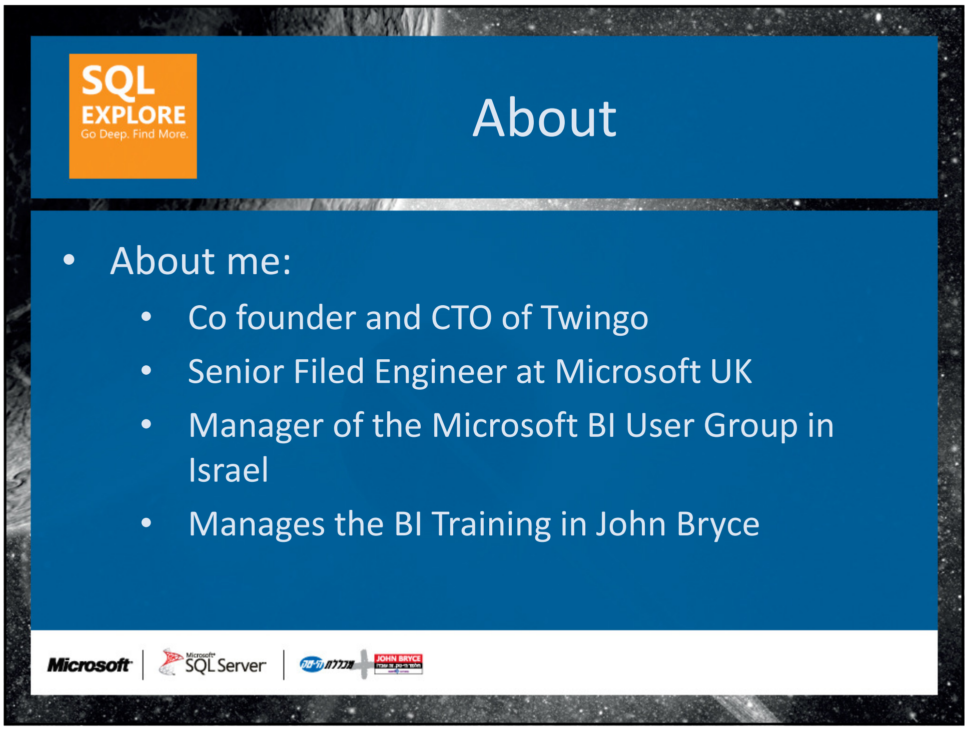
- המלצות לסוגי עיבוד
- Advanced MDX
- Actions, KPI’s, SCD and more

אבטחה

התמודדות עם תקלות נפוצות

שיפור ביצועים

Microsoft | SQL Server | John Bryce

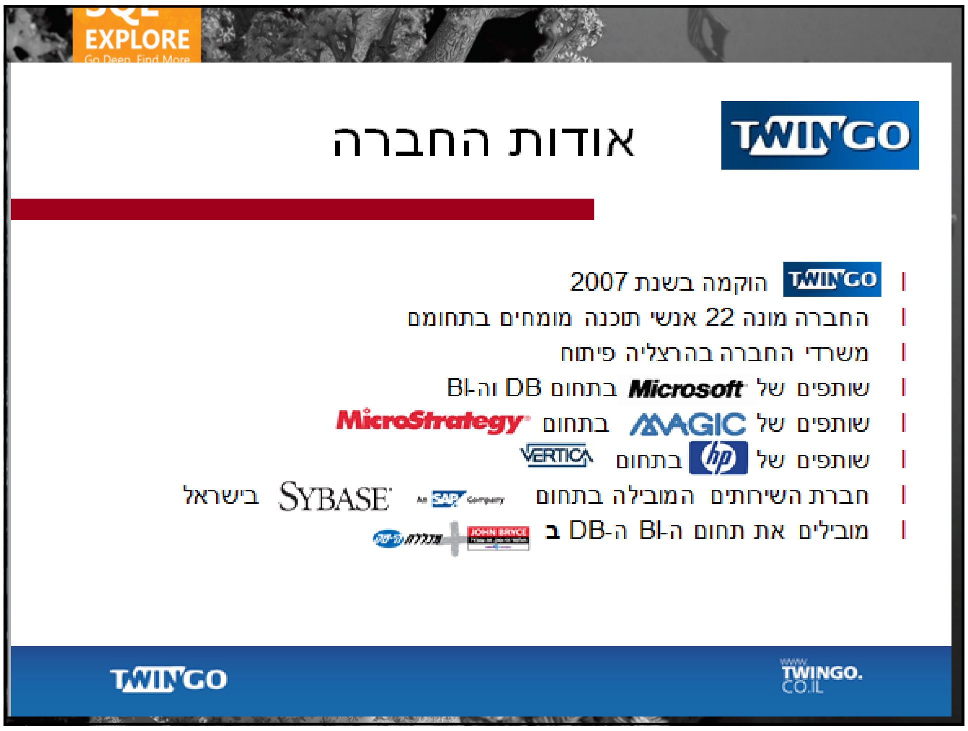


SQL EXPLORE Go Deep. Find More.

About

- About me:
  - Co founder and CTO of Twingo
  - Senior Filed Engineer at Microsoft UK
  - Manager of the Microsoft BI User Group in Israel
  - Manages the BI Training in John Bryce

Microsoft | SQL Server | John Bryce



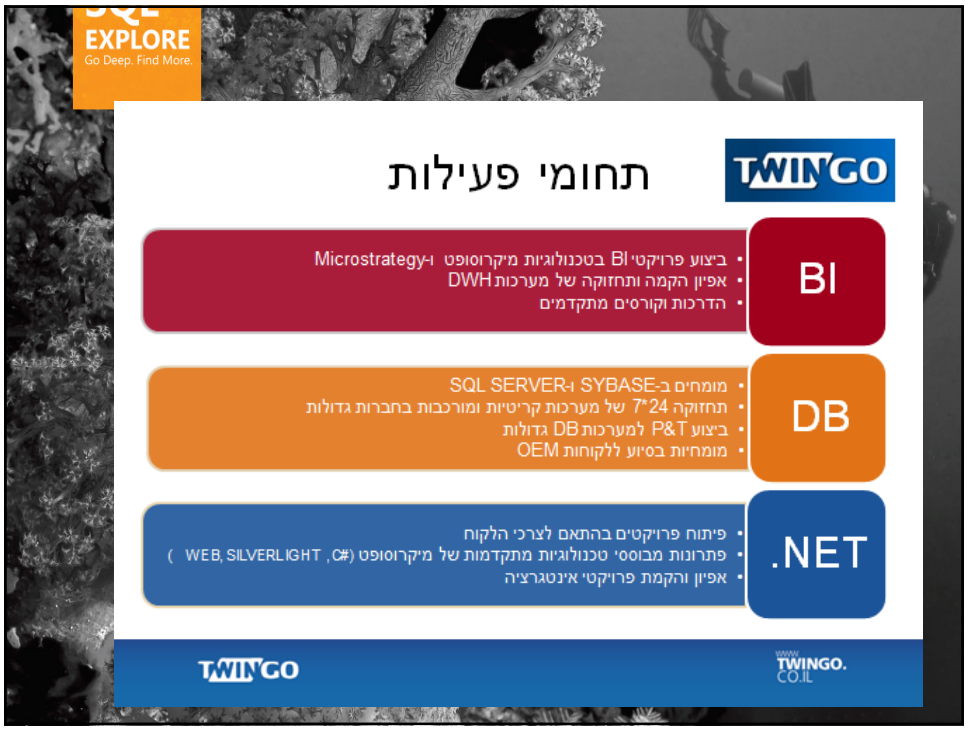
SQL EXPLORE Go Deep. Find More.

אודות החברה

TWINGO

- TWINGO הוקמה בשנת 2007
- החברה מונה 22 אנשי תוכנה מומחים בתחום
- משרדי החברה בהרצליה פיתוח
- שותפים של Microsoft בתחום DB וה-BI
- שותפים של MAGIC בתחום MicroStrategy
- שותפים של hp בתחום VERTICA
- חברת השירותים המובילה בתחום SAP SYBASE בישראל
- מובילים את תחום ה-BI ה-DB ב John Bryce

TWINGO www.TWINGO.CO.IL


EXPLORE
Go Deep. Find More.

TWINGO

תחומי פעילות

BI
- ביצוע פרויקטי BI בטכנולוגיות מיקרוסופט ו-Microstrategy
- אפיון הקמה ותחזוקת של מערכות DWH
- הדרכות וקורסים מתקדמים

DB
- מומחים ב-SYBASE ו-SQL SERVER
- תחזוקה 24*7 של מערכות קריטיות ומורכבות בחברות גדולות
- ביצוע P&T למערכות DB גדולות
- מומחיות בסיוע ללקוחות OEM

.NET
- פיתוח פרויקטים בהתאם לצרכי הלקוח
- פתרונות מבוססי טכנולוגיות מתקדמות של מיקרוסופט (C#, WEB, SILVERLIGHT)
- אפיון והקמת פרויקטי אינטגרציה

TWINGO www.TWINGO.CO.IL

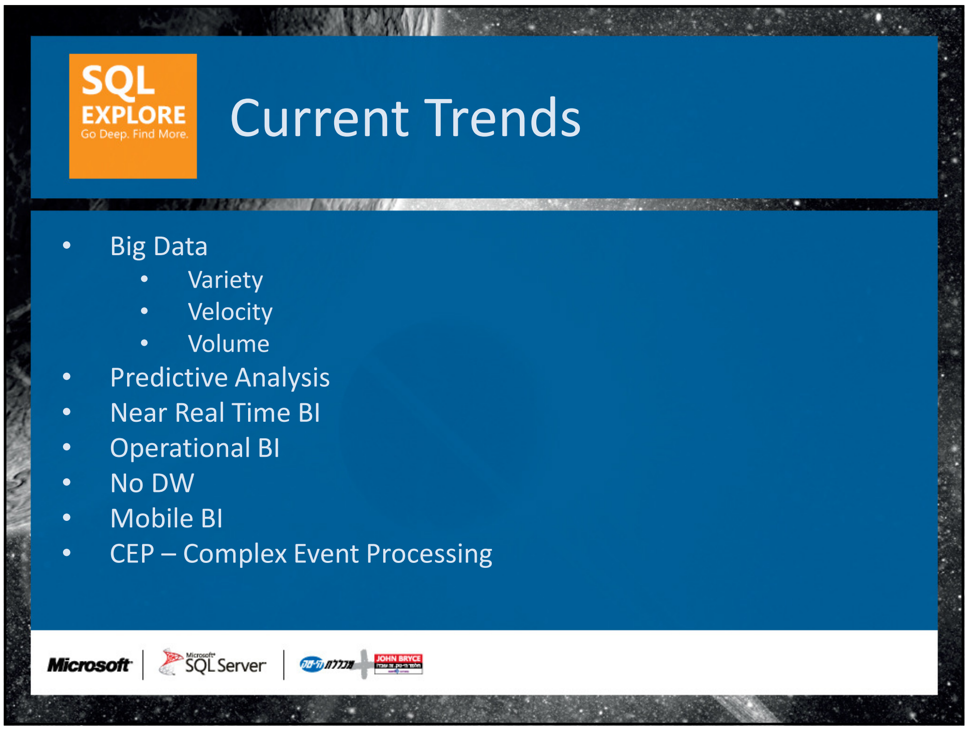
EXPLORE
Go Deep. Find More.

TWINGO

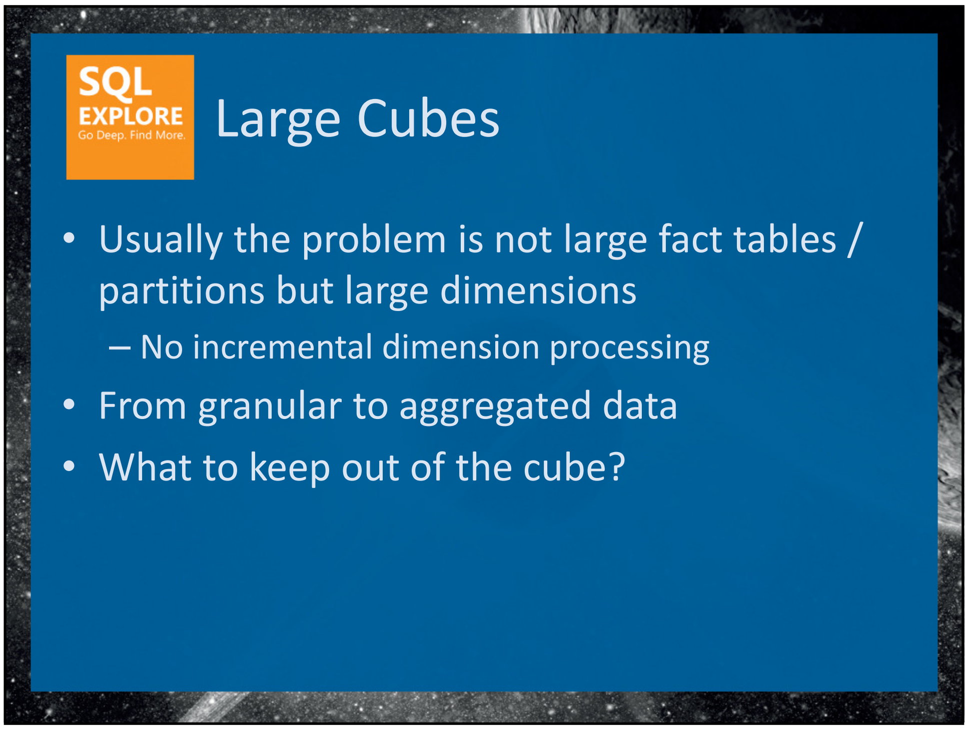
לקוחות נבחרים

דיסקונט
jajah
intel
Microsoft
סופר-פארם
PROFITECT
מפעל הפיס
ביטוח ישיר
TIS Top Image Systems
VERINT Witness Actionable Solutions
eci
APPLIED MATERIALS
JOHN BRYCE
mediamind
PrimeSense

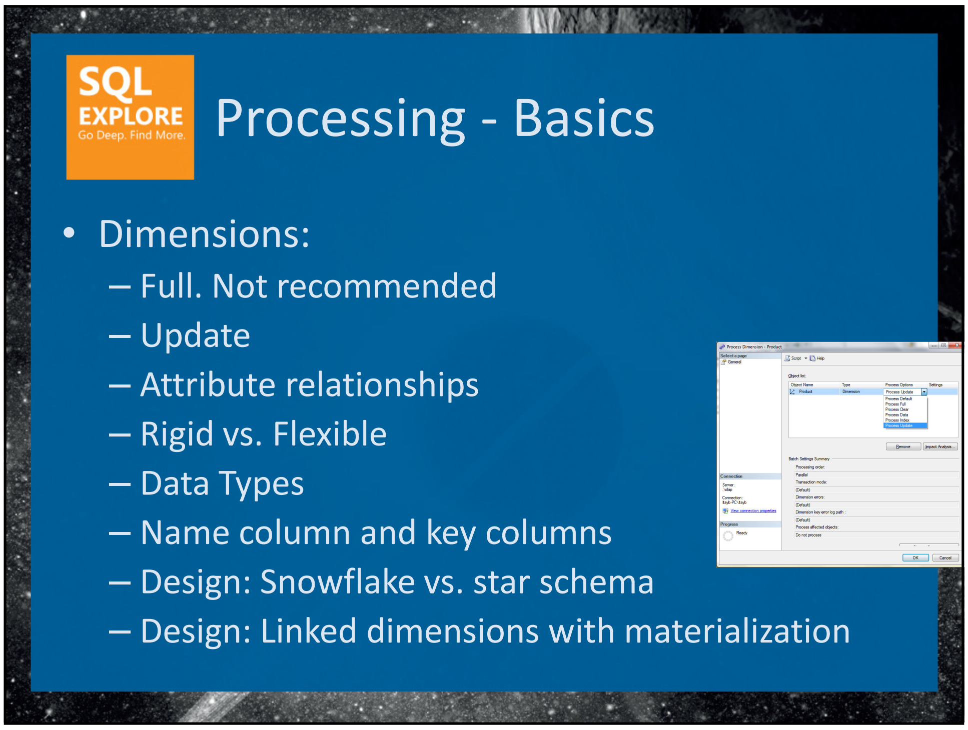
TWINGO www.TWINGO.CO.IL


Current Trends

- Big Data
  - Variety
  - Velocity
  - Volume
- Predictive Analysis
- Near Real Time BI
- Operational BI
- No DW
- Mobile BI
- CEP – Complex Event Processing

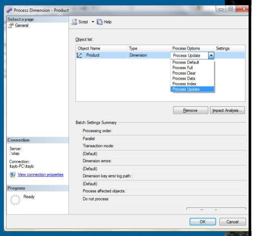

Large Cubes

- Usually the problem is not large fact tables / partitions but large dimensions
  – No incremental dimension processing
- From granular to aggregated data
- What to keep out of the cube?



Processing - Basics

- Dimensions:
  - Full. Not recommended
  - Update
  - Attribute relationships
  - Rigid vs. Flexible
  - Data Types
  - Name column and key columns
  - Design: Snowflake vs. star schema
  - Design: Linked dimensions with materialization



Process Dimension - Product

Select a page: General

Object list:

Object Name	Type	Process Options	Settings
Product	Dimension	Process Update	

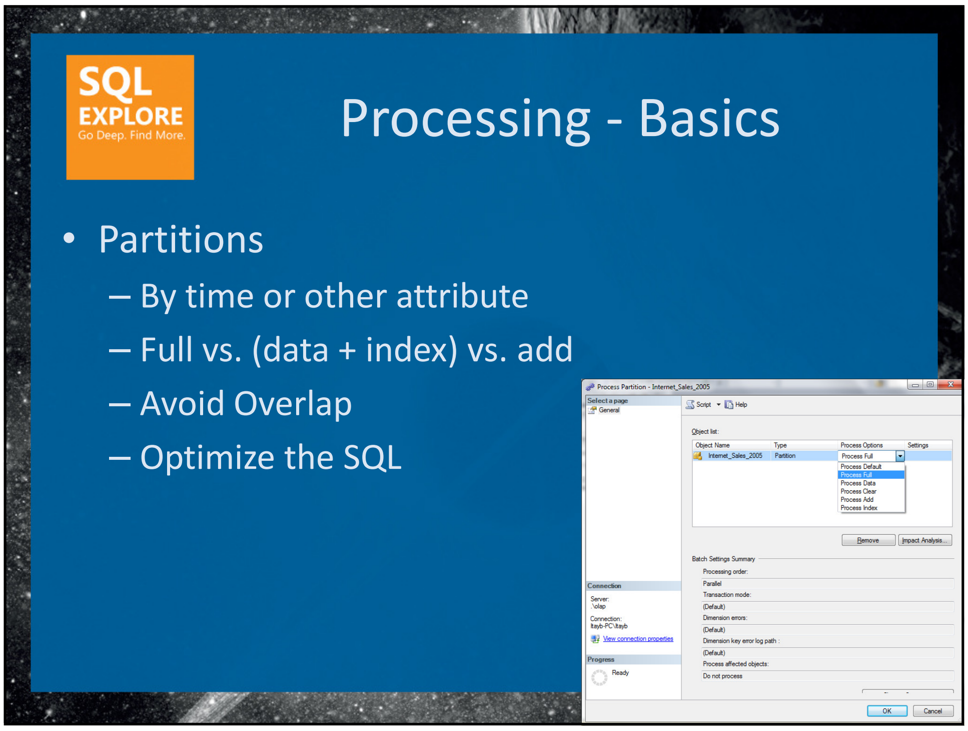
Batch Settings Summary

Processing order: Parallel
Transaction mode: (Default)
Dimension errors: (Default)
Dimension key error log path : (Default)
Process affected objects: Do not process

Connection
Server: .\olap
Connection: Itayb-PC\Itayb

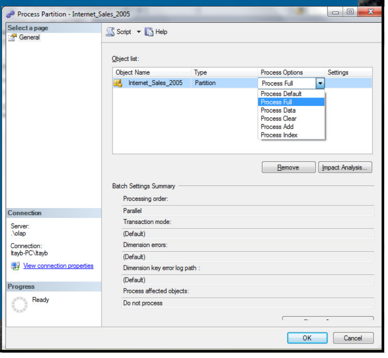
Progress: Ready

Script
Help
Process Default
Process Full
Process Clear
Process Data
Process Index
Process Update
Remove
Impact Analysis...
View connection properties
OK
Cancel



Processing - Basics

- Partitions
  - By time or other attribute
  - Full vs. (data + index) vs. add
  - Avoid Overlap
  - Optimize the SQL



Process Partition - Internet_Sales_2005

Select a page: General

Object list:

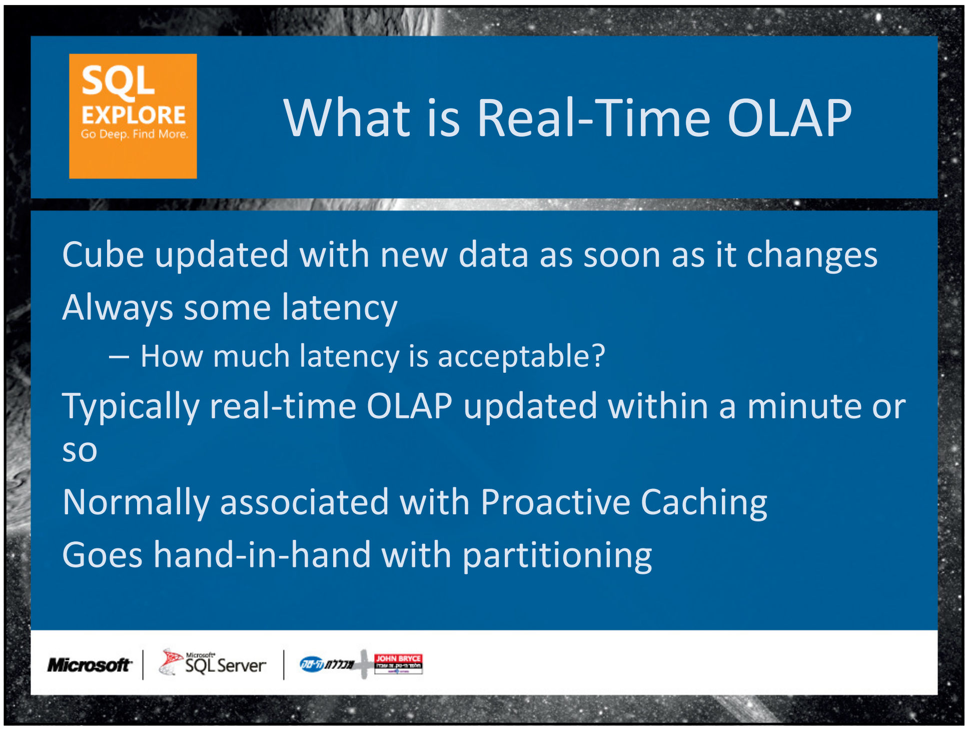
Object Name	Type	Process Options	Settings
Internet_Sales_2005	Partition	Process Full	

Batch Settings Summary

Processing order: Parallel
Transaction mode: (Default)
Dimension errors: (Default)
Dimension key error log path : (Default)
Process affected objects: Do not process

Connection
Server: .\olap
Connection: Itayb-PC\Itayb

Progress: Ready



SQL EXPLORE Go Deep. Find More.

What is Real-Time OLAP

Cube updated with new data as soon as it changes

Always some latency

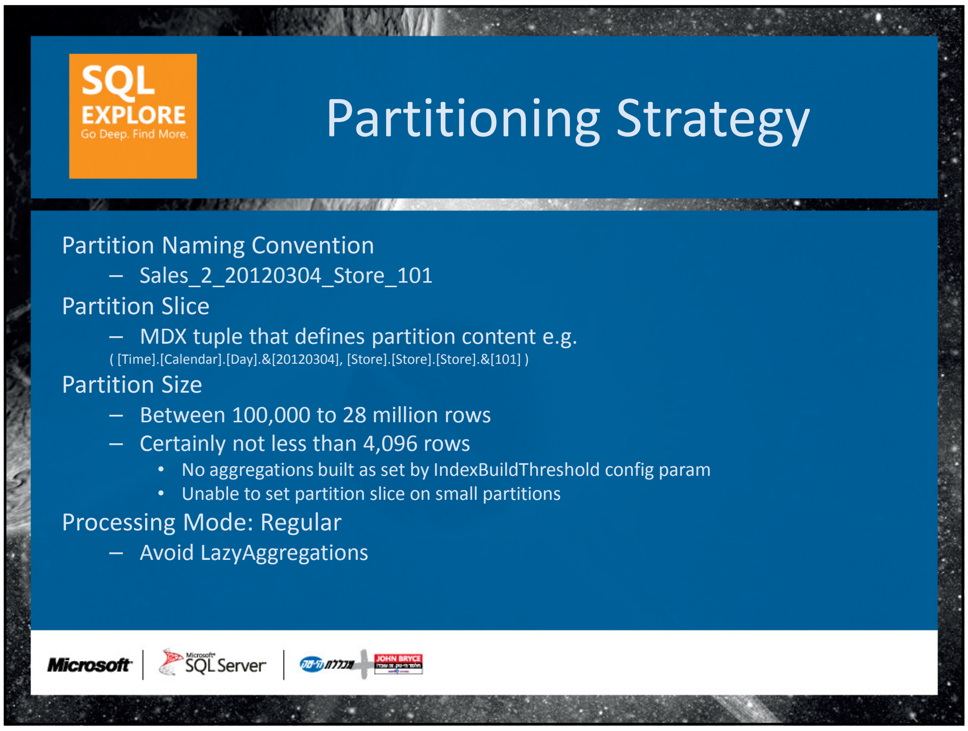
- How much latency is acceptable?

Typically real-time OLAP updated within a minute or so

Normally associated with Proactive Caching

Goes hand-in-hand with partitioning

Microsoft | SQL Server | JOHN BRYCE



SQL EXPLORE Go Deep. Find More.

Partitioning Strategy

Partition Naming Convention

- Sales_2_20120304_Store_101

Partition Slice

- MDX tuple that defines partition content e.g.

([Time].[Calendar].[Day].&[20120304], [Store].[Store].[Store].&[101])

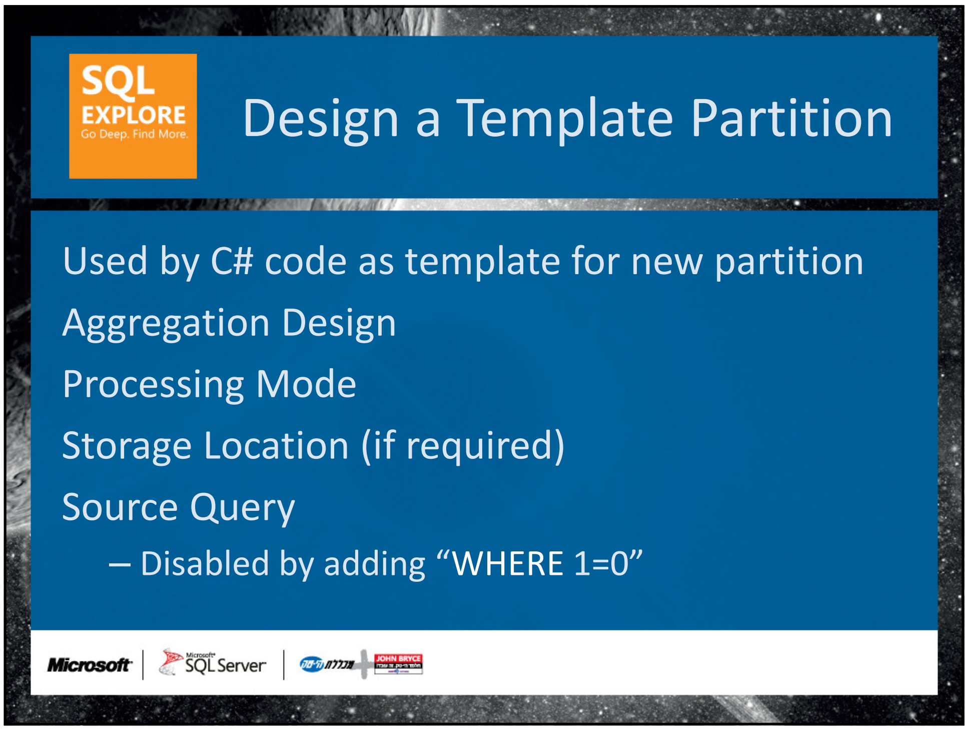
Partition Size

- Between 100,000 to 28 million rows
- Certainly not less than 4,096 rows
  - No aggregations built as set by IndexBuildThreshold config param
  - Unable to set partition slice on small partitions

Processing Mode: Regular

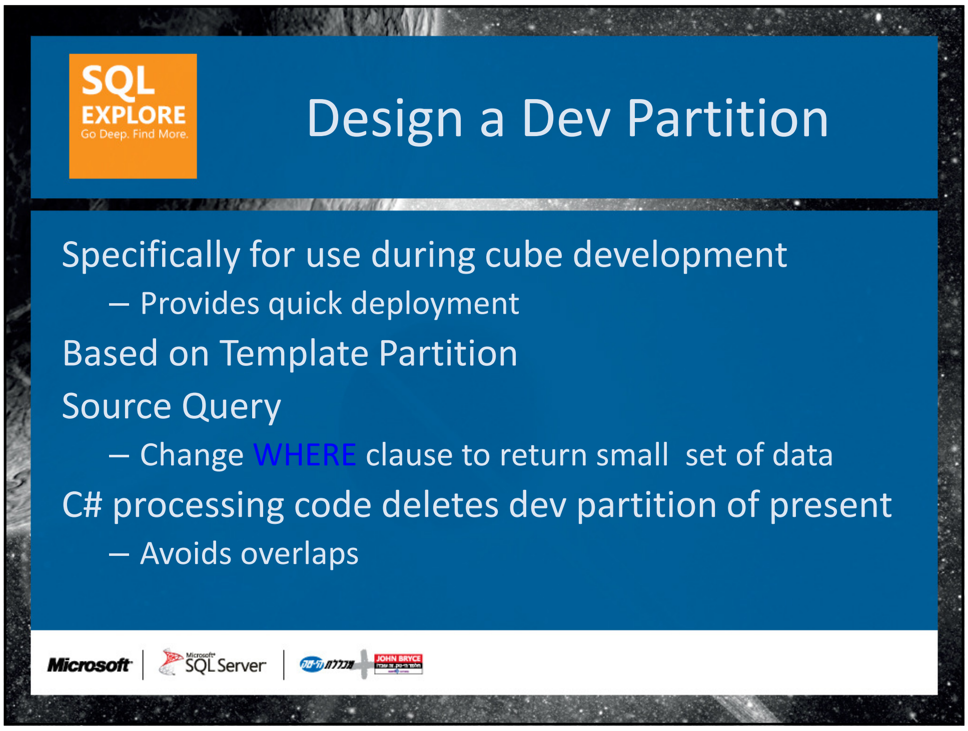
- Avoid LazyAggregations

Microsoft | SQL Server | JOHN BRYCE



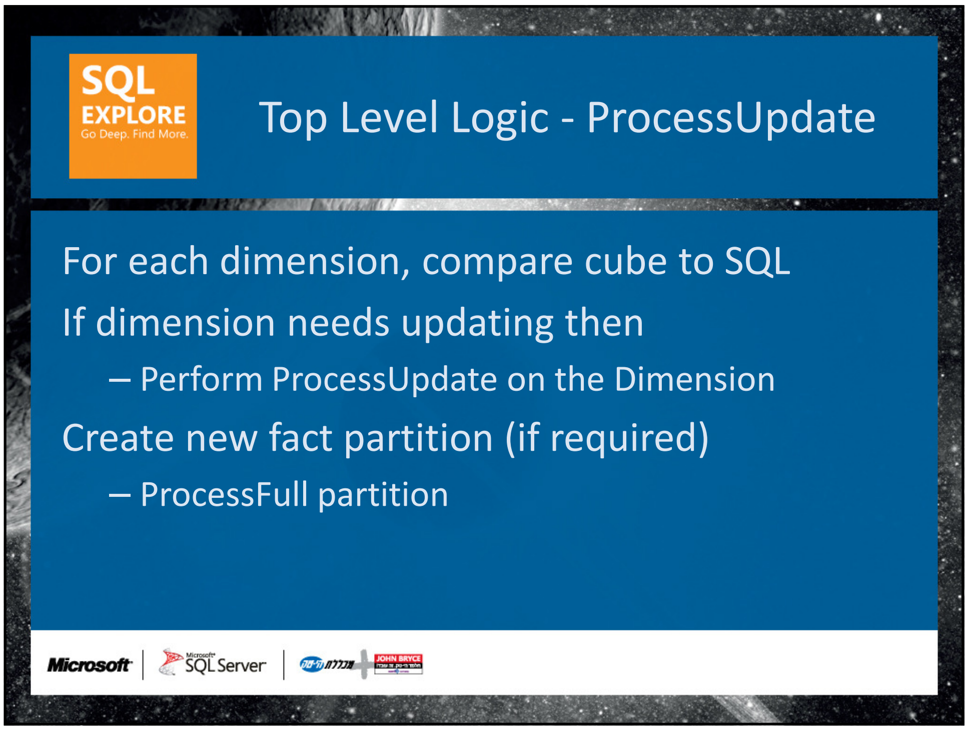
Design a Template Partition

- Used by C# code as template for new partition
- Aggregation Design
- Processing Mode
- Storage Location (if required)
- Source Query
  - Disabled by adding “WHERE 1=0”



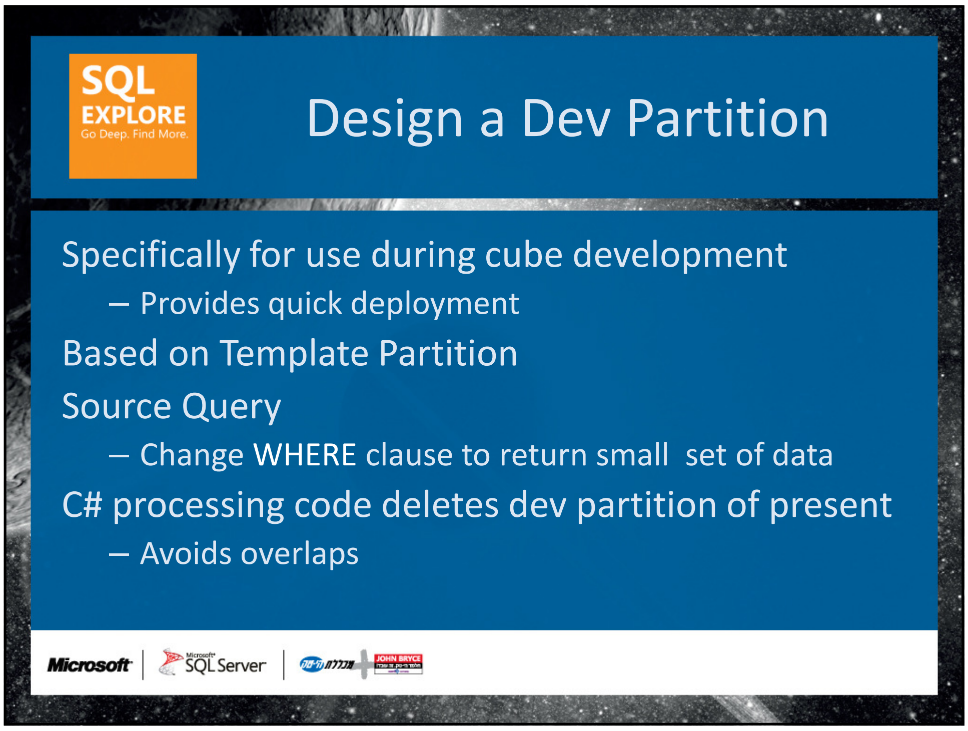
Design a Dev Partition

- Specifically for use during cube development
  - Provides quick deployment
- Based on Template Partition
- Source Query
  - Change WHERE clause to return small set of data
- C# processing code deletes dev partition of present
  - Avoids overlaps



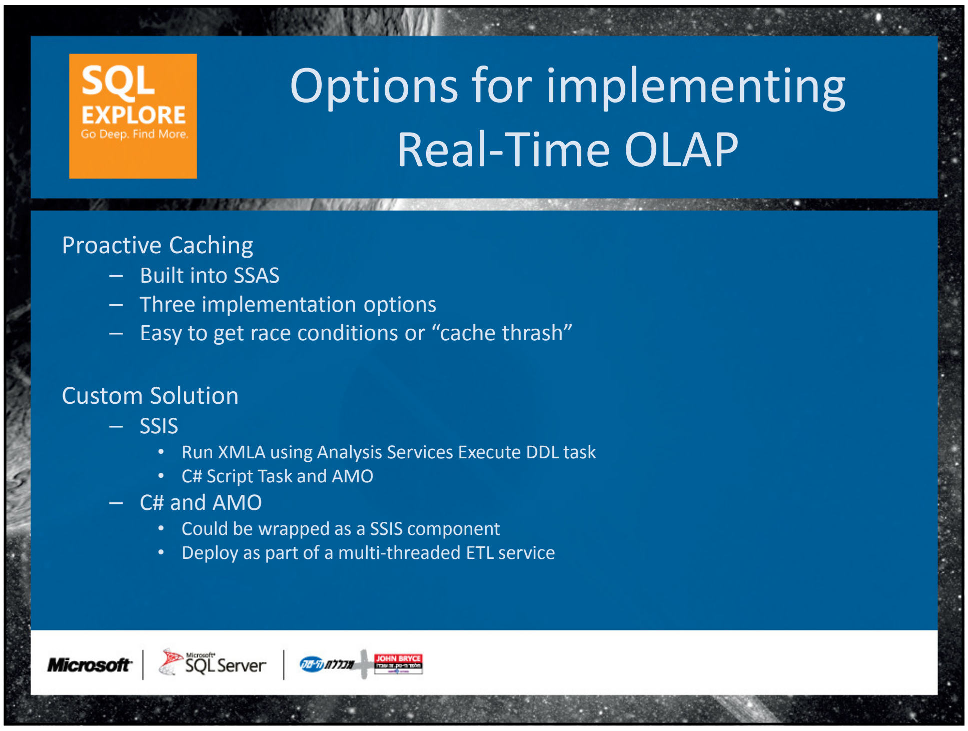
Top Level Logic - ProcessUpdate

- For each dimension, compare cube to SQL
- If dimension needs updating then
  - Perform ProcessUpdate on the Dimension
- Create new fact partition (if required)
  - ProcessFull partition



Design a Dev Partition

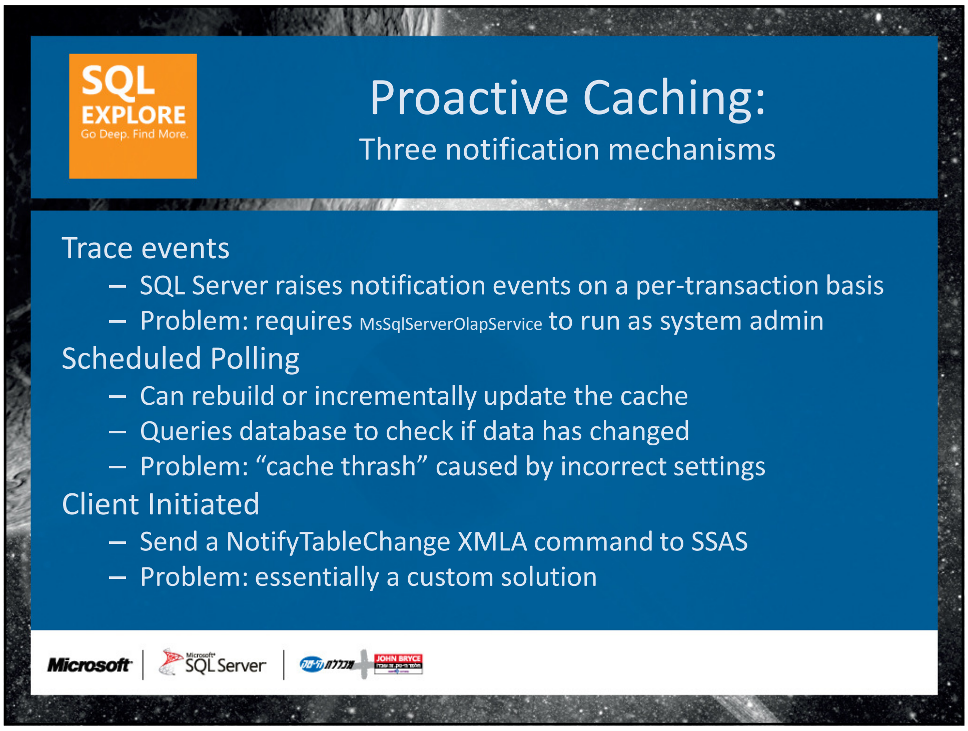
- Specifically for use during cube development
  - Provides quick deployment
- Based on Template Partition
- Source Query
  - Change WHERE clause to return small set of data
- C# processing code deletes dev partition of present
  - Avoids overlaps



Options for implementing Real-Time OLAP

Proactive Caching
- Built into SSAS
- Three implementation options
- Easy to get race conditions or “cache thrash”

Custom Solution
- SSIS
  - Run XMLA using Analysis Services Execute DDL task
  - C# Script Task and AMO
- C# and AMO
  - Could be wrapped as a SSIS component
  - Deploy as part of a multi-threaded ETL service

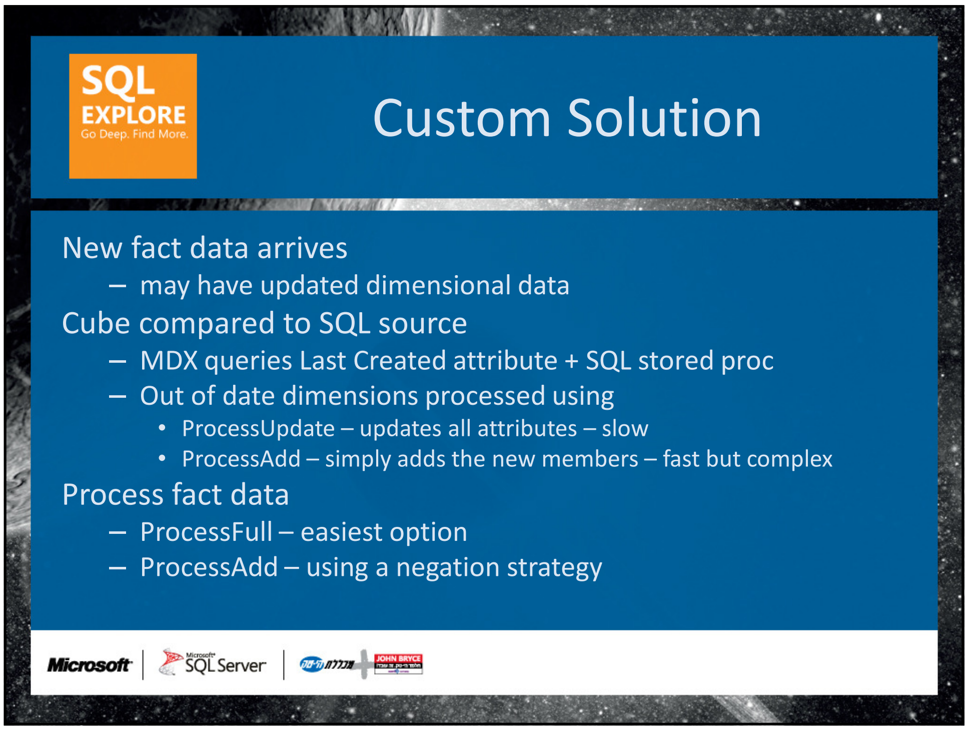


Proactive Caching:
Three notification mechanisms

Trace events
- SQL Server raises notification events on a per-transaction basis
- Problem: requires MsSqlServerOlapService to run as system admin

Scheduled Polling
- Can rebuild or incrementally update the cache
- Queries database to check if data has changed
- Problem: “cache thrash” caused by incorrect settings

Client Initiated
- Send a NotifyTableChange XMLA command to SSAS
- Problem: essentially a custom solution

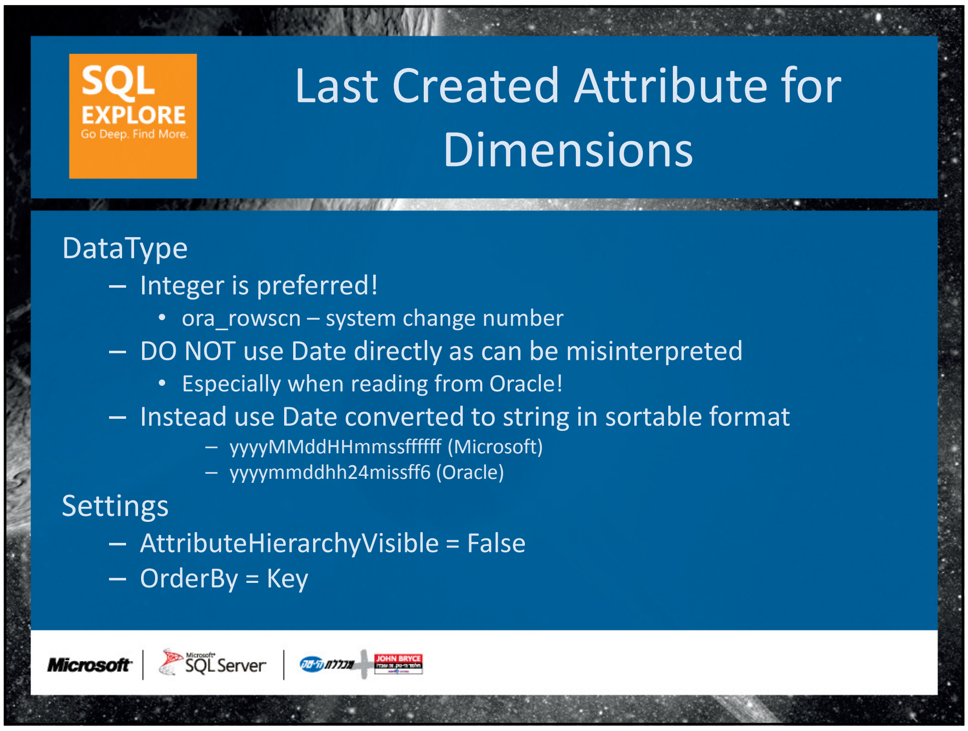


Custom Solution

New fact data arrives
- may have updated dimensional data

Cube compared to SQL source
- MDX queries Last Created attribute + SQL stored proc
- Out of date dimensions processed using
  - ProcessUpdate – updates all attributes – slow
  - ProcessAdd – simply adds the new members – fast but complex

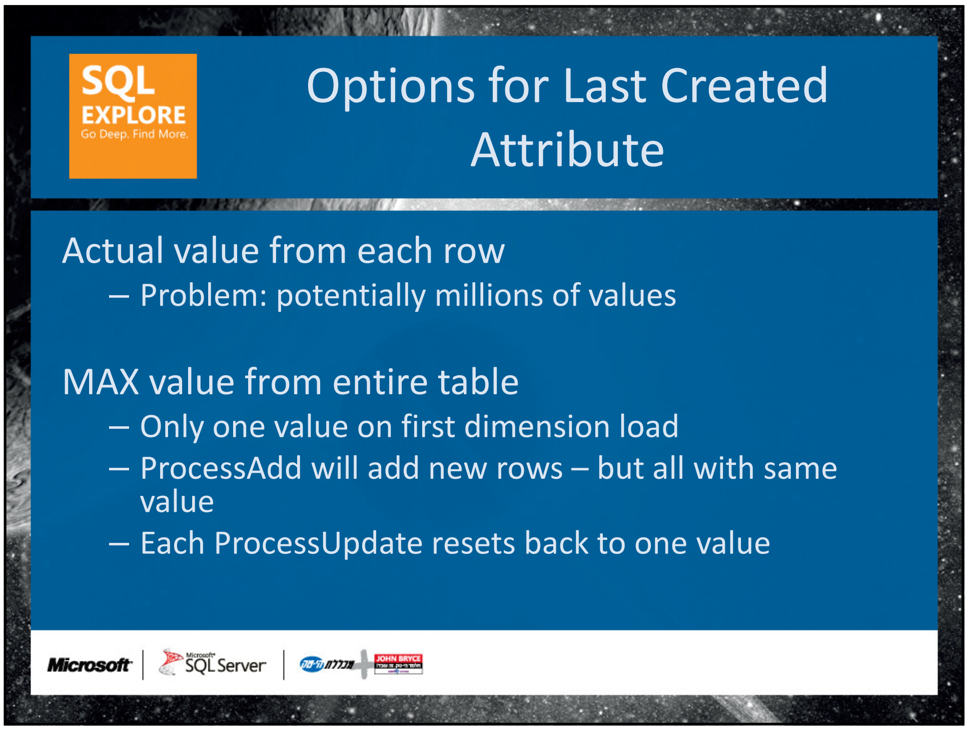
Process fact data
- ProcessFull – easiest option
- ProcessAdd – using a negation strategy



Last Created Attribute for Dimensions

DataType
- Integer is preferred!
  - ora_rowscn – system change number
- DO NOT use Date directly as can be misinterpreted
  - Especially when reading from Oracle!
- Instead use Date converted to string in sortable format
  - yyyyMMddHHmmssffffff (Microsoft)
  - yyyymmddhh24missff6 (Oracle)

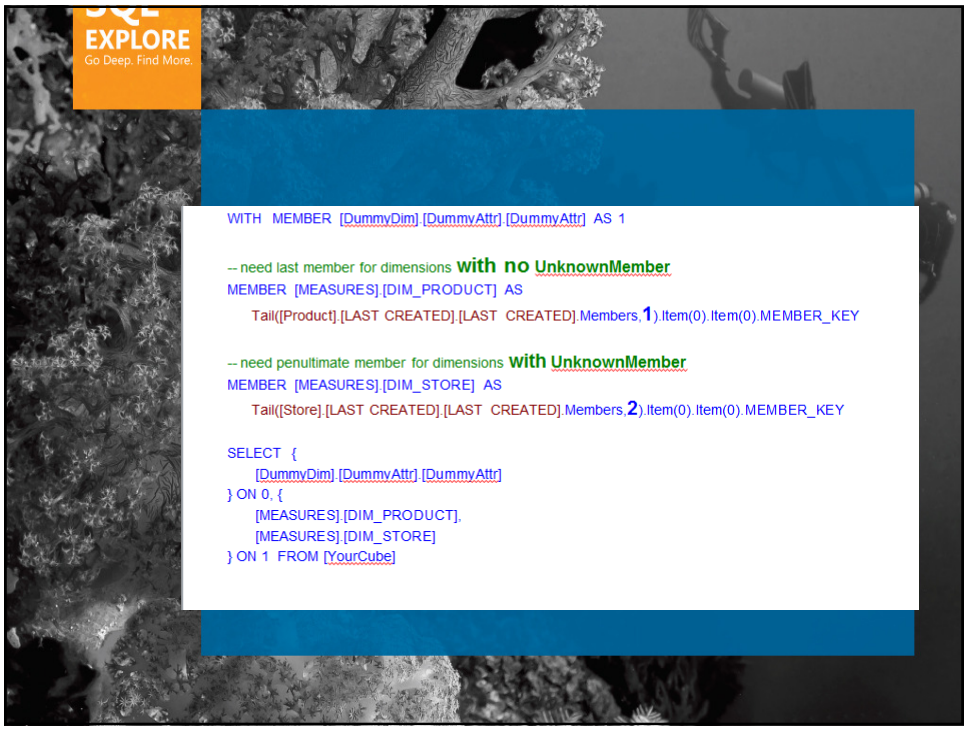
Settings
- AttributeHierarchyVisible = False
- OrderBy = Key



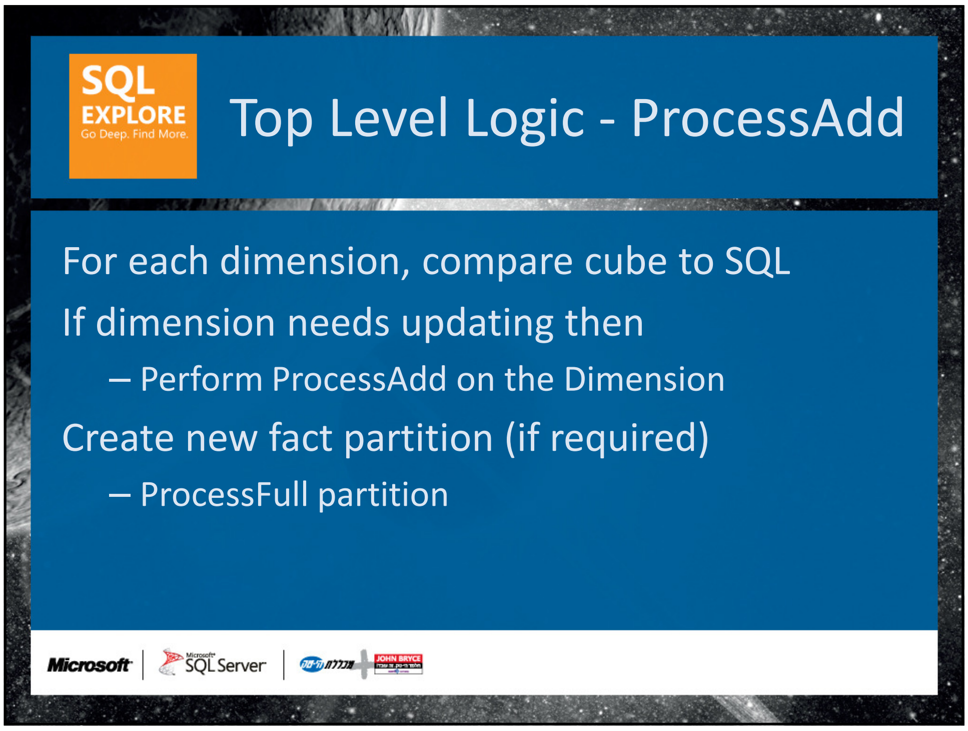
Options for Last Created Attribute

Actual value from each row
- Problem: potentially millions of values

MAX value from entire table
- Only one value on first dimension load
- ProcessAdd will add new rows – but all with same value
- Each ProcessUpdate resets back to one value



```
WITH MEMBER [DummyDim].[DummyAttr].[DummyAttr] AS 1

-- need last member for dimensions with no UnknownMember
MEMBER [MEASURES].[DIM_PRODUCT] AS
    Tail([Product].[LAST CREATED].[LAST CREATED].Members,1).Item(0).Item(0).MEMBER_KEY

-- need penultimate member for dimensions with UnknownMember
MEMBER [MEASURES].[DIM_STORE] AS
    Tail([Store].[LAST CREATED].[LAST CREATED].Members,2).Item(0).Item(0).MEMBER_KEY

SELECT {
    [DummyDim].[DummyAttr].[DummyAttr]
} ON 0, {
    [MEASURES].[DIM_PRODUCT],
    [MEASURES].[DIM_STORE]
} ON 1 FROM [YourCube]
```



Top Level Logic - ProcessAdd

For each dimension, compare cube to SQL

If dimension needs updating then

– Perform ProcessAdd on the Dimension

Create new fact partition (if required)

– ProcessFull partition

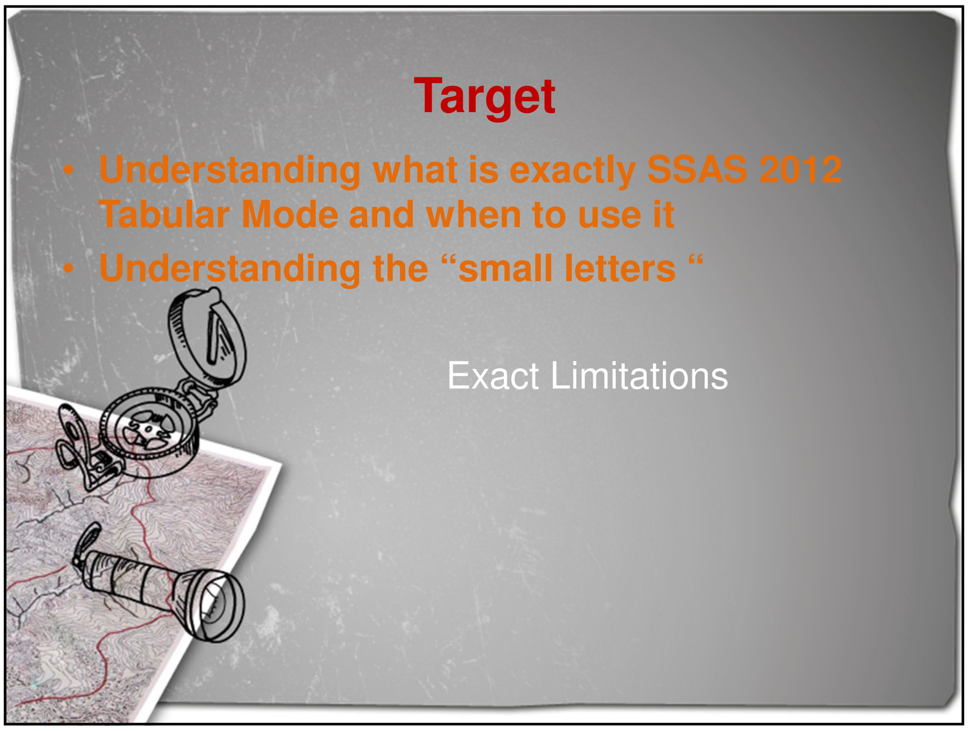
Microsoft

TWINGO

SSAS 2012 Multi Dimensional vs. Tabular Mode

Itay Braun
CTO & BI Architect, Twingo

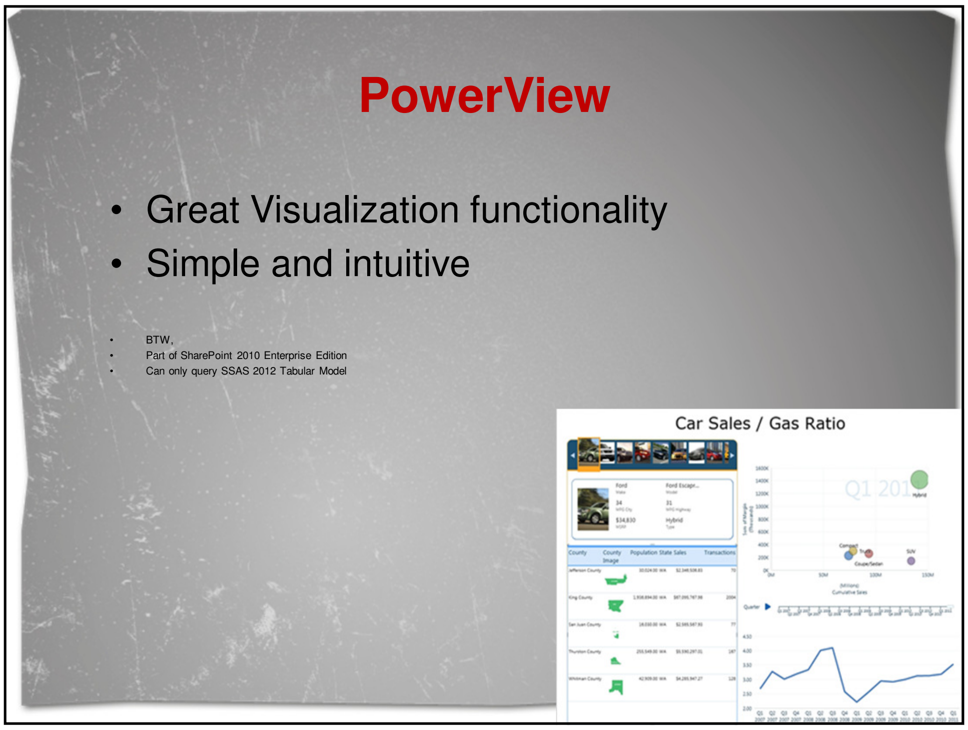
Manager of the Microsoft BI User Group in Israel
itay@twingo.co.il



Target

- Understanding what is exactly SSAS 2012 Tabular Mode and when to use it
- Understanding the "small letters "

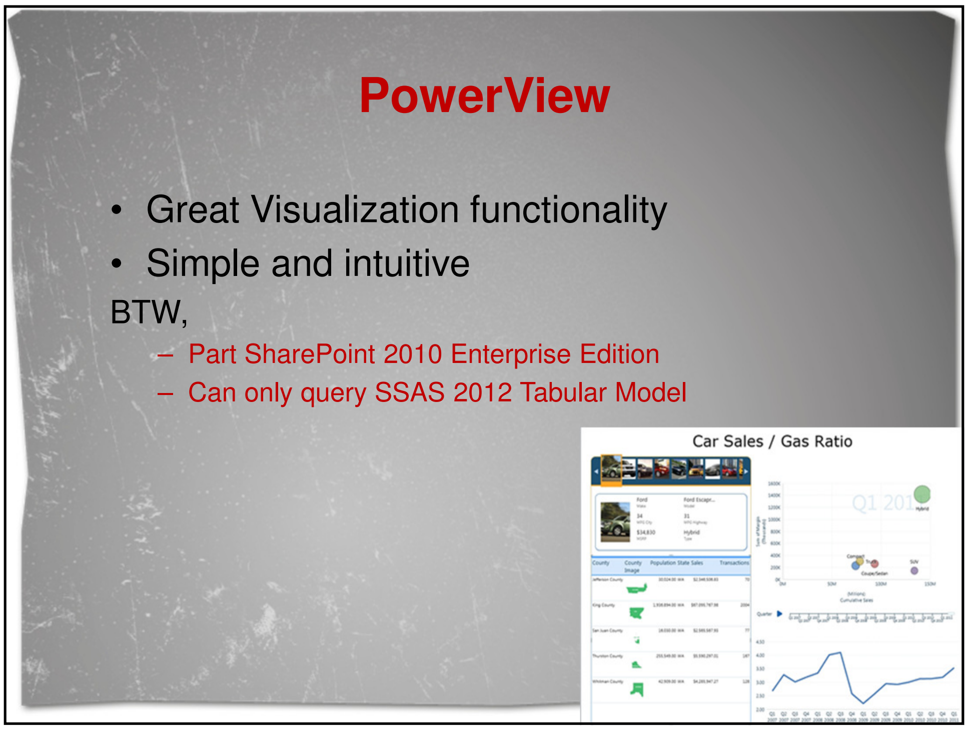
Exact Limitations



PowerView

- Great Visualization functionality
- Simple and intuitive

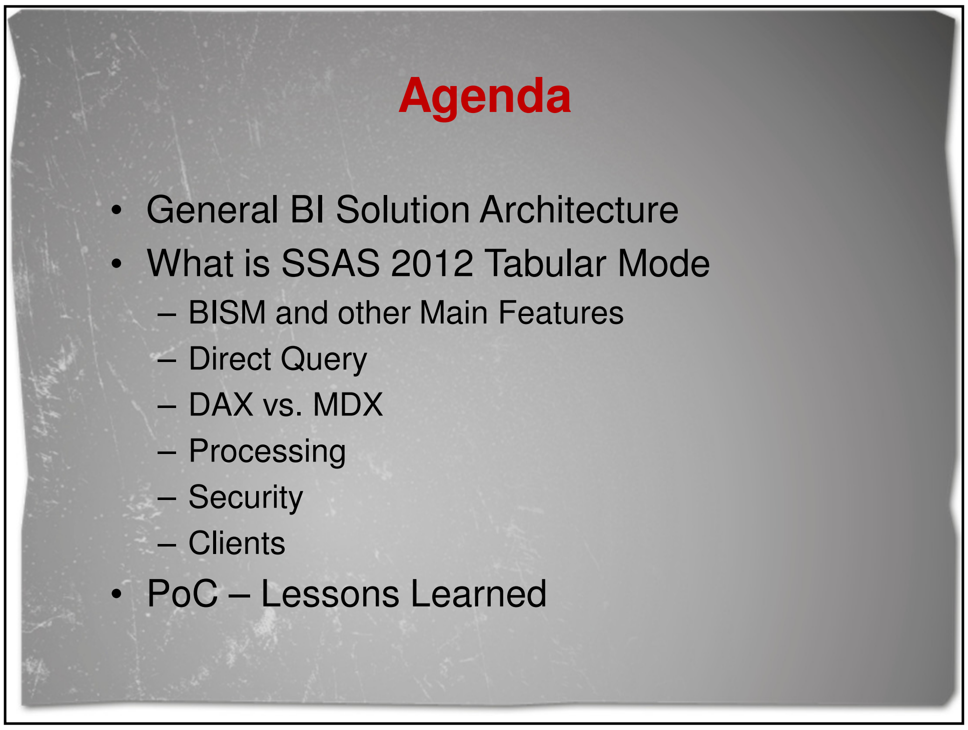
- BTW,
- Part of SharePoint 2010 Enterprise Edition
- Can only query SSAS 2012 Tabular Model



PowerView

- Great Visualization functionality
- Simple and intuitive

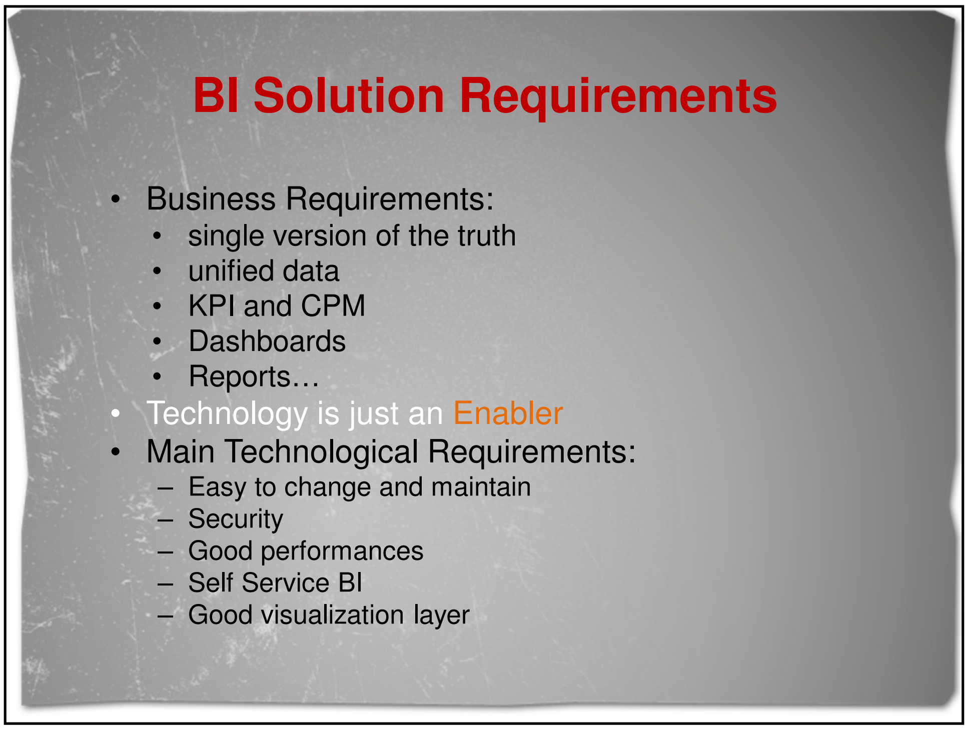
BTW,
  - Part SharePoint 2010 Enterprise Edition
  - Can only query SSAS 2012 Tabular Model



Agenda

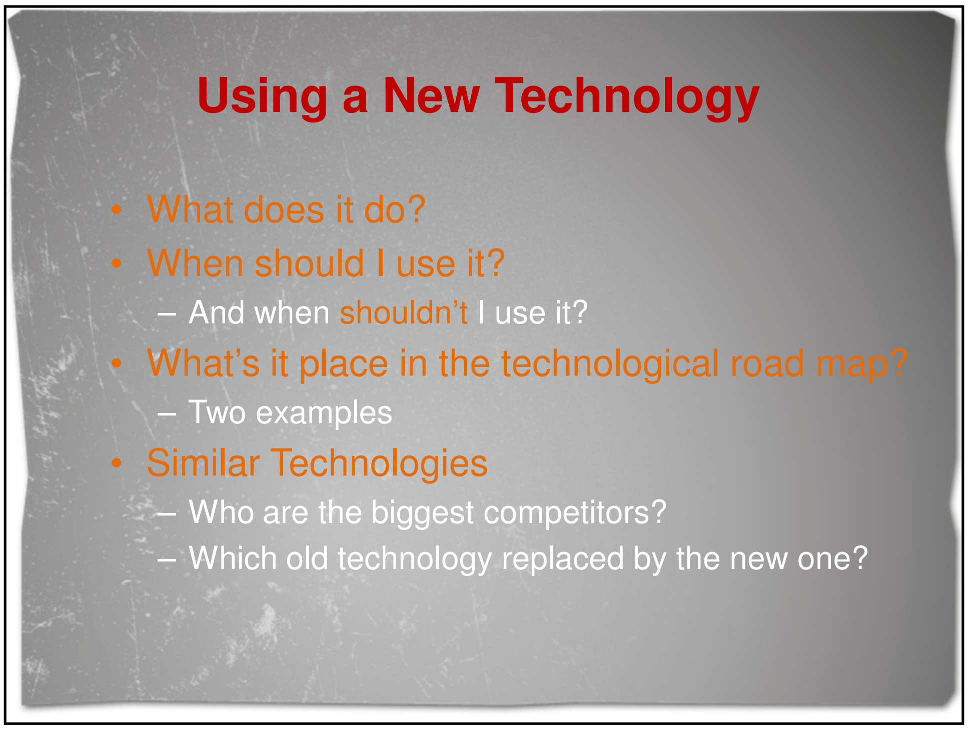
- General BI Solution Architecture
- What is SSAS 2012 Tabular Mode
  - BISM and other Main Features
  - Direct Query
  - DAX vs. MDX
  - Processing
  - Security
  - Clients
- PoC – Lessons Learned

BI Solution Architecture



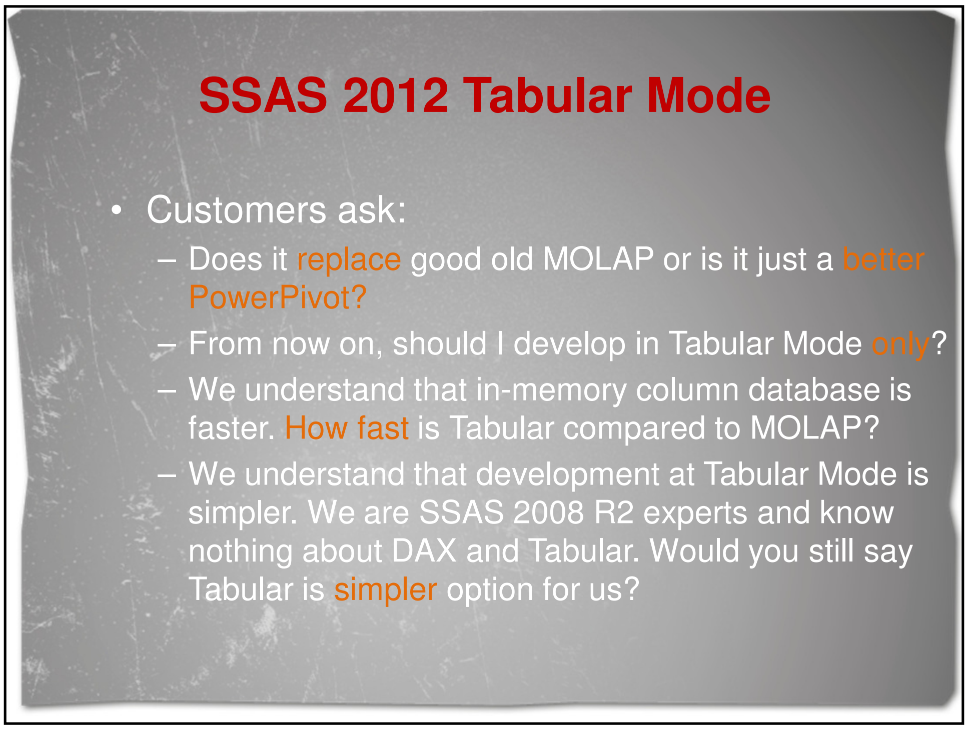
BI Solution Requirements

- Business Requirements:
  - single version of the truth
  - unified data
  - KPI and CPM
  - Dashboards
  - Reports…
- Technology is just an Enabler
- Main Technological Requirements:
  - Easy to change and maintain
  - Security
  - Good performances
  - Self Service BI
  - Good visualization layer



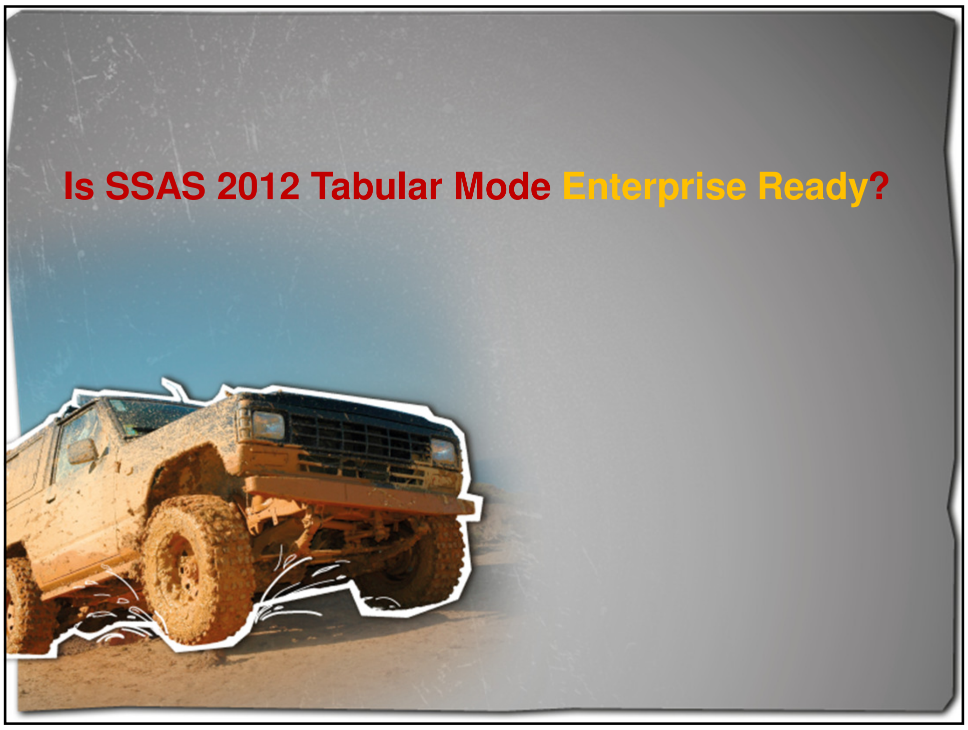
Using a New Technology

- What does it do?
- When should I use it?
  - And when shouldn't I use it?
- What's it place in the technological road map?
  - Two examples
- Similar Technologies
  - Who are the biggest competitors?
  - Which old technology replaced by the new one?



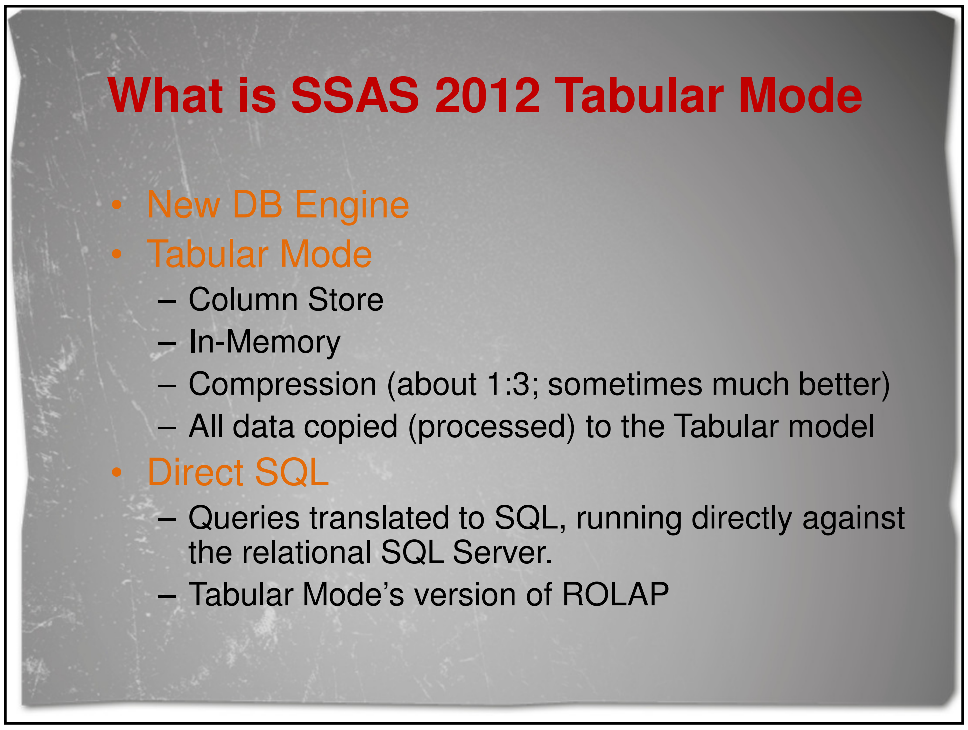
SSAS 2012 Tabular Mode

- Customers ask:
  - Does it replace good old MOLAP or is it just a better PowerPivot?
  - From now on, should I develop in Tabular Mode only?
  - We understand that in-memory column database is faster. How fast is Tabular compared to MOLAP?
  - We understand that development at Tabular Mode is simpler. We are SSAS 2008 R2 experts and know nothing about DAX and Tabular. Would you still say Tabular is simpler option for us?



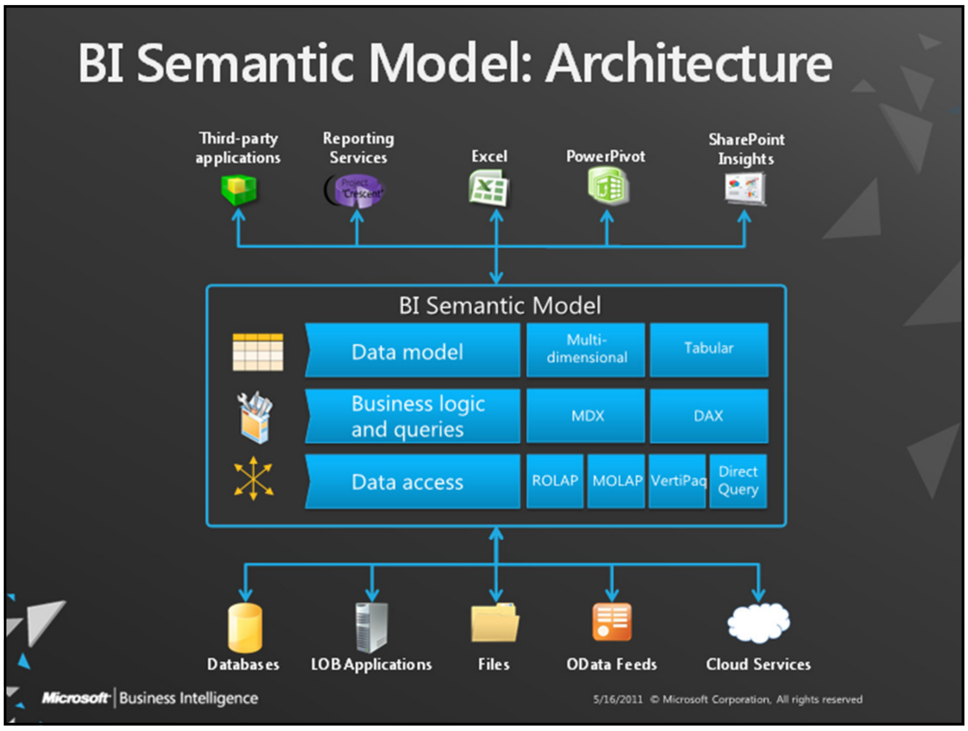
Is SSAS 2012 Tabular Mode Enterprise Ready?

WHAT IS SSAS 2012 TABULAR MODE?



What is SSAS 2012 Tabular Mode

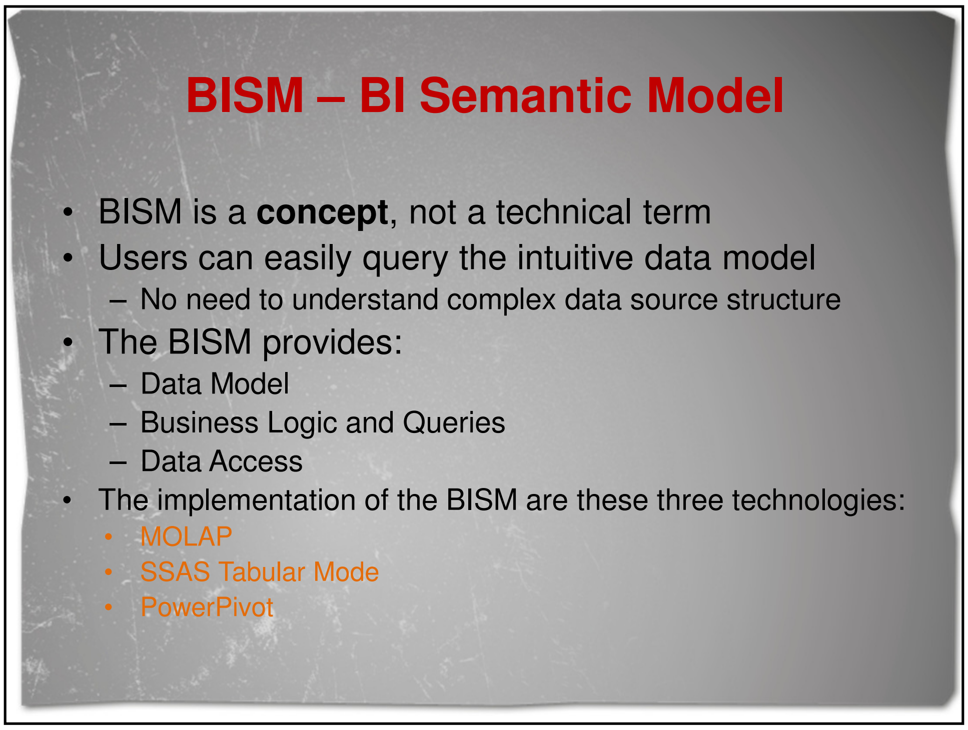
- New DB Engine
- Tabular Mode
  - Column Store
  - In-Memory
  - Compression (about 1:3; sometimes much better)
  - All data copied (processed) to the Tabular model
- Direct SQL
  - Queries translated to SQL, running directly against the relational SQL Server.
  - Tabular Mode's version of ROLAP



BI Semantic Model: Architecture

Microsoft | Business Intelligence

5/16/2011 © Microsoft Corporation. All rights reserved



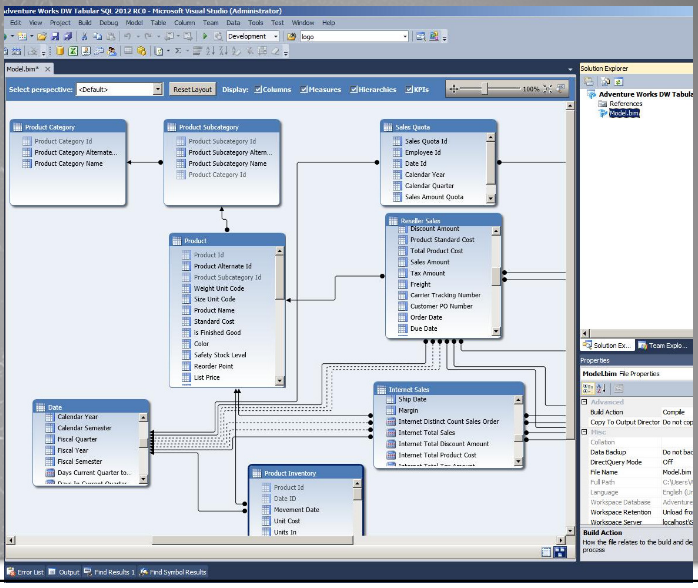
BISM – BI Semantic Model

- BISM is a **concept**, not a technical term
- Users can easily query the intuitive data model
  - No need to understand complex data source structure
- The BISM provides:
  - Data Model
  - Business Logic and Queries
  - Data Access
- The implementation of the BISM are these three technologies:
  - MOLAP
  - SSAS Tabular Mode
  - PowerPivot

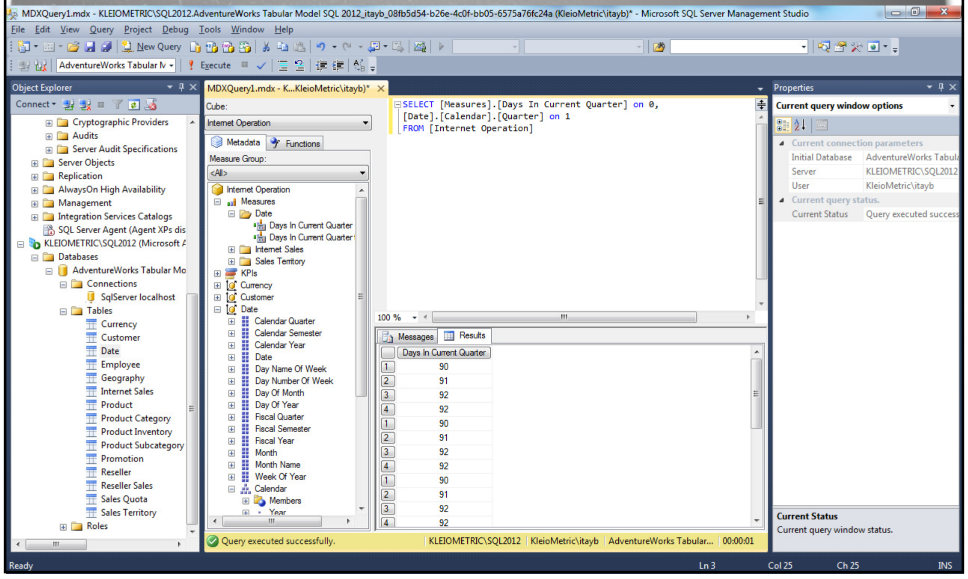
Tabular vs. Multi Dimensional

- Tabular
  - Quicker development
  - From detailed (granular, images) data to aggregate data
  - Good performances
  - Some features are missing
- Multi-Dimensional
  - Rich Multi-dimensional functionality
  - Known and well documented engine
  - Suitable for very large cubes, beyond server's memory size

Schema (SQL Server Data Tools)



Management Studio



```
SELECT [Measures].[Days In Current Quarter] on 0,
[Date].[Calendar].[Quarter] on 1
FROM [Internet Operation]
```

	Days In Current Quarter
1	90
2	91
3	92
4	92
1	90
2	91
3	92
4	92
1	90
2	91
3	92
4	92

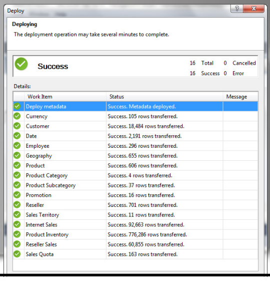
Query executed successfully. KLEIOMETRIC\SQL2012 | KleioMetric\itayb | AdventureWorks Tabular... | 00:00:01

Tabular Mode - Main Features

- Understanding the main features of a technology is the key to understand when and how to use it
  - Feature by Server Mode or Solution Type

Developing a Model

- Use the Visual Studio SQL Server Data Tools to build a BISM
- Open an existing PowerPivot Model using SSDT or Management Studio
- Deployment
  - Immediate changes implementation at the SDDT
  - Use small DB for development



Deploy

Deploying

The deployment operation may take several minutes to complete.

Success — 16 Total, 0 Cancelled, 16 Success, 0 Error

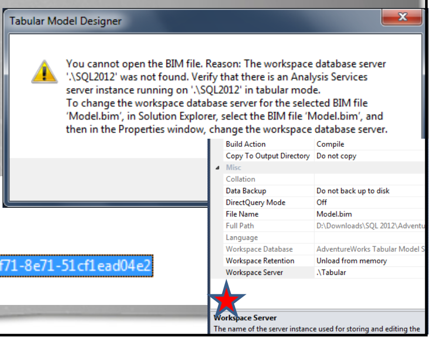
Details:

	Work Item	Status	Message
✓	Deploy metadata	Success. Metadata deployed.	
✓	Currency	Success. 105 rows transferred.	
✓	Customer	Success. 18,484 rows transferred.	
✓	Date	Success. 2,191 rows transferred.	
✓	Employee	Success. 296 rows transferred.	
✓	Geography	Success. 655 rows transferred.	
✓	Product	Success. 606 rows transferred.	
✓	Product Category	Success. 4 rows transferred.	
✓	Product Subcategory	Success. 37 rows transferred.	
✓	Promotion	Success. 16 rows transferred.	
✓	Reseller	Success. 701 rows transferred.	
✓	Sales Territory	Success. 11 rows transferred.	
✓	Internet Sales	Success. 92,663 rows transferred.	
✓	Product Inventory	Success. 776,286 rows transferred.	
✓	Reseller Sales	Success. 60,855 rows transferred.	
✓	Sales Quota	Success. 163 rows transferred.	

Workspace Database

- Created during model authoring using SSDT.
- Disappeared automatically when closing the project
- The workspace database resides in-memory.

[Workspace DB]

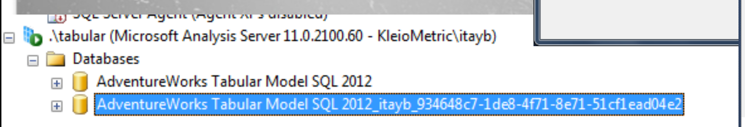


Tabular Model Designer

You cannot open the BIM file. Reason: The workspace database server '.\SQL2012' was not found. Verify that there is an Analysis Services server instance running on '.\SQL2012' in tabular mode.
To change the workspace database server for the selected BIM file 'Model.bim', in Solution Explorer, select the BIM file 'Model.bim', and then in the Properties window, change the workspace database server.

Build Action	Compile
Copy To Output Directory	Do not copy
Misc	
Collation	
Data Backup	Do not back up to disk
DirectQuery Mode	Off
File Name	Model.bim
Full Path	D:\Downloads\SQL 2012\Adventu
Language	
Workspace Database	AdventureWorks Tabular Model S
Workspace Retention	Unload from memory
Workspace Server	.\Tabular

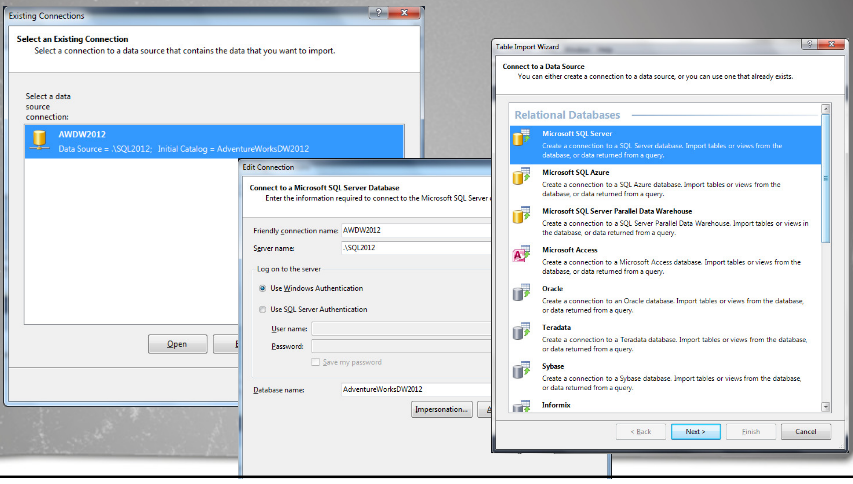
Workspace Server
The name of the server instance used for storing and editing the



.\tabular (Microsoft Analysis Server 11.0.2100.60 - KleioMetric\itayb)
Databases
AdventureWorks Tabular Model SQL 2012
AdventureWorks Tabular Model SQL 2012_itayb_934648c7-1de8-4f71-8e71-51cf1ead04e2

Data Sources

- The model can use multiple data sources



Existing Connections

Select an Existing Connection
Select a connection to a data source that contains the data that you want to import.

Select a data source connection:

AWDW2012
Data Source = .\SQL2012; Initial Catalog = AdventureWorksDW2012

Open

Edit Connection

Connect to a Microsoft SQL Server Database
Enter the information required to connect to the Microsoft SQL Server

Friendly connection name: AWDW2012
Server name: .\SQL2012

Log on to the server
[✓ Use Windows Authentication]
Use SQL Server Authentication
User name:
Password:
Save my password

Database name: AdventureWorksDW2012

Impersonation...

Table Import Wizard

Connect to a Data Source
You can either create a connection to a data source, or you can use one that already exists.

Relational Databases

Microsoft SQL Server
Create a connection to a SQL Server database. Import tables or views from the database, or data returned from a query.

Microsoft SQL Azure
Create a connection to a SQL Azure database. Import tables or views from the database, or data returned from a query.

Microsoft SQL Server Parallel Data Warehouse
Create a connection to a SQL Server Parallel Data Warehouse. Import tables or views in the database, or data returned from a query.

Microsoft Access
Create a connection to a Microsoft Access database. Import tables or views from the database, or data returned from a query.

Oracle
Create a connection to an Oracle database. Import tables or views from the database, or data returned from a query.

Teradata
Create a connection to a Teradata database. Import tables or views from the database, or data returned from a query.

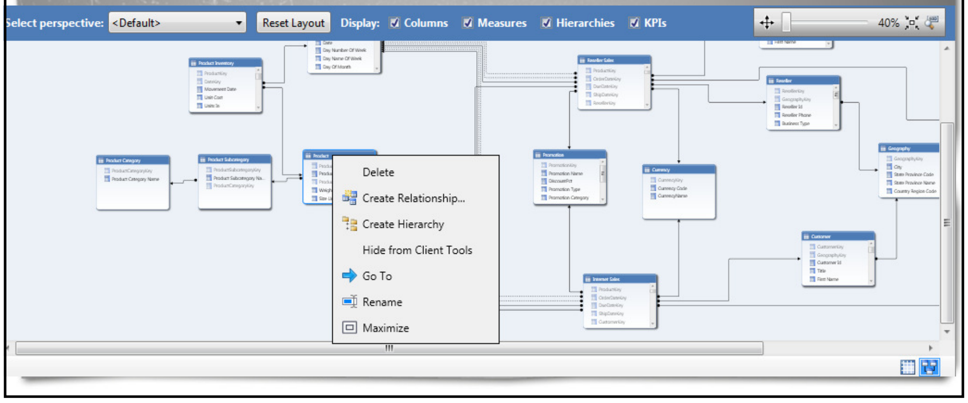
Sybase
Create a connection to a Sybase database. Import tables or views from the database, or data returned from a query.

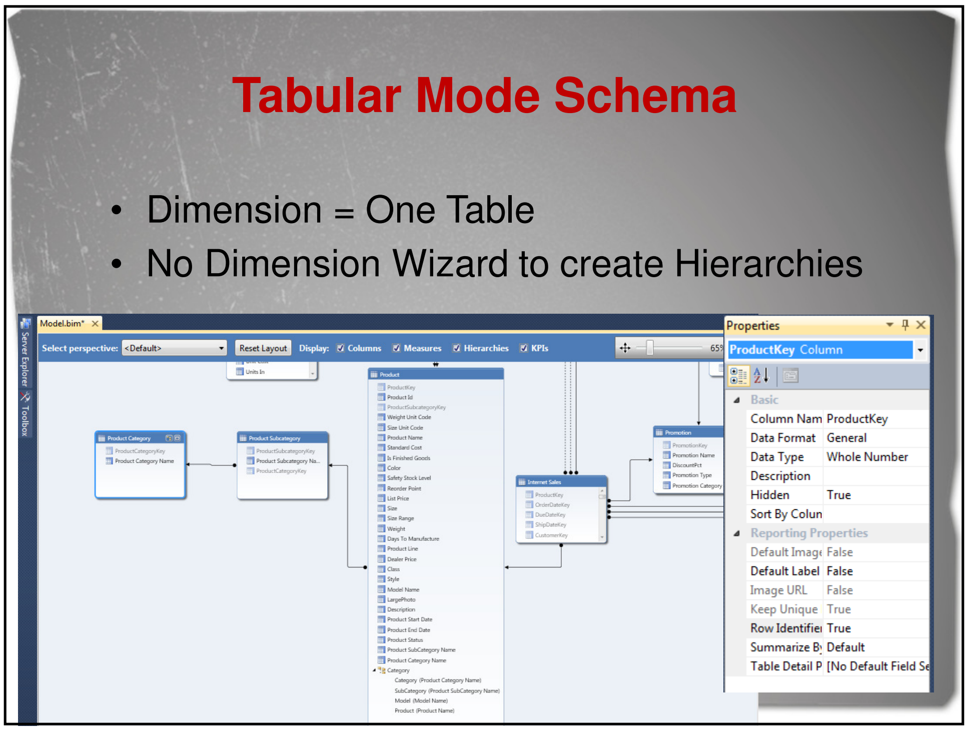
Informix

< Back | Next > | Finish | Cancel

Tabular Mode Schema vs. Data Source Views

- Much simpler than a Data Source View
- Building the AdventureWorks DB using Tabular Mode is about x4 times faster.





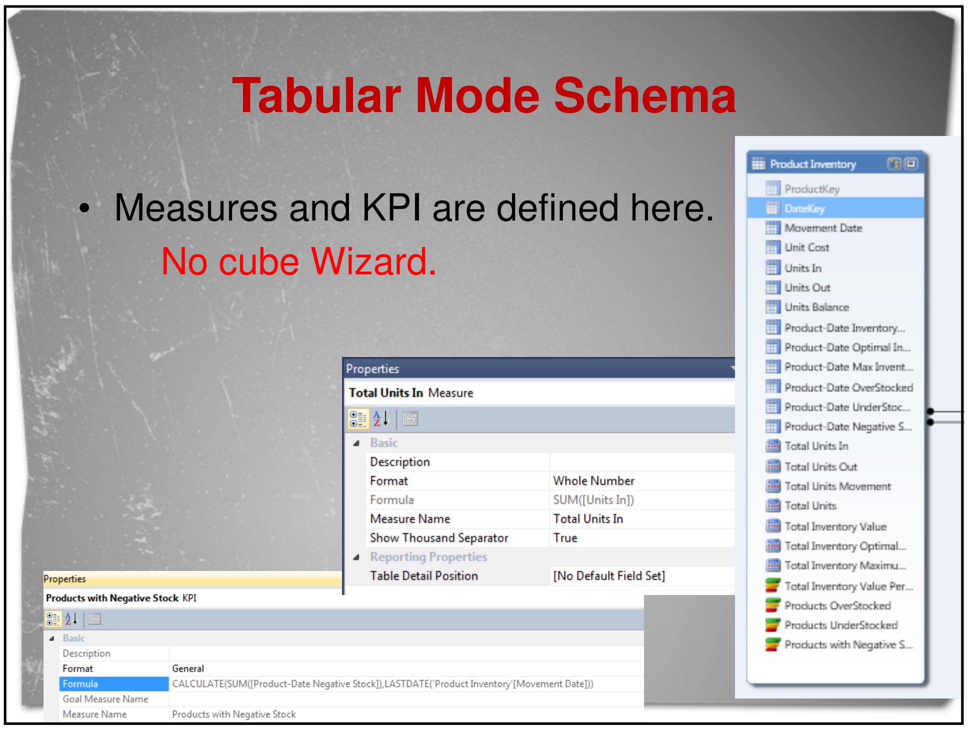
Tabular Mode Schema

- Dimension = One Table
- No Dimension Wizard to create Hierarchies

Properties

ProductKey Column

Basic	
Column Nam	ProductKey
Data Format	General
Data Type	Whole Number
Description	
Hidden	True
Sort By Colun	
Reporting Properties	
Default Image	False
Default Label	False
Image URL	False
Keep Unique	True
Row Identifier	True
Summarize By	Default
Table Detail P	[No Default Field Se



Tabular Mode Schema

- Measures and KPI are defined here.
  No cube Wizard.

Product Inventory
ProductKey
DateKey
Movement Date
Unit Cost
Units In
Units Out
Units Balance
Product-Date Inventory...
Product-Date Optimal In...
Product-Date Max Invent...
Product-Date OverStocked
Product-Date UnderStoc...
Product-Date Negative S...
Total Units In
Total Units Out
Total Units Movement
Total Units
Total Inventory Value
Total Inventory Optimal...
Total Inventory Maximu...
Total Inventory Value Per...
Products OverStocked
Products UnderStocked
Products with Negative S...

Properties

Total Units In Measure

Basic	
Description	
Format	Whole Number
Formula	SUM([Units In])
Measure Name	Total Units In
Show Thousand Separator	True
Reporting Properties	
Table Detail Position	[No Default Field Set]

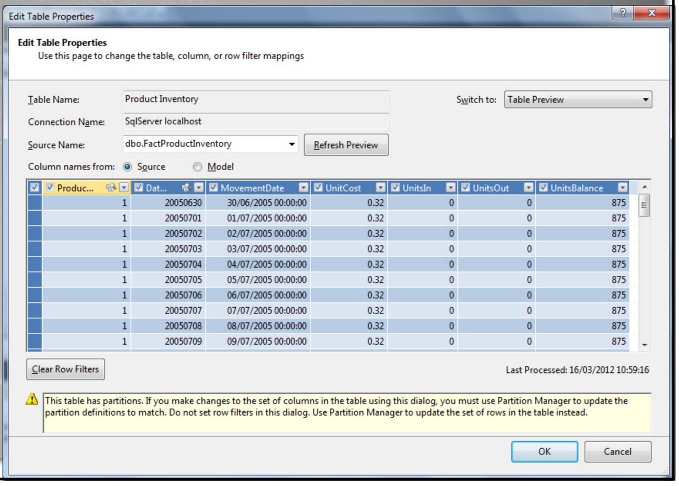
Properties

Products with Negative Stock KPI

Basic	
Description	
Format	General
Formula	CALCULATE(SUM([Product-Date Negative Stock]),LASTDATE('Product Inventory'[Movement Date]))
Goal Measure Name	
Measure Name	Products with Negative Stock

Tabular Mode Schema

- Edit Table Properties



Edit Table Properties

Edit Table Properties
Use this page to change the table, column, or row filter mappings

Table Name: Product Inventory
Connection Name: SqlServer localhost
Source Name: dbo.FactProductInventory
Switch to: Table Preview
Column names from: [✓ Source] Model

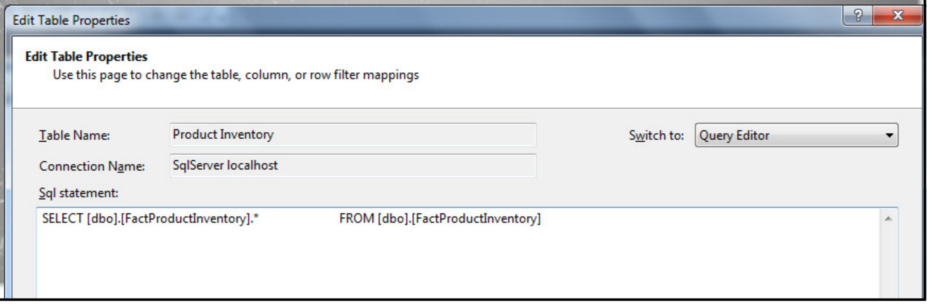
Produc...	Dat...	MovementDate	UnitCost	UnitsIn	UnitsOut	UnitsBalance
1	20050630	30/06/2005 00:00:00	0.32	0	0	875
1	20050701	01/07/2005 00:00:00	0.32	0	0	875
1	20050702	02/07/2005 00:00:00	0.32	0	0	875
1	20050703	03/07/2005 00:00:00	0.32	0	0	875
1	20050704	04/07/2005 00:00:00	0.32	0	0	875
1	20050705	05/07/2005 00:00:00	0.32	0	0	875
1	20050706	06/07/2005 00:00:00	0.32	0	0	875
1	20050707	07/07/2005 00:00:00	0.32	0	0	875
1	20050708	08/07/2005 00:00:00	0.32	0	0	875
1	20050709	09/07/2005 00:00:00	0.32	0	0	875

Last Processed: 16/03/2012 10:59:16

This table has partitions. If you make changes to the set of columns in the table using this dialog, you must use Partition Manager to update the partition definitions to match. Do not set row filters in this dialog. Use Partition Manager to update the set of rows in the table instead.

Tabular Mode Schema- Adding a column

- Adding new calculated columns in the schema is possible.
- Either add calculated columns to the source DB or to the BISM

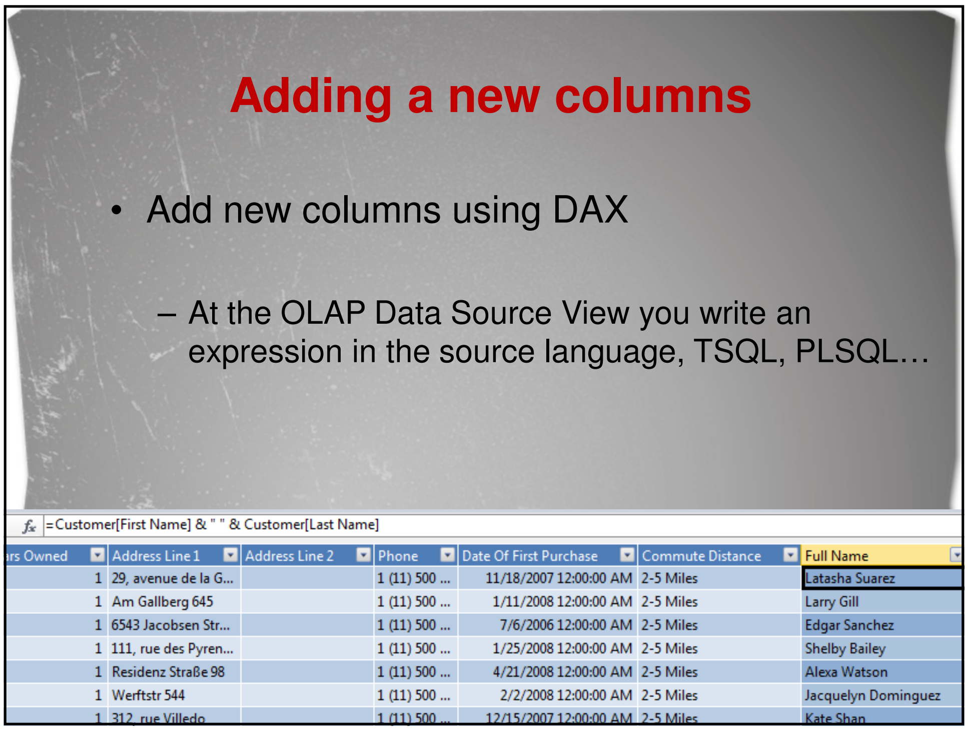


Edit Table Properties

Edit Table Properties
Use this page to change the table, column, or row filter mappings

Table Name: Product Inventory
Connection Name: SqlServer localhost
Switch to: Query Editor
Sql statement:

```
SELECT [dbo].[FactProductInventory].*            FROM [dbo].[FactProductInventory]
```

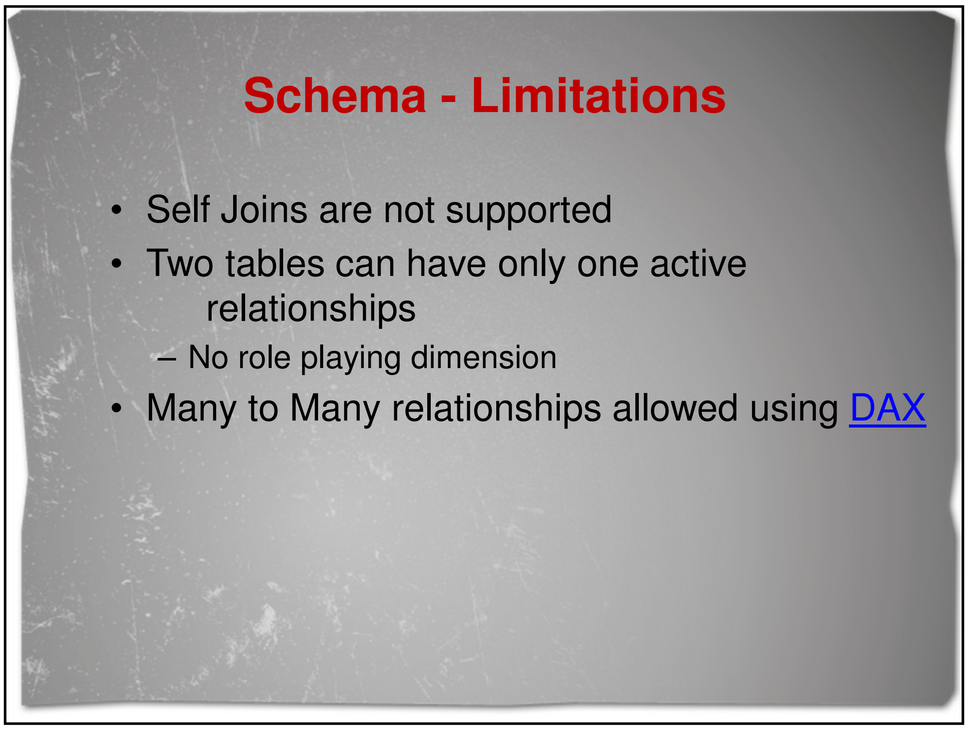


Adding a new columns

- Add new columns using DAX
  - At the OLAP Data Source View you write an expression in the source language, TSQL, PLSQL…

```
=Customer[First Name] & " " & Customer[Last Name]
```

rs Owned	Address Line 1	Address Line 2	Phone	Date Of First Purchase	Commute Distance	Full Name
1	29, avenue de la G...		1 (11) 500 ...	11/18/2007 12:00:00 AM	2-5 Miles	Latasha Suarez
1	Am Gallberg 645		1 (11) 500 ...	1/11/2008 12:00:00 AM	2-5 Miles	Larry Gill
1	6543 Jacobsen Str...		1 (11) 500 ...	7/6/2006 12:00:00 AM	2-5 Miles	Edgar Sanchez
1	111, rue des Pyren...		1 (11) 500 ...	1/25/2008 12:00:00 AM	2-5 Miles	Shelby Bailey
1	Residenz Straße 98		1 (11) 500 ...	4/21/2008 12:00:00 AM	2-5 Miles	Alexa Watson
1	Werftstr 544		1 (11) 500 ...	2/2/2008 12:00:00 AM	2-5 Miles	Jacquelyn Dominguez
1	312, rue Villedo		1 (11) 500 ...	12/15/2007 12:00:00 AM	2-5 Miles	Kate Shan



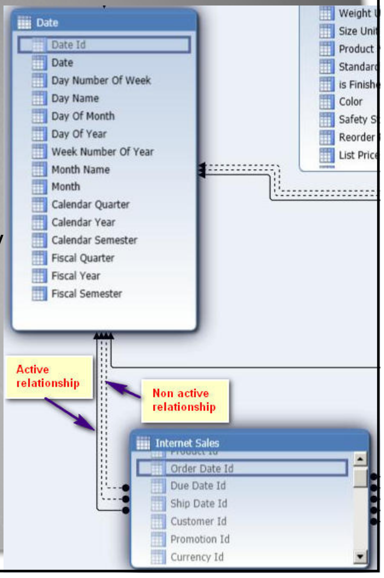
Schema - Limitations

- Self Joins are not supported
- Two tables can have only one active relationships
  - No role playing dimension
- Many to Many relationships allowed using DAX

Dimensions

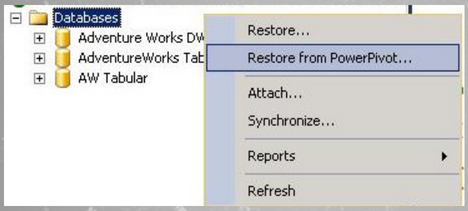
- Active / non active relationships
  - Only ONE relationship can be active
  - No Role Playing Dimensions
  - The Date Dimension can be connected by either OrderDate or ShipDate or DueDate

You'll have to build many date dimensions



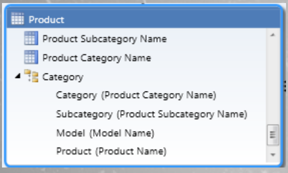
Main Features / Capabilities

- Import Existing PowerPivot model
  - Let users to quickly build model using Excel and upload it to the server in a click

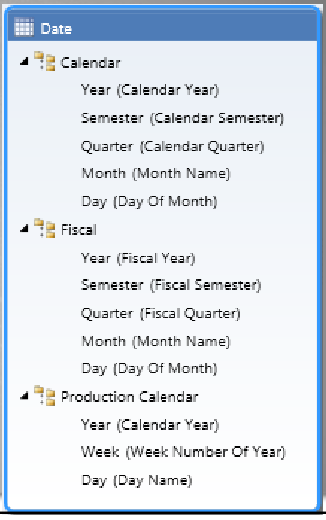


Dimensions

- Dimensions
- Multiple Hierarchies



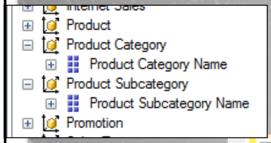
```
Product
  Product Subcategory Name
  Product Category Name
  ◢ Category
      Category (Product Category Name)
      Subcategory (Product Subcategory Name)
      Model (Model Name)
      Product (Product Name)
```



```
Date
◢ Calendar
    Year (Calendar Year)
    Semester (Calendar Semester)
    Quarter (Calendar Quarter)
    Month (Month Name)
    Day (Day Of Month)
◢ Fiscal
    Year (Fiscal Year)
    Semester (Fiscal Semester)
    Quarter (Fiscal Quarter)
    Month (Month Name)
    Day (Day Of Month)
◢ Production Calendar
    Year (Calendar Year)
    Week (Week Number Of Year)
    Day (Day Name)
```

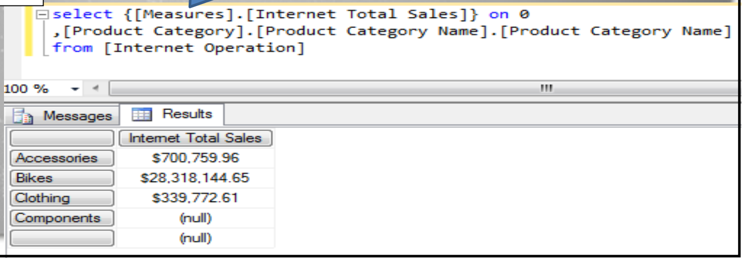
Snowflake Dimensions

- A dimension is based on one table / view only.
  - A hierarchy can be based on one Table.
  - Create a view on the data source which joins the tables.



```
⊞ Product
⊟ Product Category
    ⊞ Product Category Name
⊟ Product Subcategory
    ⊞ Product Subcategory Name
⊞ Promotion
```

Category name from Dim Category with a measure works as expected



```
select {[Measures].[Internet Total Sales]} on 0
,[Product Category].[Product Category Name].[Product Category Name]
from [Internet Operation]
```

100 %

Messages | Results

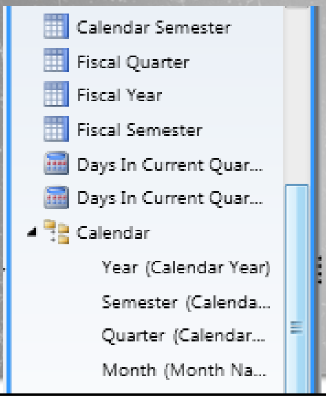
	Internet Total Sales
Accessories	$700,759.96
Bikes	$28,318,144.65
Clothing	$339,772.61
Components	(null)
	(null)

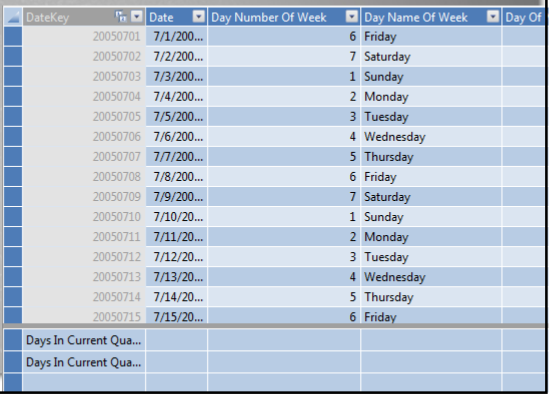
Dimensions - Limitations

- No member properties
- No Parent-Child
- No linked dimensions
- No need for Attribute Relationships
- The dimension is a table
  - A dimension can be based on one and only one table
  - Dimension names taken directly from the
- Large dimensions (million members) performed better

Measures

- Sum, min, max, count, distinct count…or complex DAX Expressions
- A table can contain both attributes and





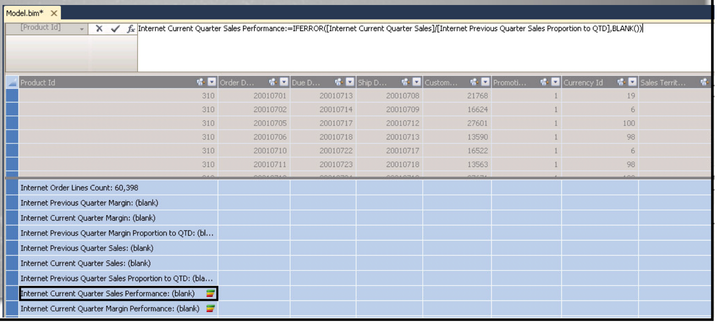
DateKey	Date	Day Number Of Week	Day Name Of Week	Day Of
20050701	7/1/200...	6	Friday	
20050702	7/2/200...	7	Saturday	
20050703	7/3/200...	1	Sunday	
20050704	7/4/200...	2	Monday	
20050705	7/5/200...	3	Tuesday	
20050706	7/6/200...	4	Wednesday	
20050707	7/7/200...	5	Thursday	
20050708	7/8/200...	6	Friday	
20050709	7/9/200...	7	Saturday	
20050710	7/10/20...	1	Sunday	
20050711	7/11/20...	2	Monday	
20050712	7/12/20...	3	Tuesday	
20050713	7/13/20...	4	Wednesday	
20050714	7/14/20...	5	Thursday	
20050715	7/15/20...	6	Friday	
Days In Current Qua...				
Days In Current Qua...				

Multi Grain Measure Groups

- Support many "Measure Groups" with different dimensionality
- For ex. Internet Sales (product level), Sales Quota (Category Level)

KPI

- Key performance Indicators are the heart of any BI solution
- Replaces existing Measures


Model.bim*

[Product Id]

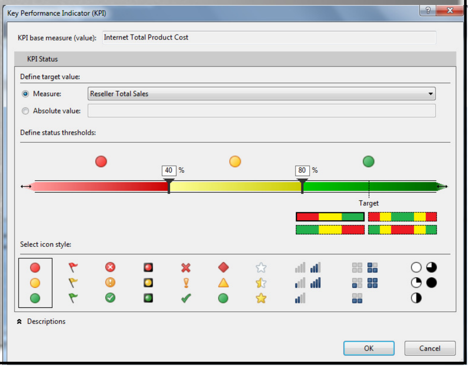
```
Internet Current Quarter Sales Performance:=IFERROR([Internet Current Quarter Sales]/[Internet Previous Quarter Sales Proportion to QTD],BLANK())
```

Product Id	Order D...	Due D...	Ship D...	Custom...	Promoti...	Currency Id	Sales Territ...
310	20010701	20010713	20010708	21768	1	19	
310	20010702	20010714	20010709	16624	1	6	
310	20010705	20010717	20010712	27601	1	100	
310	20010706	20010718	20010713	13590	1	98	
310	20010710	20010722	20010717	16522	1	6	
310	20010711	20010723	20010718	13563	1	98	

Internet Order Lines Count: 60,398
Internet Previous Quarter Margin: (blank)
Internet Current Quarter Margin: (blank)
Internet Previous Quarter Margin Proportion to QTD: (bl...
Internet Previous Quarter Sales: (blank)
Internet Current Quarter Sales: (blank)
Internet Previous Quarter Sales Proportion to QTD: (bla...
Internet Current Quarter Sales Performance: (blank)
Internet Current Quarter Margin Performance: (blank)

KPI

- Easy to create using a wizard
  - Value – existing measure
  - Target – measure or a number
  - Status – using a GUI



Key Performance Indicator (KPI)

KPI base measure (value): Internet Total Product Cost

KPI Status

Define target value:

(●) Measure: Reseller Total Sales

( ) Absolute value:

Define status thresholds:

40 % 80 %

Target

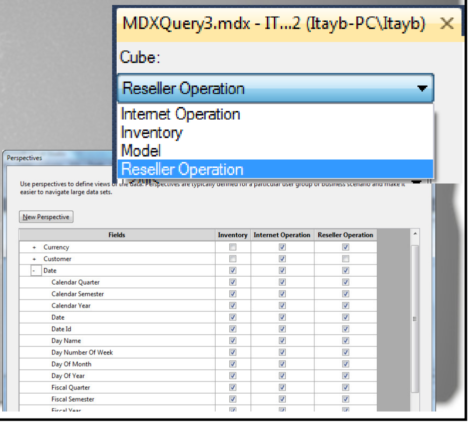
Select icon style:

Descriptions

OK Cancel

Perspectives

- Same as in SSAS Multidimensional
- Allows users to see a Database in a simpler way
- Can hide:
  - Tables (dimensions)
  - Columns (Attributes)
  - Hierarchies
  - Measures
  - KPI



MDXQuery3.mdx - IT...2 (Itayb-PC\Itayb)

Cube: Reseller Operation

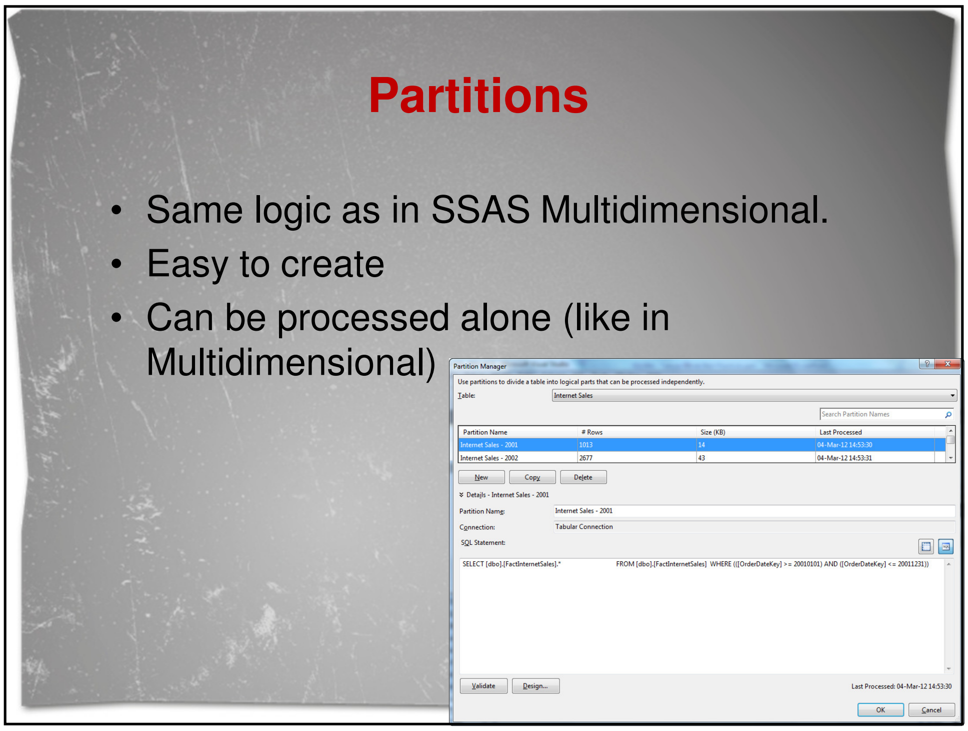
Internet Operation
Inventory
Model
Reseller Operation

Perspectives

Use perspectives to define views of the data. Perspectives are typically defined for a particular user group or business scenario and make it easier to navigate large data sets.

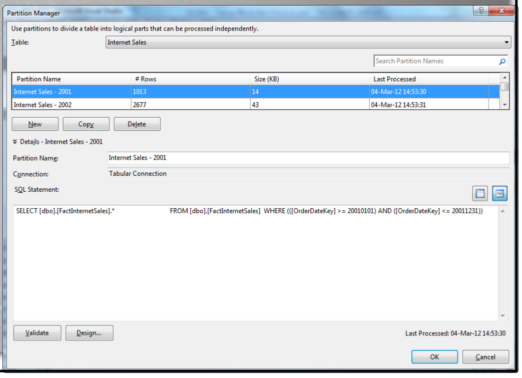
New Perspective

Fields	Inventory	Internet Operation	Reseller Operation
+ Currency	☐	✓	✓
+ Customer	☐	✓	☐
- Date	✓	✓	✓
Calendar Quarter	✓	✓	✓
Calendar Semester	✓	✓	✓
Calendar Year	✓	✓	✓
Date	✓	✓	✓
Date Id	✓	✓	✓
Day Name	✓	✓	✓
Day Number Of Week	✓	✓	✓
Day Of Month	✓	✓	✓
Day Of Year	✓	✓	✓
Fiscal Quarter	✓	✓	✓
Fiscal Semester	✓	✓	✓
Fiscal Year	✓	✓	✓



Partitions

- Same logic as in SSAS Multidimensional.
- Easy to create
- Can be processed alone (like in Multidimensional)



Partition Manager

Use partitions to divide a table into logical parts that can be processed independently.

Table: Internet Sales

Partition Name	# Rows	Size (KB)	Last Processed
Internet Sales - 2001	1013	14	04-Mar-12 14:53:30
Internet Sales - 2002	2677	43	04-Mar-12 14:53:31

Details - Internet Sales - 2001

Partition Name: Internet Sales - 2001

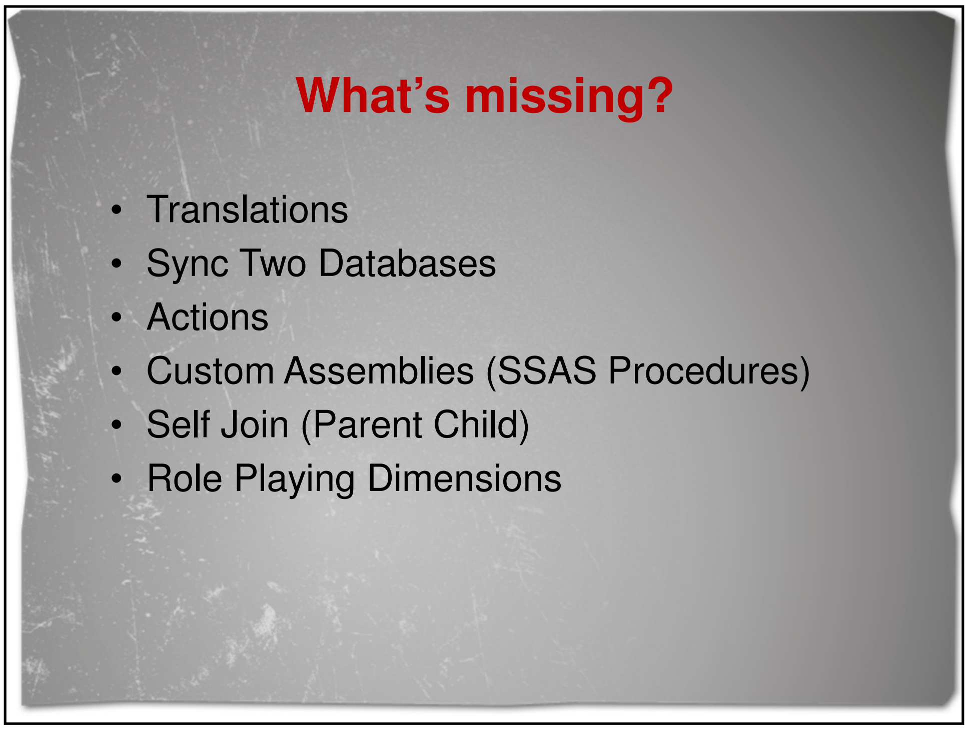
Connection: Tabular Connection

SQL Statement:

```
SELECT [dbo].[FactInternetSales].*                    FROM [dbo].[FactInternetSales]  WHERE (([OrderDateKey] >= 20010101) AND ([OrderDateKey] <= 20011231))
```

Last Processed: 04-Mar-12 14:53:30

Search Partition Names
New
Copy
Delete
Validate
Design...
OK
Cancel



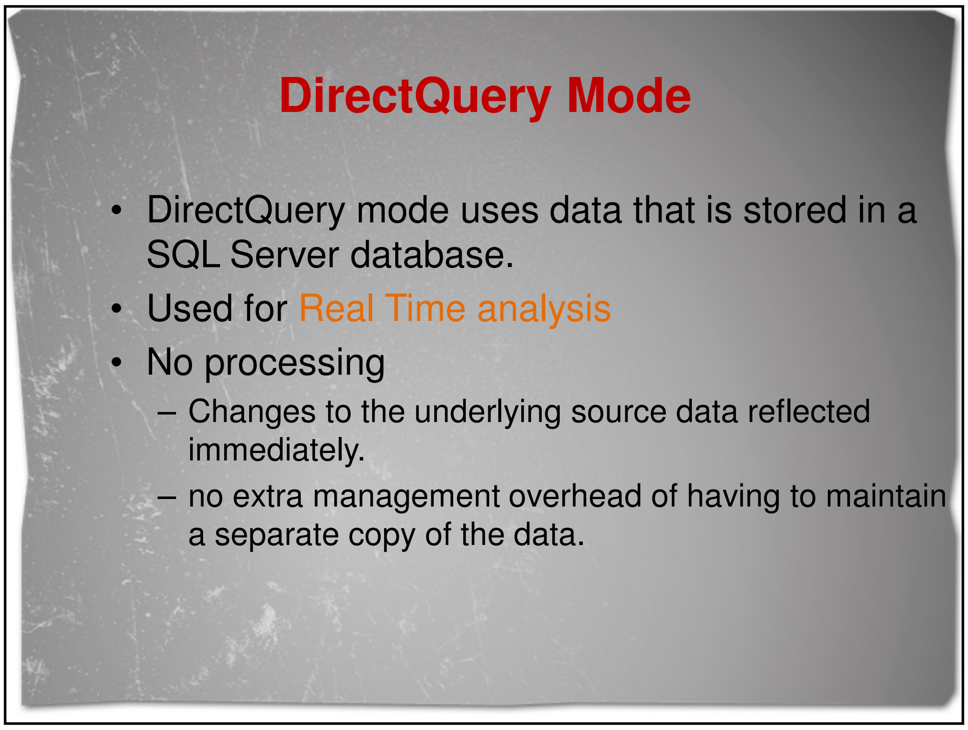
What's missing?

- Translations
- Sync Two Databases
- Actions
- Custom Assemblies (SSAS Procedures)
- Self Join (Parent Child)
- Role Playing Dimensions

More Limitations

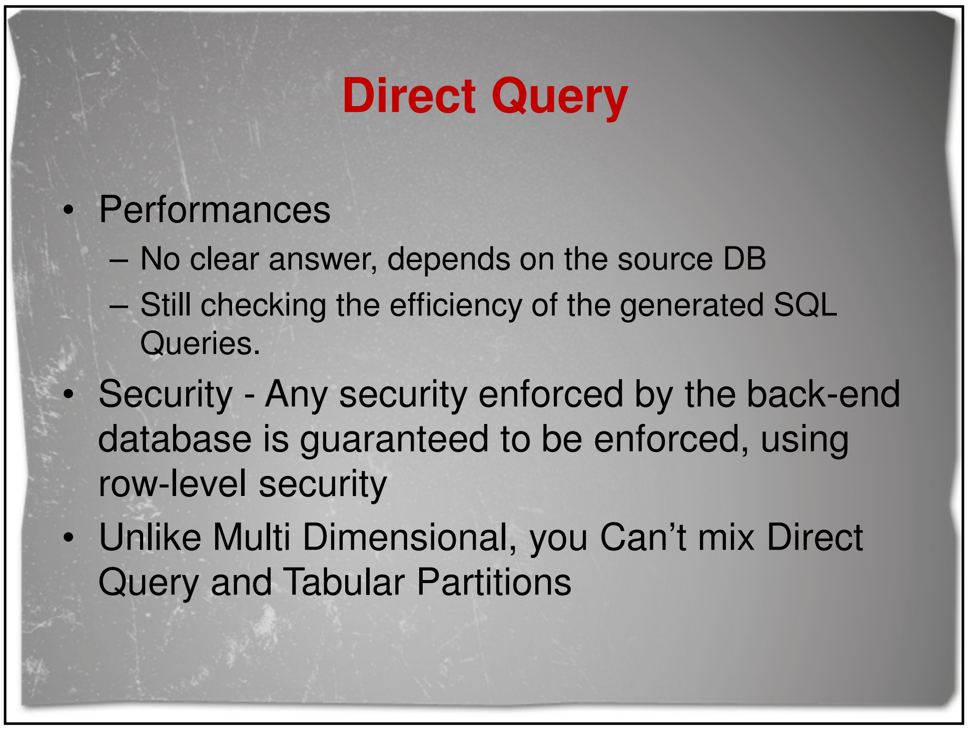
- Named Set
- Scopes
- Write Back

Direct Query



DirectQuery Mode

- DirectQuery mode uses data that is stored in a SQL Server database.
- Used for Real Time analysis
- No processing
  – Changes to the underlying source data reflected immediately.
  – no extra management overhead of having to maintain a separate copy of the data.

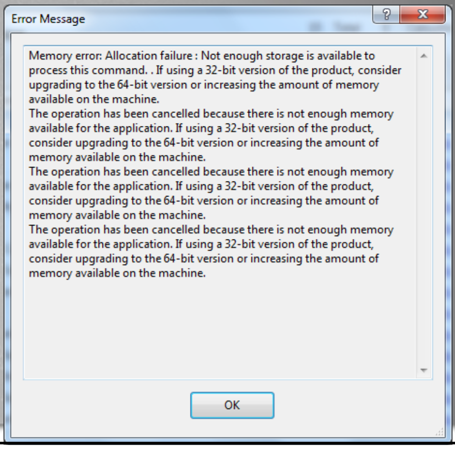


Direct Query

- Performances
  – No clear answer, depends on the source DB
  – Still checking the efficiency of the generated SQL Queries.
- Security - Any security enforced by the back-end database is guaranteed to be enforced, using row-level security
- Unlike Multi Dimensional, you Can't mix Direct Query and Tabular Partitions

DirectQuery Mode

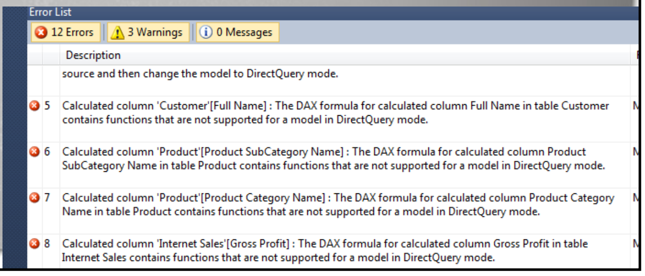
- It is possible to have a model over data sets that are too large to fit in memory on the Analysis Services server.



```
Memory error: Allocation failure : Not enough storage is available to
process this command. . If using a 32-bit version of the product, consider
upgrading to the 64-bit version or increasing the amount of memory
available on the machine.
The operation has been cancelled because there is not enough memory
available for the application. If using a 32-bit version of the product,
consider upgrading to the 64-bit version or increasing the amount of
memory available on the machine.
The operation has been cancelled because there is not enough memory
available for the application. If using a 32-bit version of the product,
consider upgrading to the 64-bit version or increasing the amount of
memory available on the machine.
The operation has been cancelled because there is not enough memory
available for the application. If using a 32-bit version of the product,
consider upgrading to the 64-bit version or increasing the amount of
memory available on the machine.
```

Direct Query Limitations

- Can only use one SQL Server DB as a source
- The entire model is either is either using DirectQuery or not.
- Limited use of DAX functions
- Client restrictions: Can only be queried by using DAX
  - Excel can't be used because it uses MDX



	Description
	source and then change the model to DirectQuery mode.
5	Calculated column 'Customer'[Full Name] : The DAX formula for calculated column Full Name in table Customer contains functions that are not supported for a model in DirectQuery mode.
6	Calculated column 'Product'[Product SubCategory Name] : The DAX formula for calculated column Product SubCategory Name in table Product contains functions that are not supported for a model in DirectQuery mode.
7	Calculated column 'Product'[Product Category Name] : The DAX formula for calculated column Product Category Name in table Product contains functions that are not supported for a model in DirectQuery mode.
8	Calculated column 'Internet Sales'[Gross Profit] : The DAX formula for calculated column Gross Profit in table Internet Sales contains functions that are not supported for a model in DirectQuery mode.

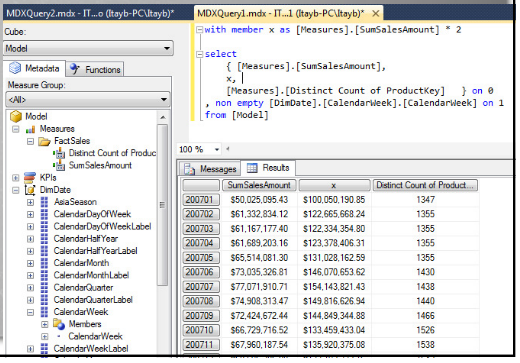
DAX vs. MDX

DAX vs. MDX

- MDX (Multi Dimensional Cube) vs. DAX
- MDX (Tabular) vs. DAX (Tabular)

Querying a Tabular Model using MDX

- Quick start – just use MDX to query the model
  - You don't have to learn DAX
  - In general speaking, MDW performs well
  - Fine tuning DAX queries

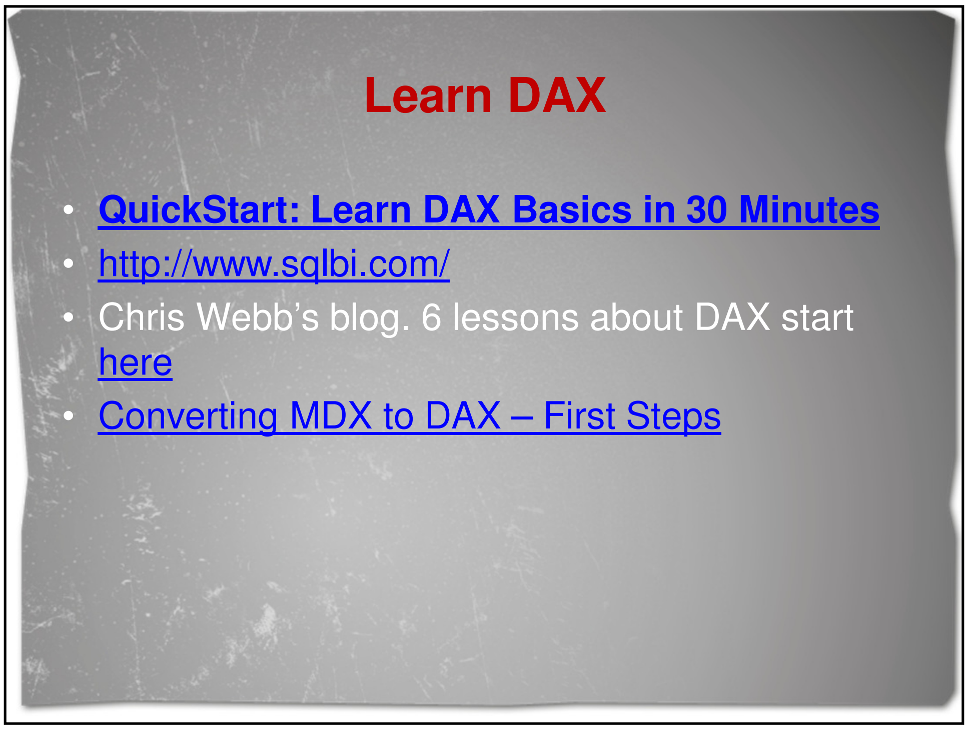


```
with member x as [Measures].[SumSalesAmount] * 2

select
    { [Measures].[SumSalesAmount],
    x,
    [Measures].[Distinct Count of ProductKey]   } on 0
, non empty [DimDate].[CalendarWeek].[CalendarWeek] on 1
from [Model]
```

	SumSalesAmount	x	Distinct Count of Product...
200701	$50,025,095.43	$100,050,190.85	1347
200702	$61,332,834.12	$122,665,668.24	1355
200703	$61,167,177.40	$122,334,354.80	1355
200704	$61,689,203.16	$123,378,406.31	1355
200705	$65,514,081.30	$131,028,162.59	1355
200706	$73,035,326.81	$146,070,653.62	1430
200707	$77,071,910.71	$154,143,821.43	1438
200708	$74,908,313.47	$149,816,626.94	1440
200709	$72,424,672.44	$144,849,344.88	1466
200710	$66,729,716.52	$133,459,433.04	1526
200711	$67,960,187.54	$135,920,375.08	1538

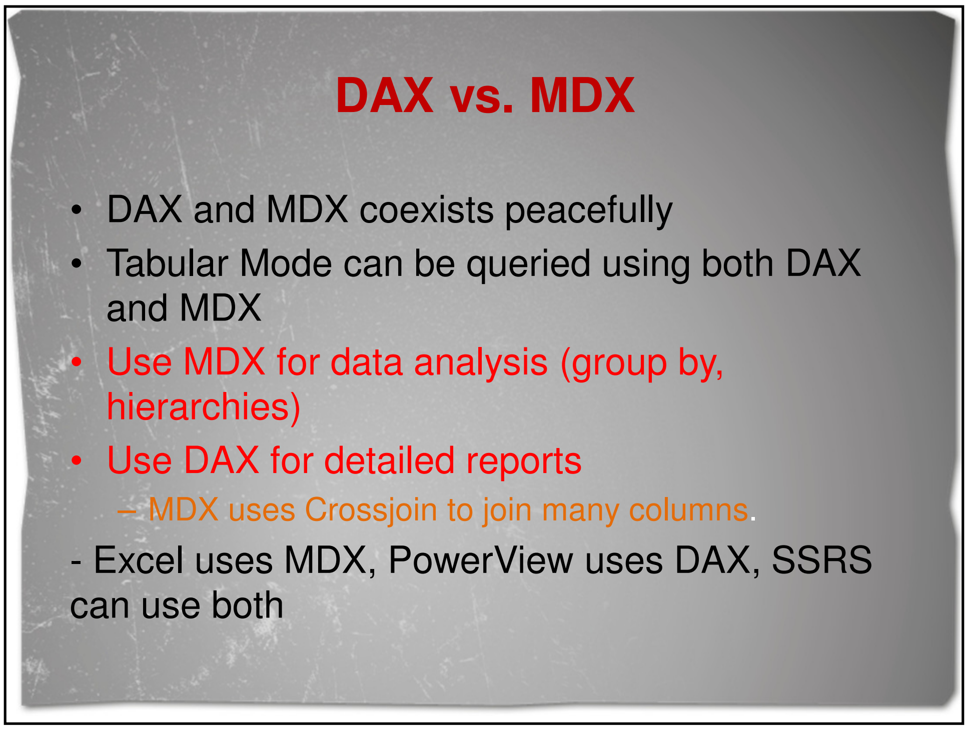
What is DAX

- Set of Excel-like formulas
- Enables advanced data modeling inside the PowerPivot for Excel or Tabular Mode
- Two types of calculations
  - **calculated columns**
  - **measures**



Learn DAX

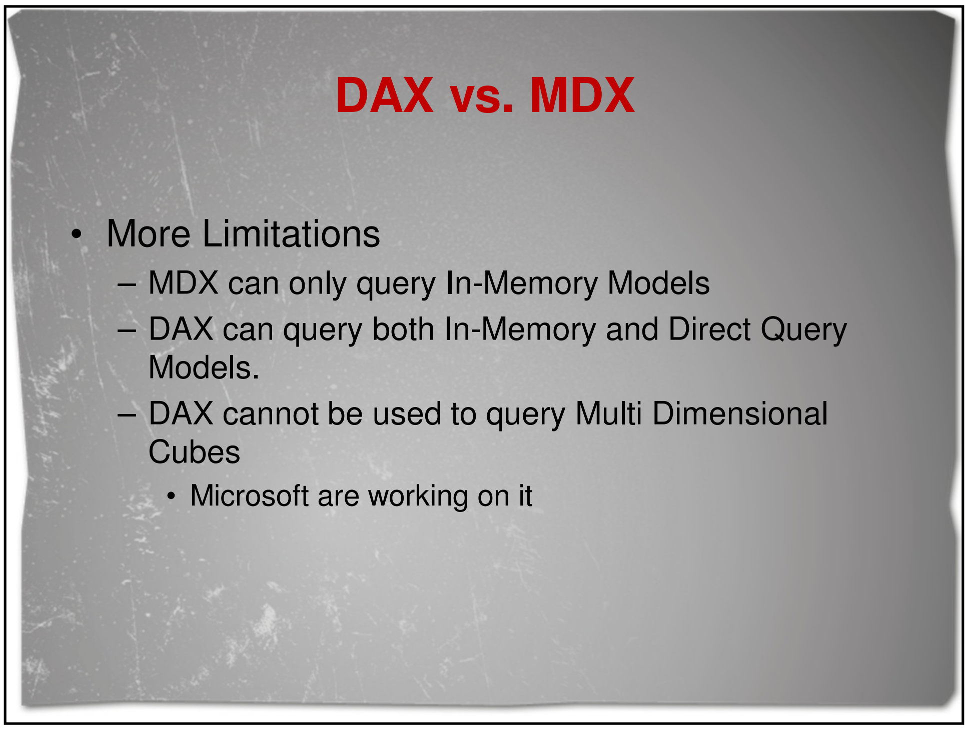
- **QuickStart: Learn DAX Basics in 30 Minutes**
- http://www.sqlbi.com/
- Chris Webb's blog. 6 lessons about DAX start here
- Converting MDX to DAX – First Steps



DAX vs. MDX

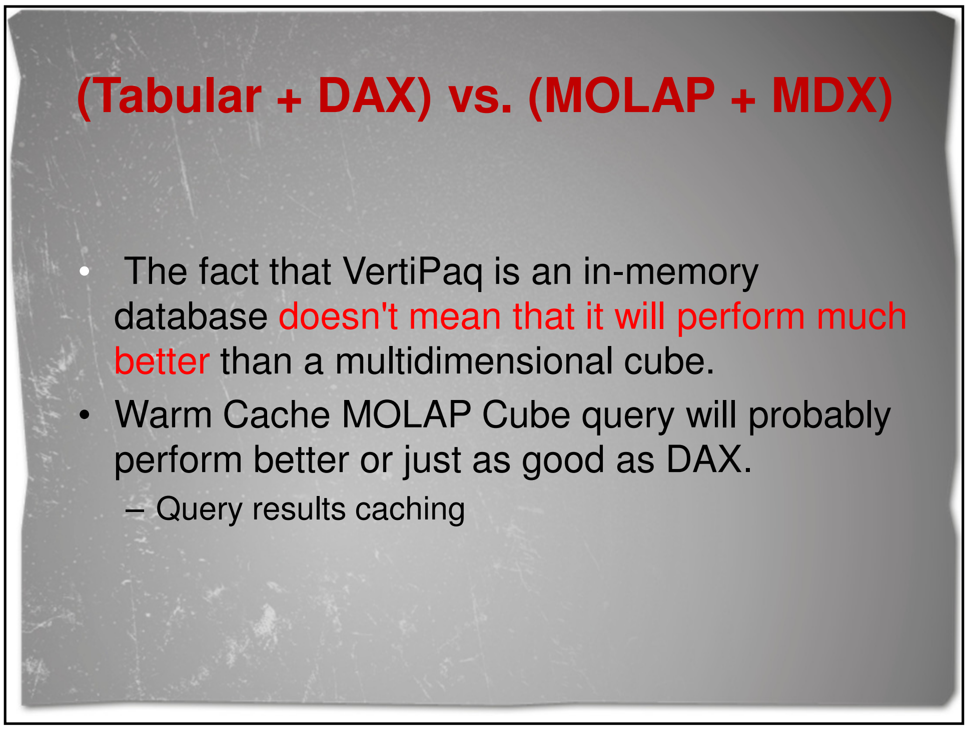
- DAX and MDX coexists peacefully
- Tabular Mode can be queried using both DAX and MDX
- Use MDX for data analysis (group by, hierarchies)
- Use DAX for detailed reports
  - MDX uses Crossjoin to join many columns.

- Excel uses MDX, PowerView uses DAX, SSRS can use both



DAX vs. MDX

- More Limitations
  - MDX can only query In-Memory Models
  - DAX can query both In-Memory and Direct Query Models.
  - DAX cannot be used to query Multi Dimensional Cubes
    - Microsoft are working on it



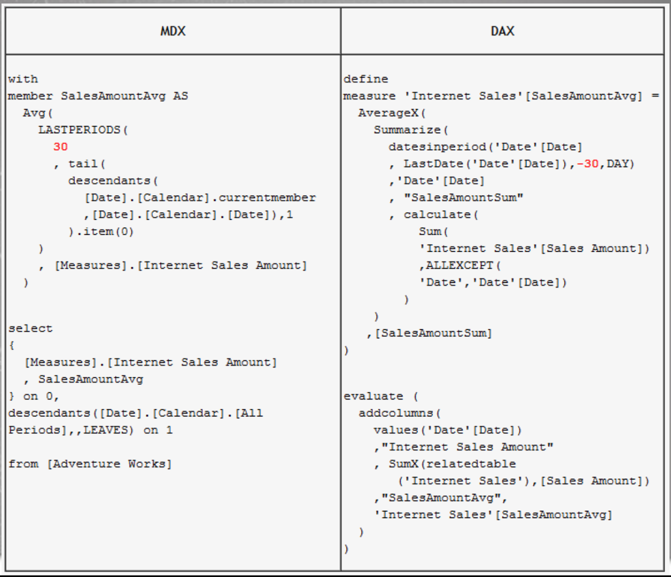
(Tabular + DAX) vs. (MOLAP + MDX)

- The fact that VertiPaq is an in-memory database doesn't mean that it will perform much better than a multidimensional cube.
- Warm Cache MOLAP Cube query will probably perform better or just as good as DAX.
  - Query results caching

(Tabular + DAX) vs. (MOLAP + MDX)

- DAX on Tabular mode usually give you good performance without special tuning.
- DAX queries are never cached
  - Second (warm) run is just as fast the first (cold) one.
- Distinct Count queries usually performed better using DAX + Tabular
  - Overall good performance of DC even with MDX

When number of days increased – the runtime of MDX(CUBE) query is better then DAX (Tabular)



MDX

```
with
member SalesAmountAvg AS
  Avg(
    LASTPERIODS(
      30
      , tail(
        descendants(
          [Date].[Calendar].currentmember
          ,[Date].[Calendar].[Date]),1
        ).item(0)
    )
    , [Measures].[Internet Sales Amount]
  )


select
{
  [Measures].[Internet Sales Amount]
  , SalesAmountAvg
} on 0,
descendants([Date].[Calendar].[All
Periods],,LEAVES) on 1

from [Adventure Works]
```

DAX

```
define
measure 'Internet Sales'[SalesAmountAvg] =
  AverageX(
    Summarize(
      datesinperiod('Date'[Date]
      , LastDate('Date'[Date]),-30,DAY)
      ,'Date'[Date]
      , "SalesAmountSum"
      , calculate(
          Sum(
          'Internet Sales'[Sales Amount])
          ,ALLEXCEPT(
          'Date','Date'[Date])
        )
    )
   ,[SalesAmountSum]
)


evaluate (
  addcolumns(
    values('Date'[Date])
    ,"Internet Sales Amount"
    , SumX(relatedtable
       ('Internet Sales'),[Sales Amount])
    ,"SalesAmountAvg",
    'Internet Sales'[SalesAmountAvg]
  )
)
```

Complex query that run on list of members (Products and Dates) and make aggregations – will return faster in MDX (Cube) (and much faster from the warm cache) than in DAX.

MDX Query

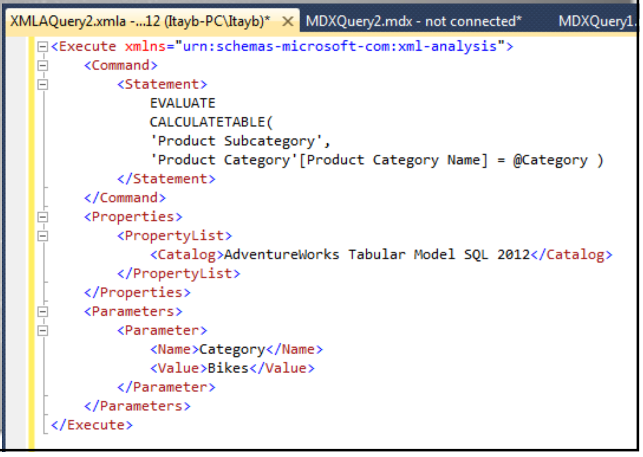
```
WITH

MEMBER diff as iif ([Measures].[Sales Amount] > ([Measures].[Sales Amount],
ParallelPeriod([Date].[Calendar].[Month])), [Measures].[Sales Amount], null)

MEMBER [Measures].SlowAvg AS

Avg

(

[Product].[Product].[Product].MEMBERS, diff

)

SELECT

[Measures].SlowAvg ON 0,

[Date].[Calendar].[Date].Members ON 1

FROM [Adventure Works];
```

Same Query in DAX

```
define measure FactResellerSales[TotalSales] = Sum([SalesAmount])

measure FactResellerSales[TotalSales - LastYear] = [TotalSales]
(SamePeriodLastYear(DimDate[FullDateAlternateKey]), All(DimDate))

measure FactResellerSales[AverageSales] = AverageX(Filter(Values(DimProduct[ProductKey]),
[TotalSales] > [TotalSales - LastYear]), [TotalSales])

evaluate addcolumns(filter(values(DimDate[DateKey]), not isblank([AverageSales])),
"AverageSalesAmount", [AverageSales])

order by [DateKey]
```

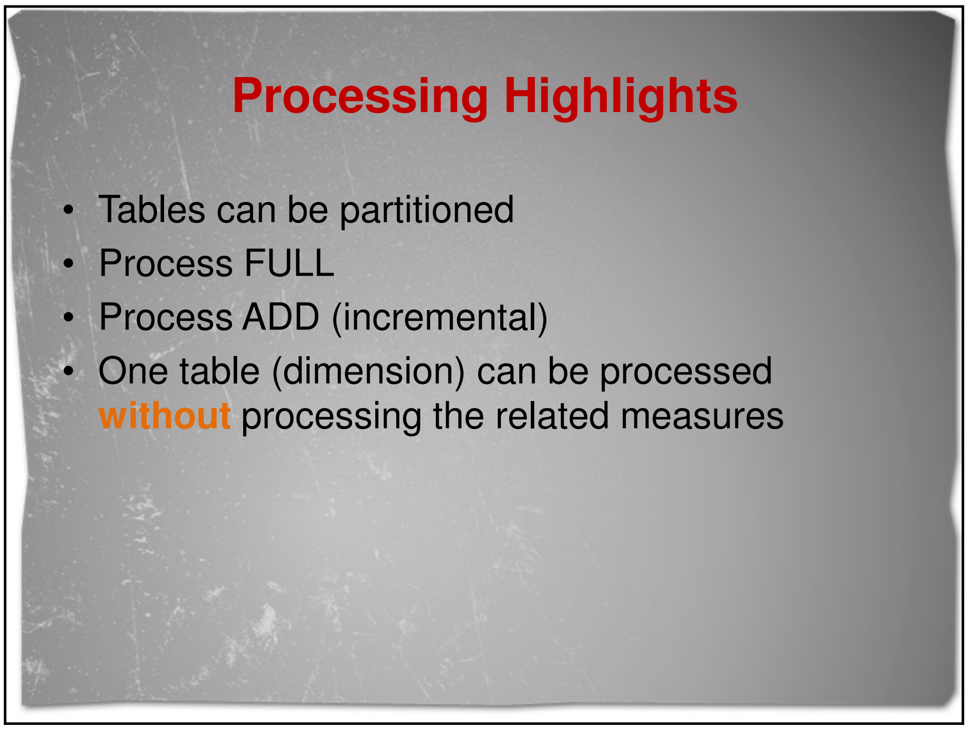
DAX in XMLA

- Easy to manage parameters
- DAX Editor sample


XMLAQuery2.xmla -...12 (Itayb-PC\Itayb)*
MDXQuery2.mdx - not connected*
MDXQuery1.

```
<Execute xmlns="urn:schemas-microsoft-com:xml-analysis">
    <Command>
        <Statement>
            EVALUATE
            CALCULATETABLE(
            'Product Subcategory',
            'Product Category'[Product Category Name] = @Category )
        </Statement>
    </Command>
    <Properties>
        <PropertyList>
            <Catalog>AdventureWorks Tabular Model SQL 2012</Catalog>
        </PropertyList>
    </Properties>
    <Parameters>
        <Parameter>
            <Name>Category</Name>
            <Value>Bikes</Value>
        </Parameter>
    </Parameters>
</Execute>
```

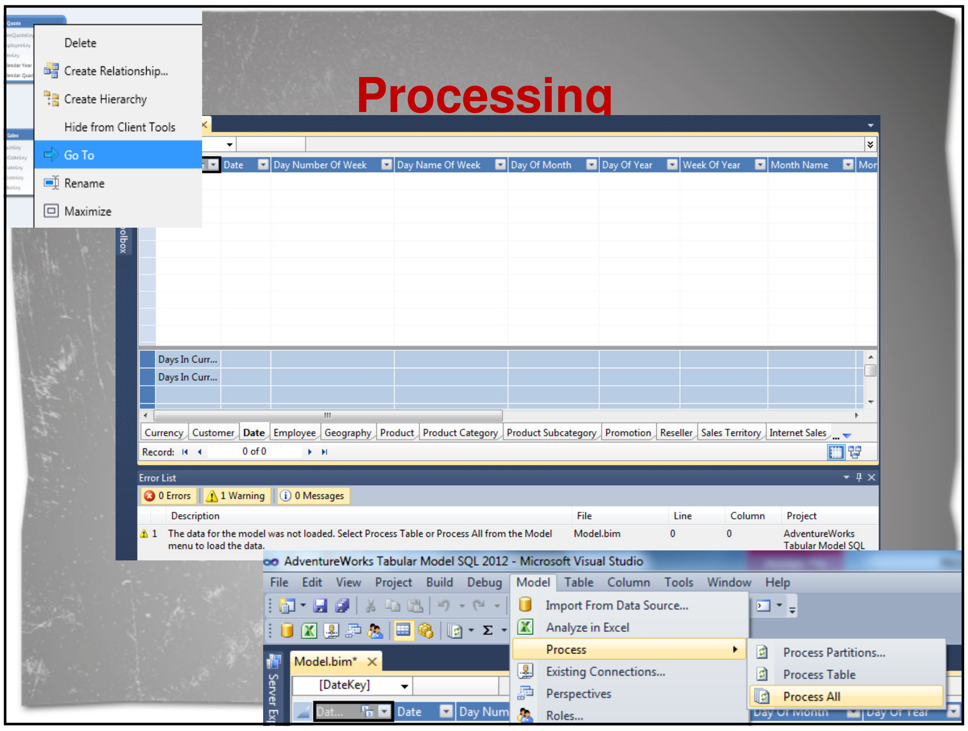
Processing



Processing Highlights

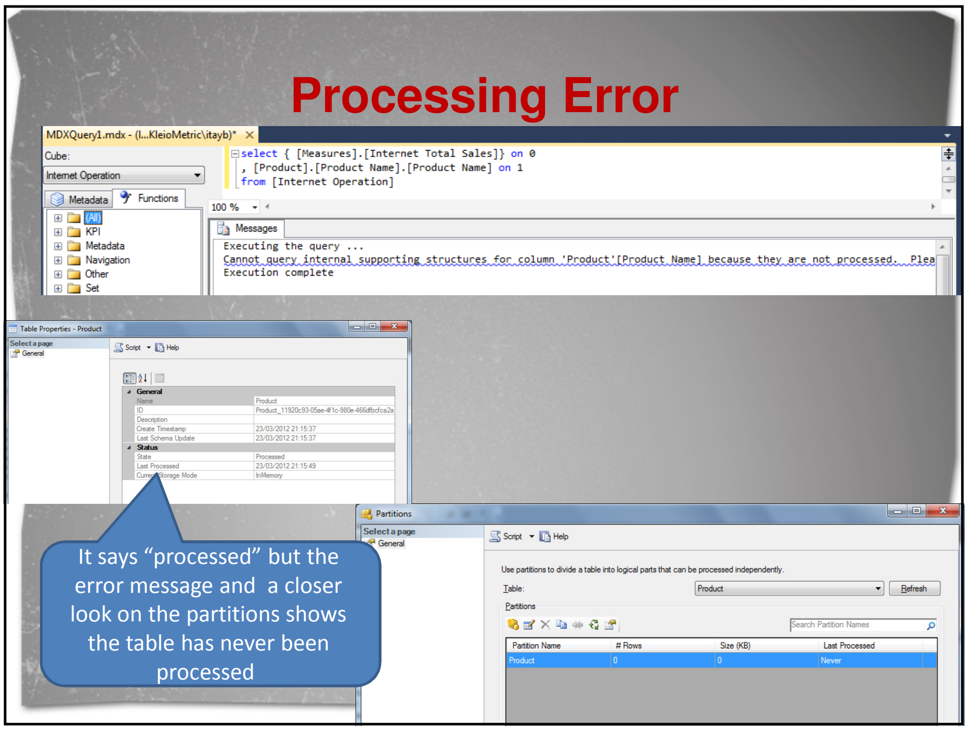
- Tables can be partitioned
- Process FULL
- Process ADD (incremental)
- One table (dimension) can be processed **without** processing the related measures

Processing



	Description	File	Line	Column	Project
1	The data for the model was not loaded. Select Process Table or Process All from the Model menu to load the data.	Model.bim	0	0	AdventureWorks Tabular Model SQL

Processing Error



```
select { [Measures].[Internet Total Sales]} on 0
, [Product].[Product Name].[Product Name] on 1
from [Internet Operation]
```

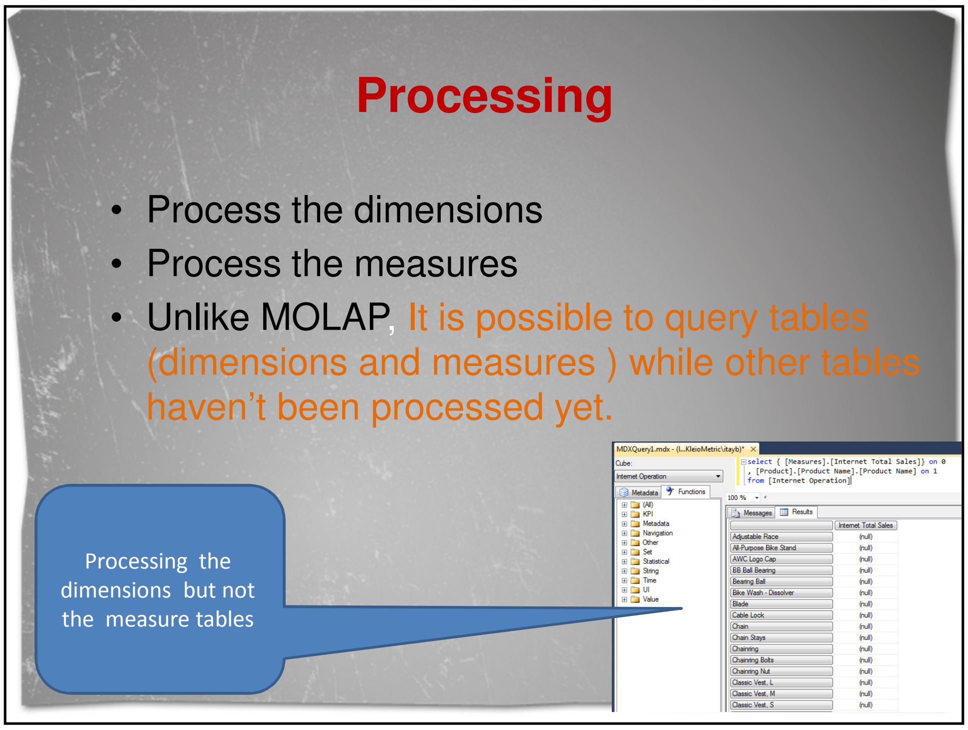
```
Executing the query ...
Cannot query internal supporting structures for column 'Product'[Product Name] because they are not processed.  Plea
Execution complete
```

General	
Name	Product
ID	Product_11920c93-05ae-4f1c-980e-466dfbcfca2a
Description	
Create Timestamp	23/03/2012 21:15:37
Last Schema Update	23/03/2012 21:15:37
Status	
State	Processed
Last Processed	23/03/2012 21:15:49
Current Storage Mode	InMemory

Use partitions to divide a table into logical parts that can be processed independently.

Partition Name	# Rows	Size (KB)	Last Processed
Product	0	0	Never

It says "processed" but the error message and a closer look on the partitions shows the table has never been processed



Processing

- Process the dimensions
- Process the measures
- Unlike MOLAP, It is possible to query tables (dimensions and measures) while other tables haven't been processed yet.

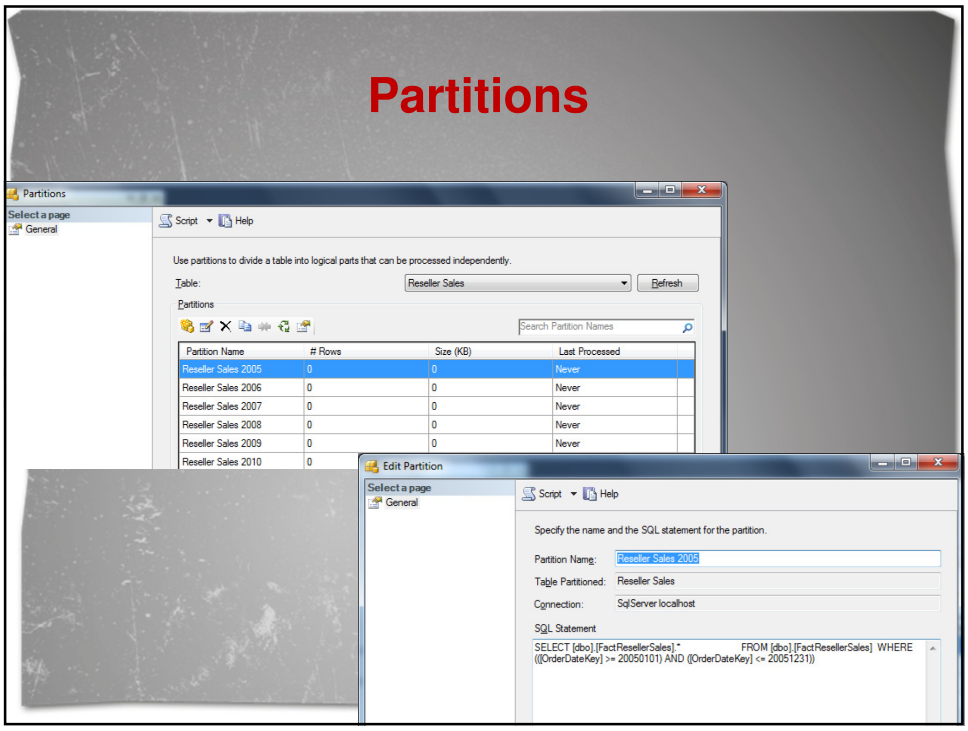
Processing the dimensions but not the measure tables

MDXQuery1.mdx - (l...KleioMetric\itayb)*

Cube: Internet Operation

```
select { [Measures].[Internet Total Sales]} on 0
, [Product].[Product Name].[Product Name] on 1
from [Internet Operation]
```

	Internet Total Sales
Adjustable Race	(null)
All-Purpose Bike Stand	(null)
AWC Logo Cap	(null)
BB Ball Bearing	(null)
Bearing Ball	(null)
Bike Wash - Dissolver	(null)
Blade	(null)
Cable Lock	(null)
Chain	(null)
Chain Stays	(null)
Chainring	(null)
Chainring Bolts	(null)
Chainring Nut	(null)
Classic Vest, L	(null)
Classic Vest, M	(null)
Classic Vest, S	(null)



Partitions

Partitions

Use partitions to divide a table into logical parts that can be processed independently.

Table: Reseller Sales

Partition Name	# Rows	Size (KB)	Last Processed
Reseller Sales 2005	0	0	Never
Reseller Sales 2006	0	0	Never
Reseller Sales 2007	0	0	Never
Reseller Sales 2008	0	0	Never
Reseller Sales 2009	0	0	Never
Reseller Sales 2010	0		

Edit Partition

Specify the name and the SQL statement for the partition.

Partition Name: Reseller Sales 2005

Table Partitioned: Reseller Sales

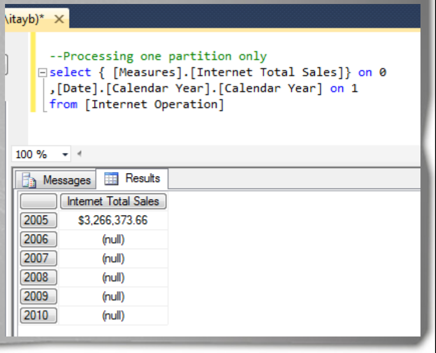
Connection: SqlServer localhost

SQL Statement

```
SELECT [dbo].[FactResellerSales].*          FROM [dbo].[FactResellerSales]  WHERE
(([OrderDateKey] >= 20050101) AND ([OrderDateKey] <= 20051231))
```

Partitions

- Every table can be partitioned.
- Since Dimension = Table, a dimension can be partitioned too.
  - Not sure this is a good idea.

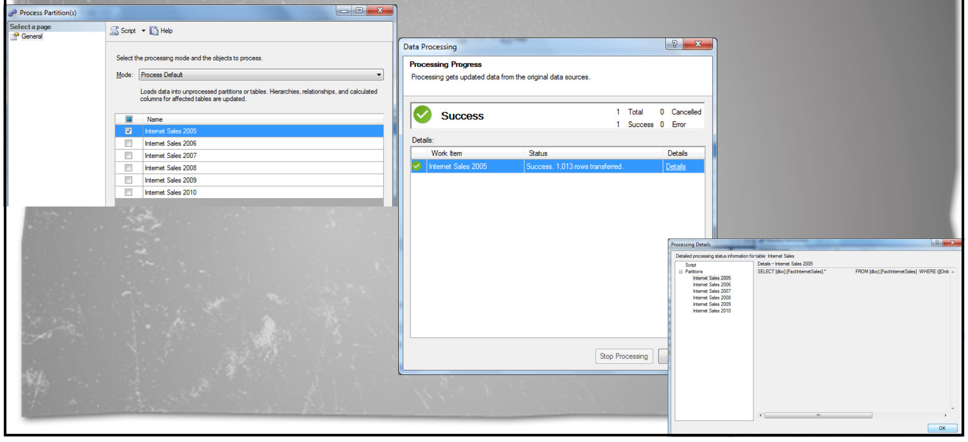


```
--Processing one partition only
select { [Measures].[Internet Total Sales]} on 0
,[Date].[Calendar Year].[Calendar Year] on 1
from [Internet Operation]
```

	Internet Total Sales
2005	$3,266,373.66
2006	(null)
2007	(null)
2008	(null)
2009	(null)
2010	(null)

Partitions

- It is possible to process only some of the partitions



Process Partition(s)

Select the processing mode and the objects to process.

Mode: Process Default

Loads data into unprocessed partitions or tables. Hierarchies, relationships, and calculated columns for affected tables are updated.

	Name
✓	Internet Sales 2005
	Internet Sales 2006
	Internet Sales 2007
	Internet Sales 2008
	Internet Sales 2009
	Internet Sales 2010

Data Processing

Processing Progress

Processing gets updated data from the original data sources.

Success — 1 Total, 0 Cancelled, 1 Success, 0 Error

Details:

Work Item	Status	Details
Internet Sales 2005	Success. 1,013 rows transferred.	Details

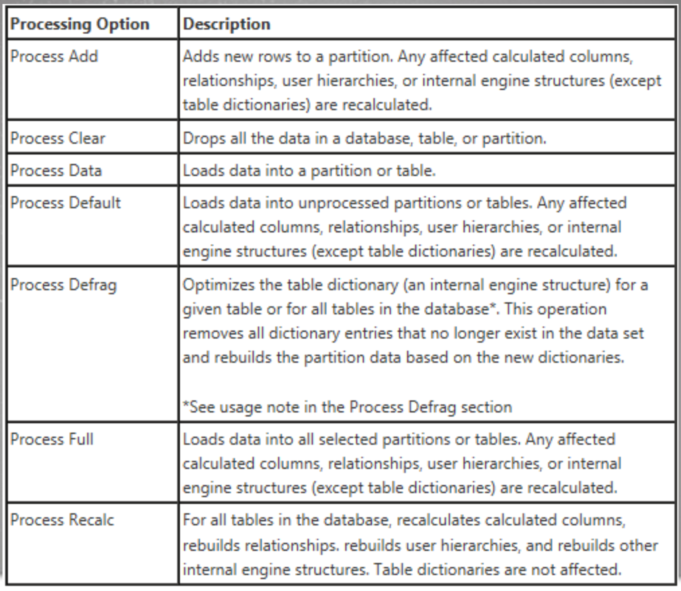
Processing Details

Detailed processing status information for table: Internet Sales

Details – Internet Sales 2005

```
SELECT [dbo].[FactInternetSales].*          FROM [dbo].[FactInternetSales]  WHERE (([Orde
```

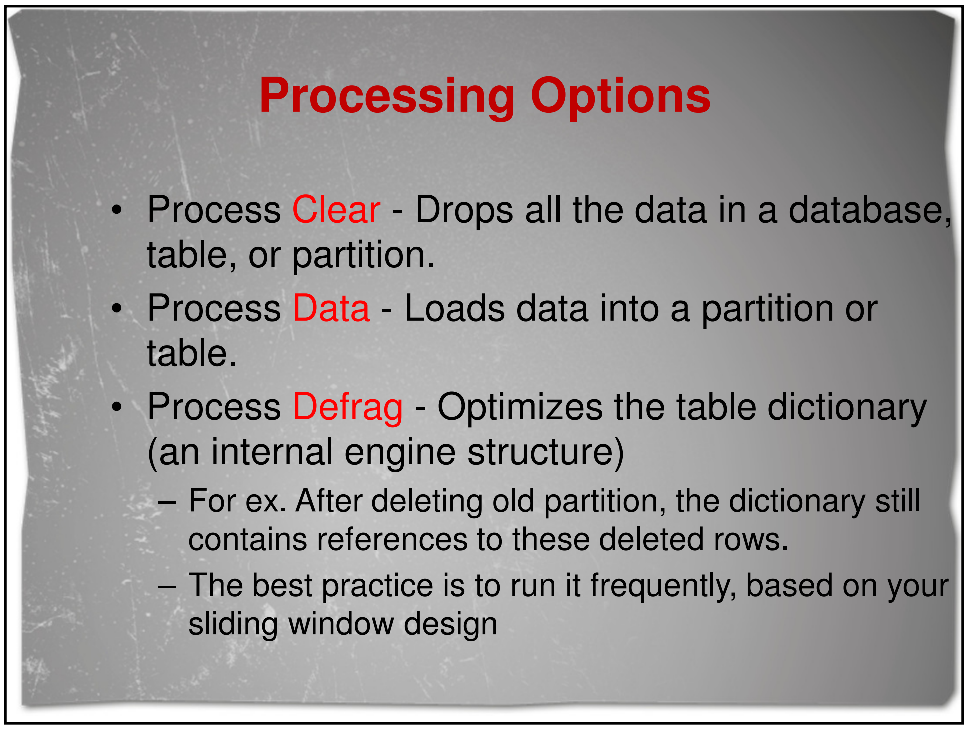
Processing Options



Processing Option	Description
Process Add	Adds new rows to a partition. Any affected calculated columns, relationships, user hierarchies, or internal engine structures (except table dictionaries) are recalculated.
Process Clear	Drops all the data in a database, table, or partition.
Process Data	Loads data into a partition or table.
Process Default	Loads data into unprocessed partitions or tables. Any affected calculated columns, relationships, user hierarchies, or internal engine structures (except table dictionaries) are recalculated.
Process Defrag	Optimizes the table dictionary (an internal engine structure) for a given table or for all tables in the database*. This operation removes all dictionary entries that no longer exist in the data set and rebuilds the partition data based on the new dictionaries. *See usage note in the Process Defrag section
Process Full	Loads data into all selected partitions or tables. Any affected calculated columns, relationships, user hierarchies, or internal engine structures (except table dictionaries) are recalculated.
Process Recalc	For all tables in the database, recalculates calculated columns, rebuilds relationships. rebuilds user hierarchies, and rebuilds other internal engine structures. Table dictionaries are not affected.

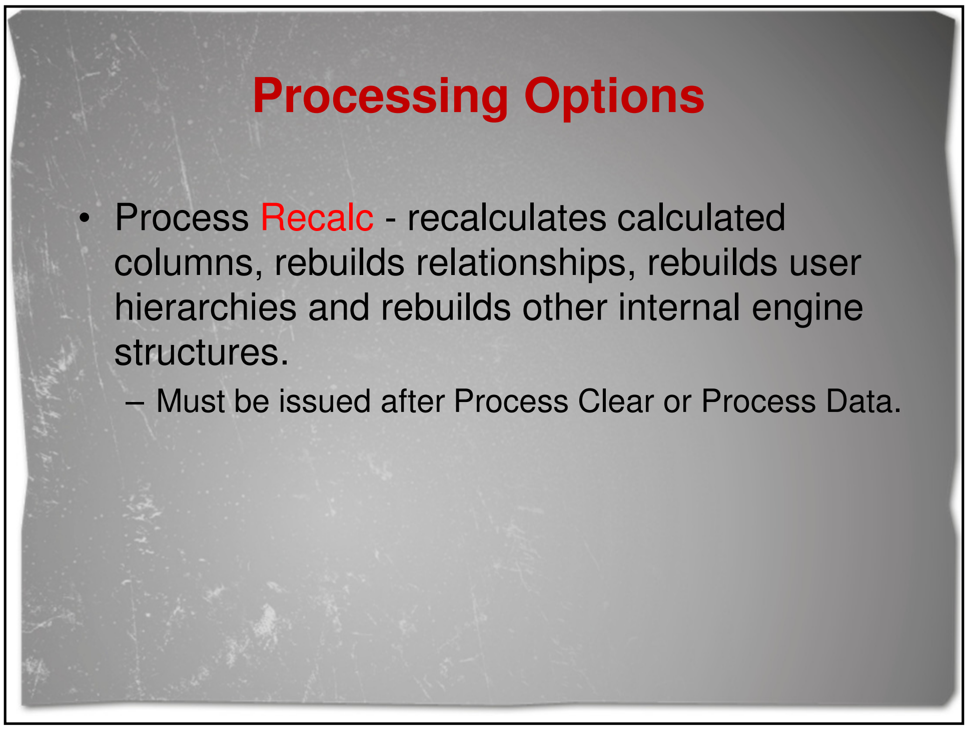
Process Full

- Processes a partition or table and all the objects that it contains.
- Drops all data, and then processes the object.
  - Any affected calculated columns, relationships, user hierarchies, or internal engine structures (except table dictionaries) are recalculated.
- This kind of processing is required when a structural change has been made to an object.



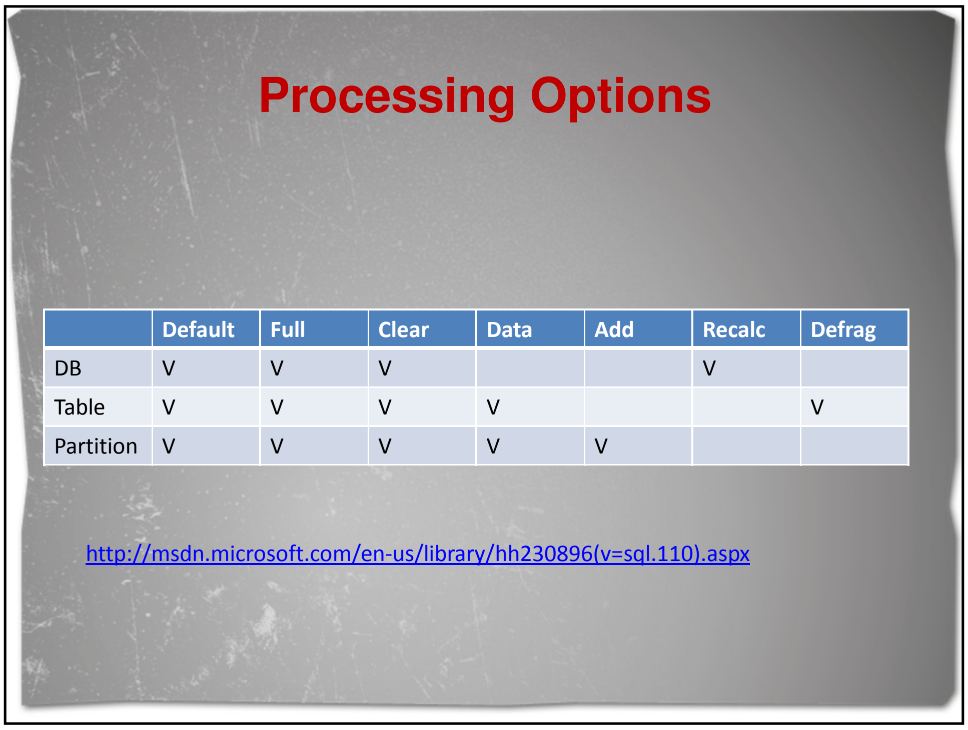
Processing Options

- Process Clear - Drops all the data in a database, table, or partition.
- Process Data - Loads data into a partition or table.
- Process Defrag - Optimizes the table dictionary (an internal engine structure)
  - For ex. After deleting old partition, the dictionary still contains references to these deleted rows.
  - The best practice is to run it frequently, based on your sliding window design



Processing Options

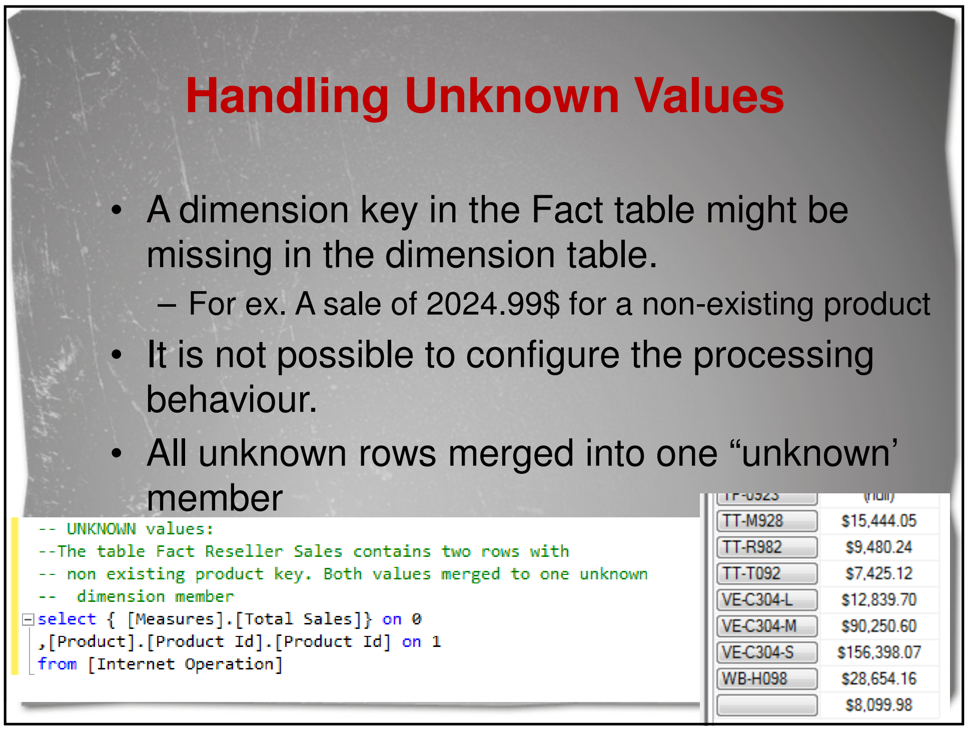
- Process Recalc - recalculates calculated columns, rebuilds relationships, rebuilds user hierarchies and rebuilds other internal engine structures.
  - Must be issued after Process Clear or Process Data.



Processing Options

	Default	Full	Clear	Data	Add	Recalc	Defrag
DB	V	V	V			V	
Table	V	V	V	V			V
Partition	V	V	V	V	V		

http://msdn.microsoft.com/en-us/library/hh230896(v=sql.110).aspx



Handling Unknown Values

- A dimension key in the Fact table might be missing in the dimension table.
  - For ex. A sale of 2024.99$ for a non-existing product
- It is not possible to configure the processing behaviour.
- All unknown rows merged into one “unknown’ member

```
-- UNKNOWN values:
--The table Fact Reseller Sales contains two rows with
-- non existing product key. Both values merged to one unknown
--  dimension member
select { [Measures].[Total Sales]} on 0
,[Product].[Product Id].[Product Id] on 1
from [Internet Operation]
```

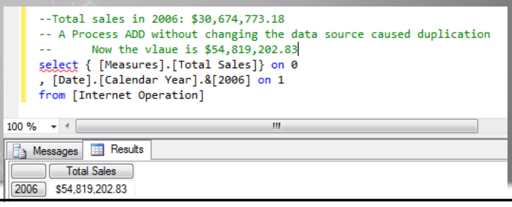
TF-0923	(null)
TT-M928	$15,444.05
TT-R982	$9,480.24
TT-T092	$7,425.12
VE-C304-L	$12,839.70
VE-C304-M	$90,250.60
VE-C304-S	$156,398.07
WB-H098	$28,654.16
	$8,099.98

Using Views in the Tabular Model

- Use views to filter small amount of data at the development phase, later change the view definition to return all data.
- Using views let you do changes in the underlying DB without affecting the Tabular Model Schema.

ProcessAdd

- The simplest solution – add a new partition + Full processing
- It is possible to run process add to load new data.
- It is under the responsibility of the DBA to avoid duplication.
- You can change the query binding using the UI.
  - Use views as much simpler query binding.



```
--Total sales in 2006: $30,674,773.18
-- A Process ADD without changing the data source caused duplication
--      Now the vlaue is $54,819,202.83
select { [Measures].[Total Sales]} on 0
, [Date].[Calendar Year].&[2006] on 1
from [Internet Operation]
```

	Total Sales
2006	$54,819,202.83

Parallel Processing

- It is possible to process many tables on parallel.
- However, only one partition of a table at a time.

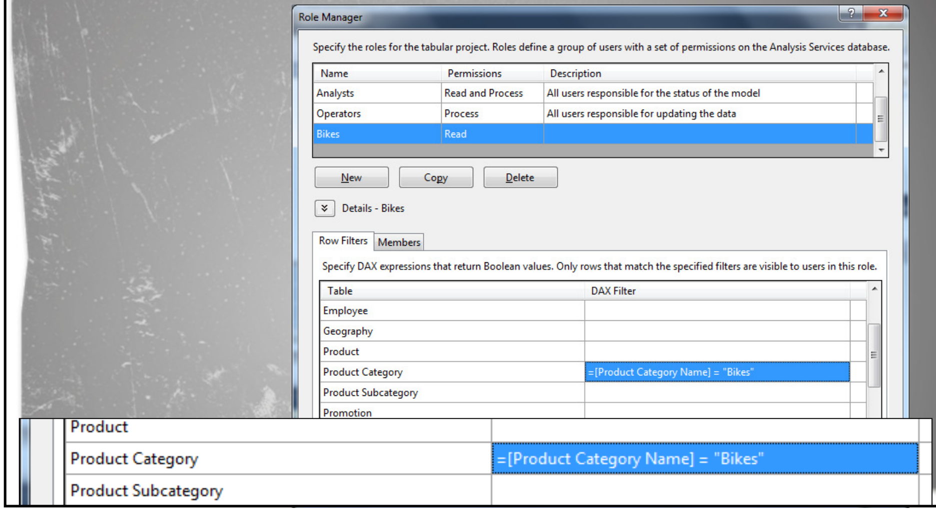
Security

Security

- Two main permissions:
  - Database Permissions
  - Allowed Row Sets
- Create roles using SSDT.
- Use DAX to restrict access
  - =Region[Country]=”USA”
- Dynamic security is also possible
  - http://msdn.microsoft.com/en-us/library/hh213165(v=sql.110).aspx

Security

- Create a role

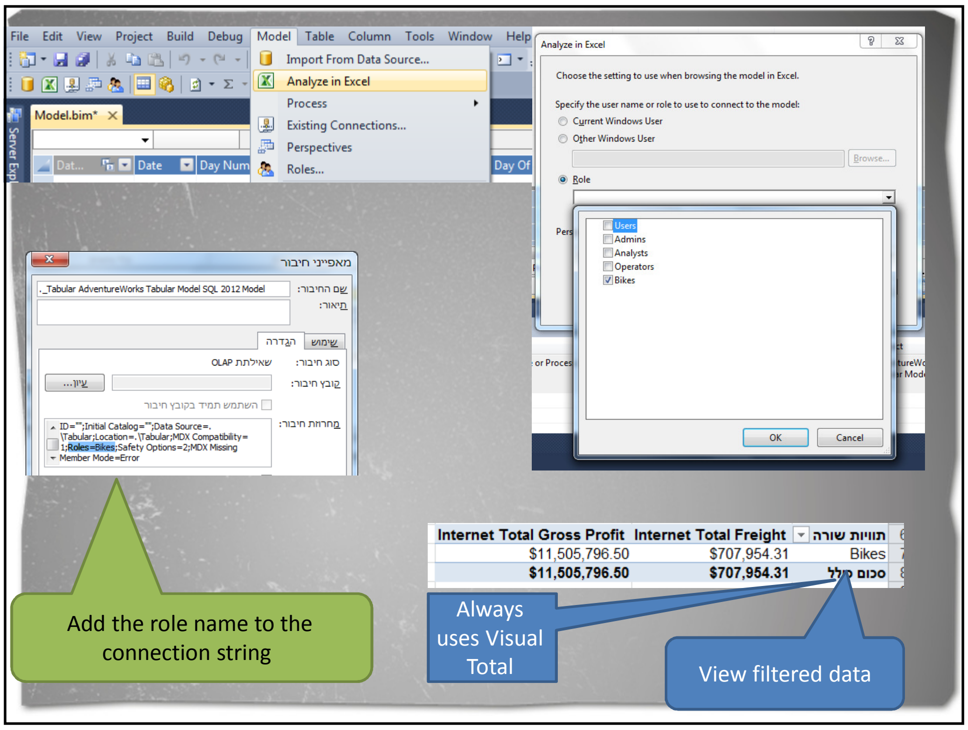


Specify the roles for the tabular project. Roles define a group of users with a set of permissions on the Analysis Services database.

Name	Permissions	Description
Analysts	Read and Process	All users responsible for the status of the model
Operators	Process	All users responsible for updating the data
Bikes	Read	

Specify DAX expressions that return Boolean values. Only rows that match the specified filters are visible to users in this role.

Table	DAX Filter
Employee	
Geography	
Product	
Product Category	=[Product Category Name] = "Bikes"
Product Subcategory	
Promotion	


File Edit View Project Build Debug Model Table Column Tools Window Help
Import From Data Source...
Analyze in Excel
Process
Existing Connections...
Perspectives
Roles...
Model.bim*
Dat... Date Day Num Day Of
Server Exp

מאפייני חיבור

שם החיבור: ._Tabular AdventureWorks Tabular Model SQL 2012 Model
תיאור:
הגדרה | שימוש
סוג חיבור: OLAP שאילתת
קובץ חיבור: עיון...
השתמש תמיד בקובץ חיבור
מחרוזת חיבור:

```
ID="";Initial Catalog="";Data Source=.
\Tabular;Location=.\Tabular;MDX Compatibility=
1;Roles=Bikes;Safety Options=2;MDX Missing
Member Mode=Error
```

Analyze in Excel

Choose the setting to use when browsing the model in Excel.

Specify the user name or role to use to connect to the model:
- Current Windows User
- Other Windows User
- [✓ Role]

- [ ] Users
- [ ] Admins
- [ ] Analysts
- [ ] Operators
- [x] Bikes

Internet Total Gross Profit	Internet Total Freight	תוויות שורה
$11,505,796.50	$707,954.31	Bikes
$11,505,796.50	**$707,954.31**	**סכום כולל**

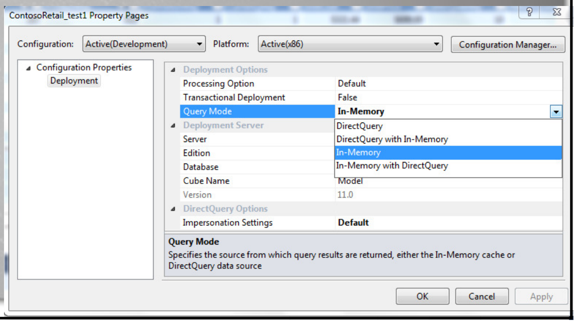
PoC – Lessons Learned

PoC – Lessons Learned

- Forget what you know about SSAS 2008, new design is needed
  - No Migration Wizard
  - Consider de-normalization.
- Server Configuration
  - Server Properties (SSAS - Tabular)
  - Memory paging allows models to be larger than the physical memory of the server

Project Properties - Query Mode

- Defualt: In-memory
- Direct query
- Direct Query with In-Memory
- In-Memory with Direct Query
- Configuration


ContosoRetail_test1 Property Pages

Configuration: Active(Development) Platform: Active(x86)

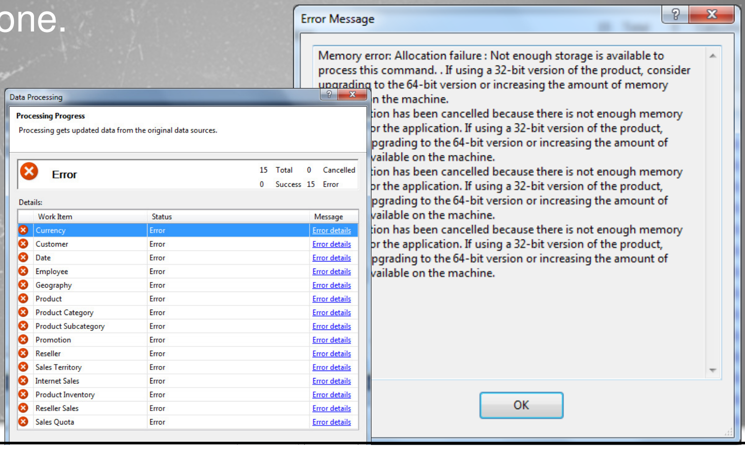
Deployment Options	
Processing Option	Default
Transactional Deployment	False
Query Mode	In-Memory
Deployment Server	
Server	
Edition	
Database	
Cube Name	Model
Version	11.0
DirectQuery Options	
Impersonation Settings	Default

DirectQuery
DirectQuery with In-Memory
In-Memory
In-Memory with DirectQuery

Query Mode
Specifies the source from which query results are returned, either the In-Memory cache or DirectQuery data source

PoC – Lessons Learned

- Full process of a whole DB might consume too much memory. Consider running queries one by one.



Memory error: Allocation failure : Not enough storage is available to
process this command. . If using a 32-bit version of the product, consider
upgrading to the 64-bit version or increasing the amount of memory
n the machine.
tion has been cancelled because there is not enough memory
or the application. If using a 32-bit version of the product,
pgrading to the 64-bit version or increasing the amount of
vailable on the machine.
tion has been cancelled because there is not enough memory
or the application. If using a 32-bit version of the product,
pgrading to the 64-bit version or increasing the amount of
vailable on the machine.
tion has been cancelled because there is not enough memory
or the application. If using a 32-bit version of the product,
pgrading to the 64-bit version or increasing the amount of
vailable on the machine.

Processing Progress

Processing gets updated data from the original data sources.

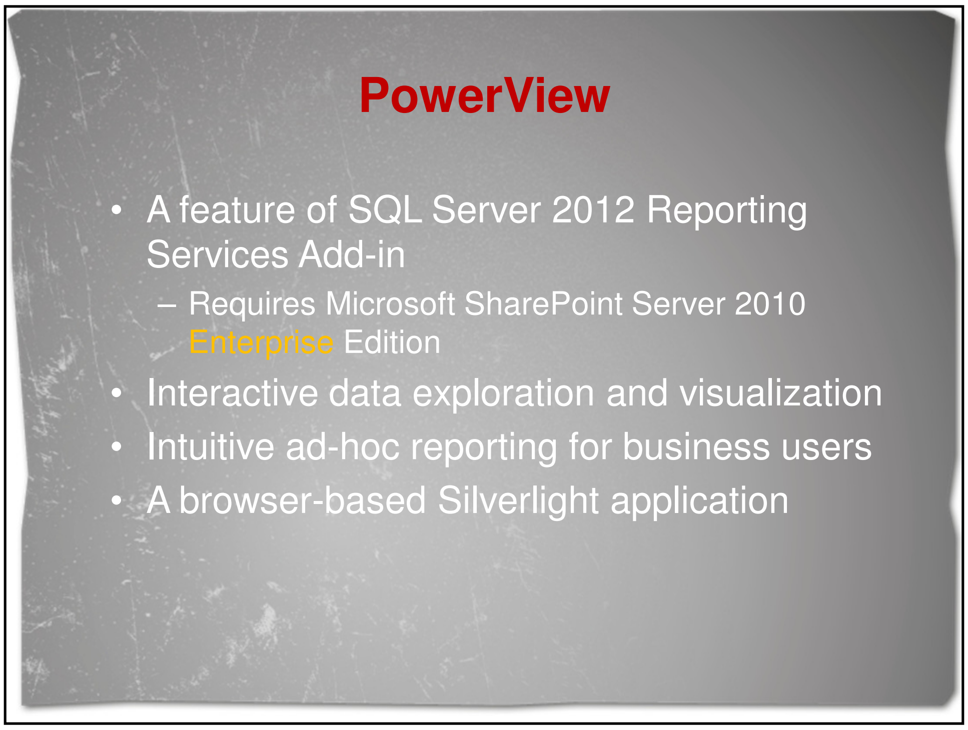
Error

15	Total	0	Cancelled
0	Success	15	Error

Details:

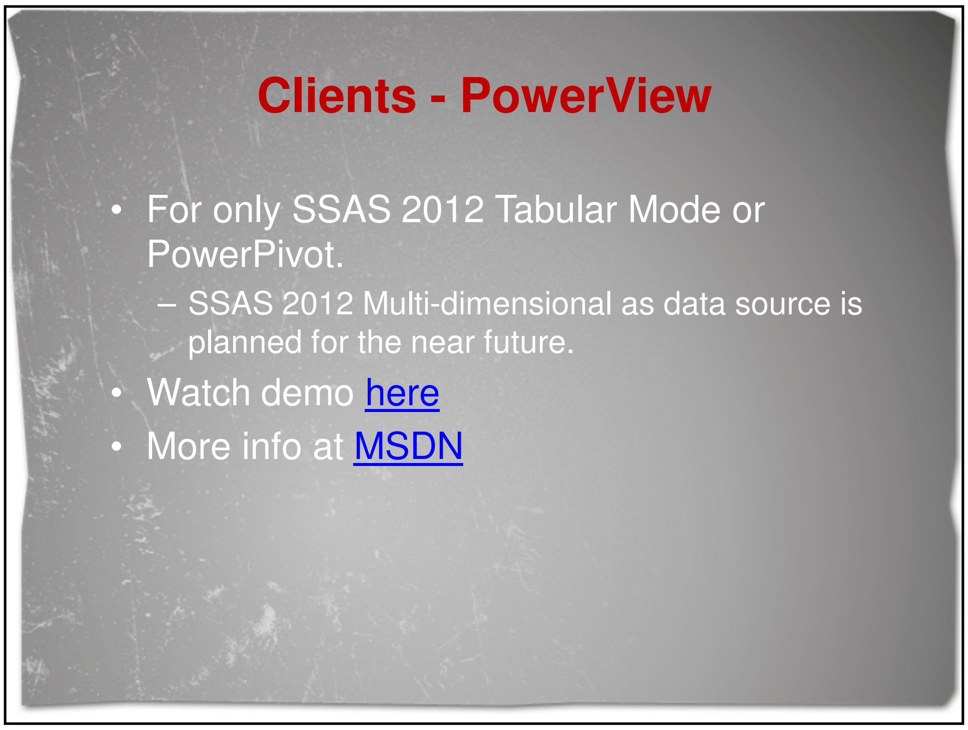
Work Item	Status	Message
Currency	Error	Error details
Customer	Error	Error details
Date	Error	Error details
Employee	Error	Error details
Geography	Error	Error details
Product	Error	Error details
Product Category	Error	Error details
Product Subcategory	Error	Error details
Promotion	Error	Error details
Reseller	Error	Error details
Sales Territory	Error	Error details
Internet Sales	Error	Error details
Product Inventory	Error	Error details
Reseller Sales	Error	Error details
Sales Quota	Error	Error details

Clients



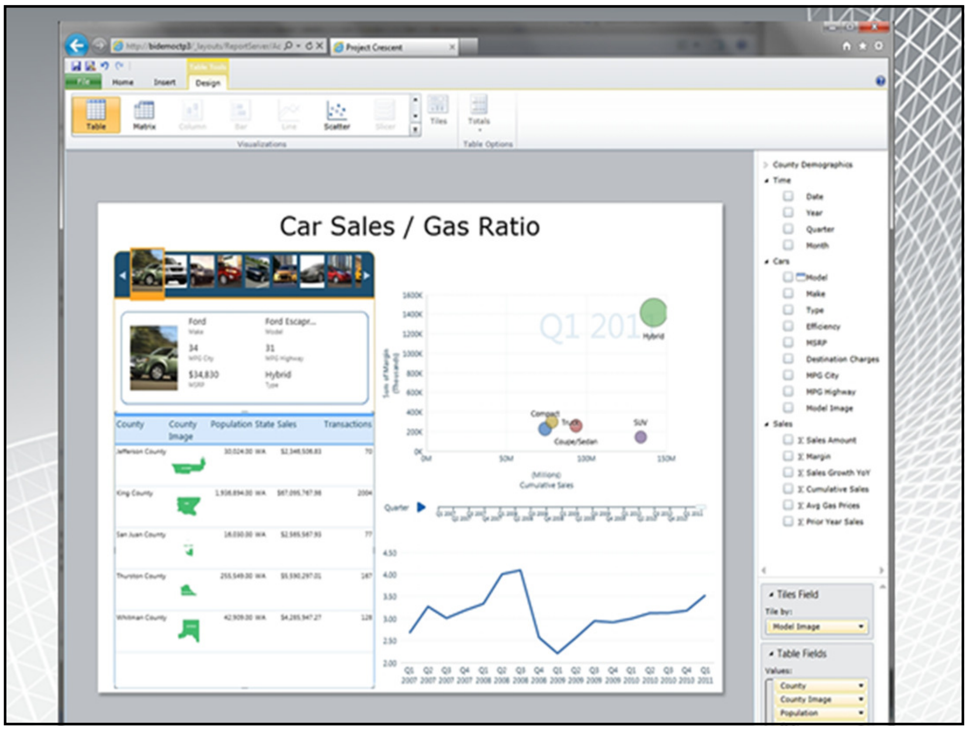
PowerView

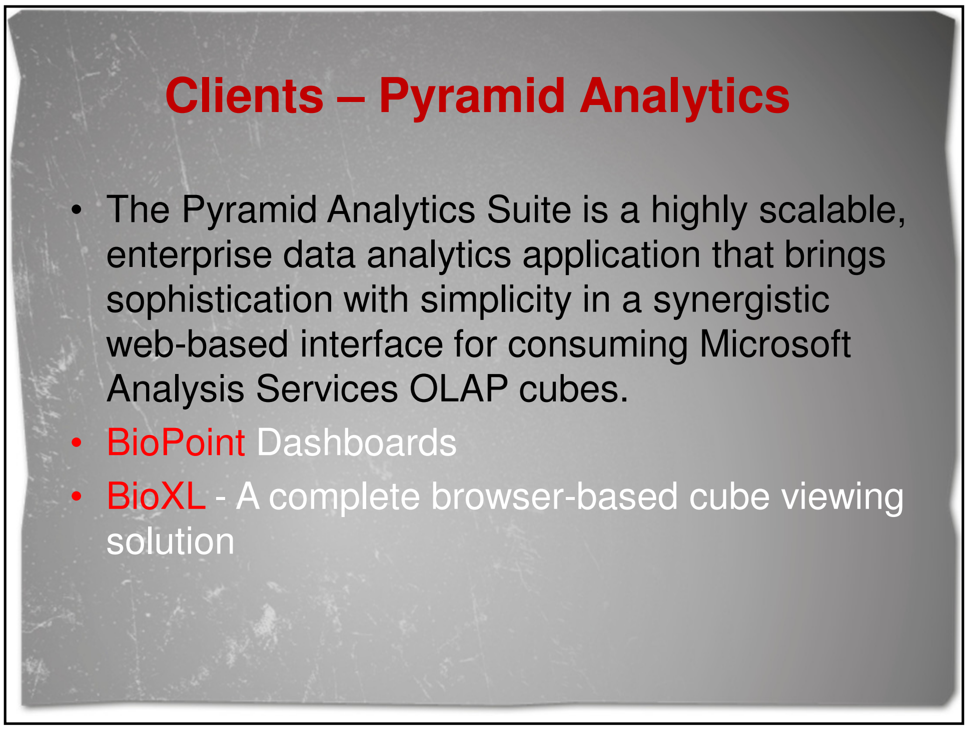
- A feature of SQL Server 2012 Reporting Services Add-in
  - Requires Microsoft SharePoint Server 2010 Enterprise Edition
- Interactive data exploration and visualization
- Intuitive ad-hoc reporting for business users
- A browser-based Silverlight application



Clients - PowerView

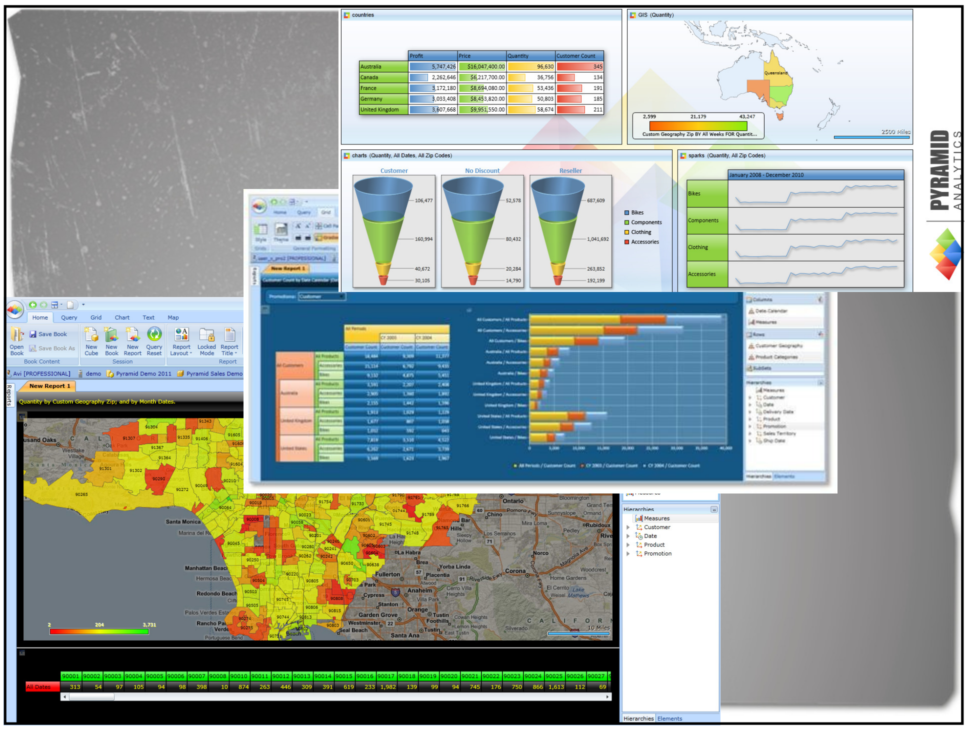
- For only SSAS 2012 Tabular Mode or PowerPivot.
  - SSAS 2012 Multi-dimensional as data source is planned for the near future.
- Watch demo here
- More info at MSDN



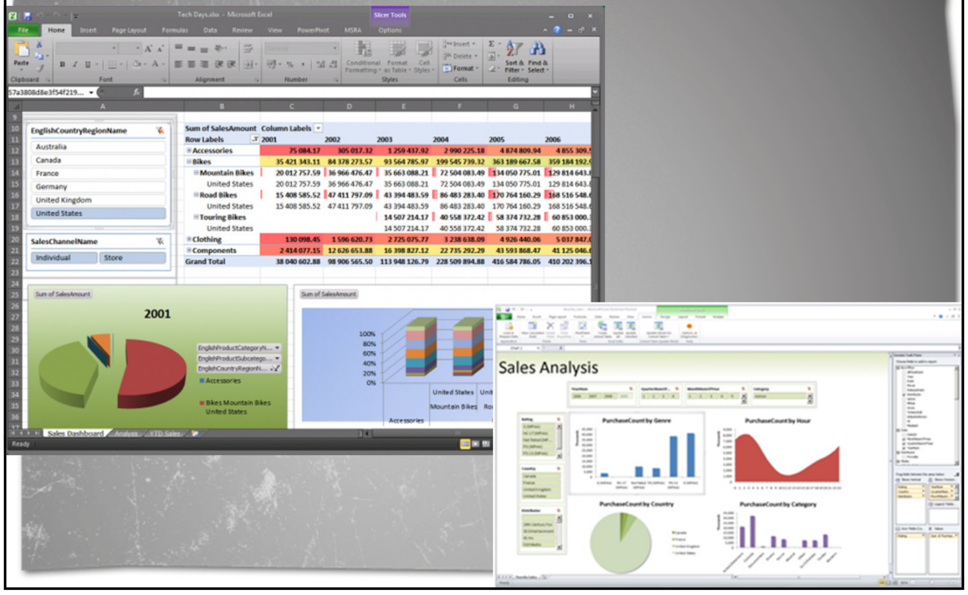


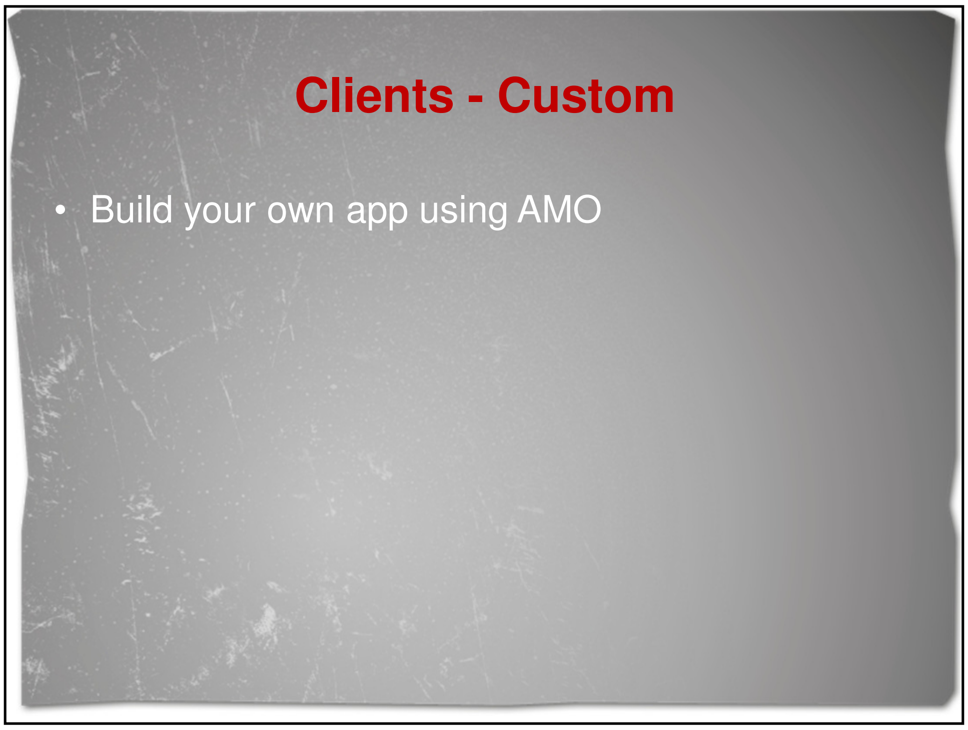
Clients – Pyramid Analytics

- The Pyramid Analytics Suite is a highly scalable, enterprise data analytics application that brings sophistication with simplicity in a synergistic web-based interface for consuming Microsoft Analysis Services OLAP cubes.
- BioPoint Dashboards
- BioXL - A complete browser-based cube viewing solution


PYRAMID ANALYTICS
countries
GIS (Quantity)
charts (Quantity, All Dates, All Zip Codes)
sparks (Quantity, All Zip Codes)
Customer
No Discount
Reseller
Bikes
Components
Clothing
Accessories
New Report 1
Quantity by Custom Geography Zip; and by Month Dates.

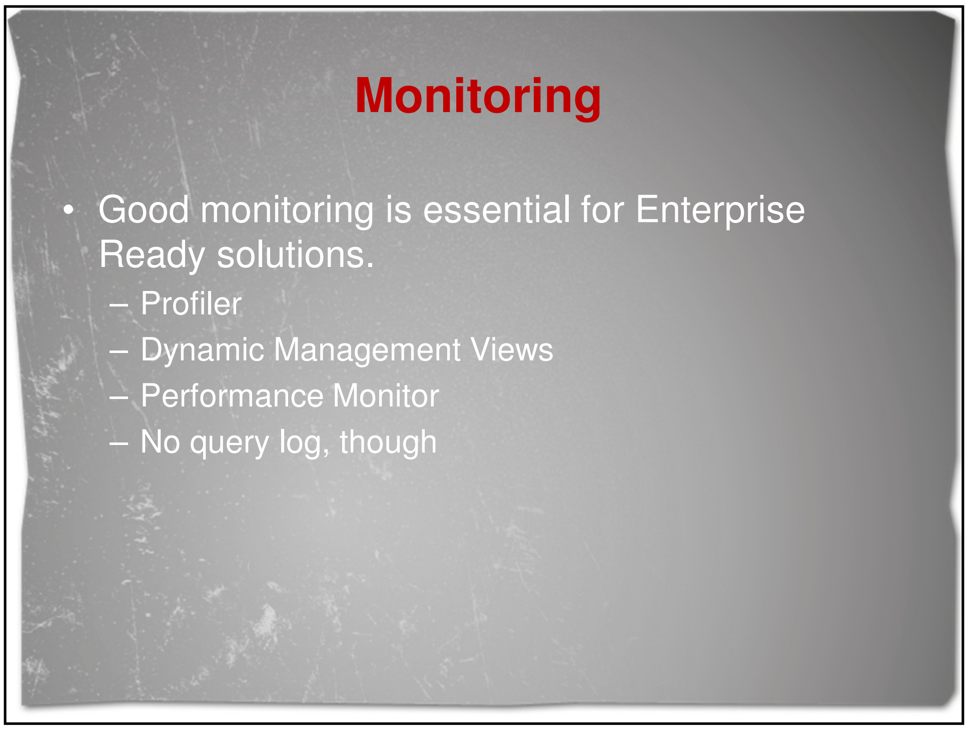
Clients – Excel 2010





Clients - Custom

- Build your own app using AMO

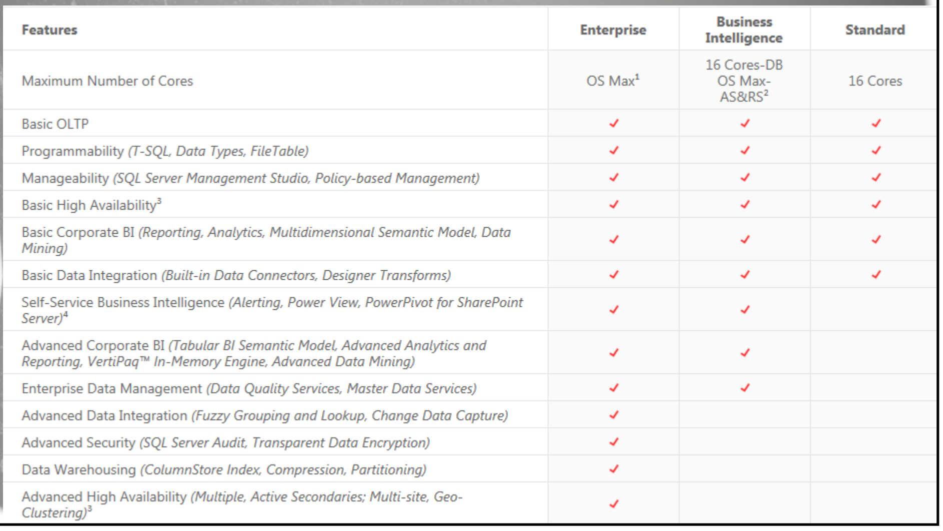


Monitoring

- Good monitoring is essential for Enterprise Ready solutions.
  - Profiler
  - Dynamic Management Views
  - Performance Monitor
  - No query log, though

Licensing

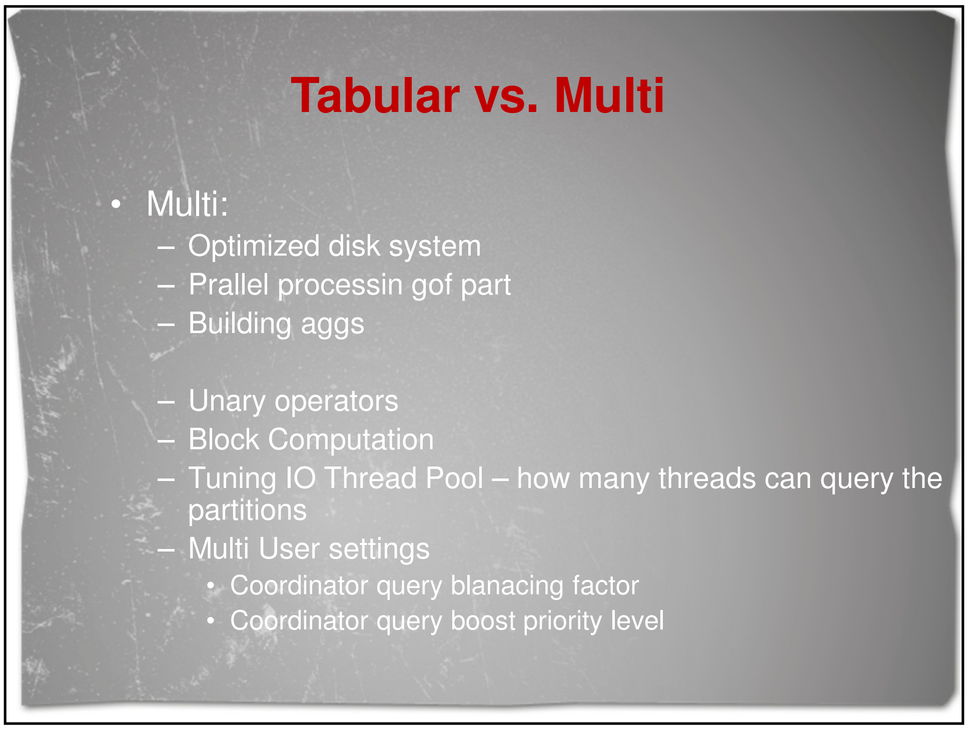
- Tabular Mode **not** in Standard Edition



Features	Enterprise	Business Intelligence	Standard
Maximum Number of Cores	OS Max[1]	16 Cores-DB OS Max-AS&RS[2]	16 Cores
Basic OLTP	✓	✓	✓
Programmability *(T-SQL, Data Types, FileTable)*	✓	✓	✓
Manageability *(SQL Server Management Studio, Policy-based Management)*	✓	✓	✓
Basic High Availability[3]	✓	✓	✓
Basic Corporate BI *(Reporting, Analytics, Multidimensional Semantic Model, Data Mining)*	✓	✓	✓
Basic Data Integration *(Built-in Data Connectors, Designer Transforms)*	✓	✓	✓
Self-Service Business Intelligence *(Alerting, Power View, PowerPivot for SharePoint Server)*[4]	✓	✓	
Advanced Corporate BI *(Tabular BI Semantic Model, Advanced Analytics and Reporting, VertiPaq™ In-Memory Engine, Advanced Data Mining)*	✓	✓	
Enterprise Data Management *(Data Quality Services, Master Data Services)*	✓	✓	
Advanced Data Integration *(Fuzzy Grouping and Lookup, Change Data Capture)*	✓		
Advanced Security *(SQL Server Audit, Transparent Data Encryption)*	✓		
Data Warehousing *(ColumnStore Index, Compression, Partitioning)*	✓		
Advanced High Availability *(Multiple, Active Secondaries; Multi-site, Geo-Clustering)*[3]	✓		

Conclusion

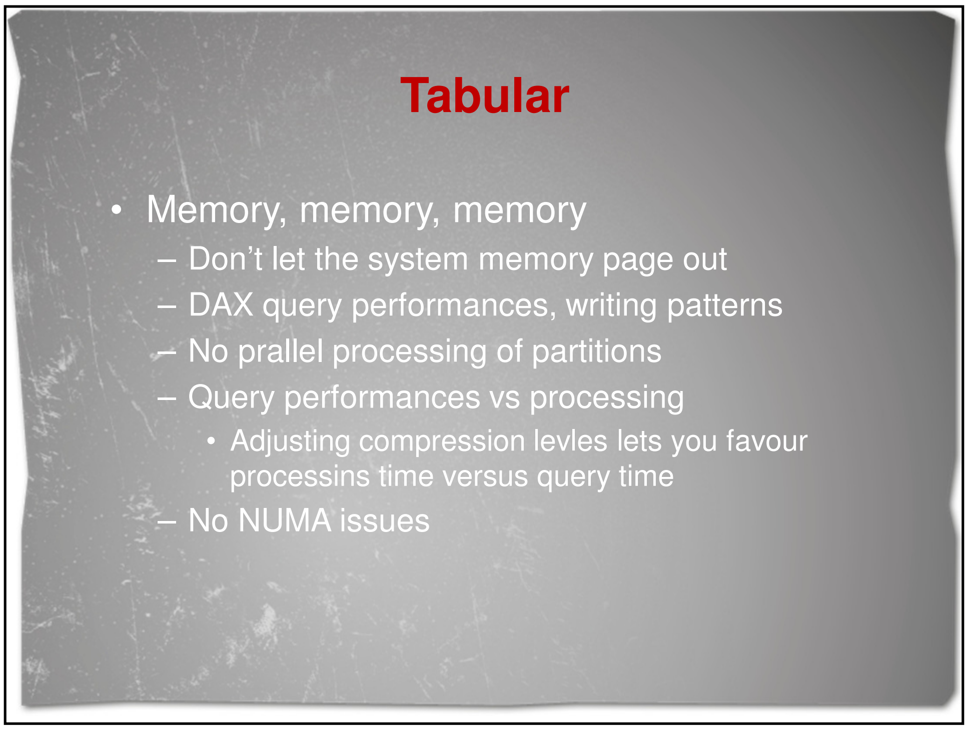
- Multi Dimensional OLAP:
  - Familiar, Huge Install base, wide knowledgebase, stable, large scale implementations, monitoring tools, many OLAP Viewers
- Tabular Mode
  - New Technology
  - Simple, easier development, sometimes faster
  - Microsoft now focusing on this technology



Tabular vs. Multi

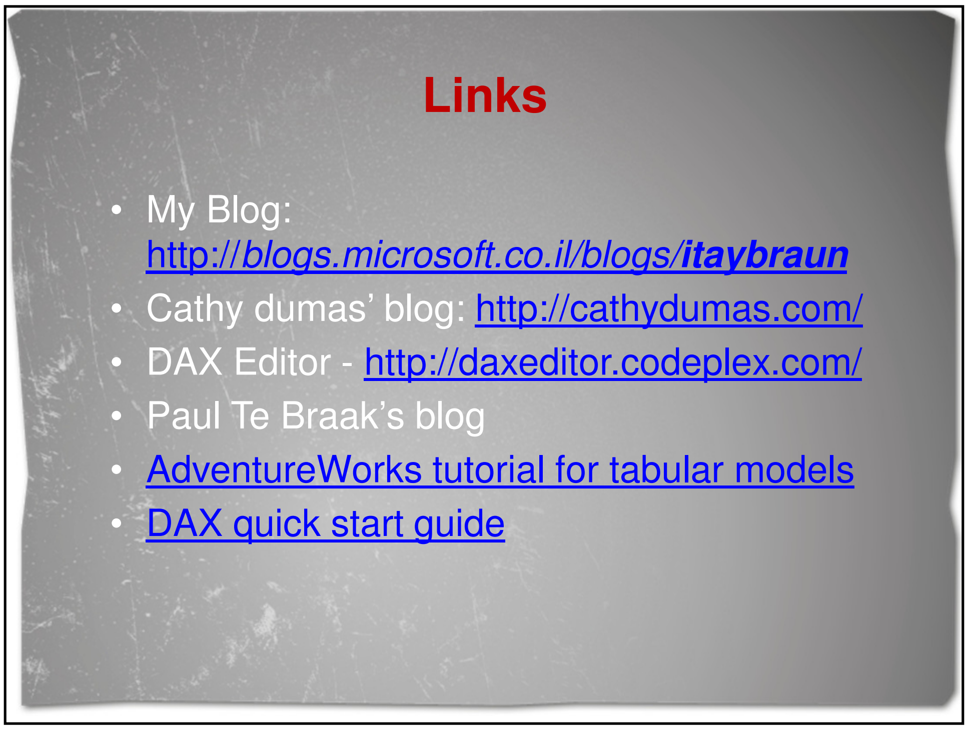
- Multi:
  - Optimized disk system
  - Prallel processin gof part
  - Building aggs

  - Unary operators
  - Block Computation
  - Tuning IO Thread Pool – how many threads can query the partitions
  - Multi User settings
    - Coordinator query blanacing factor
    - Coordinator query boost priority level



Tabular

- Memory, memory, memory
  - Don't let the system memory page out
  - DAX query performances, writing patterns
  - No prallel processing of partitions
  - Query performances vs processing
    - Adjusting compression levles lets you favour processins time versus query time
  - No NUMA issues



Links

- My Blog: *http://blogs.microsoft.co.il/blogs/**itaybraun***
- Cathy dumas' blog: http://cathydumas.com/
- DAX Editor - http://daxeditor.codeplex.com/
- Paul Te Braak's blog
- AdventureWorks tutorial for tabular models
- DAX quick start guide

Microsoft SQL Server 2005 Analysis Services Performance Guide

SQL Server Technical Article

Author: Elizabeth Vitt

Subject Matter Experts:

T.K. Anand
Sasha (Alexander) Berger
Marius Dumitru
Eric Jacobsen
Edward Melomed
Akshai Mirchandani
Mosha Pasumansky
Cristian Petculescu
Carl Rabeler
Wayne Robertson
Richard Tkachuk
Dave Wickert
Len Wyatt

Published: February 2007

Applies To: SQL Server 2005, Service Pack 2

Summary: This white paper describes how application developers can apply performance-tuning techniques to their Microsoft SQL Server 2005 Analysis Services Online Analytical Processing (OLAP) solutions.

Copyright

Table of Contents

Introduction

Fast query response times and timely data refresh are two well-established performance requirements of Online Analytical Processing (OLAP) systems. To provide fast analysis, OLAP systems traditionally use hierarchies to efficiently organize and summarize data. While these hierarchies provide structure and efficiency to analysis, they tend to restrict the analytic freedom of end users who want to freely analyze and organize data on the fly.

To support a broad range of structured and flexible analysis options, Microsoft® SQL Server™ Analysis Services (SSAS) 2005 combines the benefits of traditional hierarchical analysis with the flexibility of a new generation of attribute hierarchies. Attribute hierarchies allow users to freely organize data at query time, rather than being limited to the predefined navigation paths of the OLAP architect. To support this flexibility, the Analysis Services OLAP architecture is specifically designed to accommodate both attribute and hierarchical analysis while maintaining the fast query performance of conventional OLAP databases.

Realizing the performance benefits of this combined analysis paradigm requires understanding how the OLAP architecture supports both attribute hierarchies and traditional hierarchies, how you can effectively use the architecture to satisfy your analysis requirements, and how you can maximize the architecture's utilization of system resources.

> **Note** To apply the performance tuning techniques discussed in this white paper, you must have SQL Server 2005 Service Pack 2 installed.

To satisfy the performance needs of various OLAP designs and server environments, this white paper provides extensive guidance on how you can take advantage of the wide range of opportunities to optimize Analysis Services performance. Since Analysis Services performance tuning is a fairly broad subject, this white paper organizes performance tuning techniques into the following four segments.

Enhancing 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 OLAP implementation. Analysis Services provides a variety of mechanisms to accelerate query performance, including aggregations, caching, and indexed data retrieval. In addition, you can improve query performance by optimizing the design of your dimension attributes, cubes, and MDX queries.

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 efficient dimension design, effective aggregations, partitions, and an economical processing strategy (for example, incremental vs. full refresh vs. proactive caching).

Optimizing Special Design Scenarios – Complex design scenarios require a distinct set of performance tuning techniques to ensure that they are applied successfully, especially if you combine a complex design with large data volumes. Examples of complex design components include special aggregate functions, parent-child hierarchies, complex dimension relationships, and "near real-time" data refreshes.

Tuning Server Resources – Analysis Services operates within the constraints of available server resources. Understanding how Analysis Services uses memory, CPU, and disk resources can help you make effective server management decisions that optimize querying and processing performance.

Three appendices provide links to additional resources, information on various partition storage modes, and guidance on using the Aggregation Utility that is a part of SQL Server 2005 Service Pack 2 samples.

Enhancing Query Performance

Querying is the operation where Analysis Services provides data to client applications according to the calculation and data requirements of a MultiDimensional eXpressions (MDX) query. Since query performance directly impacts the user experience, this section describes the most significant opportunities to improve query performance. Following is an overview of the query performance topics that are addressed in this section:

Understanding the querying architecture - The Analysis Services querying architecture supports three major operations: session management, MDX query execution, and data retrieval. Optimizing query performance involves understanding how these three operations work together to satisfy query requests.

Optimizing the dimension design - A well-tuned dimension design is perhaps one of the most critical success factors of a high-performing Analysis Services solution. Creating attribute relationships and exposing attributes in hierarchies are design choices that influence effective aggregation design, optimized MDX calculation resolution, and efficient dimension data storage and retrieval from disk.

Maximizing the value of aggregations - Aggregations improve query performance by providing precalculated summaries of data. To maximize the value of aggregations, ensure that you have an effective aggregation design that satisfies the needs of your specific workload.

Using partitions to enhance query performance - Partitions provide a mechanism to separate measure group data into physical units that improve query performance, improve processing performance, and facilitate data management. Partitions are naturally queried in parallel; however, there are some design choices and server property optimizations that you can specify to optimize partition operations for your server configuration.

Writing efficient MDX - This section describes techniques for writing efficient MDX statements such as: 1) writing statements that address a narrowly defined calculation space, 2) designing calculations for the greatest re-usage across multiple users, and 3) writing calculations in a straight-forward manner to help the Query Execution Engine select the most efficient execution path.

Understanding the querying 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. Figure 1 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.

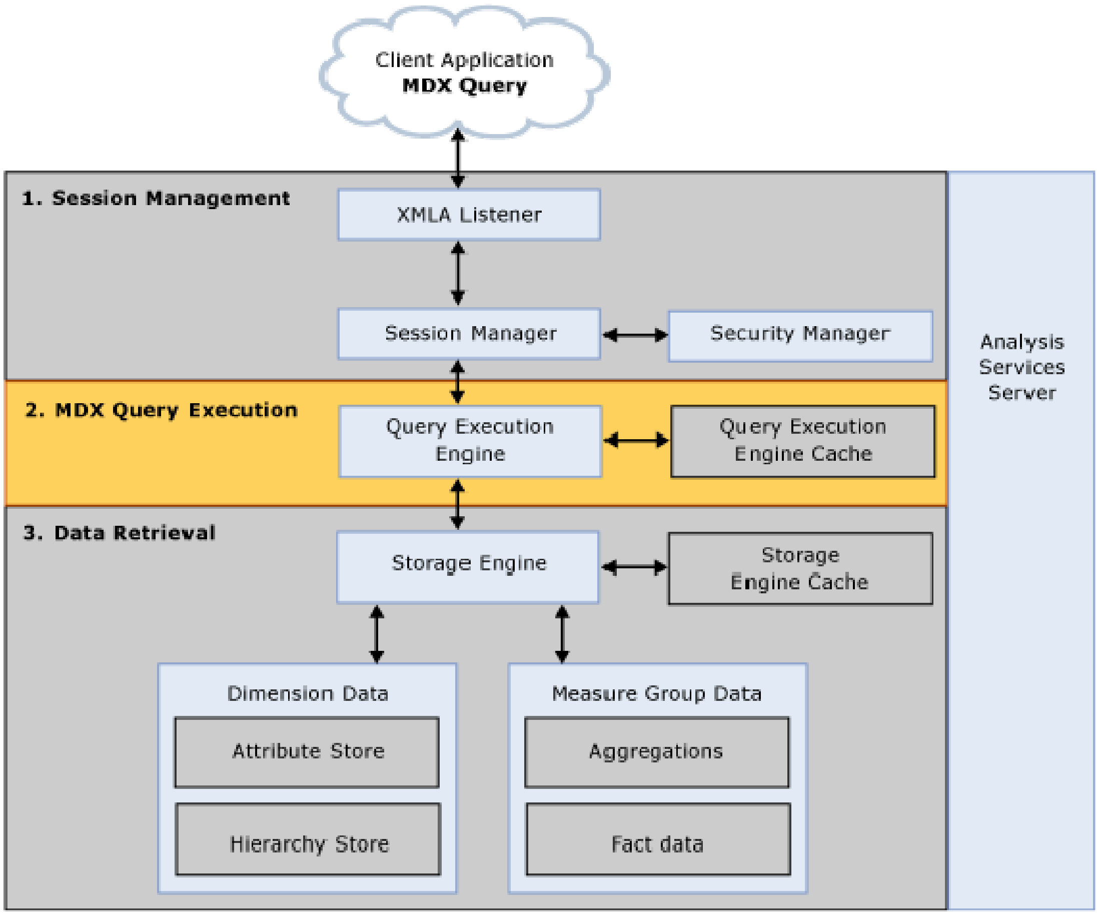


Figure 1 Analysis Services querying architecture

Session management

Client applications communicate with Analysis Services using XML for Analysis (XMLA) over TCP IP or HTTP. Analysis Services provides an XMLA listener component that handles all XMLA communications between Analysis Services and its clients. The Analysis Services Session Manager controls how clients connect to an Analysis Services instance. Users authenticated by Microsoft® Windows and who have rights to Analysis Services 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 reuses the session for all of the user's requests. The session provides the context under which client queries are executed by the Query Execution Engine. A session exists until it is either closed by the client application, or until the server needs to expire it. For more information regarding the longevity of sessions, see Monitoring the timeout of idle sessions in this white paper.

MDX query execution

The primary operation of the Query Execution Engine is to execute MDX queries. This section provides an overview of how the Query Execution Engine executes queries. To learn more details about optimizing MDX, see Writing efficient MDX later in this white paper.

While the actual query execution process is performed in several stages, from a performance perspective, the Query Execution engine must consider two basic requirements: retrieving data and producing the result set.

1. **Retrieving data**—To retrieve the data requested by a query, the Query Execution Engine decomposes each MDX query into data requests. To communicate with the Storage Engine, the Query Execution Engine must translate the data requests into subcube requests that the Storage Engine can understand. A subcube represents a logical unit of querying, caching, and data retrieval. An MDX query may be resolved into one or more subcube requests depending on query granularity and calculation complexity. Note that the word *subcube* is a generic term. For example, the subcubes that the Query Execution Engine creates during query evaluation are not to be confused with the subcubes that you can create using the MDX CREATE SUBCUBE statement.
2. **Producing the result set**—To manipulate the data retrieved from the Storage Engine, the Query Execution Engine uses two kinds of execution plans to calculate results: it can bulk calculate an entire subcube, or it can calculate individual cells. In general, the subcube evaluation path is more efficient; however, the Query Execution Engine ultimately selects execution plans based on the complexities of each MDX query. Note that a given query can have multiple execution plans for different parts of the query and/or different calculations involved in the same query. Moreover, different parts of a query may choose either one of these two types of execution plans independently, so there is not a single global decision for the entire query. For example, if a query requests resellers whose year-over-year profitability is greater than 10%, the Query Execution Engine may use one execution plan to calculate each reseller's year-over-year profitability and another execution plan to only return those resellers whose profitability is greater than 10%.

   When you execute an MDX calculation, the Query Execution Engine must often execute the calculation across more cells than you may realize. Consider the example where you have an MDX query that must return the calculated year-to-date sales across the top five regions. While it may seem like you are only returning five cell values, Analysis Services must execute the calculation across additional cells in order to determine the top five regions and also to return their year to date sales. A general MDX optimization technique is to write MDX queries in a way that minimizes the amount of data that the Query Execution Engine must evaluate. To learn more about this MDX optimization technique, see Specifying the calculation space later in this white paper.

   As the Query Execution Engine evaluates cells, it uses the Query Execution Engine cache and the Storage Engine Cache to store calculation results. The primary benefits of the cache are to optimize the evaluation of calculations and to support the re-usage of calculation results across users. To optimize cache re-usage, the Query Execution Engine manages three cache scopes that determine the level of cache reusability: global scope, session scope, and query scope. For more information on cache sharing, see Taking advantage of the Query Execution Engine cache in this white paper.

Data retrieval: dimensions

During data retrieval, the Storage Engine must efficiently choose the best mechanism to fulfill the data requests for both dimension data and measure data.

To satisfy requests for dimension data, the Storage Engine extracts data from the dimension attribute and hierarchy stores. As it retrieves the necessary data, the Storage Engine uses dynamic on-demand caching of dimension data rather than keeping all dimension members statically mapped into memory. The Storage Engine simply brings members into memory as they are needed. Dimension data structures may reside on disk, in Analysis Services memory, or in the Windows operating system file cache, depending on memory load of the system.

As the name suggests, the Dimension Attribute Store contains all of the information about dimension attributes. The components of the Dimension Attribute Store are displayed in Figure 2.

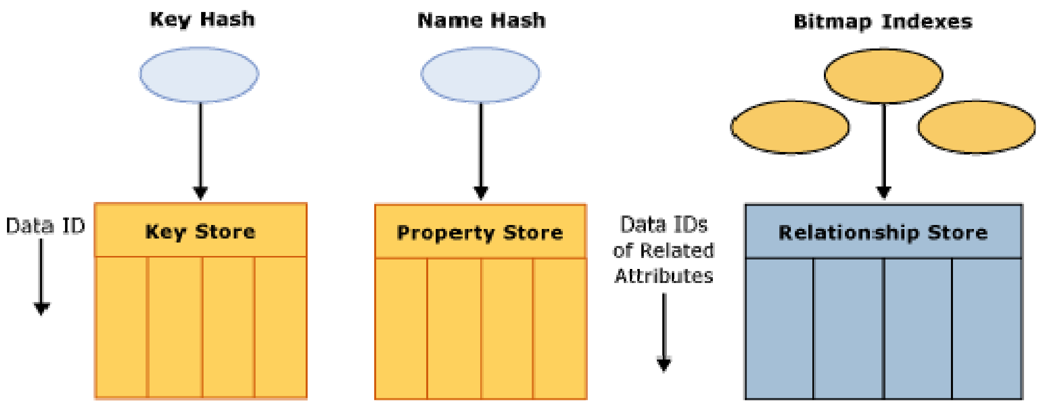


Figure 2 Dimension Attribute Store

As displayed in the diagram, the Dimension Attribute Store contains the following components for each attribute in the dimension:

- **Key store**—The key store contains the attribute's key member values as well as an internal unique identifier called a *DataID*. Analysis Services assigns the DataID to each attribute member and uses the same DataID to refer to that member across all of its stores.
- **Property store**—The property store contains a variety of attribute properties, including member names and translations. These properties map to specific DataIDs. DataIDs are contiguously allocated starting from zero. Property stores as well as relationship stores (relationship stores are discussed in more detail later) are physically ordered by DataID, in order to ensure fast random access to the contents, without additional index or hash table lookups.
- **Hash tables**—To facilitate attribute lookups during querying and processing, each attribute has two hash tables which are created during processing and persisted on disk. The Key Hash Table indexes members by their unique keys. A Name Hash Table indexes members by name.

- **Relationship store**—The Relationship store contains an attribute's relationships to other attributes. More specifically, the Relationship store stores each source record with DataID references to other attributes. Consider the following example for a product dimension. Product is the key attribute of the dimension with direct attribute relationships to color and size. The data instance of product Sport Helmet, color of Black, and size of Large may be stored in the relationship store as 1001, 25, 5 where 1001 is the DataID for Sport Helmet, 25 is the Data ID for Black, and 5 is the Data ID for Large. Note that if an attribute has no attribute relationships to other attributes, a Relationship Store is not created for that particular attribute. For more information on attribute relationships, see Identifying attribute relationships in this whitepaper.
- **Bitmap indexes**—To efficiently locate attribute data in the Relationship Store at querying time, the Storage Engine creates bitmap indexes at processing time. For each DataID of a related attribute, the bitmap index states whether or not a page contains at least one record with that DataID. For attributes with a very large number of DataIDs, the bitmap indexes can take some time to process. In most scenarios, the bitmap indexes provide significant querying benefits; however, there is a design scenario where the querying benefit that the bitmap index provides does not outweigh the processing cost of creating the bitmap index in the first place. It is possible to remove the bitmap index creation for a given attribute by setting the **AttributeHierarchyOptimizedState** property to **Not Optimized**. For more information on this design scenario, see Reducing attribute overhead in this white paper.

In addition to the attribute store, the Hierarchy Store arranges attributes into navigation paths for end users as displayed in Figure 3.

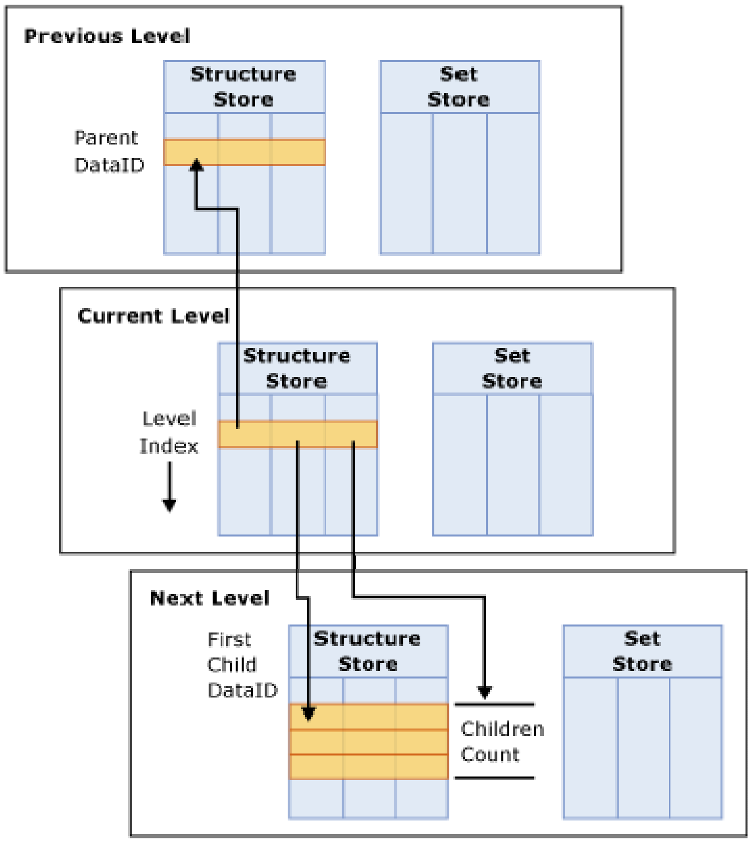


Figure 3 Hierarchy stores

The Hierarchy store consists of the following primary components:

- **Set Store**—The Set Store uses DataIDs to construct the path of each member, mapped from the first level to the current level. For example, All, Bikes, Mountain Bikes, Mountain Bike 500 may be represented as 1,2,5,6 where 1 is the DataID for All, 2 is the DataID for Bikes, 5 is the DataID for Mountain Bikes, and 6 is the DataID for Mountain Bike 500.
- **Structure Store**—For each member in a level, the Structure Store contains the DataID of the parent member, the DataID of the first child, and the total children count. The entries in the Structure Store are ordered by each member's Level index. The Level index of a member is the position of a member in the level as specified by the ordering settings of the dimension. To better understand the Structure Store, consider the following example. If Bikes contains 3 children, the entry for Bikes in the Structure Store would be 1,5,3 where 1 is the DataID for the Bike's parent, All, 5 is the DataID for Mountain Bikes, and 3 is the number of Bike's children.

Note that only natural hierarchies are materialized in the hierarchy store and optimized for data retrieval. Unnatural hierarchies are not materialized on disk. For more information on the best practices for designing hierarchies, see Using hierarchies effectively.

Data retrieval: measure group data

For data requests, the Storage Engine retrieves measure group data that is physically stored in partitions. A partition contains two categories of measure group data: fact data and aggregations. To accommodate a variety of data storage architectures, each partition can be assigned a different storage mode that specifies where fact data and aggregations are stored. From a performance perspective, the storage mode that provides the fastest query performance is the Multidimensional Online Analytical Processing (MOLAP) storage mode. In MOLAP, the partition fact data and aggregations are stored in a compressed multidimensional format that Analysis Services manages. For most implementations, MOLAP storage mode should be used; however, if you require additional information about other partition storage modes, see Appendix B. If you are considering a "near real-time" deployment, see Near real-time data refreshes in this white paper.

In MOLAP storage, the data structures for fact data and aggregation data are identical. Each is divided into segments. A segment contains a fixed number of records (typically 64 KB) divided into 256 pages with 256 records in each. Each record stores all of the measures in the partition's measure group and a set of internal DataIDs that map to the granularity attributes of each dimension. Only records that are present in the relational fact table are stored in the partition, resulting in highly compressed data files.

To efficiently fulfill data requests, the Storage Engine follows an optimized process to satisfy the request by using three general mechanisms, Storage Engine Cache, aggregations, and fact data represented in Figure 4:.

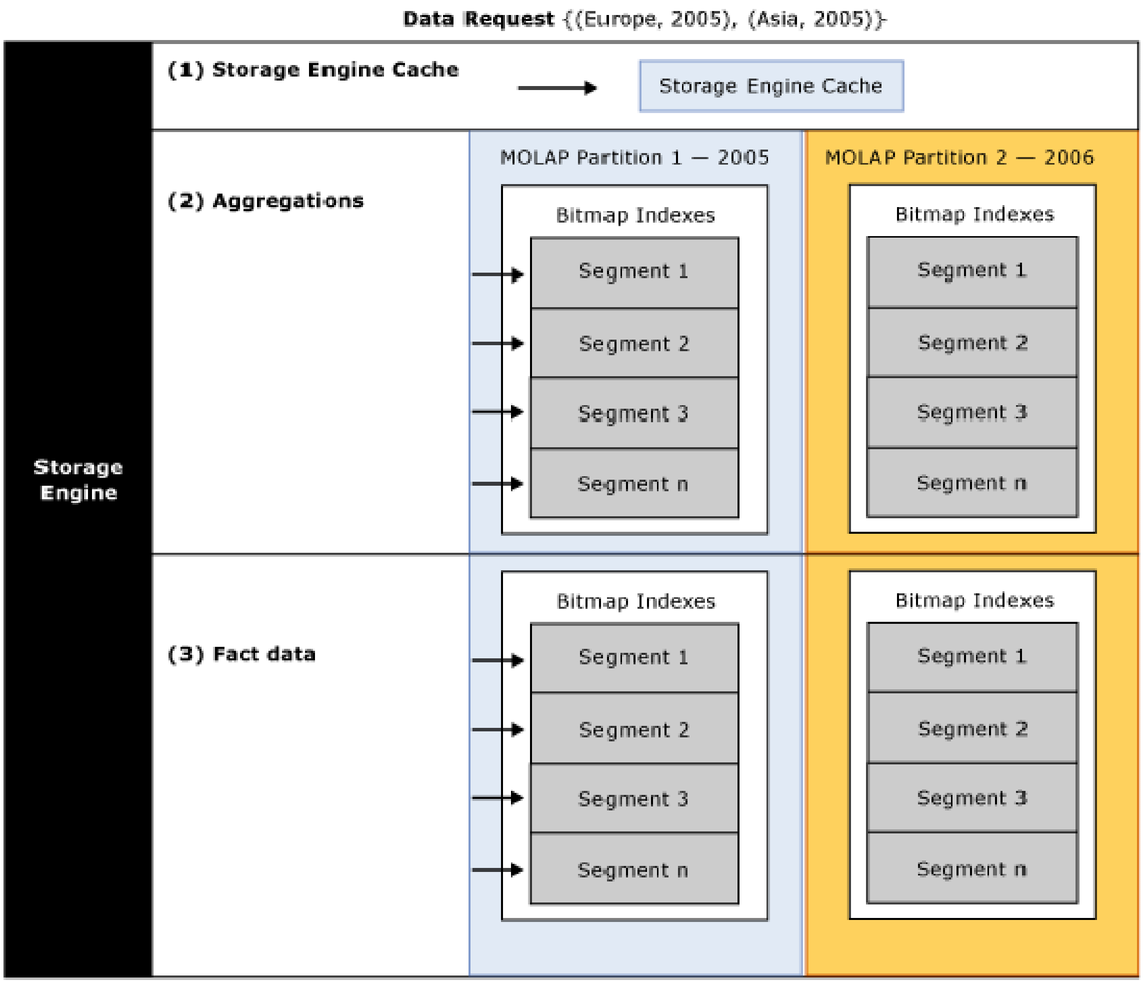


Figure 4 Satisfying data requests

Figure 4 presents a data request for {(Europe, 2005), (Asia, 2005)}. To fulfill this request, the Storage Engine chooses among the following approaches:

1. **Storage Engine cache**—The Storage Engine first attempts to satisfy the data request using the Storage Engine cache. Servicing a data request from the Storage Engine cache provides the best query performance. The Storage Engine cache always resides in memory. For more information on managing the Storage Engine cache, see Memory demands during querying.
2. **Aggregations**—If relevant data is not in the cache, the Storage Engine checks for a precalculated data aggregation. In some scenarios, the aggregation may exactly fit the data request. For example, an exact fit occurs when the query asks for sales by category by year and there is an aggregation that summarizes sales by category by year. While an exact fit is ideal, the Storage Engine can use also data aggregated at a lower level, such as sales aggregated by month and category or sales aggregated by quarter and item. The Storage Engine then summarizes the values on the fly to produce sales by category by year. For more information on how to design aggregations to improve performance, see Maximizing the value of aggregations.

3. **Fact data**—If appropriate aggregations do not exist for a given query, the Storage Engine must retrieve the fact data from the partition. The Storage Engine uses many internal optimizations to effectively retrieve data from disk including enhanced indexing and clustering of related records. For both aggregations and fact data, different portions of data may reside either on disk or in the Windows operating system file cache, depending on memory load of the system.

A key performance tuning technique for optimizing data retrieval is to reduce the amount of data that the Storage Engine needs to scan by using multiple partitions that physically divide your measure group data into distinct data slices. Using multiple partitions can not only enhance querying speed, but they can also provide greater scalability, facilitate data management, and optimize processing performance.

From a querying perspective, the Storage Engine can predetermine the data stored in each MOLAP partition and optimize which MOLAP partitions it scans in parallel. In the example in Figure 4, a partition with 2005 data is displayed in blue and a partition with 2006 data is displayed in orange. The data request displayed in the diagram {(Europe, 2005), (Asia, 2005)} only requires 2005 data. Consequently, the Storage Engine only needs to go to the 2005 partition. To maximize query performance, the Storage Engine uses parallelism, such as scanning partitions in parallel, wherever possible. To locate data in a partition, the Storage Engine queries segments in parallel and uses bitmap indexes to efficiently scan pages to find the desired data.

Partitions are a major component of high-performing cubes. For more information on the broader benefits of partitions, see the following sections in this white paper:

- For more information on how bitmap indexes are created and used in partitions, see Partition-processing jobs.
- To find out more detailed information about the querying benefits of partitioning, see Using partitions to enhance query performance.
- To learn more about the processing benefits of using partitions, see Using partitions to enhance processing performance.

Optimizing the dimension design

A well-tuned dimension design is one of the most critical success factors of a high-performing Analysis Services solution. The two most important techniques that you can use to optimize your dimension design for query performance are:

- Identifying attribute relationships
- Using hierarchies effectively

Identifying attribute relationships

A typical data source of an Analysis Services dimension is a relational data warehouse dimension table. In relational data warehouses, each dimension table typically contains a primary key, attributes, and, in some cases, foreign key relationships to other tables.

Table 1 Column properties of a simple Product dimension table

Dimension table column	Column type	Relationship to primary key	Relationship to other columns
Product Key	Primary Key	Primary Key	---
Product SKU	Attribute	1:1	---
Description	Attribute	1:1	---
Color	Attribute	Many:1	---
Size	Attribute	Many:1	Many: 1 to Size Range
Size Range	Attribute	Many:1	
Subcategory	Attribute	Many:1	Many: 1 to Category
Category	Attribute	Many:1	

Table 1 displays the design of a simple product dimension table. In this simple example, the product dimension table has one primary key column, the product key. The other columns in the dimension table are attributes that provide descriptive context to the primary key such as product SKU, description, and color. From a relational perspective, all of these attributes either have a many-to-one relationship to the primary key or a one-to-one relationship to the primary key. Some of these attributes also have relationships to other attributes. For example, size has a many-to-one relationship with size range and subcategory has a many-to-one relationship with category.

Just as it is necessary to understand and define the functional dependencies among fields in relational databases, you must also follow the same practices in Analysis Services. Analysis Services must understand the relationships among your attributes in order to correctly aggregate data, effectively store and retrieve data, and create useful aggregations. To help you create these associations among your dimension attributes, Analysis Services provides a feature called *attribute relationships*. As the name suggests, an attribute relationship describes the relationship between two attributes.

When you initially create a dimension, Analysis Services auto-builds a dimension structure with many-to-one attribute relationships between the primary key attribute and every other dimension attribute as displayed in Figure 5.

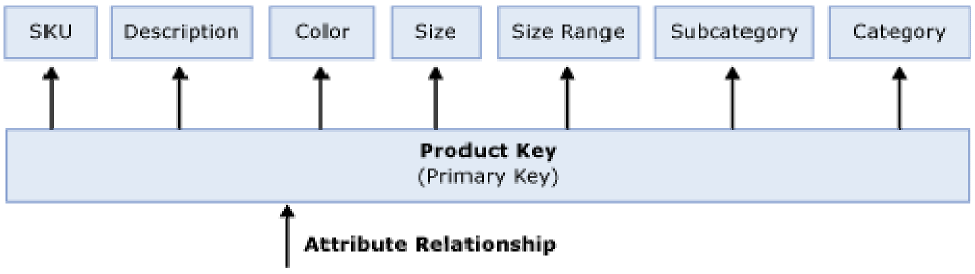


Figure 5 Default attribute relationships

The arrows in Figure 5 represent the attribute relationships between product key and the other attributes in the dimension. While the dimension structure presented in Figure 5 provides a valid representation of the data from the product dimension table, from a performance perspective, it is not an optimized dimension structure since Analysis Services is not aware of the relationships among the attributes.

With this design, whenever you issue a query that includes an attribute from this dimension, data is always summarized from the primary key and then grouped by the attribute. So if you want sales summarized by Subcategory, individual product keys are grouped on the fly by Subcategory. If your query requires sales by Category, individual product keys are once again grouped on the fly by Category. This is somewhat inefficient since Category totals could be derived from Subcategory totals. In addition, with this design, Analysis Services doesn’t know which attribute combinations naturally exist in the dimension and must use the fact data to identify meaningful member combinations. For example, at query time, if a user requests data by Subcategory and Category, Analysis Services must do extra work to determine that the combination of Subcategory: Mountain Bikes and Category: Accessories does not exist.

To optimize this dimension design, you must understand how your attributes are related to each other and then take steps to let Analysis Services know what the relationships are.

To enhance the product dimension, the structure in Figure 6 presents an optimized design that more effectively represents the relationships in the dimension.

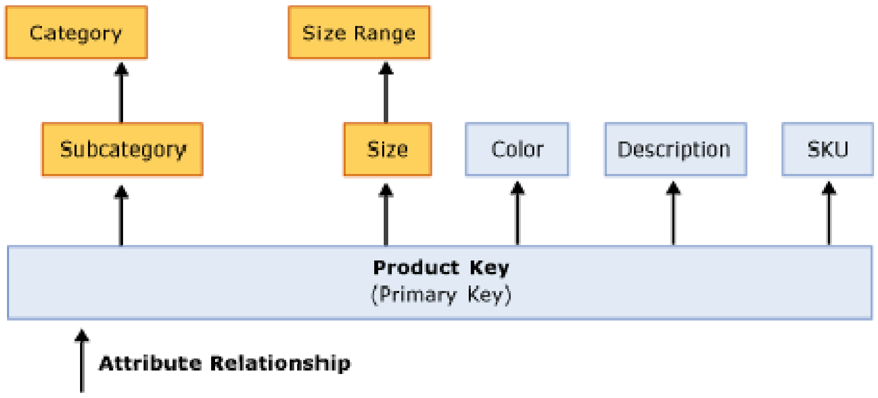


Figure 6 Product dimension with optimized attribute relationships

Note that the dimension design in Figure 6 is different than the design in Figure 5. In Figure 5, the primary key has attribute relationships to every other attribute in the dimension. In Figure 6, two new attribute relationships have been added between Size and Size Range and Subcategory and Category.

The new relationships between Size and Size Range and Subcategory and Category reflect the many-to-one relationships among the attributes in the dimension. Subcategory has a many-to-one relationship with Category. Size has a many-to-one relationship to Size Range. These new relationships tell Analysis Services how the nonprimary key attributes (Size and Size Range, and Subcategory and Category) are related to each other.

Typically many-to-one relationships follow data hierarchies such as the hierarchy of products, subcategories, and categories depicted in Figure 6. While a data hierarchy can commonly suggest many-to-one relationships, do not automatically assume that this is always the case. Whenever you add an attribute relationship between two attributes, it is important to first verify that the attribute data strictly adheres to a many-to-one relationship. As a general rule, you should create an attribute relationship from attribute A to attribute B if and only if the number of distinct (a, b) pairs from A and B is the same (or smaller) than the number of distinct members of A. If you create an attribute relationship and the data violates the many-to-one relationship, you will receive incorrect data results.

Consider the following example. You have a time dimension with a month attribute containing values such as January, February, March and a year attribute containing values such as 2004, 2005, and 2006. If you define an attribute relationship between the month and year attributes, when the dimension is processed, Analysis Services does not know how to distinguish among the months for each year. For example, when it comes across the January member, it does not which year should it roll it up to. The only way to ensure that data is correctly rolled up from month to year is to change the definition of the month attribute to month and year. You make this definition change by changing the **KeyColumns** property of the attribute to be a combination of month and year.

The **KeyColumns** property consists of a source column or combination of source columns (known as a *collection*) that uniquely identifies the members for a given attribute. Once you define attribute relationships among your attributes, the importance of the **KeyColumns** property is highlighted. For every attribute in your dimension, you must ensure that the **KeyColumns** property of each attribute uniquely identifies each attribute member. If the **KeyColumns** property does not uniquely identify each member, duplicates encountered during processing are ignored by default, resulting in incorrect data rollups.

Note that if the attribute relationship has a default **Type** of Flexible, Analysis Services does not provide any notification that it has encountered duplicate months and incorrectly assigns all of the months to the first year or last year depending on data refresh technique. For more information on the **Type** property and the impact of your data refresh technique on key duplicate handling, see Optimizing dimension inserts, updates, and deletes in this white paper.

Regardless of your data refresh technique, key duplicates typically result in incorrect data rollups and should be avoided by taking the time to set a unique **KeyColumns** property. Once you have correctly configured the **KeyColumns** property to uniquely define an attribute, it is a good practice to change the default error configuration for the dimension so that it no longer ignores duplicates. To do this, set the **KeyDuplicate** property from **IgnoreError** to **ReportAndContinue** or **ReportAndStop**. With this change, you can be alerted of any situation where the duplicates are detected.

Whenever you define a new attribute relationship, it is critical that you remove any redundant relationships for performance and data correctness. In Figure 6, with the new attribute relationships, the Product Key no longer requires direct relationships to Size Range or Category. As such, these two attribute relationships have been removed. To help you identify redundant attribute relationships, Business Intelligence Development Studio provides a visual warning to alert you about the redundancy; however, it does not require you to eliminate the redundancy. It is a best practice to always manually remove the redundant relationship. Once you remove the redundancy, the warning disappears.

Even though Product Key is no longer directly related to Size Range and Category, it is still indirectly related to these attributes through a chain of attribute relationships. More specifically, Product Key is related to Size Range using the chain of attribute relationships that link Product Key to Size and Size to Size Range. This chain of attribute relationships is also called *cascading attribute relationships*.

With cascading attribute relationships, Analysis Services can make better performance decisions concerning aggregation design, data storage, data retrieval, and MDX calculations. Beyond performance considerations, attribute relationships are also used to enforce dimension security and to join measure group data to nonprimary key granularity attributes. For example, if you have a measure group that contains sales data by Product Key and forecast data by Subcategory, the forecast measure group will only know how to roll up data from Subcategory to Category if attribute relationships exists between Subcategory and Category.

The core principle behind designing effective attribute relationships is to create the most efficient dimension model that best represents the semantics of your business. While this section provides guidelines and best practices for optimizing your dimension design, to be successful, you must be extremely familiar with your data and the business requirements that the data must support before considering how to tune your design.

Consider the following example. You have a time dimension with an attribute called Day of Week. This attribute contains seven members, one for each day of the week, where the Monday member represents all of the Mondays in your time dimension. Given what you learned from the month / year example, you may think that you should immediately change the **KeyColumns** property of this attribute to concatenate the day with the calendar date or some other attribute. However, before making this change, you should consider your business requirements. The day-of-week grouping can be valuable in some analysis scenarios such as analyzing retail sales patterns by the day of the week. However, in other applications, the day of the week may only be interesting if it is concatenated with the actual calendar date. In other words, the best design depends on your analysis scenario. So while it is important to follow best practices for modifying dimension properties and creating efficient attribute relationships, ultimately you must ensure that your own business requirements are satisfied. For additional thoughts on various dimension designs for a time dimension, see the blog Time calculations in UDM: Parallel Period.

Using hierarchies effectively

In Analysis Services, attributes can be exposed to users by using two types of hierarchies: attribute hierarchies and user hierarchies. Each of these hierarchies has a different impact on the query performance of your cube.

Attribute hierarchies are the default hierarchies that are created for each dimension attribute to support flexible analysis. For non parent-child hierarchies, each attribute hierarchy consists of two levels: the attribute itself and the All level. The All level is automatically exposed as the top level attribute of each attribute hierarchy.

Note that you can disable the All attribute for a particular attribute hierarchy by using the **IsAggregatable** property. Disabling the All attribute is generally not advised in most design scenarios. Without the All attribute, your queries must always slice on a specific value from the attribute hierarchy. While you can explicitly control the slice by using the **Default Member** property, realize that this slice applies across all queries regardless of whether your query specifically references the attribute hierarchy. With this in mind, it is never a good idea to disable the All attribute for multiple attribute hierarchies in the same dimension.

From a performance perspective, attributes that are only exposed in attribute hierarchies are not automatically considered for aggregation. This means that no aggregations include these attributes. 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 somewhat slow.

To enhance performance, it is possible to flag an attribute as an aggregation candidate by using the **Aggregation Usage** property. For more detailed information on this technique, see Suggesting aggregation candidates in this white paper. However, before you modify the **Aggregation Usage** property, you should consider whether you can take advantage of user hierarchies.

In user hierarchies, attributes are arranged into predefined multilevel navigation trees to facilitate end user analysis. Analysis Services enables you to build two types of user hierarchies: natural and unnatural hierarchies, each with 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 most scenarios, natural hierarchies follow the chain of many-to-one relationships that "naturally" exist in your data. In the product dimension example discussed earlier in Figure 6, you may decide to create a Product Grouping hierarchy that from bottom-to-top consists of Products, Product Subcategories, and Product Categories. In this scenario, from the bottom of the hierarchy to the top of the hierarchy, each attribute is directly related to the attribute in the next level of the hierarchy. In an alternative design scenario, you may have a natural hierarchy that from bottom to top consists of Products and Product Categories. Even though Product Subcategories has been removed, this is still a natural hierarchy since Products is indirectly related to Product Category via cascading attribute relationships.

- 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 Size Range and Category or vice versa.

From a performance perspective, natural hierarchies behave very differently than unnatural hierarchies. 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. This is a very important characteristic of natural hierarchies, important enough that you should consider creating natural hierarchies wherever possible. For more information on aggregation candidates, see Suggesting 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. An alternative to using unnatural hierarchies is to cross-join the data by using MDX at query time. The performance of the unnatural hierarchies vs. cross-joins at query time is relatively similar. Unnatural hierarchies simply provide the added benefit of reusability and central management.

To take advantage of natural hierarchies, you must make sure that you have correctly set up cascading attribute relationships for all attributes participating in the hierarchy. Since creating attribute relationships and creating hierarchies are two separate operations, it is not uncommon to inadvertently miss an attribute relationship at some point in the hierarchy. If a relationship is missing, Analysis Services classifies the hierarchy as an unnatural hierarchy, even if you intended it be a natural hierarchy.

To verify the type of hierarchy that you have created, Business Intelligence Development Studio issues a warning icon ⚠ whenever you create a user hierarchy that is missing one or more attribute relationships. The purpose of the warning icon is to help identify situations where you have intended to create a natural hierarchy but have inadvertently missed attribute relationships. Once you create the appropriate attribute relationships for the hierarchy in question, the warning icon disappears. If you are intentionally creating an unnatural hierarchy, the hierarchy continues to display the warning icon to indicate the missing relationships. In this case, simply ignore the warning icon.

In addition, while this is not a performance issue, be mindful of how your attribute hierarchies, natural hierarchies, and unnatural hierarchies are displayed to end users in your front end tool. For example, if you have a series of geography attributes that are generally queried by using a natural hierarchy of Country/Region, State/Province, and City, you may consider hiding the individual attribute hierarchies for each of these attributes in order to prevent redundancy in the user experience. To hide the attribute hierarchies, use the **AttributeHierarchyVisible** property.

Maximizing the value of aggregations

An aggregation is a precalculated summary of data that Analysis Services uses to enhance query performance. More specifically, an aggregation summarizes measures by a combination of dimension attributes.

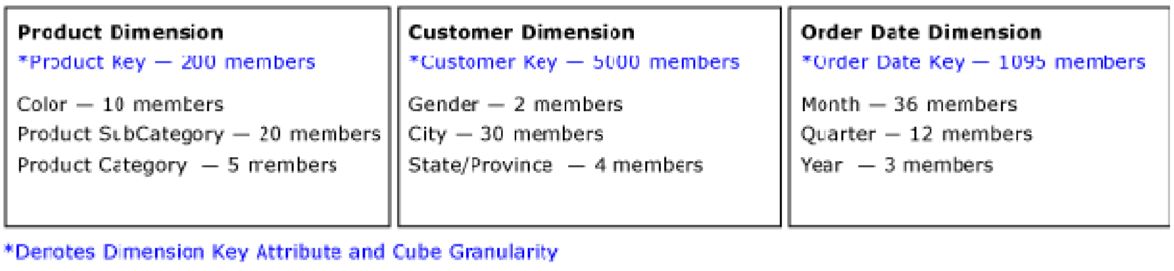
Designing aggregations is the process of selecting the most effective aggregations for your querying workload. As you design aggregations, you must consider the querying benefits that aggregations provide compared with the time it takes to create and refresh the aggregations. On average, having more aggregations helps query performance but increases the processing time involved with building aggregations.

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 on how you can improve query performance using multiple partitions, see Using partitions to enhance query performance.

How aggregations help

While pre-aggregating data to improve query performance sounds reasonable, how do aggregations actually help Analysis Services satisfy queries more efficiently? The answer is simple. Aggregations reduce the number of records that the Storage Engine needs to scan from disk in order to satisfy a query. To gain some perspective on how this works, first consider how the Storage Engine satisfies a query against a cube with no aggregations.

While you may think that the number of measures and fact table records are the most important factors in aggregating data, dimensions actually play the most critical role in data aggregation, determining how data is summarized in user queries. To help you visualize this, Figure 7 displays three dimensions of a simple sales cube.



Product Dimension	**Customer Dimension**	**Order Date Dimension**
*Product Key — 200 members	*Customer Key — 5000 members	*Order Date Key — 1095 members
Color — 10 members	Gender — 2 members	Month — 36 members
Product SubCategory — 20 members	City — 30 members	Quarter — 12 members
Product Category — 5 members	State/Province — 4 members	Year — 3 members

*Denotes Dimension Key Attribute and Cube Granularity

Figure 7 Product, Customer, and Order Date dimensions

Each dimension has four attributes. At the grain of the cube, there are 200 individual products, 5,000 individual customers, and 1,095 order dates. The maximum potential number of detailed values in this cube is the Cartesian product of these numbers: 200 * 5000 * 1095 or 109,500,000 theoretical combinations. This theoretical value is only possible if every customer buys every product on every day of every year, which is unlikely. In reality, the data distribution is likely a couple of orders of magnitude lower than the theoretical value. For this scenario, assume that the example cube has 1,095,000 combinations at the grain, a factor of 100 lower than the theoretical value.

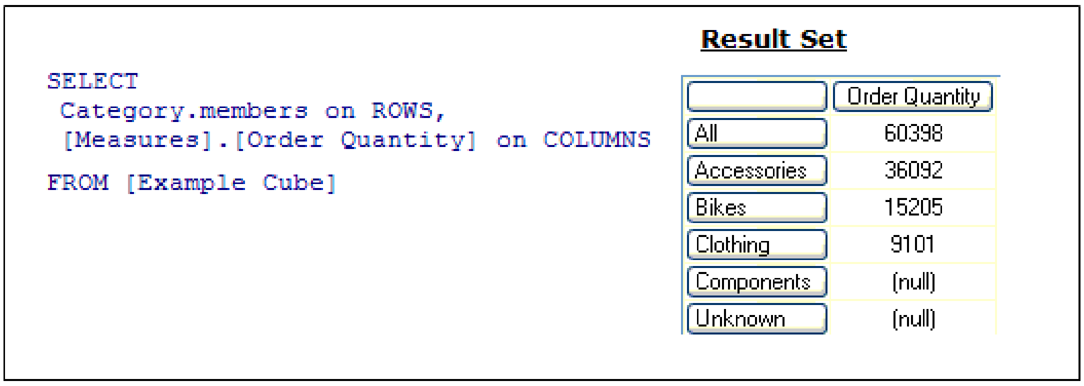
Querying the cube at the grain is uncommon, given that such a large result set (1,095,000 cells) is probably not useful for end users. For any query that is not at the cube grain, the Storage Engine must perform an on-the-fly summarization of the detailed cells by the other dimension attributes, which can be costly to query performance. To optimize this summarization, Analysis Services uses aggregations to precalculate and store summaries of data during cube processing. With the aggregations readily available at query time, query performance can be improved greatly.

Continuing with the same cube example, if the cube contains an aggregation of sales by the month and product subcategory attributes, a query that requires sales by month by product subcategory can be directly satisfied by the aggregation without going to the fact data. The maximum number of cells in this aggregation is 720 (20 product subcategory members * 36 months, excluding the All attribute). While the actual number cells in the aggregation is again dependent on the data distribution, the maximum number of cells, 720, is considerably more efficient than summarizing values from 1,095,000 cells.

In addition, the benefit of the aggregation applies beyond those queries that directly match the aggregation. Whenever a query request is issued, the Storage Engine attempts to use any aggregation that can help satisfy the query request, including aggregations that are at a finer level of detail. For these queries, the Storage Engine simply summarizes the cells in the aggregation to produce the desired result set. For example, if you request sales data summarized by month and product category, the Storage Engine can quickly summarize the cells in the month and product subcategory aggregation to satisfy the query, rather than re-summarizing data from the lowest level of detail. To realize this benefit, however, requires that you have properly designed your dimensions with attribute relationships and natural hierarchies so that Analysis Services understands how attributes are related to each other. For more information on dimension design, see Optimizing the dimension design.

How the Storage Engine uses aggregations

To gain some insight into how the Storage Engine uses aggregations, you can use SQL Server Profiler to view how and when aggregations are used to satisfy queries. Within SQL Server Profiler, there are several events that describe how a query is fulfilled. The event that specifically pertains to aggregation hits is the **Get Data From Aggregation** event. Figure 8 displays a sample query and result set from an example cube.



```
SELECT
 Category.members on ROWS,
 [Measures].[Order Quantity] on COLUMNS
FROM [Example Cube]
```

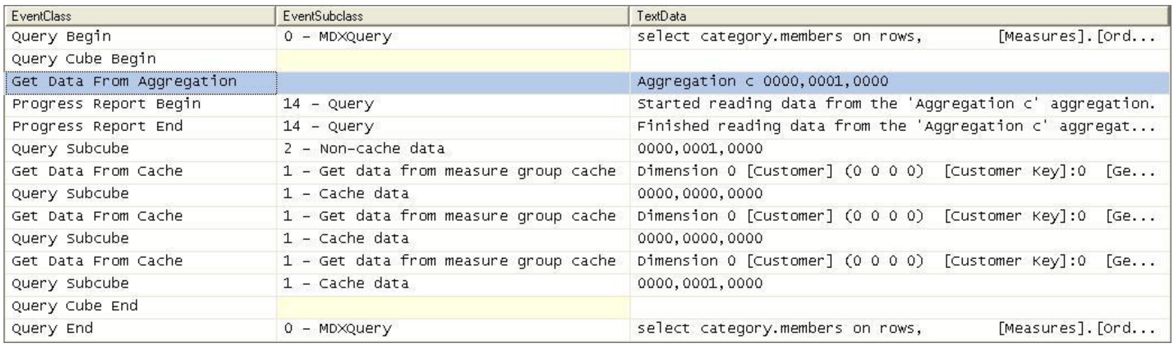
Result Set

	Order Quantity
All	60398
Accessories	36092
Bikes	15205
Clothing	9101
Components	(null)
Unknown	(null)

Figure 8 Sample query and result set

For the query displayed in Figure 8, you can use SQL Server Profiler to compare how the query is resolved in the following two scenarios:

- Scenario 1—Querying against a cube where an aggregation satisfies the query request.
- Scenario 2—Querying against a cube where no aggregation satisfies the query request.



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

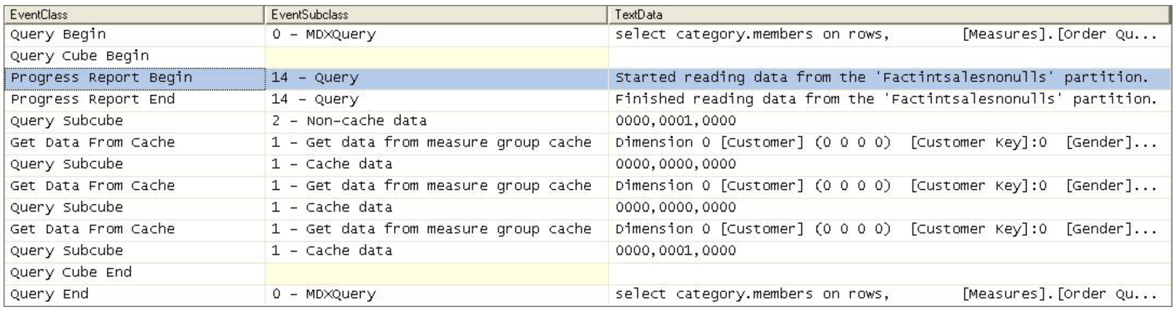
Figure 9 displays a SQL Server Profiler trace of the query's resolution against a cube with aggregations. In the SQL Server Profiler trace, you can see the operations that the Storage Engine performs to produce the result set.

To satisfy the query, the following operations are performed:

1. After the query is submitted, the Storage Engine gets data from Aggregation C 0000, 0001, 0000 as indicated by the **Get Data From Aggregation** event.

   Aggregation C is the name of the aggregation. Analysis Services assigns the aggregation with a unique name in hexadecimal format. Note that aggregations that have been migrated from earlier versions of Analysis Services use a different naming convention.

2. In addition to the aggregation name, Aggregation C, Figure 9 displays a vector, **000, 0001, 0000** , that describes the content of the aggregation. More information on what this vector actually means is described in How to interpret aggregations.
3. The aggregation data is loaded into the Storage Engine measure group cache.
4. Once in the measure group cache, the Query Execution Engine retrieves the data from the cache and returns the result set to the client.



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

Figure 10 displays a SQL Server Profiler trace for the same query against the same cube but this time, the cube has no aggregations that can satisfy the query request.

To satisfy the query, the following operations are performed:

1. After the query is submitted, rather than retrieving data from an aggregation, the Storage Engine goes to the detail data in the partition.
2. From this point, the process is the same. The data is loaded into the Storage Engine measure group cache.
3. Once in the measure group cache, the Query Execution Engine retrieves the data from the cache and returns the result set to the client.

To summarize these two scenarios, when SQL Server Profiler displays **Get Data From Aggregation**, this indicates an aggregation hit. With an aggregation hit, the Storage Engine can retrieve part or all of the data answer from the aggregation and does not need to go to the data detail. Other than fast response times, aggregation hits are a primary indication of a successful aggregation design.

To help you achieve an effective aggregation design, Analysis Services provides tools and techniques to help you create aggregations for your query workload. For more information on these tools and techniques, see Which aggregations are built in this white paper. Once you have created and deployed your aggregations, SQL Server Profiler provides excellent insight to help you monitor aggregation usage over the lifecycle of the application.

Why not create every possible aggregation?

Since aggregations can significantly improve query performance, you may wonder why not create every possible aggregation? Before answering this question, first consider what creating every possible aggregation actually means in theoretical terms.

Note that the goal of this theoretical discussion is to help you understand how aggregations work in an attribute-based architecture. It is not meant to be a discussion of how Analysis Services actually determines which aggregations are built. For more information on this topic, see Which aggregations are built in this white paper.

Generally speaking, an aggregation summarizes measures by a combination of attributes. From an aggregation perspective, for each attribute, there are two levels of detail: the attribute itself and the All attribute. Figure 11 displays the levels of detail for each attribute in the product dimension.

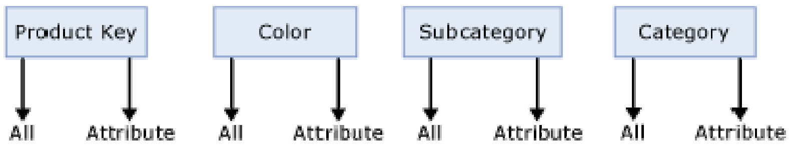


Figure 11 Attribute levels for the product dimension

With four attributes and two levels of detail (All and attribute), the total possible combination for the product dimension is 2*2*2*2 or 2^4 = 16 vectors or potential aggregations.

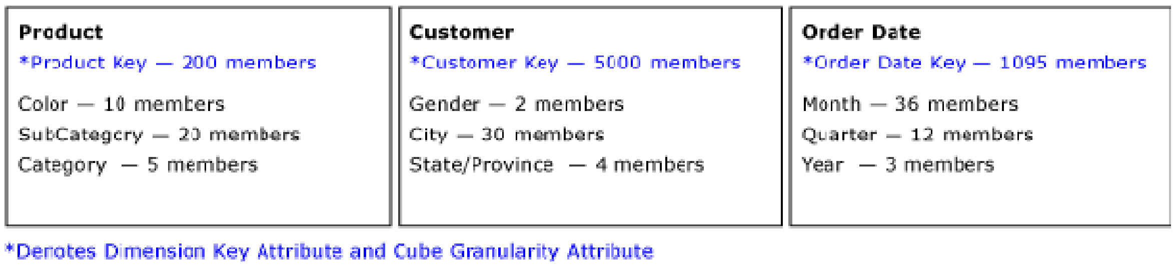


Figure 12 (3) dimensions with (4) attributes per dimension

If you apply this logic across all attributes displayed in Figure 12, the total number of possible aggregations can be represented as follows:

Total Number of Aggregations

2 (product key) *2 (color) *2 (product subcategory) * 2 (product category)*

2 (customer key) *2 (gender) *2 (city) *2 (state/province) *

2 (order date key) * 2 (month) * 2 (quarter)* 2 (year)

= 2^12

= 4096

Based on this example, the total potential aggregations of any cube can be expressed as 2^ (total number of attributes). While a cube with twelve attributes produces 4,096 theoretical aggregations, a large scale cube may have hundreds of attributes and consequently an exponential increase in the number of aggregations. A cube with 100 attributes, for example, would have 1.26765E+30 theoretical aggregations!

The good news is that this is just a theoretical discussion. Analysis Services only considers a small percentage of these theoretical aggregations, and eventually creates an even smaller subset of aggregations. As a general rule, an effective Analysis Services aggregation design typically contains tens or hundreds of aggregations, not thousands.

With that in mind, the theoretical discussion reminds us that as you add additional attributes to your cube, you are potentially increasing the number of aggregations that Analysis Services must consider. Furthermore, since aggregations are created at the time of cube processing, too many aggregations can negatively impact processing performance or require excessive disk space to store. As a result, ensure that your aggregation design supports the required data refresh timeline.

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 cube. To find the order of dimensions in the cube, use one of the following two techniques. With the cube opened in SQL Server Business Intelligence Development Studio, you can review the order of dimensions in a cube on the **Cube Structure** tab. The order of dimensions in the cube is displayed in the Dimensions pane, on both the **Hierarchies** tab and the **Attributes** tab. As an alternative, you can review the order of dimensions listed in the cube XML file.

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 following subcube definition (0000, 0001, 0001) describes an aggregation for:

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.

Which aggregations are built

To decide which aggregations are considered and created, Analysis Services provides an aggregation design algorithm that uses a cost/benefit analysis to assess the relative value of each aggregation candidate.

- **Aggregation Cost**—The cost of an aggregation is primarily influenced by the aggregation size. To calculate the size, Analysis Services gathers statistics including source record counts and member counts, as well as design metadata including the number of dimensions, measures, and attributes. Once the aggregation cost is calculated, Analysis Services performs a series of tests to evaluate the cost against absolute cost thresholds and to evaluate the cost compared to other aggregations.
- **Aggregation Benefit**—The benefit of the aggregation depends on how well it reduces the amount of data that must be scanned during querying. For example, if you have 1,000,000 data values that are summarized into an aggregation of fifty values, this aggregation greatly benefits query performance. Remember that Analysis Services can satisfy queries by using an aggregation that matches the query subcube exactly, or by summarizing data from an aggregation at a lower level (a more detailed level). As Analysis Services determines which aggregations should be built, the algorithm needs to understand how attributes are related to each other so

it can detect which aggregations provide the greatest coverage and which aggregations are potentially redundant and unnecessary.

To help you build aggregations, Analysis Services exposes the algorithm using two tools: the Aggregation Design Wizard and the Usage-Based Optimization Wizard.

- The **Aggregation Design Wizard** designs aggregations based on your cube design and data distribution. Behind the scenes, it selects aggregations using a cost/benefit algorithm that accepts inputs about the design and data distribution of the cube. The Aggregation Design Wizard can be accessed in either Business Intelligence Development Studio or SQL Server Management Studio.
- The **Usage-Based Optimization Wizard** designs aggregations based on query usage patterns. The Usage-Based Optimization Wizard uses the same cost/benefit algorithm as the Aggregation Design Wizard except that it provides additional weighting to those aggregation candidates that are present in the Analysis Services query log. The Usage-Based Optimization Wizard can be accessed in either Business Intelligence Development Studio (BIDS) or SQL Server Management Studio. To use the Usage-Based Optimization Wizard, you must capture end-user queries and store the queries in a query log. To set up and configure the query log for an instance of Analysis Services, you can access a variety of configuration settings in SQL Server Management Studio to control the sampling frequency of queries and the location of the query log.

In those special scenarios when you require finer grained control over aggregation design, SQL Server Service Pack 2 samples includes an advanced Aggregation utility. Using this advanced tool, you can manually create aggregations without using the aggregation design algorithm. For more information on the Aggregation utility, see Appendix C.

How to impact aggregation design

To help Analysis Services successfully apply the aggregation design algorithm, you can perform the following optimization techniques to influence and enhance the aggregation design. (The sections that follow describe each of these techniques in more detail).

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.

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.

Adopting an aggregation design strategy – To help you design the most effective aggregations for your implementation, it is useful to adopt an aggregation design strategy that leverages the strengths of each of the aggregation design methods at various stages of your development lifecycle.

Suggesting aggregation candidates

When Analysis Services designs aggregations, the aggregation design algorithm does not automatically consider every attribute for aggregation. Remember the discussion of the potential number of aggregations in a cube? If Analysis Services were to consider every attribute for aggregation, it would take too long to design the aggregations, let alone populate them with data. To streamline this process, Analysis Services uses the **Aggregation Usage** property to determine which attributes it should automatically consider for aggregation. For every measure group, verify the attributes that are automatically considered for aggregation and then determine whether you need to suggest additional aggregation candidates.

The 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 may 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. As you may guess, this is the default value of the **Aggregation Usage** property.

The default rule is highly conservative about which attributes are considered for aggregation. Therefore, it is extremely important that you understand how the default rule works. The default rule is broken down into four constraints:

1. **Default Constraint 1—Unrestricted for the Granularity and All Attributes** - For the dimension attribute that is the measure group granularity attribute and the All attribute, apply **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.

   To help you visualize how Default Constraint 1 is applied, Figure 13 displays a product dimension with six attributes. Each attribute is displayed as an attribute hierarchy. In addition, three user hierarchies are included in the dimension. Within the user hierarchies, there are two natural hierarchies displayed in blue and one unnatural hierarchy displayed in grey. In addition to the All attribute (not pictured in the diagram), the attribute in yellow, Product Key, is the only aggregation candidate that is considered after the first constraint is applied. Product Key is the granularity attribute for the measure group.

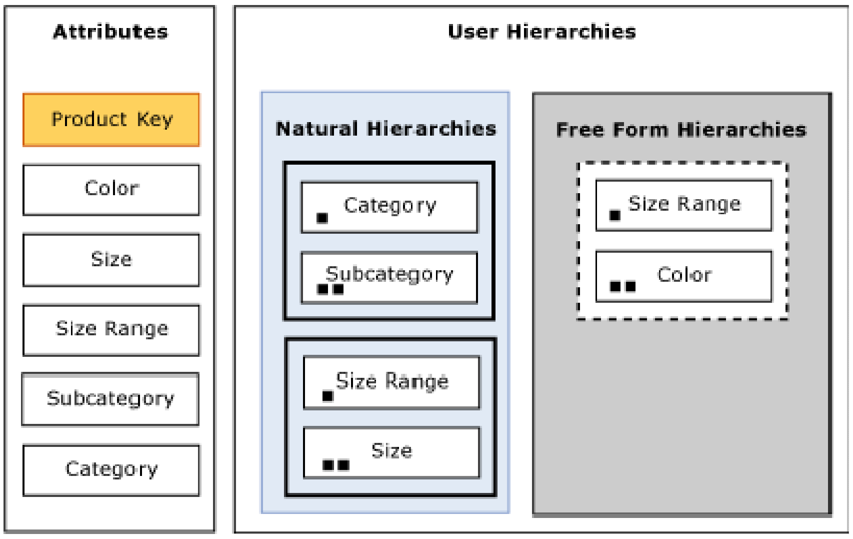


Figure 13 Product dimension aggregation candidates after applying Default Constraint 1

2. **Default Constraint 2—None for Special Dimension Types -** For all attributes (except All) in many-to-many, nonmaterialized reference dimensions, and data mining dimensions, use **None**. The product dimension in Figure 13 is a standard dimension. Therefore it is not affected by constraint 2. For more information on many-to-many and reference dimensions, see Complex dimension relationships.
3. **Default Constraint 3—Unrestricted for Natural Hierarchies -** For all user hierarchies, apply a special scanning process to identify the attributes in natural hierarchies. As you recall, a natural hierarchy is a user hierarchy where all attributes participating in the hierarchy contain attribute relationships at every level of the hierarchy.

   To identify the natural hierarchies, Analysis Services scans each user hierarchy starting at the top level and then moves down through the hierarchy to the bottom level. For each level, it checks whether the attribute of the current level is linked to the attribute of the next level via a direct or indirect attribute relationship, for every attribute that pass the natural hierarchy test, apply **Unrestricted,** except for nonaggregatable attributes, which are set to **Full**.
4. **Default Constraint 4—None For Everything Else**. For all other dimension attributes, apply **None**. In this example, the color attribute falls into this bucket since it is only exposed as an attribute hierarchy.

Figure 14 displays what the Product dimension looks like after all constraints in the default rule have been applied. The attributes in yellow highlight the aggregation candidates.

- As a result of Default Constraint 1, the Product Key and All attributes have been identified as candidates.

- As a result of Default Constraint 3, the Size, Size Range, Subcategory, and Category attributes have also been identified as candidates.
- After Default Constraint 4 is applied, Color is still not considered for any aggregation.

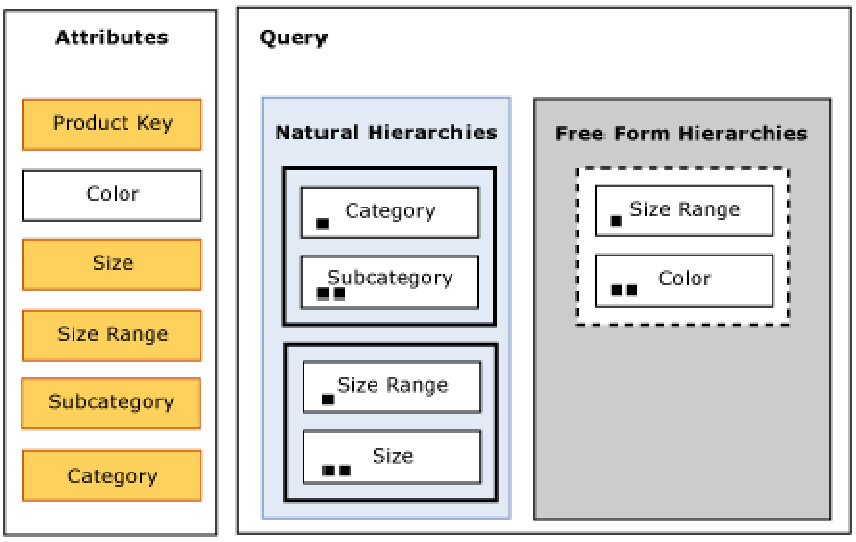
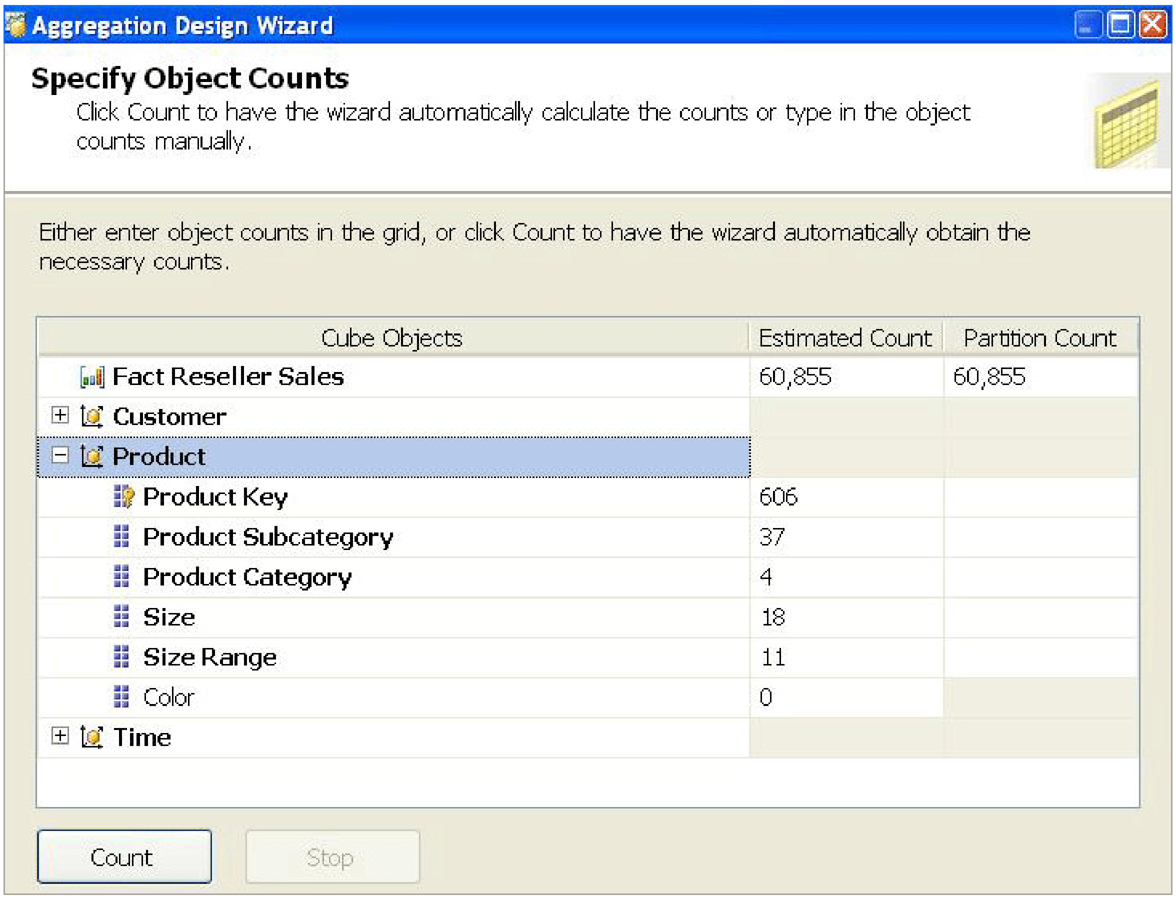


Figure 14 Product dimension aggregation candidates after all application of all default constraints

While the diagrams are helpful to visualize what happens after the each constraint is applied, you can view the specific aggregation candidates for your own implementation when you use the Aggregation Design Wizard to design aggregations.

Figure 15 displays the Specify Object Counts page of the Aggregation Design Wizard. On this Wizard page, you can view the aggregation candidates for the Product dimension displayed in Figure 14. The bold attributes in the Product Dimension are the aggregation candidates for this dimension. The Color attribute is not bold because it is not an aggregation candidate. This Specify Object Counts page is discussed again in Specifying statistics about cube metadata, which describes how you can update statistics to improve aggregation design.



Specify Object Counts

Click Count to have the wizard automatically calculate the counts or type in the object counts manually.

Either enter object counts in the grid, or click Count to have the wizard automatically obtain the necessary counts.

Cube Objects	Estimated Count	Partition Count
Fact Reseller Sales	60,855	60,855
⊞ Customer		
⊟ Product		
Product Key	606	
Product Subcategory	37	
Product Category	4	
Size	18	
Size Range	11	
Color	0	
⊞ Time		

Figure 15 Aggregation candidates in the Aggregation Design Wizard

Influencing aggregation candidates

In light of the behavior of the **Aggregation Usage** property, following are some guidelines that you can adopt to influence the aggregation candidates for your implementation. Note that by making these modifications, you are influencing the aggregation candidates, not guaranteeing that a specific aggregation is going to be created. The aggregation must still be evaluated for its relative cost and benefit before it is created. The guidelines have been organized into three design scenarios:

- **Dimensions with no user hierarchies**—If your dimension ONLY has attribute hierarchies with no user-defined hierarchies, by default the only attribute that is considered for aggregation is the cube granularity attribute plus the All attribute. As such, you may want to consider adding some natural hierarchies to your design. In addition to the **Aggregation Usage** benefits, your users may enjoy the analysis experience of using predefined navigation paths.
- **Attribute only exposed in an attribute hierarchy**—If a given attribute is only exposed as an attribute hierarchy such as Color in Figure 14, you may want to change its **Aggregation Usage** property as follows:
    - 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 be tempted to set all of your attribute hierarchies to **Unrestricted**. While this scenario seems reasonable, you may quickly find yourself in a position where the Wizard takes a very long time to consider all of the possible candidates and create an aggregation design. In large cubes, the Wizard can take at least an hour to complete the design and considerably much more time to process. As such, you should set the property to **Unrestricted** only for the commonly queried attribute hierarchies. The general rule is five to ten **Unrestricted** attributes per dimension.
  - 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 very rare case and should only be used for attributes that have a relatively small number of members.

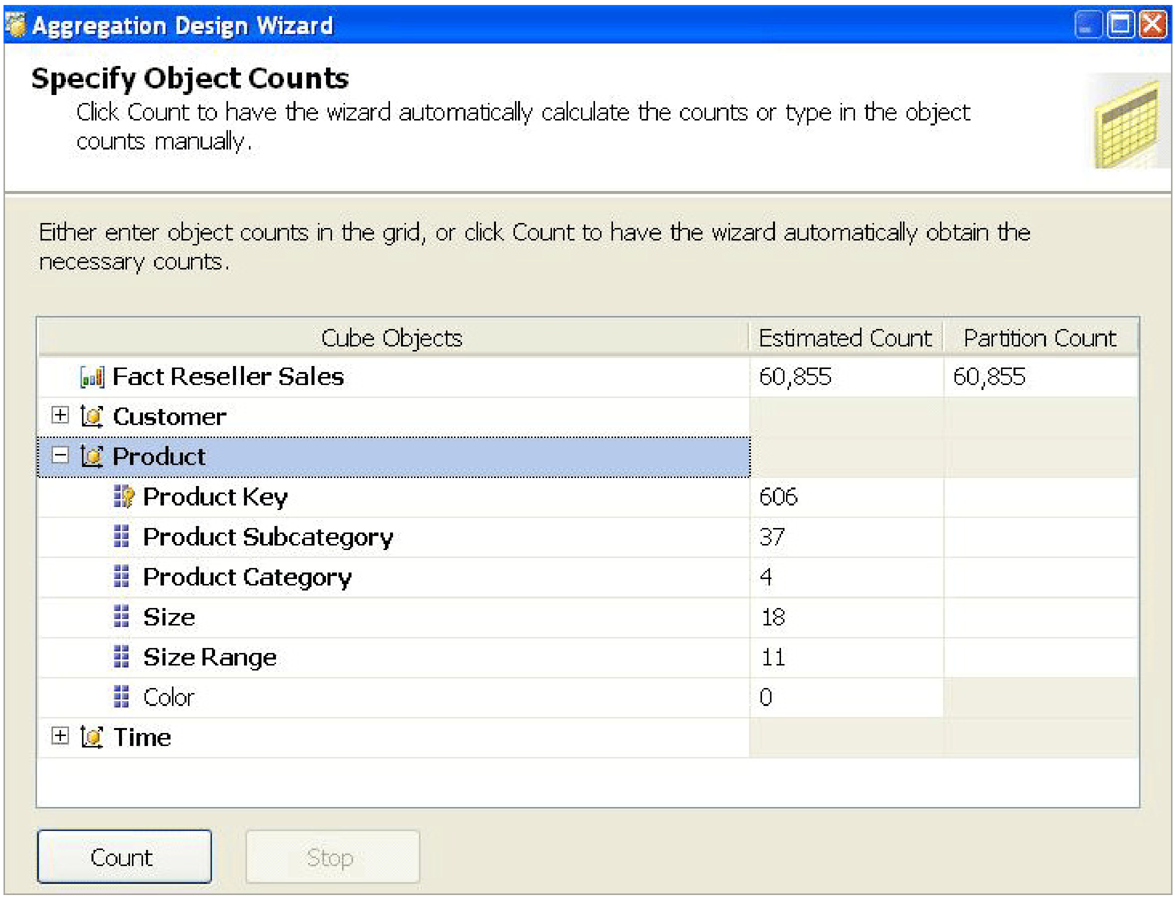
- **Infrequently used attributes**—For attributes participating in natural hierarchies, you may want to change the **Aggregation Usage** property from **Default** to **None** if users would only infrequently use it. Using this approach can help you reduce the aggregation space and get to the five to 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. Another example where you may want to consider setting the **Aggregation Usage** property to **None** is when you have a natural hierarchy where the number of members from one level to the next level is almost identical. For example, if you have 20 product subcategories and 18 product categories, it may make sense to set the product category attribute to None since the I/O penalty of aggregating 20 members up to 18 members is negligible. Generally speaking, if the data does not support at least a 2:1 ratio, you should consider setting the **Aggregation Usage** to **None**.

Specifying statistics about cube data

Once the aggregation design algorithm has identified the aggregation candidates, it performs a cost/benefit analysis of each aggregation. In order 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 record counts. Ensuring that your metadata is up-to-date can improve the effectiveness of your aggregation design.

You can define the fact table source record count in the **EstimatedRows** property of each measure group, and you can define attribute member count in the **EstimatedCount** property of each attribute.

You can modify these counts in the **Specify Counts** page of the Aggregation Design Wizard as displayed in Figure 16.


Aggregation Design Wizard

Specify Object Counts

Click Count to have the wizard automatically calculate the counts or type in the object counts manually.

Either enter object counts in the grid, or click Count to have the wizard automatically obtain the necessary counts.

Cube Objects	Estimated Count	Partition Count
Fact Reseller Sales	60,855	60,855
⊞ Customer		
⊟ Product		
Product Key	606	
Product Subcategory	37	
Product Category	4	
Size	18	
Size Range	11	
Color	0	
⊞ Time		

Figure 16 Specify object counts in the Aggregation Design Wizard

If the count is NULL (i.e., you did not define it during design), clicking the **Count** button populates the counts for each aggregation candidate as well as the fact table size. If the count is already populated, clicking the **Count** button does not update the counts. Rather you must manually change the counts either in the dialog box or programmatically. This is significant when you design aggregations on a small data set and then move the cube to a production database. Unless you update the counts, any aggregation design is built by using the statistics from the development data set.

In addition, when you use multiple partitions to physically divide your data, it is important that the partition counts accurately reflect the data in the partition and not the data across the measure group. So if you create one partition per year, the partition count for the year attribute should be 1. Any blank counts in the Partition Count column use the Estimated Count values, which apply to the entire fact table.

Using these statistics, Analysis Services compares the cost of each aggregation to predefined cost thresholds to determine whether or not an aggregation is too expensive to build. If the cost is too high, it is immediately discarded. One of the most important cost thresholds is known as the one-third rule. Analysis Services never builds an aggregation that is greater than one third of the size of the fact table. In practical terms, the one-third rule typically prevents the building of aggregations that include one or more large attributes.

As the number of dimension members increases at deeper levels in a cube, it becomes less likely that an aggregation will contain these lower levels because of the one-third rule. The aggregations excluded by the one-third rule are those that would be almost as large as the fact level itself and almost as expensive for Analysis Services to use for query resolution as the fact level. As a result, they add little or no value.

When you have dimensions with a large number of members, this threshold can easily be exceeded at or near the leaf level. For example, you have a measure group with the following design:

- Customer dimension with 5,000,000 individual customers organized into 50,000 sales districts and 5,000 sales territories
- Product dimension with 10,000 products organized into 1,000 subcategories and 30 categories
- Time dimension with 1,095 days organized into 36 months and 3 years
- Sales fact table with 12,000,000 sales records

If you model this measure group using a single partition, Analysis Services does not consider any aggregation that exceeds 4,000,000 records (one third of the size of the partition). For example, it does not consider any aggregation that includes the individual customer, given that the customer attribute itself exceeds the one-third rule. In addition, it does not consider an aggregation of sales territory, category, and month since the total number of records of that aggregation could potentially be 5.4 million records consisting of 5,000 sales territories, 30 categories, and 36 months.

If you model this measure group using multiple partitions, you can impact the aggregation design by breaking down the measure group into smaller physical components and adjusting the statistics for each partition.

For example, if you break down the measure group into 36 monthly partitions, you may have the following data statistics per partition:

- Customer dimension has 600,000 individual customers organized into 10,000 sales districts and 3,000 sales territories
- Product dimension with 7,500 products organized into 700 subcategories and 25 categories
- Time dimension with 1 month and 1 year
- Sales fact table with 1,000,000 sales records

With the data broken down into smaller components, Analysis Services can now identify additional useful aggregations. For example, the aggregation for sales territory, category, and month is now a good candidate with (3000 sales territories *25 categories *1 month) or 75,000 records which is less than one third of the partition size of 1,000,000 records. While creating multiple partitions is helpful, it is also critical that you update the member count and partition count for the partition as displayed in Figure 16. If you do not update the statistics, Analysis Services will not know that the partition contains a reduced data set. Note that this example has been provided to illustrate how you can use multiple partitions to impact aggregation design. For practical partition sizing guidelines, including the recommended number of records per partition, see Designing partitions in this white paper.

Note that you can examine metadata stored on, and retrieve support and monitoring information from, an Analysis Services instance by using XML for Analysis (XMLA) schema rowsets. Using this technique, you can access information regarding partition record counts and aggregation size on disk to help you get a better sense of the footprint of Analysis Services cubes.

Adopting an aggregation design strategy

The goal of an aggregation design strategy is to help you design and maintain aggregations throughout your implementation lifecycle. From an aggregation perspective, the cube lifecycle can be broken down into two general stages: initial aggregation design and ongoing tuning based on query patterns.

Initial Aggregation Design

The most effective aggregation designs are those that are customized for the querying patterns of your user base. Unfortunately, when you initially deploy a cube, you probably will not have query usage data available. As such, it is not possible to use the Usage-Based Optimization Wizard. However, because Analysis Services generally resolves user queries faster with some aggregations than with none, you should initially design a limited number of aggregations by using the Aggregation Design Wizard. The number of initial aggregations that you should design depends on the complexity and size of the cube (the fact size).

- **Small cubes—**With a small cube, an effective aggregation strategy is to initially design aggregations using the Aggregation Design Wizard to achieve a 20 to 30 percent increase in performance. Note that if the design has too many attributes to consider for aggregation, there is a chance that the Aggregation Design Wizard will stop before reaching the designed percent performance improvement and the user interface may not visibly show any percent performance increase. In this scenario, the large number of attributes has resulted in many possible aggregations, and the number of aggregations that have actually been created are a very small percentage of the total possible aggregations.
- **Large and complex cubes—**With a large and complex cube, it takes Analysis Services a long time just to design a small percentage of the possible aggregations. Recall that the number of theoretical aggregations in a cube can be expressed as 2^ (total number of Unrestricted attributes). A complex cube with five dimensions that each contain eight attributes has 2^40 or 1.1 trillion aggregation candidates (given that every attribute is an aggregation candidate). With this number, if you assume that the Aggregation Design Wizard can examine 1000 aggregations per second (which is a very generous estimate), it will take the Aggregation Design Wizard approximately 35 years to consider a trillion possible aggregations. Furthermore, a large number of aggregations takes a long time to calculate and consumes a large amount of disk space. While this is a theoretical example, an effective approach with real world cubes that are large and complex is to initially design aggregations to achieve a small performance increase (less than 10 percent and possibly even 1 or 2 percent with very complex cubes) and then allow the Aggregation Design Wizard to run for no more than 15 minutes.
- **Medium complexity cubes—**With a medium-complexity cube, design aggregations to achieve a 10 to 20 percent increase in performance. Then, allow the wizard to run

for no more than 15 minutes. While it is difficult to define what constitutes a high-complexity cube versus a medium-complexity cube, consider this general guideline: a high-complexity cube is a cube contains more than ten **Unrestricted** attributes in any given dimension.

Before you create initial aggregations with the Aggregation Design Wizard, you should evaluate the application **Aggregation Usage** property and modify its value as necessary to minimize aggregations that are rarely used and to maximize the probability of useful aggregations. Using **Aggregation Usage** is equivalent to providing the aggregation algorithm with "hints" about which attributes are frequently and infrequently queried. For specific guidelines on modifying the **Aggregation Usage** property, see Influencing aggregation candidates.

After you design aggregations for a given partition, it is a good practice to evaluate the size of the aggregation files. The total size of all aggregation files for a given partition should be approximately one to two times the size of the source fact table. If the aggregations are greater than two times the size of the fact table, you are likely spending a long time processing your cube to build relatively large aggregation files. During querying, you can potentially experience performance issues when large aggregation files cannot be effectively loaded into memory due to lack of system resources. If you experience these issues, it is good practice to consider reducing the number of aggregation candidates.

Ongoing tuning based on query patterns

After users have queried the cube for a sufficient period of time to gather useful query pattern data in the query log (perhaps a week or two), use the Usage-Based Optimization Wizard to perform a usage-based analysis for designing additional aggregations that would be useful based on actual user query patterns. You can then process the partition to create the new set of aggregations. As usage patterns change, use the Usage-Based Optimization Wizard to update additional aggregations.

Remember that to use the Usage-Based Optimization Wizard, you must capture end-user queries and store the queries in a query log. Logging queries requires a certain amount of overhead so it is generally recommended that you turn off logging and then turn it back on periodically when you need to tune aggregations based on query patterns.

As an alternative to using the Usage-Based Optimization Wizard, if you require finer grained control over aggregation design, SQL Server Service Pack 2 samples includes an advanced Aggregation Utility that allows you to create specific aggregations from the query log without using the aggregation design algorithm. For more information on the Aggregation Utility, see Appendix C.

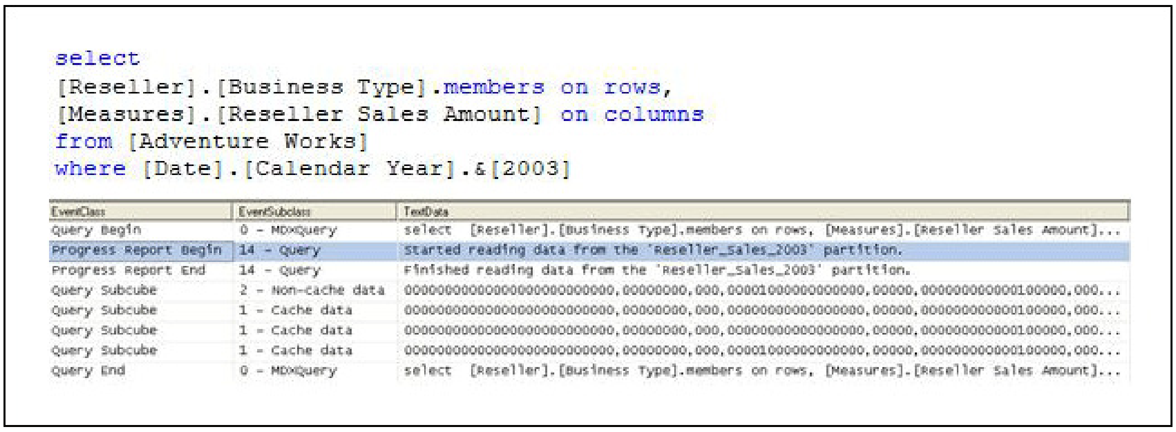
Using partitions to enhance query performance

Partitions separate measure group data into physical 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. The Using partitions to enhance processing performance section discusses the processing and data management benefits of partitions.

How partitions are used in querying

When you query a cube, the Storage Engine 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 go to the fact data. If you have one partition, Analysis Services must scan though all of the fact data in the partition to find the data that you are interested in. While the Storage Engine can query the partition in parallel and use bitmap indexes to speed up data retrieval, with just one partition, performance is not going to be optimal.

As an alternative, you can use multiple partitions to break up your measure group into separate physical components. Each partition can be queried separately and the Storage Engine can query only the partition(s) that contain the relevant data.



```
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	000000000000000000000000000,00000000,000,0000100000000000,00000,000000000000100000,000...
Query Subcube	1 - Cache data	000000000000000000000000000,00000000,000,0000000000000000,00000,000000000000100000,000...
Query Subcube	1 - Cache data	000000000000000000000000000,00000000,000,0000000000000000,00000,000000000000100000,000...
Query Subcube	1 - Cache data	000000000000000000000000000,00000000,000,0000100000000000,00000,000000000000100000,000...
Query End	0 - MDXQuery	select [Reseller].[Business Type].members on rows, [Measures].[Reseller Sales Amount]...

Figure 17 Intelligent querying by partitions

Figure 17 displays a query requesting Reseller Sales Amount by Business Type from a cube called Adventure Works as well as the SQL Server Profiler trace that describes how the query was satisfied. 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 does not have to scan data from other partitions. The SQL Server Profiler trace for this query demonstrates how the query needs to read data only from the 2003 Reseller Sales partition.

Designing partitions

If you have some idea how users query the data, you can partition data in a manner that matches common queries. This may be somewhat difficult if user queries do not follow common patterns. A very common choice for partitions is to select an element of time such as day, month, quarter, year or some combination of time elements. Many queries contain a time element, so partitioning by time often benefits query performance.

When you set up your partitions, you must bind each partition to a source table, view, or source query that contains the subset of data for that partition. For MOLAP partitions, during processing Analysis Services internally identifies the slice 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 slice definition is persisted as a subcube. Knowing this information, the Storage Engine can optimize which partitions it scans during querying by only choosing those partitions that are relevant to the query. For ROLAP and proactive caching partitions, you must manually identify the slice in the properties of the partition.

As you design partitions, use the following guidelines for creating and managing partitions:

- When you decide how to break down your data into partitions, you are generally weighing out partition size vs. number of partitions. Partition size is a function of the number of records in the partition as well as the size of the aggregation files for that partition. Even though the segments in a partition are queried in parallel, if the aggregation files for a partition cannot be effectively managed in memory, you can see significant performance issues during querying.
- In general. the number of records per partition should not exceed 20 million. In addition, the size of a partition should not exceed 250 MB. If the partition exceeds either one of these thresholds, consider breaking the partition into smaller components to reduce the amount of time spent scanning the partition. While having multiple partitions is generally beneficial, having too many partitions, e.g., greater than a few hundred, can also affect performance negatively.
- If you have several partitions that are less than 50 MB or 2 million records per partition, consider consolidating them into one partition. In addition, it is generally not a good practice to create a partition that has less than 4,096 records. In this scenario, given that the record count is so small, the Storage Engine does not create aggregations or indexes for the partition and therefore does not set the auto-slice. Note that this record count threshold is controlled by the **IndexBuildThreshold** property in the msmdsrv.ini file. Falling below this threshold is generally not an issue in production environments since partition data sets are typically much larger than 4,096 records.
- When you define your partitions, remember that they do not have to contain uniform datasets. For example, for a given measure group, you may have three yearly partitions, 11 monthly partitions, three weekly partitions, and 1–7 daily partitions. The value of using heterogeneous partitions with different levels of detail is that you can more easily manage the loading of new data without disturbing existing partitions. In addition, you can design aggregations for groups of partitions that share the same level of detail (more information on this in the next section).
- Whenever you use multiple partitions for a given measure group, you must ensure that you update the data statistics for each partition. More specifically, it is important to ensure that the partition data and member counts accurately reflect the specific data in the partition and not the data across the entire measure group. For more information on how to update partition counts, see Specifying statistics about cube data.
- For distinct count measure groups, consider specifically defining your partitions to optimize the processing and query performance of distinct counts. For more information on this topic, see Optimizing distinct count.

Aggregation considerations for multiple partitions

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. While the flexibility can definitely help you enhance performance, too many aggregation designs across your partitions can introduce overhead.

To help guide your aggregation design, the following are general guidelines for you to consider. When you have less than ten partitions, you should typically have no more than two aggregation designs per measure group. With less than 50 partitions, you typically want no more than three aggregation designs per measure group. For greater than 50 partitions, you want no more than four aggregation designs.

While each partition can have a different aggregation design, it is a good practice to group your partitions based on the data statistics of the partition so that you can apply a single aggregation design to a group of similar partitions.

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.

The newest aggregation design can also be copied to a *base partition*. This base partition holds no data—it serves only to hold the current aggregation design. When it is time to add a new partition (for example, at the start of a new month), the base partition can be cloned to a new partition. When the slice is set on the new partition, it is ready to take data as the current partition. Following an initial full process, the current partition can be incrementally updated for the remainder of the period. For more information on processing techniques, see Refreshing partitions efficiently.

Writing efficient MDX

When the Query Execution Engine executes an MDX query, it translates the query into Storage Engine data requests and then compiles the data to produce a query result set.

During query execution, the Query Execution Engine also executes any calculations that are directly or indirectly referenced, such as calculated members, semi-additive measures, and MDX Script scope assignments. Whenever you directly or indirectly reference calculations in your query, you must consider the impact of the calculations on query performance.

This section presents techniques for writing efficient MDX statements in common design scenarios. The section assumes that the reader has some knowledge of MDX.

Specifying the calculation space

When you need to create MDX calculations that apply business rules to certain cells in a cube, it is critical to write efficient MDX code that effectively specifies the calculation space for each rule.

Before you learn about these MDX coding techniques, it is important to be familiar with some common scenarios where conditional business rules are relevant. Following is a description of a growth calculation where you want to apply unique rules to different time periods. To calculate growth from a prior period, you may think that the logical expression for this calculation is current period minus prior period. In general, this expression is valid; however, it is not exactly correct in the following three situations.

- **The First Period**—In the first time period of a cube, there is no prior period. Since the prior period does not exist for the first period, the expression of current period minus prior period evaluates to current period, which can be somewhat misleading. To avoid end-user confusion, for the prior period, you may decide to apply a business rule that replaces the calculation with NULL.
- **The All Attribute**—You need to consider how the calculation interacts with the All. In this scenario, the calculation simply does not apply to the All, so you decide to apply a rule that sets the value for All to NA.
- **Future Periods**—For time periods that extend past the end of the data range, you must consider where you want the calculation to stop. For example, if your last period of data is December of 2006, you want to use a business rule to only apply the calculation to time periods before and including December of 2006.

Based on this analysis, you need to apply four business rules for the growth from prior period calculation. Rule 1 is for the first period, rule 2 is for the All, rule 3 is for the future time periods and rule 4 is for the remaining time periods. The key to effectively applying these business rules is to efficiently identify the calculation space for each rule. To accomplish this, you have two general design choices:

- You can create a calculated member with an IIF statement that uses conditional logic to specify the calculation space for a given period.
- Alternatively you can create a calculated member and use an MDX Script scope assignment to specify the calculation space for a given period.

In a simple cube with no other calculations, the performance of these two approaches is approximately equal. However, if the calculated member directly or indirectly references any other MDX calculations such as semi-additive measures, calculated members, or MDX Script scope assignments, you can definitely see a performance difference between the two approaches. To see how MDX Script scope assignments can be used to perform the growth calculation, you can use the Business Intelligence Wizard to generate time intelligence calculations in your cube.

To better understand the performance benefit of the scope assignment technique used by the Business Intelligence Wizard, consider the following illustrative example of an effective scope assignment. You require a new calculated member called Weekend Gross Profit. The Weekend Gross Profit is derived from the Gross Profit calculated member. To calculate the Weekend Gross Profit, you must sum the Gross Profit calculated member for the days in a sales weekend. This seems easy enough but there are different business rules that apply to each sales territory to as follows:

- For all of the stores in the North America sales territory, the Weekend Gross Profit should sum the Gross Profit for days 5, 6, and 7, i.e., Friday, Saturday, and Sunday.
- For all of the stores in the Pacific sales territory, the Weekend Gross Profit should be NULL. You want to set it to NULL because the Pacific territory is in the process of closing of all of its stores and the Gross Profit numbers are significantly skewed due to their weekend clearance sales.
- For all other territories, the Weekend Gross Profit should sum the Gross Profit for days 6 and 7, i.e., Saturday and Sunday.

To satisfy the conditions of this scenario, following are two design options that you can choose.

Option 1—Calculated Member

You can use a calculated member with an IIF statement to apply the conditional summing of Gross Profit. In this scenario, the Query Execution Engine must evaluate the calculation space at runtime on a cell-by-cell basis based on the IIF specification. As a result, the Query Execution Engine uses a less optimized code path to execute the calculation that increases query response time.

```
with member [Option 1 Weekend Gross Profit] as
iif (ancestor([Sales Territory].[Sales Territory].CurrentMember,
              [Sales Territory].[Sales Territory].[Group])
        IS [Sales Territory].[Sales Territory].[Group].&[North America],
        Sum({[Date].[Day of Week].&[5],
             [Date].[Day of Week].&[6],
             [Date].[Day of Week].&[7]},
             [Measures].[Reseller Gross Profit]),
   iif (ancestor([Sales Territory].[Sales Territory].CurrentMember,
                 [Sales Territory].[Sales Territory].[Group])
        IS [Sales Territory].[Sales Territory].[Group].&[Pacific],
        NULL,
        Sum({[Date].[Day of Week].&[6],
             [Date].[Day of Week].&[7]},
             [Measures].[Reseller Gross Profit])))
```

Option 2—Scope assignment with a Calculated Member

In Option 2, you use a calculated member with a scope assignment to apply the conditional summing of Gross Profit.

```
CREATE MEMBER CURRENTCUBE.[MEASURES].[Option 2 Weekend Gross Profit] AS
     Sum({[Date].[Day of Week].&[7],
          [Date].[Day of Week].&[6] },
     [Measures].[Reseller Gross Profit]);

Scope ([Option 2 Weekend Gross Profit],
       Descendants([Sales Territory].[Sales Territory Group].&[North
       America]));
       This = Sum({[Date].[Day of Week].&[7],
                   [Date].[Day of Week].&[6],
                   [Date].[Day of Week].&[5] },
           [Measures].[Reseller Gross Profit]);
End Scope;

Scope ([Option 2 Weekend Gross Profit],
       Descendants([Sales Territory].[Sales Territory Group].&[Pacific]));
        This = NULL;
End Scope;
```

In this example, this option is significantly faster than the first option. The reason this option is faster is because the two scope subcube definitions (the left hand side of each scope assignment) enable the Query Execution Engine to know ahead of time the calculation space for each business rule. Using this information, the Query Execution Engine can select an optimized execution path to execute the calculation on the specified range of cells. As a general rule, it is a best practice to always try to simplify calculation expressions by moving the complex parts into multiple Scope definitions whenever possible.

Note that in this Scope assignment, the **Descendants** function represents the Analysis Services 2000 approach to using MDX functions to navigate a dimension hierarchy. An alternative approach in Analysis Services 2005 is to simply use Scope with the attribute hierarchies. So instead of using Scope (Descendants([Sales Territory].[Pacific])), you can use Scope ([Sales Territory].[Pacific]).

To enhance performance, wherever possible, use the scope subcube definition to narrowly define the calculation space. While this is the ideal scenario, there are situations where it is not possible. The most common scenario is when you need to define a calculation on an arbitrary collection of cells ([Measures].[Reseller Sales Amount] >500). This type of expression is not allowed in the scope subcube definition since the definition must be static. In this scenario, the solution is to define a broader calculation space in the scope definition, and then use the scope MDX expression (the right hand side of the scope assignment) to narrow down the cube space using the IIF statement. The following is an example of how this statement can be structured to apply a weighting factor to the North American sales:

```
SCOPE ([Measures].[Sales Amount],

       [Sales Territory].[Sales Territory].[Group].&[North America])

THIS =

   IIF ([Measures].[Sales Amount] > 1000, [Measures].[Sales Amount],
        [Measures].[Sales Amount]*1.2);
END SCOPE
```

From a performance perspective, MDX scope assignments provide an efficient alternative to using IIF to apply unique business rules to certain cells in a cube. Whenever you need to conditionally apply calculations, you should consider this approach.

Removing empty tuples

When you write MDX statements that use a set function such as **Crossjoin**, **Descendants**, or **Members**, the default behavior of the function is to return both empty and nonempty tuples. From a performance perspective, empty tuples can not only increase the number of rows and/or columns in your result set, but they can also increase query response time.

In many business scenarios, empty tuples can be removed from a set without sacrificing analysis capabilities. Analysis Services provides a variety of techniques to remove empty tuples depending on your design scenario.

Analysis Services ground rules for interpreting null values

Before describing these techniques, it is important to establish some ground rules about how empty values are interpreted in various design scenarios.

- **Missing fact table records**—Earlier in the white paper, it was stated that only records that are present in the relational fact table are stored in the partition. For example, a sales fact table only stores records for those customers with sales for a particular product. If a customer never purchased a product, a fact table record will not exist for that customer and product combination. The same holds true for the Analysis Services partition. If a fact table record does not exist for a particular combination of dimension members, the cube cells for these dimension members are considered empty.
- **Null values in measures**—For each fact table measure that is loaded into the partition, you can decide how Analysis Services interprets null values. Consider the following example. Your sales fact table contains a record that has a sales amount of 1000 and a discount amount of null. When discount is loaded into the cube, by default it is interpreted as a zero, which means that it is not considered empty. How Analysis Services interprets null values is controlled by a property called **NullProcessing**. The **NullProcessing** property is set on a measure-by-measure basis. By default, it is set to Automatic which means that Analysis Services converts the null values to zero. If you want to preserve the null value from the source system, such as in the example of the discount measure, configure the **NullProcessingProperty** of that measure to **Preserve** instead of **Automatic**.
- **Null values in calculations**—In calculations, it is important to understand how Nulls are evaluated. For example, 1 minus Null equals 1, not Null. In this example, the Null is treated like a zero for calculation purposes. which may or may not be what you want to use. To explicitly test whether a tuple is null, use the ISEmpty function within an IIF statement to conditionally handle empty tuples.

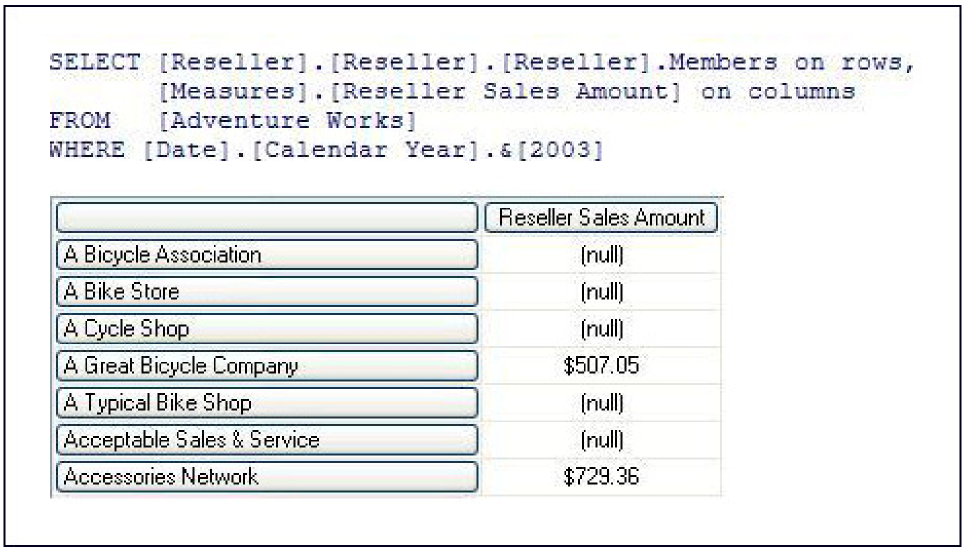
- **Empty members**—When writing calculations that reference dimension members, you may need to handle scenarios where specific members do not exist, such as the parent of the All. In this scenario, the ISEmpty function is not appropriate as it tests for empty cells. Rather in this scenario you want to use the IS operator to test whether the member IS NULL.

General techniques for removing empty tuples

The following describes the most general techniques for removing empty tuples:

- **Non Empty Keyword**—When you want to remove empty rows or columns from the axes of an MDX query, you can use the NON EMPTY keyword. Most client applications use the NON EMPTY keyword to remove empty cells in query result sets. The NON EMPTY keyword is applied to an axis and takes effect after the query result set is determined, i.e., after the Query Execution Engine completes the axis tuples with the current members taken from the rows axis, columns axis, and WHERE clause (as well as the default members of the attribute hierarchies not referenced in the query).

  Consider the following example displayed in Figure 18. Note that only a subset of the query results is shown.



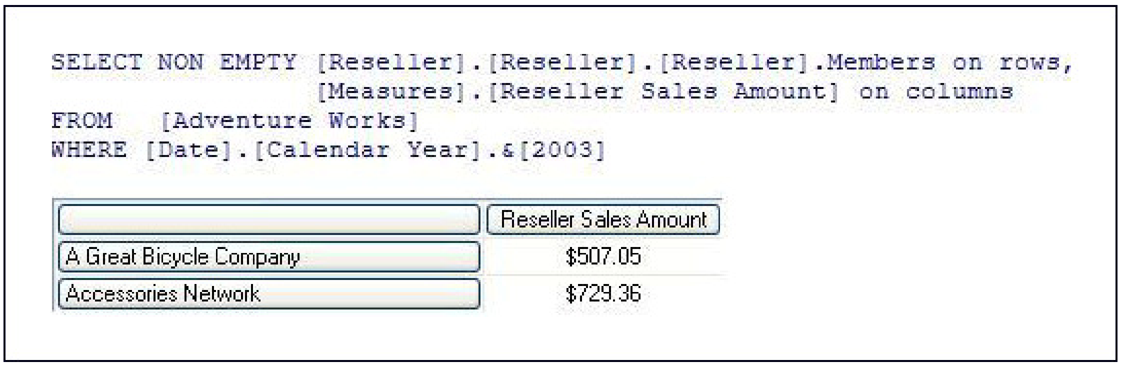
```
SELECT [Reseller].[Reseller].[Reseller].Members on rows,
       [Measures].[Reseller Sales Amount] on columns
FROM   [Adventure Works]
WHERE [Date].[Calendar Year].&[2003]
```

	Reseller Sales Amount
A Bicycle Association	(null)
A Bike Store	(null)
A Cycle Shop	(null)
A Great Bicycle Company	$507.05
A Typical Bike Shop	(null)
Acceptable Sales & Service	(null)
Accessories Network	$729.36

Figure 18 – Query without Non Empty Keyword

In Figure 18, the rows axis returns a complete list of resellers, regardless of whether or not they have sales in 2003. For each reseller, the columns axis displays the reseller's 2003 sales. If the reseller had no sales in 2003, the reseller is still returned in the list, it is just returned with a (null) value.

To remove resellers who do not have sales in 2003, you can use the NON EMPTY keyword as displayed in Figure 19. Note that only a subset of the query results is shown.



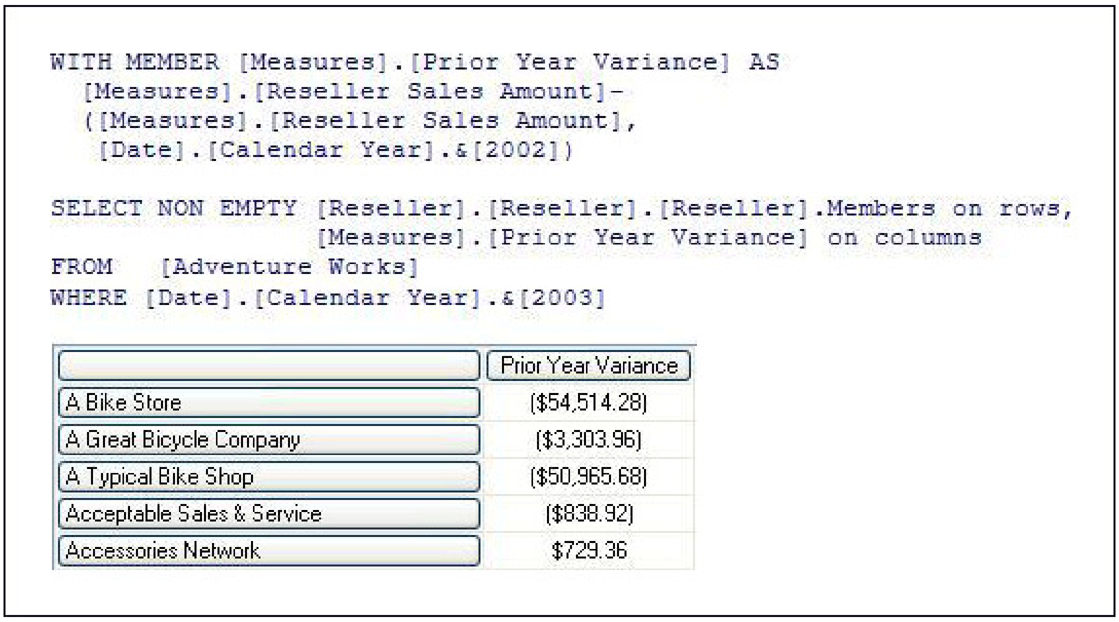
```
SELECT NON EMPTY [Reseller].[Reseller].[Reseller].Members on rows,
                 [Measures].[Reseller Sales Amount] on columns
FROM   [Adventure Works]
WHERE [Date].[Calendar Year].&[2003]
```

	Reseller Sales Amount
A Great Bicycle Company	$507.05
Accessories Network	$729.36

Figure 19 – Query with Non Empty Keyword

In Figure 19, the NON EMPTY keyword removes the resellers that do not have sales, i.e., null sales for 2003. To apply the NON EMPTY keyword, the Query Execution Engine must completely evaluate all cells in the query before it can remove the empty tuples.

If the query references a calculated member, the Query Execution Engine must evaluate the calculated member for all cells in the query and then remove the empty cells. Consider the example displayed in Figure 20. Note that only a subset of the query results is shown. In this example, you have modified the reseller query, replacing Reseller Sales Amount with the calculated measure Prior Year Variance. To produce the query result set, Analysis Services first obtains a complete set of resellers and then removes those resellers that have an empty prior year variance.



```
WITH MEMBER [Measures].[Prior Year Variance] AS
  [Measures].[Reseller Sales Amount]-
  ([Measures].[Reseller Sales Amount],
   [Date].[Calendar Year].&[2002])

SELECT NON EMPTY [Reseller].[Reseller].[Reseller].Members on rows,
                 [Measures].[Prior Year Variance] on columns
FROM   [Adventure Works]
WHERE [Date].[Calendar Year].&[2003]
```

	Prior Year Variance
A Bike Store	($54,514.28)
A Great Bicycle Company	($3,303.96)
A Typical Bike Shop	($50,965.68)
Acceptable Sales & Service	($838.92)
Accessories Network	$729.36

Figure 20 – Query with Non Empty Keyword and Calculated Measure

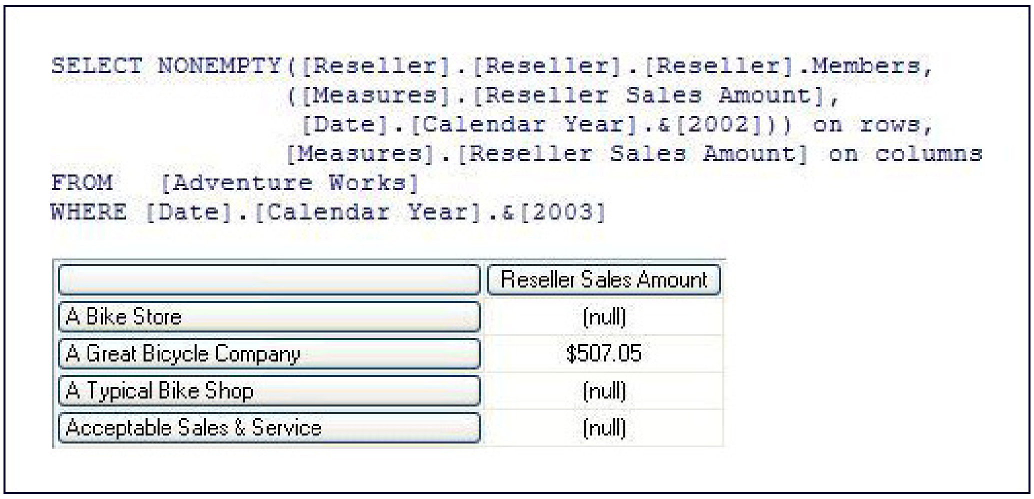
Given the ground rules about null interpretation, it is necessary to point out when the prior year variance is going to be null.

    - Acceptable Sales & Service has no sales in 2003 but has sales of $838.92 in 2002. The Prior Year Variance calculation is null minus $838.92 which evaluates to -$838.92. Since the value is not null, that reseller is returned in the result set.
    - Accessories Network has $729.36 sales in 2003 but no sales in 2002. The Prior Year Variance calculation is $729.36 minus null which evaluates to $729.36. Since the value is not null, the reseller is returned in the result set.
    - If a reseller has no sales in either 2003 or 2002, the calculation is null minus null, which evaluates to null. The reseller is removed from the result set.

- **NonEmpty Function**—The **NonEmpty** function, **NonEmpty()**, is similar to the NON EMPTY keyword but it provides additional flexibility and more granular control. While the NON EMPTY keyword can only be applied to an axis, the **NonEmpty** function can be applied to a set. This is especially useful when you are writing MDX calculations.

  As an additional benefit, the **NonEmpty** function allows you use an MDX expression to evaluate the empty condition against a business rule. The business rule can reference any tuple, including calculated members. Note that if you do not specify the MDX expression in the **NonEmpty** function, the **NonEmpty** function behaves just like the NON EMPTY keyword, and the empty condition is evaluated according to the context of the query.

  Continuing with the previous examples regarding resellers, you want to change the previous queries to apply your own business rule that decides whether or not a reseller is returned in the query. The business rule is as follows. You want a list of all resellers that had a sale in 2002. For each of these resellers, you only want to display their sales for 2003. To satisfy this requirement, you use the query displayed in Figure 21. Note that only a subset of the query results is shown.



```
SELECT NONEMPTY([Reseller].[Reseller].[Reseller].Members,
               ([Measures].[Reseller Sales Amount],
                [Date].[Calendar Year].&[2002])) on rows,
               [Measures].[Reseller Sales Amount] on columns
FROM   [Adventure Works]
WHERE [Date].[Calendar Year].&[2003]
```

	Reseller Sales Amount
A Bike Store	(null)
A Great Bicycle Company	$507.05
A Typical Bike Shop	(null)
Acceptable Sales & Service	(null)

Figure 21 Query with NonEmpty Function

In this query, the **NonEmpty** function returns those resellers that had a sale in 2002. For each of these resellers, their 2003 sales are displayed. Note that query produces a different list of resellers than the resellers returned in the NON EMPTY keyword example. The Accessories Network reseller has been removed because it only has sales in 2003 with no sales in 2002.

Note that an alternative way to write this query is to use the FILTER expression, such as the following:

```
SELECT FILTER ([Reseller].[Reseller].[Reseller].Members,
       NOT ISEmpty(([Measures].[Reseller Sales Amount],
                    [Date].[Calendar Year].&[2002]))) on rows,
       [Measures].[Reseller Sales Amount] on columns
FROM   [Adventure Works]
WHERE [Date].[Calendar Year].&[2003]
```

In this query, FILTER is used to return only those resellers who had a sale in 2002. FILTER was commonly used in prior versions of Analysis Services. For simple expressions like the one depicted in the above example, Analysis Services actually re-writes the query behind-the-scenes using the **NonEmpty** function. For other more complicated expressions, it is advisable to use the **NonEmpty** function in place of the FILTER expression. In Analysis Services 2005, the **NonEmpty** function provides a more optimized alternative to using the FILTER expression to check for empty cells.

As you use NON EMPTY keyword and **NonEmpty** function to remove empty tuples, consider the following guidelines:

- The **NonEmpty** function and NON EMPTY keyword will have approximately the same performance when the parameters passed to **NonEmpty** function coincide with the query axes of a NON EMPTY keyword query.
- In common scenarios, **Non Empty** is normally used instead of **NonEmpty()** at the top level of SELECT query axes.
- In calculations or query sub expressions, **NonEmpty()** is the recommended approach to achieve similar semantics. When using **NonEmpty()**, pay extra attention to making sure that the current cell context used by **NonEmpty()** is the intended one, possibly by specifying additional sets and members as the second parameter of the function.

- **Non_Empty_Behavior (NEB)**—Whether or not an expression resolves to null is important for two major reasons. First, most client applications use the NON EMPTY key word in a query. If you can tell the Query Execution Engine that you know an expression will evaluate to null, it does not need to be computed and can be eliminated from the query results before the expression is evaluated. Second, the Query Execution Engine can use the knowledge of a calculation's Non_Empty_Behavior (NEB) even when the NON EMPTY keyword is not used. If a cell's expression evaluates to null, it does not have to be computed during query evaluation.

  Note that the current distinction between how the Query Execution Engine uses an expression's NEB is really an artifact of the Query Execution Engine design. This distinction indicates whether one or both optimizations are used, depending how the NEB calculation property is defined. The first optimization is called NonEmpty Optimization and the second optimization is called Query Execution Engine Optimization.

  When an expression's NEB is defined, the author is guaranteeing that the result is null when the NEB is null and consequently the result set is not null when NEB is not null. This information is used internally by the Query Execution Engine to build the query plan.

To use NEB for a given calculation, you provide an expression that defines the conditions under which the calculation is guaranteed to be empty. The reason why NEB is an advanced setting is because it is often difficult to correctly identify the conditions under which the calculation is guaranteed to be empty. If you incorrectly set NEB, you will receive incorrect calculation results. As such, the primary consideration of using NEB is to ensure first and foremost that you have defined the correct expression, before taking into account any performance goals.

The NEB expression can be a fact table measure, a list of two or more fact table measures, a tuple, or a single measure set. To help you better understand which optimizations are used for each expression; consider the guidelines in Table 2.

Table 2 NEB guidelines

Calculation Type	NEB Expressions	Query Execution Engine Optimization Support	NonEmpty Optimization Support	Example
Calculated Measure	Constant measure	Yes	Yes	With Member Measures.DollarSales As Measures.Sales / Measures.ExchangeRate, NEB = Measures.Sales
Calculated Measure	(List of two or more constant measure references)	No	Yes	With Member Measures.Profit As Measures.Sales – Measures.Cost, NEB = {Measures.Sales, Measures.Cost}

Any (calculated member, script assignment, calculated cell)	Constant tuple reference
	Constant single-measure set
		Yes	No	Scope [Measures].[store cost];
				This = iif([Measures].[Exchange Rate]>0, [Measures].[Store Cost]/[Measures].[Exchange Rate], null);
				
				Non_Empty_Behavior(This) = [Measures].[Store Cost];
				End Scope;

In addition to understanding the guidelines for the NEB expression, it is important to consider how the expression is applied for various types of calculation operations.

- **Scenario 1—Addition or Subtraction:** Example - Measures.M1 + or – Measures.M2. The following guidelines apply for addition and subtraction:
  - In general, you MUST specify both measures in the NEB expression for correctness reasons.

  - In particular, if both measures belong to the same measure group, it may be possible to specify just one of them in NEB expression if the data supports it. This could result in better performance.
- **Scenario 2—Multiplication.** Example - Measures.M1 * Measures.M2. The following guidelines apply for multiplication:
  - In general, you CANNOT specify any correct NEB expression for this calculation.
  - In particular, if it is guaranteed that one of the measures is never null (e.g., a currency exchange rate), you MAY specify the other measure in NEB expression.
  - In particular, if it is guaranteed that, for any given cell, either both measures are null, or both are non-null (e.g. they belong to the same measure group), you MAY specify both measures in the NEB expression, OR specify a single measure.
- **Scenario 3—Division.** Example - Measures.M1 / Measures.M2. In this scenario, you MUST specify the first measure (the numerator, M1) in NEB.

Evaluating empty by using a measure group

In some design scenarios, you may be able to optimize the removal of empty tuples by evaluating the empty condition against an entire measure group. In other words, if a tuple corresponds to a fact data record in the measure group, the tuple is included. If the tuple does not have a fact data record, it is excluded. To apply this syntax, you can use a special version of the **Exists** function, `Exists (Set,, “Measure Group”)`. Note that this special version of the **Exists** function actually behaves very differently from the regular **Exists** function and includes a third parameter of type **string** where you can specify the name of the desired measure group.

While this approach can be very powerful in removing empty tuples, you must evaluate whether it generates the correct result set. Consider the following example. The sales fact table contains a record corresponding to a reseller’s purchase. In this record, Sales Amount has a value of 500 while Discount Amount is null. To write a query where you only see the resellers with discounts, you cannot use this approach since the reseller still exists in measure group whether or not the reseller’s Discount Amount is null. To satisfy this query, you will need to use the NON EMPTY keyword or **NonEmpty** function as long as you have properly configured the **Null Processing** property for the measures in that measure group.

When you do have scenarios where you can apply the **Exists** function, keep in mind that the **Exists** function ignores all calculated members. In addition, note that the **Exists** function used with a measure group specification replaces the deprecated **NonEmptyCrossJoin** function, which was used in prior versions of Analysis Services to achieve similar functionality.

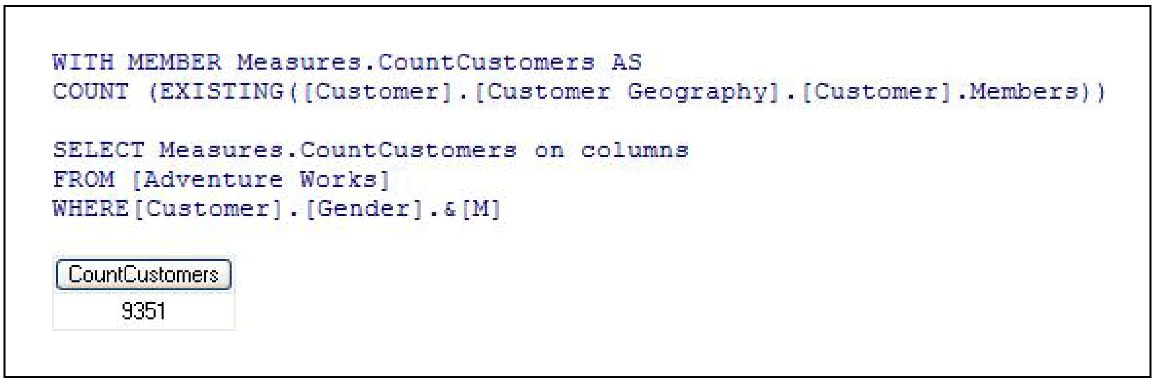
Removing empty member combinations

Analysis Services 2005 provides a rich attribute architecture that allows you to analyze data across multiple attributes at a given time. When you write queries that involve multiple attributes, there are some optimizations that you should be aware of to ensure that the queries are efficiently evaluated.

- **Autoexists—Autoexists** is applied behind the scenes whenever you use the **Crossjoin** function to cross join two attribute hierarchies from the same dimension or, more broadly speaking, whenever you **Crossjoin** sets with common dimensionality. **Autoexists** retains only the members that exist with each other so that you do not see empty member combinations that never occur such as (Seattle, Scotland). In some design scenarios, you may have a choice as to whether you create two attributes in a given dimension or model them as two separate dimensions. For example, in an employee dimension you may consider whether or not you should include department in that dimension or add department as another dimension. If you include department in the same dimension, you can take advantage of Autoexists to remove empty combinations of employees and departments.

- **Exists** function—Using the **Exists** function in the form of `Exists (Set1, Set2)`, you can remove tuples from one set that do not exist in another set by taking advantage of **Autoexists**. For example, `Exists (Customer.Customer.Members, Customer.Gender.Male)` only returns male customers.
- **EXISTING operator**—The EXISTING operator is similar to the **Exists** function but it uses as the filter set, the current coordinate specified in your MDX query. Since it uses the current coordinate, it reflects whatever you are slicing by in your query.

In Figure 22, a calculated measure counts a set of customers defined by the EXISTING operator and the WHERE clause of the query slices on Male customers. The result set of this query is a total count of male customers.



```
WITH MEMBER Measures.CountCustomers AS
COUNT (EXISTING([Customer].[Customer Geography].[Customer].Members))

SELECT Measures.CountCustomers on columns
FROM [Adventure Works]
WHERE[Customer].[Gender].&[M]
```

CountCustomers
9351

Figure 22 Calculated Measure Using Existing Operator

Summarizing data with MDX

Analysis Services naturally aggregates measures across the attributes of each dimension. While measures provide tremendous analytical value, you may encounter scenarios when you want to perform additional aggregations of data, either aggregating MDX calculations or aggregating subsets of the cube that satisfy specific business rules.

Consider the following examples where you may need to aggregate data in MDX:

- **Performing time series analysis**—You need to sum reseller profit for all time periods in this year up to and include the current time period.
- **Aggregating custom sets**—You need to average year-over-year sales variance across all resellers in the USA who sold more than 10,000 units.
- **Aggregating calculations at dimension leaves**—You need to aggregate the hours worked by each employee multiplied by the employee’s hourly rate.

While using MDX calculated measures with functions like **Sum**, **Average**, and **Aggregate** is a valid approach to summarizing data in these scenarios, from a performance perspective, summarizing data through MDX is not a trivial task and can potentially result in slow performance in large-scale cubes or cubes with many nested MDX calculations. If you experience performance issues summarizing data in MDX, you may want to consider the following design alternatives:

Create a named calculation in the data source view

Depending on the scenario, you should first consider whether you need to perform additional aggregations in MDX or whether you can take advantage of the natural aggregation of the cube. Consider the Profit Margin calculation. Profit is defined by Revenue minus Cost. Assuming that these two measures are stored in the same fact table, instead of defining a calculated member that is calculated on the fly, you can move the Profit calculation to a measure. In the data source view you can create a named calculation on the fact table that defines Profit as Revenue minus Cost. Then, you can add the named calculation as a measure in your cube to be aggregated just like every other measure.

Generally speaking, any time that you have a calculated member that performs addition and subtraction from columns on the same fact table, you can move the calculation to measure. If you do this, keep in mind that SQL operations on NULL data are not identical to MDX. In addition, even though you can add the calculation to the fact table, you must also evaluate the impact of additional measures on the cube. Whenever you query a measure group, even if you only request one measure, all measures in that measure group are retrieved from the Storage Engine and loaded into the data cache. The more measures you have, the greater the resource demands. Therefore, it is important to evaluate the performance benefits on a case by case basis.

Use measure expressions

Measure expressions are calculations that the Storage Engine can perform. Using measure expressions, you can multiply or divide data from two different measure groups at the measure group leaves and then aggregate the data as a part of normal cube processing. The classic example of this is when you have a weighting factor stored in one measure group, such as currency rates, and you want to apply that to another measure group such as sales. Instead of creating an MDX calculation that aggregates the multiplication of these two measures at the measure group leaves, you can use measure expressions as an optimized solution. This kind of calculation is perfect for a measure expression since it is somewhat more difficult to accomplish in the data source view given that the measures are from different source fact tables and likely have different granularities. To perform the calculation in the data source view, you can use a named query to join the tables; however, the measure expression typically provides a more efficient solution.

While measure expressions can prove to be very useful, note that when you use a measure expression, the Storage Engine evaluates the expression in isolation of the Query Execution Engine. If any of the measures involved in the measure expression depend on MDX calculations, the Storage Engine evaluates the measure expressions without any regard for the MDX calculations, producing an incorrect result set. In this case, rather than using a measure expression, you can use a calculated member or scope assignment.

Use semiadditive measures and unary operators

Instead of writing complex MDX calculations to handle semiadditive measures, you can use the semiadditive aggregate functions like **FirstChild**, **LastChild**, etc. Note that semiadditive functions are a feature of SQL Server Enterprise Edition. In addition to using semiadditive measures, in finance applications, instead of writing complicated MDX expressions that apply custom aggregation logic to individual accounts, you can use unary operators with parent-child hierarchies to apply a custom roll up operator for each account. Note that while parent-child hierarchies are restricted in their aggregation design, they can be faster and less complex than custom MDX. For more information on parent-child hierarchies, see Parent-child hierarchies in this white paper.

Move numeric attributes to measures

You can convert numeric attributes to measures whenever you have an attribute such as Population or Salary that you need to aggregate. Instead of writing MDX expressions to aggregate these values, consider defining a separate measure group on the dimension table containing the attribute and then defining a measure on the attribute column. So for example, you can replace Sum(Customer.City.Members, Customer.Population.MemberValue) by adding a new measure group on the dimension table with a Sum measure on the Population column.

Aggregate subsets of data

In many scenarios, you want to aggregate subsets of data that meet specific business rules. Before you write a complex filter statement to identify the desired set, evaluate whether you can substitute a filter expression by using **Crossjoin** or **Exists** with specific members of your attribute hierarchies.

Consider the following examples.

- If you want a set of resellers with 81 to 100 employees and you have the number of employees stored in a separate attribute, you can easily satisfy this request with the following syntax:

  ```
  Exists([Reseller].[Reseller].members,

          [Reseller].[Number of Employees].[81]:

          [Reseller].[Number of Employees].[100])
  ```

  In this example, you use **Exists** with a range for the Number of Employees attribute hierarchy to filter resellers in the 81–100 employee range. The value of this approach is that you can arbitrarily set the ranges based on user requests.

- If your reseller dimension is large and the ranges that you need to calculate are fixed and commonly used across all users, you can pre-build an attribute that groups the resellers according to employee size ranges. With this new attribute, the previous statement could be written as follows:

  ```
  Exists([Reseller].[Reseller].members,

          [Reseller].[Employee Size Range].[81 to 100])
  ```

- Now if you want the sum of the gross profit for resellers with 81 to 100 employees, you can satisfy this request with the following solutions.

  For the range of values, you can use the following syntax.

```
Sum([Reseller].[Number of Employees].[81]:
    [Reseller].[Number of Employees].[100],
 [Measures].[Reseller Gross Profit])
```

For the custom range attribute, you can use the following syntax.

```
([Reseller].[Number of Employees].[81 - 100],
 [Measures].[Reseller Gross Profit])
```

Note that in both of these solutions, the set of resellers is not necessary.

Using an attribute hierarchy to slice data sets provides a superior solution to aggregating a set filtered on member property values, which was a common practice in prior versions of Analysis Services. Retrieving a member's properties can be slow since each member needs to be retrieved as well as its property value. With attribute hierarchies, you can leverage the normal aggregation of the cube.

Taking advantage of the Query Execution Engine cache

During the execution of an MDX query, the Query Execution Engine stores calculation results in the Query Execution Engine cache. The primary benefits of the cache are to optimize the evaluation of calculations and to support the re-usage of calculation results across users. To understand how the Query Execution Engine uses calculation 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 Execution Engine caches the Profit Margin values for each Sales Territory, assuming that the Sales Territory has a nonempty Profit Margin. To manage the re-usage of the cached results across users, the Query Execution Engine uses scopes. Note that Query Execution scopes should not be confused with the SCOPE keyword in an MDX script. Each Query Execution Engine scope maintains its own cache and has the following characteristics.

- **Query Scope**—The query scope contains any calculations created within a query by using the WITH keyword. The query scope is created on demand and terminates when the query is over. Therefore, the cache of the query scope is not shared across queries in a session.
- **Session Scope**—The session scope contains calculations created in a given session by using the CREATE statement. The cache of the session scope is reused from request to request in the same session, but is not shared across sessions.
- **Global Scope**—The global scope contains the MDX script, unary operators, and custom rollups for a given set of security permissions. The cache of the global scope can be shared across sessions if the sessions share the same security roles.

The scopes are tiered in terms of their level of re-usage. The query scope is considered to be the lowest scope, because it has no potential for re-usage. The global scope is considered to be the highest scope, because it has the greatest potential for re-usage.

During execution, every MDX query must reference all three scopes to identify all of the potential calculations and security conditions that can impact the evaluation of the query. For example, if you have a query that contains a query calculated member, to resolve the query, the Query Execution Engine creates a query scope to resolve the query calculated member, creates a session scope to evaluate session calculations, and creates a global scope to evaluate the MDX script and retrieve the security permissions of the user who submitted the query. Note that these scopes are created only if they aren't already built. Once they are built for a session, they are usually just re-used for subsequent queries to that cube.

Even though a query references all three scopes, it can only use the cache of a single scope. This means that on a per-query basis, the Query Execution Engine must select which cache to use. The Query Execution Engine always attempts to use the cache of the highest possible scope depending on whether or not the Query Execution Engine detects the presence of calculations at a lower scope.

If the Query Execution Engine detects any calculations that are created at the query scope, it always uses the query scope cache, even if a query references calculations from the global scope. If there are no query-scope calculations, but there are session-scope calculations, the Query Execution Engine uses the cache of session scope. It does not matter whether or not the session calculations are actually used in a query. The Query Execution Engine 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 scoped calculations, the global cache is not used, even if you do not specifically use the session calculation in a given query.

There are other calculation scenarios that impact how the Query Execution Engine 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. This means that there is no guarantee of what the stored procedure will return. As a result, nothing will be cached globally or in the session cache. Rather, the calculations will only be stored in the query cache. In addition, the following scenarios determine how the Query Execution Engine caches calculation results:

- If you enable visual totals for the session by setting the default MDX **Visual Mode** property of the Analysis Services connection string to 1, the Query Execution Engine 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 Execution Engine uses the query cache.
- Queries that use the subselect syntax (SELECT FROM SELECT) or are based on a session subcube (CREATE SUBCUBE) could cause the query cache to be used.
- Arbitrary shape sets can only use the query cache when they are used in a subselect, in the WHERE clause, or in a calculated member **Aggregate** expression referenced in the WHERE clause. An example of arbitrary shape set is a set of multiple members from different levels of a parent-child hierarchy.

Based on this behavior, when your querying workload can benefit from re-using 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 a 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 a many security roles or you are using dynamic security, the opportunity to re-use calculation results across users is lessened and the performance benefits associated with re-using the Query Execution Engine cache are not as high.

Applying calculation best practices

While many MDX recommendations must be evaluated in the context of a design scenario, the following best practices are optimization techniques that apply to most MDX calculations regardless of the scenario.

Use the Format String property

Instead of applying conditional logic to return customized values if the cell is EMPTY or 0, use the **Format String** property. The **Format String** property provides a mechanism to format the value of a cell. You can specify a user-defined formatting expression for positive values, negative values, zeros, and nulls. The **Format String** display property has considerably less overhead than writing a calculation or assignment that must invoke the Query Execution Engine. Keep in mind that your front-end tool must support this property.

Avoid late-binding functions

When writing MDX calculations against large data sets involving multiple iterations, avoid referencing late binding functions whose metadata cannot be evaluated until run time. Examples of these functions include: **LinkMember**, **StrToSet**, **StrToMember**, **StrToValue**, and **LookupCube**. Because they are evaluated at run time, the Query Execution Engine cannot select the most efficient execution path.

Eliminate redundancy

When you use a function that has default arguments such as Time.CurrentMember, you can experience performance benefits if you do not redundantly specify the default argument. For example, use PeriodsToDate([Date].[Calendar].[Calendar Year]) instead of PeriodsToDate([Date].[Calendar].[Calendar Year], [Date].Calendar.CurrentMember). To take advantage of this benefit, you must ensure that you only have one default Time Hierarchy in your application. Otherwise, you must explicitly specify the member in your calculation.

Ordering expression arguments

When writing calculation expressions like "expr1 * expr2", make sure the expression sweeping the largest area/volume in the cube space and having the most Empty (Null) values is on the left side. For instance, write "Sales * ExchangeRate" instead of "ExchangeRate * Sales", and "Sales * 1.15" instead of "1.15 * Sales". This is because the Query Execution Engine iterates the first expression over the second expression. The smaller the area in the second expression, the fewer iterations the Query Execution Engine needs to perform, and the faster the performance.

Use IS

When you need to check the value of a member, use IIF [Customer].[Company] IS [Microsoft] and not IIF [Customer].[Company].Name = "Microsoft". The reason that IS is faster is because the Query Execution Engine does not need to spend extra time translating members into strings.

Tuning Processing Performance

Processing is the general operation that loads data from one or more data sources into one or more Analysis Services objects. While OLAP systems are not generally judged by how fast they process data, processing performance impacts how quickly new data is available for querying. While every application has different data refresh requirements, ranging from monthly updates to "near real-time" data refreshes, the faster the processing performance, the sooner users can query refreshed data.

Note that "near real-time" data processing is considered to be a special design scenario that has its own set of performance tuning techniques. For more information on this topic, see Near real-time data refreshes.

To help you effectively satisfy your data refresh requirements, the following provides an overview of the processing performance topics that are discussed in this section:

Understanding the processing architecture – For readers unfamiliar with the processing architecture of Analysis Services, this section provides an overview of processing jobs and how they apply to dimensions and partitions. Optimizing processing performance requires understanding how these jobs are created, used, and managed during the refresh of Analysis Services objects.

Refreshing dimensions efficiently – 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: optimizing SQL source queries, reducing attribute overhead, and preparing each dimension attribute to efficiently handle inserts, updates, deletes as necessary.

Refreshing partitions efficiently – 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, using multiple partitions, effectively handling data inserts, updates, and deletes, and evaluating the usage of rigid vs. flexible aggregations.

Understanding the processing architecture

Processing is typically described in the simple terms of loading data from one or more data sources into one or more Analysis Services objects. While this is generally true, Analysis Services provides the ability to perform a broad range of processing operations to satisfy the data refresh requirements of various server environments and data configurations.

Processing job overview

To manage processing operations, Analysis Services uses centrally controlled jobs. A processing job is a generic unit of work generated by a processing request. Note that while jobs are a core component of the processing architecture, jobs are not only used during processing. For more information on how jobs are used during querying, see Job architecture.

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 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 a **ProcessFull**, **ProcessUpdate**, or **ProcessIndexes**. For example, when you issue a **ProcessFull** 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** 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.

Dimension processing jobs

During the processing of MOLAP dimensions, jobs are used to extract, index, and persist data in a series of dimension stores. For more information on the structure and content of the dimension stores, see Data retrieval: dimensions. To create these dimension stores, the Storage Engine uses the series of jobs displayed in Figure 23.

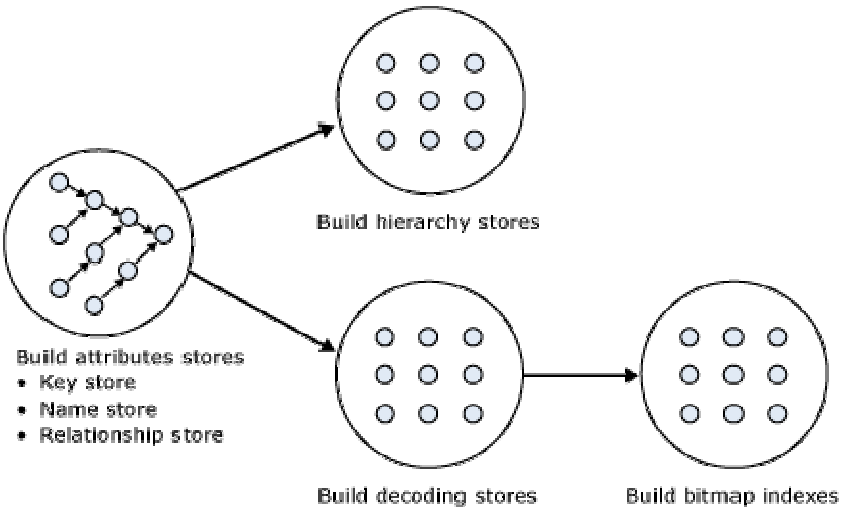


Figure 23 Dimension processing jobs

Build attribute stores

For each attribute in a dimension, a job is instantiated to extract and persist the attribute members into an attribute store. As stated earlier, the attribute store primarily consists of the key store, name store, and relationship store. While Analysis Services is capable of processing multiple attributes in parallel, it requires that an order of operations must be maintained. The order of operations is determined by the attribute relationships in the dimension. The relationship store defines the attribute’s relationships to other attributes. In order for this store to be built correctly, for a given attribute, all dependent attributes must already be processed before its relationship store is built. To provide the correct workflow, the Storage Engine analyzes the attribute relationships in the dimension, assesses the dependencies among the attributes, and then creates an execution tree that indicates the order in which attributes can be processed, including those attributes that can be processed in parallel.

Figure 24 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 that the dimension has been configured using cascading attribute relationships which is a best practice for all dimension designs.

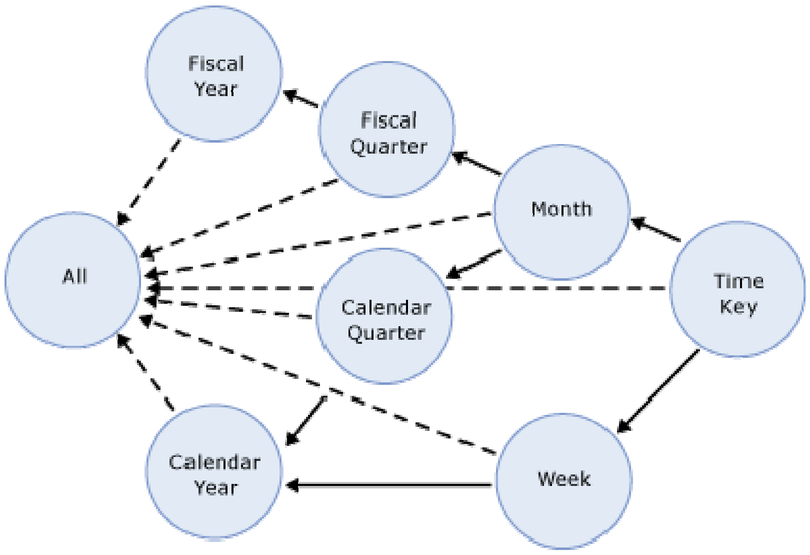


Figure 24 Execution tree example

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 primary key attribute always being processed last since 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 since it has the most members and all other jobs (hierarchy, decoding, bitmap indexes) are waiting for it to complete. For more information about the importance of using cascading attribute relationships, see Identifying attribute relationships.

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. For more information on hierarchy stores, see Data retrieval: dimensions.

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 DataIDs, 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. For more information on this design scenario, see Reducing attribute overhead.

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:

- A **ProcessFull** command discards all storage contents of the dimension and rebuilds them. Behind the scenes, **ProcessFull** executes all dimension processing jobs and performs an implicit **ProcessClear** to discard the storage contents of all dependent partitions. This means that whenever you perform a **ProcessFull** of a dimension, you need to perform a **ProcessFull** on dependent partitions to bring the cube back online.
- **ProcessData** discards all storage contents of the dimension and rebuilds only the attribute and hierarchy stores. **ProcessData** is a component of the **ProcessFull** operation. **ProcessData** also clears partitions.
- **ProcessIndexes** requires that a dimension already has attribute and hierarchy stores built. **ProcessIndexes** preserves the data in these stores and then rebuilds the bitmap indexes. **ProcessIndexes** is a component of the **ProcessFull** operation.
- **ProcessUpdate** does not discard the dimension storage contents, unlike **ProcessFull**. Rather, 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. A **ProcessUpdate** can handle inserts, updates, and deletions depending on the type of attribute relationships (rigid vs. flexible) in the dimension. Note that **ProcessUpdate** will drop invalid aggregations and indexes, requiring you to take action to rebuild the aggregations in order to maintain query performance. For more information on applying **ProcessUpdate**, see Evaluating rigid vs. flexible aggregations.
- **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 flexible aggregations.

For a more comprehensive list of processing commands, see the Analysis Services 2005 Processing Architecture white paper located on the Microsoft Developer Network (MSDN).

Partition-processing jobs

During partition processing, source data is extracted and stored on disk using the series of jobs displayed in Figure 25.

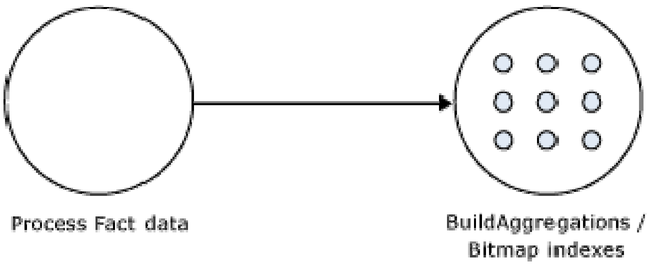


Figure 25 Partition processing jobs

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.

During the processing of fact data, a potential bottleneck may be the source SQL statement. For techniques to optimize the source SQL statement, see Optimizing the source query.

Build aggregations and bitmap indexes

Aggregations are built in memory during processing. While too few aggregations may have little impact on query performance, excessive aggregations can increase processing time without much added value on query performance. As a result, care must be taken to ensure that your aggregation design supports your required processing window. For more information on deciding which aggregations to build, see Adopting an aggregation design strategy.

If aggregations do not fit in memory, chunks are written to temp files and merged at the end of the process. Bitmap indexes are built on the fact and aggregation data and written to disk on a segment by segment basis.

Partition-processing commands

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

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

- **ProcessFull** discards the storage contents of the partition and rebuilds them. Behind the scenes, a **ProcessFull** executes **ProcessData** and **ProcessIndexes** jobs.
- **ProcessData** discards the storage contents of the object and rebuilds only the fact data.

- **ProcessIndexes** requires that a partition already has its data built. **ProcessIndexes** preserves the data and any existing aggregations and bitmap indexes and creates any missing aggregations or bitmap indexes.
- **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. This command is exposed in Business Intelligence Development Studio and SQL Server Management Studio as **ProcessIncremental.**

For a more comprehensive list of processing commands, see Analysis Services 2005 Processing Architecture white paper on MSDN.

Executing processing jobs

To manage dependencies among jobs, the Analysis Services server organizes jobs into a processing schedule. Dimensions, for example, must always be processed first, given the inherent dependency of partitions on dimensions.

Jobs without dependencies can be executed in parallel as long as there are available system resources to carry out the jobs. For example, multiple dimensions can be processed in parallel, multiple measure groups can be processed in parallel, and multiple partitions within a measure group can be processed in parallel. Analysis Services performs as many operations as it can in parallel based on available resources and the values of the following three properties: the **CoordinatorExecutionMode** server property, and the **MaxParallel** processing command, and the **Threadpool\Process\MaxThreads** server property. For more information on these properties, see Maximize parallelism during processing in this white paper.

In addition, before executing a processing job, Analysis Services verifies the available memory. Each job requests a specific amount of memory from the Analysis Services memory governor. If there is not enough memory available to fulfill the memory request, the memory governor can block the job. This behavior can be especially relevant in memory-constrained environments when you issue a processing request that performs multiple intensive operations such as a **ProcessFull** on a large partition that contains a complex aggregation design. For more information on optimizing this scenario, see Tuning memory for partition processing.

In situations where you are performing querying and processing operations at the same time, a long-running query can block a processing operation and cause it to fail. When you encounter this, the processing operation unexpectedly cancels, returning a generic error message. During the execution of a query, Analysis Services takes a read database commit lock. During processing, Analysis Services requires a write database commit lock. The **ForceCommitTimeout** server property identifies the amount of time a process operation waits before killing any blocking read locks. Once the timeout threshold has been reached, all transactions holding the lock will fail. The default value for this property is 30,000 milliseconds (30 seconds). Note that this property can be modified in the msmdsrv.ini configuration file; however, it is generally not recommended that you modify this setting. Rather, it is simply important to understand the impact of long-running queries on concurrent processing to help you troubleshoot any unexpected processing failures.

Refreshing dimensions efficiently

As stated previously, 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. To accomplish this, you can perform the following techniques: optimize SQL source queries, reduce attribute overhead, and prepare each dimension attribute to efficiently handle inserts, updates, deletes, as necessary.

Optimizing the source query

During processing, you can optimize the extraction of dimension source data using the following techniques:

Use OLE DB Providers over .NET Data Providers

Since the Analysis Services runtime is written in native code, OLE DB Providers offer performance benefits over .NET Data Providers. When you use .NET Data Providers, data has to be marshaled between the .NET managed memory space and the native memory space. Since OLE DB Providers are already in native code, they provide significant performance benefits over.NET Data Providers and should be used whenever possible.

Use attribute relationships to 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 is used to merge the data streams. The OPENROWSET function provides a mechanism to access data from multiple data sources. For each attribute, a separate OPENROWSET-derived table is used. In this situation, the processing for the key attribute is extremely slow since it must access multiple OPENROWSET derived tables.

Tune the Processing Group property

When a dimension is processed, the default behavior is to issue a separate SQL statement that retrieves a distinct set of members for each attribute. This behavior is controlled by the **Processing Group** property, which is automatically set to **ByAttribute**. In most scenarios, **ByAttribute** provides the best processing performance; however, there are a few niche scenarios where it can be useful to change this property to **ByTable**. In **ByTable**, Analysis Services issues a single SQL statement on a per-table basis to extract a distinct set of all dimension attributes. This is potentially beneficial in scenarios where you need to process many high cardinality attributes and you are waiting a long time for the select distinct to complete for each attribute. Note that whenever **ByTable** is used, the server changes its processing behavior to use a multi-pass algorithm. A similar algorithm is also used for very large dimensions when the hash tables of all related attributes do not fit into memory. The algorithm can be very expensive in some scenarios because Analysis Services must read and store to disk all the data from the table in multiple data stores, and then iterate over it for each attribute. Therefore, whatever performance benefit you gain for quickly evaluating the SQL statement could be counteracted by the other processing steps. So while **ByTable** has the potential to be faster, it is only appropriate in scenarios where you believe that issuing one SQL statement performs significantly better than issuing multiple smaller SQL statements. In addition, note that if you use **ByTable**, you will see duplicate member messages when processing the dimension if you have configured the **KeyDuplicate** property to **ReportAndContinue** or **ReportAndStop**. In this scenario, these duplicate error messages are false positives resulting from the fact that the SQL statement is no longer returning a distinct set of members for each attribute. To better understand how these duplicates occur, consider the following example. You have a customer dimension table that has three attributes: customer key, customer name, and gender. If you set the **Processing Group** property to **ByTable**, one SQL statement is used to extract all three attributes. Given the granularly of the SQL statement, in this scenario, you will see duplicate error messages for the gender attribute if you have set the **KeyDuplicate** property to raise an error. Again, these error messages are likely false positives; however, you should still examine the messages to ensure that there are no unexpected duplicate values in your data.

Reducing 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. Once you have removed extraneous attributes, you can apply a series of techniques to optimize the processing of remaining attributes.

Use the KeyColumns and NameColumn properties effectively

When you add a new attribute to a dimension, two properties are used to define the attribute. The **KeyColumns** property specifies one or more source fields that uniquely identify each instance of the attribute and 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.

Analysis Services provides the ability to source the **KeyColumns** and **NameColumn** properties from different source columns. This is useful when you have a single entity like a product that is identified by two different attributes: a surrogate key and a descriptive product name. When users want to slice data by products, they may find that the surrogate key lacks business relevance and will choose to use the product name instead.

From a processing perspective, it is a best practice to assign a numeric source field to the **KeyColumns** property rather than a string property. Not only can this reduce processing time, in some scenarios it can also reduce the size of the dimension. This is especially true for attributes that have a large number of members, i.e., greater than 1 million members.

Rather than using a separate attribute to store a descriptive name, you can use the **NameColumn** property to display a descriptive field to end users. In the product example, this means you can assign the surrogate key to the **KeyColumns** property and use the product name to the **NameColumn** property. This eliminates the need for the extraneous name attribute, making your design more efficient to query and process.

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 querying time, the bitmap indexes for these attributes are not useful in speeding up retrieval since the Storage Engine still must sift through a large number of distinct values.

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. To avoid spending time building unnecessary bitmap indexes for the social security number attribute, it is possible to disable its bitmap indexes by setting the **AttributeHierarchyOptimizedState** property to **Not Optimized**.

Turn off the attribute hierarchy and use member properties

As an alternative to attribute hierarchies, member properties provide a different mechanism to expose dimension information. For a given attribute, member properties are automatically created for every 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, once 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 the same manner as attribute hierarchies and user hierarchies. To query a member property, you must query the properties of the attribute that contains the member property. For example, if you require the work phone number for a customer, you must query the properties of customer. As a convenience, most front-end tools easily display member properties in their user interfaces.

In general, querying member properties can be slower than querying attribute hierarchies because member properties are not indexed and do not participate in aggregations. The actual impact to query performance depends on how you are going to use the attribute. 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. However, if you are simply displaying the work phone number on a one-off basis for a particular customer and you are spending large amounts of time in processing, disabling the attribute hierarchy and using a member property provides a good alternative.

Optimizing dimension inserts, updates, and deletes

Dimension data refreshes can be generally handled via three processing operations:

- **ProcessFull**—Erases and rebuilds the dimension data and structure.
- **ProcessUpdate**—Implements inserts, updates, and deletes based on the types of attribute relationships in the dimension. Information on the different types of attribute relationships is included later in this section.
- **ProcessAdd**—Provides an optimized version of **ProcessUpdate** to only handle data insertions.

As you plan a dimension refresh, in addition to selecting a processing operation, you must also assess how each attribute and attribute relationship is expected to change over time. More specifically, every attribute relationship has a **Type** property that determines how the attribute relationship should respond to data changes. The **Type** property can either be set to flexible or rigid where flexible is the default setting for every attribute relationship.

Use flexible relationships

Flexible relationships permit you to make a variety of data changes without requiring you to use a **ProcessFull** to completely rebuild the dimension every time you make a change. Remember that as soon as you implement a **ProcessFull** on a dimension, the cube is taken offline and you must perform a **ProcessFull** on every dependent partition in order to restore the ability to query the cube.

For attributes with flexible relationships, inserts, updates, and deletions can be handled by using the **ProcessUpdate** command. The **ProcessUpdate** command allows you to keep the cube "online" while the data changes are made. By default, every attribute relationship is set to flexible, although it may not always be the best choice in every design scenario. Using flexible relationships is appropriate whenever you expect an attribute to have frequent data changes and you do not want to experience the impacts of performing a **ProcessFull** on the dimension and cube. For example, if you expect products to frequently change from one category to another, you may decide to keep the flexible relationship between product and category so that you only need to perform a **ProcessUpdate** to implement the data change.

The tradeoff to using flexible relationships is their impact on data aggregations. For more information on flexible aggregations, see Evaluating rigid vs. flexible aggregations.

Note that in some processing scenarios, flexible relationships can "hide" invalid data changes in your dimension. As stated in the Identifying attribute relationships section, whenever you use attribute relationships, you must verify that each attribute's **KeyColumns** property uniquely identifies each attribute member. If the **KeyColumns** property does not uniquely identify each member, duplicates encountered during processing are ignored by default, resulting in incorrect data rollups. More specifically, when Analysis Services encounters a duplicate member, it picks an arbitrary member depending on the processing technique that you have selected. If you perform a **ProcessFull**, it selects the first member it finds. If you perform a **ProcessUpdate**, it selects the last member it finds. To avoid this scenario, follow the best practice of always assigning the **KeyColumns** property with a column or combination of columns that unique identifies the attribute. If you follow this practice, you will not encounter this problem. In addition, it is a good practice to change the default error configuration to no longer ignore duplicates. To accomplish this, set the **KeyDuplicate** property from **IgnoreError** to **ReportAndContinue** or **ReportAndStop**. With this change, you can be alerted of any situation where duplicates are detected. However, this option may give you false positives in some cases (e.g., ByTable processing). For more information, see Optimizing the source query.

Use rigid relationships

For attributes with rigid relationships, inserts can be handled by using a **ProcessAdd** or **ProcessUpdate** to the dimension; however, updates and deletions require a **ProcessFull** of the dimension and consequently require a **ProcessFull** of the dependent partitions. As such, rigid relationships are most appropriate for attributes that have zero or infrequent updates or deletions. For example, in a time dimension you may assign a rigid relationship between month and quarter since the months that belong to a given quarter do not change.

If you want to assign a relationship as rigid (remember that the relationships are flexible by default), you must ensure that the source data supports the rigid relationship, i.e., no changes can be detected when you perform a **ProcessAdd** or **ProcessUpdate**. If Analysis Services detects a change, the dimension process fails and you must perform a **ProcessFull**. In addition, if you use rigid relationships, duplicate members are never tolerated for a given attribute, unlike in flexible relationships where an arbitrary member is selected during processing. Therefore, you must also ensure that your **KeyColumns** property is correctly configured.

Similar to flexible relationships, when you define rigid relationships, you must also understand their impact on data aggregations. For more information on rigid aggregations, see Evaluating rigid vs. flexible aggregations.

Refreshing partitions efficiently

The performance goal of partition processing is to refresh fact data and aggregations in an efficient manner that satisfies your overall data refresh requirements. To help you refresh your partitions, the following techniques are discussed in this topic: optimizing SQL source queries, using multiple partitions, effectively handling data inserts, updates, and deletes, and evaluating the usage of rigid vs. flexible aggregations.

Optimizing the source query

To enhance partition processing, there are two general best practices that you can apply for optimizing the source query that extracts fact data from the source database.

Use OLE DB Providers over .NET Data Providers

This is the same recommendation provided for the dimension processing. Since the Analysis Services runtime is written in native code, OLE DB Providers offer performance benefits over.NET Data Providers. When you use .NET Data Providers, data has to be marshaled between the .NET managed memory space and the native memory space. Since OLE DB Providers are already in native code, they provide significant performance benefits over the .NET Data Providers.

Use query bindings for partitions

A partition can be bound to either a source table or a source query. When you bind to a source query, as long as you return the correct number of columns expected in the partition, you can create a wide range of SQL statements to extract source data. Using source query bindings provides greater flexibility than using a named query in the data source view. For example, using query binding, you can point to a different data source using four-part naming and perform additional joins as necessary.

Using partitions to enhance processing performance

In the same manner that using multiple partitions can reduce the amount of data that needs to be scanned during data retrieval (as discussed in How partitions are used in querying), using multiple partitions can enhance processing performance by providing you with the ability to process smaller data components of a measure group in parallel.

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. When you initially create a cube, you must perform a **ProcessFull** on all measure groups in that cube.

If you process partitions from different client sessions, 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. If this is the case, you should ensure that you have at least one processed partition per measure group before you begin parallel operations. 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. In the majority of scenarios, 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. For more information on using **MaxParallel**, see Maximize parallelism during processing.

After initially loading your cube, multiple partitions are useful when you need to perform targeted data refreshes. Consider the following example. If you have a sales cube with a single partition, every time that you add new data such as a new day's worth of sales, you must not only refresh the fact data, but you must also refresh the aggregations to reflect the new data totals. This can be costly and time consuming depending on how much data you have in the partition, the aggregation design for the cube, and the type of processing that you perform.

With multiple partitions, you can isolate your data refresh operations to specific partitions. For example, when you need to insert new fact data, an effective technique involves creating a new partition to contain the new data and then performing a **ProcessFull** on the new partition. Using this technique, you can avoid impacting the other existing partitions. Note that it is possible to use XMLA scripts to automate the creation of a new partition during the refresh of your relational data warehouse.

Optimizing data inserts, updates, and deletes

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

Inserts

If you have a browseable 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** 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 a **ProcessFull** is technically not necessary given that you are only doing inserts. To optimize processing for insert operations, you can use a **ProcessAdd**.
- **ProcessAdd**—Use this operation to append data to the existing partition files. If you frequently perform a **ProcessAdd**, it is advised that you periodically perform a **ProcessFull** in order to rebuild and re-compress the partition data files. **ProcessAdd** internally creates a temp partition and merges it. This results in data fragmentation over time and the need to periodically perform a **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 a **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.

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**. You can also have an audit trail of the changes that you have made.

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, one 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** of the dimension (as long 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** or **Visible**. This is because the server does not know if there is a fact record assigned to the deleted member. With this property set appropriately, it will associate it to the unknown member at query time.

Evaluating rigid vs. flexible aggregations

Flexible and rigid attribute relationships not only impact how you process dimensions, but they also impact how aggregations are refreshed in a partition. Aggregations can either be categorized as rigid or flexible depending on the relationships of the attributes participating in the aggregation. For more information on the impact of rigid and flexible relationships, see Optimizing dimension inserts, updates, and deletes.

Rigid aggregations

An aggregation is *rigid* when all of the attributes participating in the aggregation have rigid direct or indirect relationships to the granularity attribute of a measure group. For all attributes in the aggregation, a check is performed to verify that all relationships are rigid. If any are flexible, the aggregation is flexible.

Flexible aggregations

An aggregation is *flexible* when one or more of the attributes participating in the aggregation have flexible direct or indirect relationships to the key attribute.

If you perform a **ProcessUpdate** on a dimension participating in flexible aggregations, whenever deletions or updates are detected for a given attribute, the aggregations for that attribute as well as any related attributes in the attribute chain are automatically dropped. The aggregations are not automatically recreated unless you perform one of the following tasks:

- Perform a **ProcessFull** on the cube, measure group, or partition. This is the standard **ProcessFull** operation that drops and re-creates the fact data and all aggregations in the partition.
- Perform a **ProcessIndexes** on the cube, measure group, or partition. By performing **ProcessIndexes**, you will only build whatever aggregations or indexes that need to be re-built.
- Configure Lazy Processing for the cube, measure group, or partition. If you configure Lazy Processing, the dropped aggregations are recalculated as a background task. While the flexible aggregations are being recalculated, users can continue to query the cube (without the benefit of the flexible aggregations). While the flexible aggregations are being recalculated, queries that would benefit from the flexible aggregations run slower because Analysis Services resolves these queries by scanning the fact data and then summarizing the data at query time. As the flexible aggregations are recalculated, they become available incrementally on a partition-by-partition basis. For a given cube, Lazy Processing is not enabled by default. You can configure it for a cube, measure group, or partition by changing the **ProcessingMode** property from **Regular** to **LazyAggregations**. To manage Lazy Processing, there are a series of server properties such as the **LazyProcessing \ MaxObjectsInParallel** setting, which controls the number of objects that can be lazy processed at a given time. By default it is set to 2. By increasing this number, you increase the number of objects processed in parallel; however, this also impacts query performance and should therefore be handled with care.
- Process affected objects. When you perform a **ProcessUpdate** on a dimension, you can choose whether or not to **Process Affected Objects**. If you select this option, you can trigger the update of dependent partitions within the same operation.

Note that if you do not follow one of the above techniques, and you perform a **ProcessUpdate** of a dimension that results in a deletion or update, the flexible aggregations for that attribute and all related attributes in the attribute chain are automatically deleted and not re-created, resulting in poor query performance. This is especially important to note because by default **<u>every</u>** aggregation is flexible since every attribute relationship type is set to **Flexible**.

As a result, great care must be taken to ensure that your refresh strategy configures the appropriate attribute relationships for your data changes and effectively rebuilds any flexible aggregations on an ongoing basis.

Optimizing Special Design Scenarios

Throughout this whitepaper, specific techniques and best practices are identified for improving the processing and query performance of Analysis Services OLAP databases. In addition to these techniques, there are specific design scenarios that require special performance tuning practices. Following is an overview of the design scenarios that are addressed in this section:

Special aggregate functions – Special aggregate functions allow you to implement distinct count and semiadditive data summarizations. Given the unique nature of these aggregate functions, special performance tuning techniques are required to ensure that they are implemented in the most efficient manner.

Parent-child hierarchies – Parent-child hierarchies have a different aggregation scheme than attribute and user hierarchies, requiring that you consider their impact on query performance in large-scale dimensions.

Complex Dimension Relationships – Complex dimension relationships include many-to-many relationships and reference relationships. While these relationships allow you to handle a variety of schema designs, complex dimension relationships also require you to assess how the schema complexity is going to impact processing and/or query performance.

Near real-time data refreshes - In some design scenarios, "near real-time" data refreshes are a necessary requirement. Whenever you implement a "near real-time" solution requiring low levels of data latency, you must consider how you are going to balance the required latency with querying and processing performance.

Special aggregate functions

Aggregate functions are the most common mechanism to summarize data. Aggregate functions are uniformly applied across all attributes in a dimension and can be used for additive, semiadditive, and nonadditive measures. Aggregate functions can be categorized into two general groups: traditional aggregate functions and special aggregate functions.

- Traditional aggregate functions consist of a group of functions that follow the general performance tuning recommendations described in other sections of this white paper. Traditional aggregate functions include **Sum**, **Count**, **Min**, and **Max**.
- Special aggregate functions require unique performance-tuning techniques. They include **DistinctCount** and a collection of semiadditive aggregate functions. The semiadditive functions include: **FirstChild**, **LastChild**, **FirstNonEmpty**, **LastNonEmpty**, **ByAccount**, and **AverageOfChildren**.

Optimizing distinct count

DistinctCount is a nonadditive aggregate function that counts the unique instances of an entity in a measure group. While distinct count is a very powerful analytical tool, it can have significant impact on processing and querying performance because of its explosive impact on aggregation size. When you use a distinct count aggregate function, it increases the size of an aggregation by the number of unique instances that are distinctly counted.

To better understand how distinct count impacts aggregation size, consider the following example. You have a measure group partition with an aggregation that summarizes sales amount by product category and year. You have ten product categories with sales for ten years, producing a total of 100 values in the aggregation. When you add a distinct count of customers to the measure group, the aggregation for the partition changes to include the customer key of each customer who has sales for a specific product category and year. If there are 1,000 customers, the number of potential values in the aggregation increases from 100 to 100,000 values, given that every customer has sales for every product category in every year. (Note that the actual number of values would be less than 100,000 due to natural data sparsity. At any rate, the value is likely to be a number much greater than 100.) While this additional level of detail is necessary to efficiently calculate the distinct count of customers, it introduces significant performance overhead when users request summaries of sales amount by product category and year.

With the explosive impact of distinct counts on aggregations, it is a best practice to separate each distinct count measure into its own measure group with the same dimensionality as the initial measure group. Using this technique, you can isolate the distinct count aggregations and maintain a separate aggregation design for non-distinct count measures.

Note that when you use Business Intelligence Development Studio to create a new measure, if you specify the distinct count aggregate function at the time of the measure creation, the Analysis Services Cube Designer automatically creates a separate measure group for the distinct count measure. However, if you change the aggregate function for an existing measure to distinct count, you must manually reorganize the distinct count measure into its own measure group.

When you are distinctly counting large amount of data, you may find that the aggregations for the distinct count measure group are not providing significant value, since the fact data must typically be queried to calculate the distinct counts. For a distinct count measure group, you should consider partitioning the measure group using data ranges of the distinct count field. For example, if you are performing a distinct count of customers using customer ID, consider partitioning the distinct count measure group by customer ID ranges. In this scenario, partition 1 may contain customer IDs between 1 and 1000. Partition 2 may contain customer IDs between 1001 and 2000. Partition 3 may contain customer IDs between 2001 and 3000, etc. This partitioning scheme improves query parallelism since the server does not have to coordinate data across partitions to satisfy the query.

For larger partitions, to further enhance performance, it may be advantageous to consider an enhanced partitioning strategy where you partition the distinct count measure group by both the distinct count field as well as your normal partitioning attribute, i.e., year, month, etc. As you consider your partitioning strategy for distinct count, keep in mind that when you partition by multiple dimensions, you may quickly find out that you have too many partitions. With too many partitions, in some scenarios you can actually negatively impact query performance if Analysis Services needs to scan and piece together data from multiple partitions at query time. For example, if you partition by sales territory and day, when users want the distinct count of customers by sales territory by year, all of the daily partitions in a specific year must be accessed and assimilated to return the correct value. While this will naturally happen in parallel, it can potentially result in additional querying overhead. To help you determine the size of and scope of your partitions, see the general sizing guidance in the Designing partitions section.

From a processing perspective, whenever you process a partition that contains a distinct count, you will notice an increase in processing time over your other partitions of the same dimensionality. It takes longer to process because of the increased size of the fact data and aggregations. The larger the amount of data that requires processing, the longer it takes to process, and the greater potential that Analysis Services may encounter memory constraints and will need to use temporary files to complete the processing of the aggregation. You must therefore assess whether your aggregation design is providing you with the most beneficial aggregations. To help you determine this, you can use the Usage-Based Optimization Wizard to customize your aggregation design to benefit your query workload. If you want further control over the aggregation design, you may want to consider creating custom aggregations by using the Aggregation Utility described in Appendix C. In addition, you must ensure that the system has enough resources to process the aggregations. For more information on this topic, see Tuning Server Resources.

The other reason that processing time may be slower is due to the fact that an ORDER BY clause is automatically added to the partition's source SQL statement. To optimize the performance of the ORDER BY in the source SQL statement, you can place a clustered index on the source table column that is being distinctly counted. Keep in mind that this is only applicable in the scenario where you have one distinct count per fact table. If you have multiple distinct counts off of a single fact table, you need to evaluate which distinct count will benefit most from the clustered index. For example, you may choose to apply the clustered index to the most granular distinct count field.

Optimizing semiadditive measures

Semiadditive aggregate functions include **ByAccount**, **AverageOfChildren**, **FirstChild**, **LastChild**, **FirstNonEmpty**, and **LastNonEmpty**. To return the correct data values, semiadditive functions must always retrieve data that includes the granularity attribute of the time dimension. Since semiadditive functions require more detailed data and additional server resources, to enhance performance, semiadditive measures should not be used in a cube that contains ROLAP dimensions or linked dimensions. ROLAP and linked dimensions also require additional querying overhead and server resources. Therefore, the combination of semiadditive measures with ROLAP and/or linked dimensions can result in poor query performance and should be avoided where possible.

Aggregations containing the granularity attribute of the time dimension are extremely beneficial for efficiently satisfying query requests for semiadditive measures. Aggregations containing other time attributes are never used to fulfill the request of a semiadditive measure.

To further explain how aggregations are used to satisfy query requests for semiadditive measures, consider the following example. In an inventory application, when you request a product's quantity on hand for a given month, the inventory balance for the month is actually the inventory balance for the final day in that month. To return this value, the Storage Engine must access partition data at the day level. If there is an appropriate aggregation that includes the day attribute, the Storage Engine will attempt to use that aggregation to satisfy the query. Otherwise the query must be satisfied by the partition fact data. Aggregations at the month and year cannot be used.

Note that if your cube contains only semiadditive measures, you will never receive performance benefits from aggregations created for nongranularity time attributes. In this scenario, you can influence the aggregation designer to create aggregations only for the granularity attribute of the time dimension (and the All attribute) by setting the **Aggregation Usage** to **None** for the nongranularity attributes of the time dimension. Keep in mind that the **Aggregation Usage** setting automatically applies to all measure groups in the cube, so if the cube contains only semiadditive measures, such as in inventory applications, the **Aggregation Usage** setting can be uniformly applied across all measure groups.

If you have a mixed combination of semiadditive measures and other measures across measure groups, you can adopt the following technique to apply unique aggregation usage settings to specific measure groups. First, adjust the **Aggregation Usage** settings for a dimension so that they fit the needs of a particular measure group and then design aggregations for that measure group only. Next, change the **Aggregation Usage** settings to fit the needs of the next measure group and then design aggregations for that measure group only. Using this approach, you can design aggregations measure group-by-measure group, and even partition-by-partition if desired.

In addition to this technique, you can also optimize the aggregation design using the Usage-Based Optimization Wizard to create only those aggregations that provide the most value for your query patterns. If you want further control over the aggregation design, you may want to consider creating custom aggregations using the Aggregation Utility described in Appendix C.

Parent-child hierarchies

Parent-child hierarchies are hierarchies with a variable number of levels, as determined by a recursive relationship between a child attribute and a parent attribute. Parent-child hierarchies are typically used to represent a financial chart of accounts or an organizational chart. In parent-child hierarchies, aggregations are created only for the key attribute and the top attribute, i.e., the All attribute unless it is disabled. As such, refrain from using parent-child hierarchies that contain large numbers of members at intermediate levels of the hierarchy. Additionally, you should limit the number of parent-child hierarchies in your cube.

If you are in a design scenario with a large parent-child hierarchy (greater than 250,000 members), you may want to consider altering the source schema to re-organize part or all of the hierarchy into a user hierarchy with a fixed number of levels. Once 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.

Complex dimension relationships

The flexibility of Analysis Services enables you to build cubes from a variety of source schemas. For more complicated schemas, Analysis Services supplies many-to-many relationships and reference relationships to help you model complex associations between your dimension tables and fact tables. While these relationships provide a great deal of flexibility, to use them effectively, you must evaluate their impact on processing and query performance.

Many-to-many relationships

In Analysis Services, many-to-many relationships allow you to easily model complex source schemas. To use many-to-many relationships effectively, you must be familiar with the business scenarios where these relationships are relevant.

Background information on many-to-many relationships

In typical design scenarios, fact tables are joined to dimension tables via many-to-one relationships. More specifically, each fact table record can join to only one dimension table record and each dimension table record can join to multiple fact table records. Using this many-to-one design, you can easily submit queries that aggregate data by any dimension attribute.

Many-to-many design scenarios occur when a fact table record can potentially join to multiple dimension table records. When this situation occurs, it is more difficult to correctly query and aggregate data since the dimension contains multiple instances of a dimension entity and the fact table cannot distinguish among the instances.

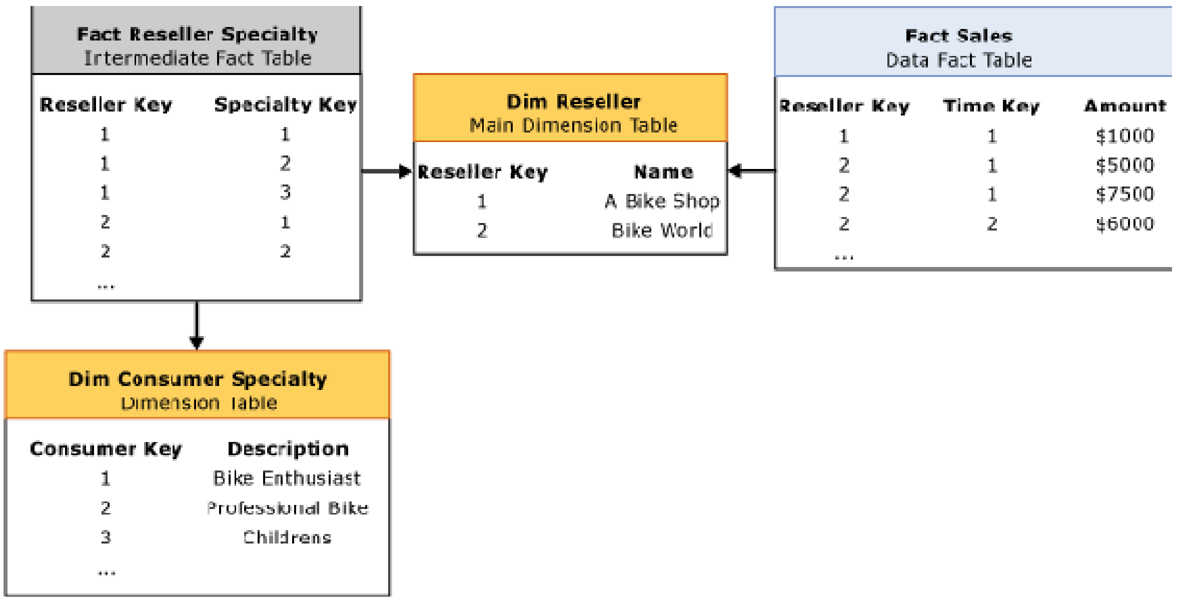
To better understand how many-to-many relationships impact data analysis, consider the following example. You have a reseller dimension where the primary key is the individual reseller. To enhance your analysis, you want to add the consumer specialty attribute to the reseller dimension. Upon examining data, you notice that each reseller can have multiple consumer specialties. For example, A Bike Store has two consumer specialties: Bike Enthusiast and Professional Bike. Your sales fact table, however, only tracks sales by reseller, not by reseller and consumer specialty. In other words, the sales data does not distinguish between A Bike Shop with a Bike Enthusiast specialty and A Bike Shop with a Professional Bike specialty. In the absence of some kind of weighting factor to allocate the sales by consumer specialty, it is a common practice to repeat the sales values for each reseller/consumer specialty combination. This approach works just fine when you are viewing data by consumer specialty and reseller; however, it poses a challenge when you want to examine data totals as well as when you want to analyze data by other attributes.

Continuing with the reseller example, if you add the new consumer specialty attribute to the reseller dimension table, whenever the fact table is joined to the dimension table via the individual reseller, the number of records in the result set is multiplied by the number of combinations of reseller and consumer specialty. When the data from this result set is aggregated, it will be inflated. In the example of A Bike Shop, sales data from the fact table will be double-counted regardless of which reseller attribute you group by. This double-counting occurs because A Bike Shop appears twice in the table with two consumer specialties. This data repeating is okay when you are viewing the breakdown of sales by consumer specialty and reseller; however, any sales totals or queries that do not include consumer specialty will be incorrectly inflated.

To avoid incorrectly inflating your data, many-to-many relationships are typically modeled in the source schema using multiple tables as displayed in Figure 26.

- A main dimension table contains a unique list of primary key values and other attributes that have a one-to-one relationship to the primary key. To enable the dimension table to maintain a one-to-many relationship with the fact table, the multivalued attribute is not included in this dimension. In the reseller example displayed in Figure 26, Dim Reseller is the main dimension table, storing one record per reseller. The multivalued attribute, consumer specialty, is not included in this table.

- A second dimension table contains a unique list of values for the multivalued attribute. In the reseller example, the Dim Consumer Specialty dimension table stores the distinct list of consumer specialties.
- An intermediate fact table maps the relationship between the two dimension tables. In the reseller example, Fact Reseller Specialty is an intermediate fact table that tracks the consumer specialties of each reseller.
- The data fact table stores a foreign key reference to the primary key of the main dimension table. In the reseller example, the Fact Sales table stores the numeric sales data and has a many-to-one relationship with Dim Reseller via the Reseller Key.



Fact Reseller Specialty
Intermediate Fact Table

Reseller Key	Specialty Key
1	1
1	2
1	3
2	1
2	2
...	

Dim Reseller
Main Dimension Table

Reseller Key	Name
1	A Bike Shop
2	Bike World

Fact Sales
Data Fact Table

Reseller Key	Time Key	Amount
1	1	$1000
2	1	$5000
2	1	$7500
2	2	$6000
...		

Dim Consumer Specialty
Dimension Table

Consumer Key	Description
1	Bike Enthusiast
2	Professional Bike
3	Childrens
...	

Figure 26 Many-to-many relationship

Using this design, you have maintained the many-to-one relationship between Fact Sales and Dim Reseller. This means that if you ignore Dim Consumer Specialty and Fact Reseller Specialty, you can easily query and aggregate data from Fact Sales by any attribute in the Dim Reseller table without incorrectly inflating values. However this design does not fully solve the problem, since it is still tricky to analyze sales by an attribute in Dim Consumer Specialty. Remember that you still do not have sales broken down by consumer specialty; i.e., sales is only stored by reseller and repeats for each reseller / consumer specialty combination. For queries that summarize sales by consumer specialty, the data totals are still inflated unless you can apply a *distinct sum*.

To help you accomplish this distinct sum, Analysis Services provides built-in support for many-to-many relationships. Once a many-to-many relationship is defined, Analysis Services can apply distinct sums to correctly aggregate data where necessary. To obtain this benefit in the reseller example, first create the Sales measure group that includes the Dim Reseller dimension and sales measures from the Fact Sales table. Next, you can use a many-to-many relationship to relate the Sales measure group to the Consumer Specialty many-to-many dimension. To set up this relationship, you must identify an intermediate measure group that can be used to map Sales data to Consumer Specialty. In this scenario, the intermediate measure group is Reseller Specialty, sourced from the Fact Reseller Specialty fact table.

Performance considerations

During processing, the data and intermediate measure groups are processed independently. Fact data and aggregations for the data measure group do not include any attributes from the many-to-many dimension. When you query the data measure group by the many-to-many dimension, a run-time "join" is performed between the two measure groups using the granularity attributes of each dimension that the measure groups have in common. In the example in Figure 26, when users want to query Sales data by Consumer Specialty, a run-time join is performed between the Sales measure group and Reseller Specialty measure group using the Reseller Key of the Reseller dimension. From a performance perspective, the run-time join has the greatest impact on query performance. More specifically, if the intermediate measure group is larger than the data measure group or if the many-to-many dimension is generally large (at least one million members), you can experience query performance issues due to the amount of data that needs to be joined at run time. To optimize the run-time join, review the aggregation design for the intermediate measure group to verify that aggregations include attributes from the many-to-many dimension. In the example in Figure 26, aggregations for the intermediate measure group should include attributes from the Consumer Specialty dimension such as the description attribute. While many-to-many relationships are very powerful, to avoid query performance issues, in general you should limit your use of many-to-many relationships to smaller intermediate measure groups and dimensions.

Reference relationships

In traditional dimension design scenarios, all dimension tables join directly to the fact table by means of their primary keys. In snowflake dimension designs, multiple dimension tables are chained together, with the chained dimensions joining indirectly to the fact table by means of a key in another dimension table. These chained dimensions are often called *snowflake dimension tables*. Figure 27 displays an example of snowflake dimension tables. Each table in the snowflake is linked to a subsequent table via a foreign key reference.

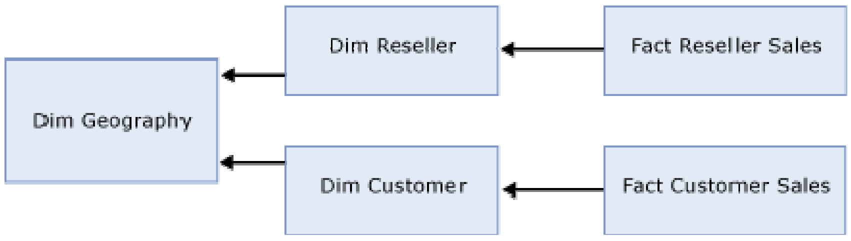


Figure 27 Snowflake dimension tables

In Figure 27, the Dim Reseller table has a snowflake relationship to the Dim Geography table. In addition, the Dim Customer table has a snowflake relationship to the Dim Geography table. From a relational point of view, if you want to analyze customer sales by geography, you must join Dim Geography to Fact Customer Sales via Dim Customer. If you want to analyze reseller sales by geography, you must join Dim Geography to Fact Reseller Sales via Dim Reseller.

Within Analysis Services, dimensions are "joined" to measure groups by specifying the dimension's relationship type. Most dimensions have regular relationships where a dimension table is joined directly to the fact table. However, for snowflake dimension table scenarios, such as the one depicted in Figure 27, there are two general design techniques that you can adopt as described in this section.

Option 1 - Combine attributes

For each dimension entity that joins to the fact table, create a single OLAP dimension that combines attributes from all of the snowflake dimension tables, and then join each dimension to measure group using a regular relationship type. If you have multiple dimension entities that reference the same snowflake tables, attributes from the shared snowflake tables are repeated across the OLAP dimensions.

To apply this technique to the Figure 27 example, create two OLAP dimensions: 1) a Reseller dimension that contains attributes from both the Dim Reseller and Dim Geography tables, and 2) a Customer dimension that contains attributes from Dim Customer and Dim Geography. Note that attributes from Dim Geography are repeated across both the Reseller and Customer dimensions.

For the Reseller Sales measure group, use a Regular relationship for the Reseller dimension. For the Customer Sales measure group, use a regular relationship Customer dimension. Remember that the relationship between a dimension and a measure group defines how the dimension data is to be "joined" to the fact data. A regular relationship means that you have defined a direct relationship between one or more dimension columns and one or more measure group columns.

With this design, for each OLAP dimension, all of the snowflake dimension tables are joined together at processing time and the OLAP dimension is materialized on disk. As with any other processing operation, you can control whether the processing should remove missing keys or use the unknown member for any records that do join across all tables.

From a performance perspective, the benefit of this approach is the ability to create natural hierarchies and use aggregations. Since each dimension has a regular relationship to the measure group, to enhance query performance, aggregations can be designed for attributes in each dimension, given the proper configuration of attribute relationships. In addition, during querying, you can take advantage of **Autoexists** optimizations that naturally occur between attributes within a dimension. For more information on **Autoexists**, see Removing empty tuples.

If you use this approach, you must also consider the impact of increasing the number of attributes that the aggregation design algorithm must consider. By repeating attributes across multiple dimensions, you are creating more work for the aggregation design algorithm which could negatively impact processing times.

Option 2 - Use a reference relationship

An alternative design approach to combining attributes involves reference relationships. Reference relationships allow you to indirectly relate OLAP dimensions to a measure group using an intermediate dimension. The intermediate dimension creates a "join" path that the measure group can use to relate its data to each reference dimension.

To apply this technique to the example in Figure 27, create three separate OLAP dimensions for customer, reseller, and geography. The following describes how these dimensions can be related to each measure group (Reseller Sales and Customer Sales):

- The Customer Sales measure group contains a regular relationship to the Customer dimension and contains a reference relationship to the Geography dimension. The reference relationship uses Customer as an intermediate dimension to assign sales to specific geographies.
- The Reseller Sales measure group contains a regular relationship to the Reseller dimension and contains a reference relationship to the Geography dimension. The reference relationship uses Reseller as an intermediate dimension to assign sales to specific geographies.

To use this technique effectively, you must consider the impacts of reference relationships on processing and query performance. During processing, each dimension is processed independently. No attributes from the reference dimension are automatically considered for aggregation. During querying, measure group data is joined to the reference dimension as necessary by means of the intermediate dimension. For example, if you query customer sales data by geography, a run-time join must occur from the Customer Sales measure group to Geography via the Customer dimension. This process can be somewhat slow for large dimensions. In addition, any missing attribute members that are encountered during querying are automatically assigned to the unknown member in order to preserve data totals.

To improve the query performance of reference relationships, you can choose to materialize them. Note that by default, reference relationships are not materialized. When a reference relationship is materialized, the joining across dimensions is performed during processing as opposed to querying. In addition, the attributes in the materialized reference dimensions follow the aggregation rules of standard dimensions. For more information on these rules, see The aggregation usage rules. Since the join is performed during processing and aggregations are possible, materialized reference relationships can significantly improve query performance when compared to unmaterialized relationships.

Some additional considerations apply to materialized reference relationships. During processing, the reference dimension is processed independently. At this time, if any row in the measure group does not join to the reference dimension, the record is removed from the partition. Note that this is different behavior than the unmaterialized reference relationship where missing members are assigned to the unknown member.

To better understand how missing members are handled for materialized relationships, consider the following example. If you have a sales order in the Customer Sales fact table that maps to a specific customer but that customer has a missing geography, the record cannot join to the Geography table and is rejected from the partition. Therefore, if you have referential integrity issues in your source data, materializing the reference relationship can result in missing data from the partition for those fact records that do not join to the reference dimension. To counteract this behavior and handle missing values, you can create your own unknown dimension record in the reference dimension table and then assign that value to all records missing reference values during your extraction, transformation, and loading (ETL) processes. With the unknown record in place for missing values, at processing time, all customer records can successfully join to the reference dimension.

Option Comparison - Combining attributes vs. using reference relationships

When you compare these two design alternatives, it is important to assess the overall impacts on processing and query performance. When you combine attributes, you can benefit from creating natural hierarchies and using aggregations to improve query performance. When you use reference relationships, a reference dimension can only participate in aggregations if it is materialized. These aggregations will not take into account any hierarchies across dimensions, since the reference dimension is analyzed separately from the other dimensions. In light of this information, the following guidelines can help you decide which approach to adopt:

- If your dimension is frequently queried and can benefit from natural hierarchies and aggregations, you should combine attributes from snowflake dimension tables into your normal dimension design.
- If the dimension is not frequently queried and is only used for one-off analysis, you can use unmaterialized reference relationships to expose the dimension for browsing without the overhead of creating aggregations for dimension attributes that are not commonly queried. If the intermediate dimensions are large, to optimize the query performance you can materialize the reference relationship.

As an additional design consideration, note that the example in Figure 27 includes snowflake tables with two fact tables / measure groups. When you have snowflake tables that join to a single fact table/ measure group, as depicted in Figure 28, the only available design option is to combine attributes**.**

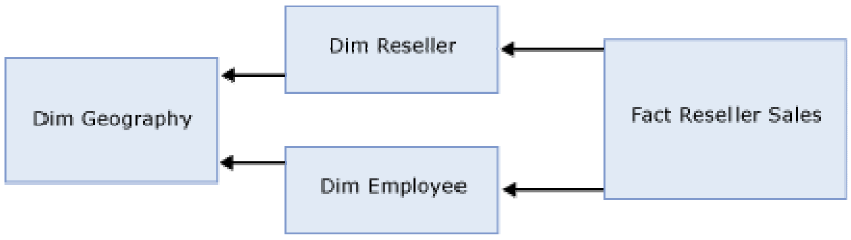


Figure 28 One measure group

Reference relationships are not applicable to this design scenario because Analysis Services only allows one reference relationship per dimension per measure group. In the example in Figure 28, this means that when you define Dim Geography as a reference dimension to Reseller Sales, you must either select Dim Reseller or Dim Employee as the intermediate dimension. This either/or selection is not likely going to satisfy your business requirements. As a result, in this scenario, you can use design option 1, combining attributes, to model the snowflake dimension tables.

Near real-time data refreshes

Whenever you have an application that requires a low level of latency, such as in near real-time data refreshes, you must consider a special set of performance tuning techniques that can help you balance low levels of data latency with optimal processing and query performance.

Generally speaking, low levels of data latency include hourly, minute, and second refresh intervals. When you need to access refreshed data on an hourly basis, for example, your first instinct may be to process all of your dimensions and measure groups every hour. However, if it takes 30 minutes to process all database objects, to meet the hourly requirement, you would need to reprocess every 30 minutes. To further complicate this, you would also need to assess the performance impact of any concurrent querying operations that are competing for server resources.

Instead of processing all measure groups and dimensions to meet a low latency requirement, to improve performance and enhance manageability, you can use partitions to isolate the low latency data from the rest of the data. Typically this means that you create one or more near real-time partitions that are constantly being updated with refreshed data. Isolating the low latency data follows normal best practices for partitioning so that you can process and update the near real-time partitions without impacting the other partitions. Using this solution, you can also reduce the amount of data that needs to be processed in near real-time.

With partitions in place, the next step is to decide on how you are going to refresh the data in your near real-time partition(s). Remember that partitions require you to process them in order to refresh the data. Depending on the size of your near real-time partition(s) and the required frequency of the data updates, you can either schedule the refresh of a partition on a periodic basis or you can enable Analysis Services to manage the refresh of a partition using proactive caching.

Proactive caching is a feature in Analysis Services that transparently synchronizes and maintains a partition or dimension much like a cache. A proactive caching partition or dimension is commonly referred to as a cache, even though it is still considered to be a partition or dimension. Using proactive caching, you can enable Analysis Services to automatically detect a data change in the source data, to incrementally update or rebuild the cache with the refreshed data, and to expose the refreshed data to end users.

Even though proactive caching automates much of the refresh work for you, from a performance perspective, you must still consider the typical parameters that impact processing performance such as the frequency of source system updates, the amount of time it takes to update or rebuild the cache, and the level of data latency that end users are willing to accept.

From a performance perspective, there are three groups of settings that impact query responsiveness and processing performance for proactive caching: notification settings, refresh settings, and availability settings.

Notification settings

Notification settings impact how Analysis Services detects data changes in the source system. To satisfy the needs of different data architectures, Analysis Services provides a few mechanisms that you can use to notify Analysis Services of data changes. From a performance perspective, the **Scheduled polling** mechanism provides the most flexibility by allowing you to either rebuild or incrementally update the cache. Incremental updates improve proactive caching performance by reducing the amount of data that needs to be processed. For proactive caching partitions, incremental updates use a **ProcessAdd** to append new data to the cache. For proactive caching dimensions, a **ProcessUpdate** is performed. If you use **Scheduled polling** without incremental updates, the cache is always completely rebuilt.

Refresh settings

Refresh settings impact when Analysis Services rebuilds or updates the cache. The two most important refresh settings are **Silence Interval** and **Silence Override Interval**.

- **Silence Interval** defines how long Analysis Services waits from the point at which it detects a change to when it begins refreshing the cache. Since Analysis Services does not officially know when source data changes are finished, the goal is for Analysis Services to refresh the cache when there is a lull in the propagation of source data changes. The lull is defined by a period of silence or no activity that must expire before Analysis Services refreshes the cache. Consider the example where the **Silence Interval** is set to ten seconds and your cache is configured to be fully rebuilt during refresh. In this scenario, once Analysis Services detects a data change, a time counter starts. If ten seconds pass and no additional changes are detected, Analysis Services rebuilds the cache. If Analysis Services detects additional changes during that time period, the time counter resets each time a change is detected.
- **Silence Override Interval** determines how long Analysis Services waits before performing a forced refresh of the cache. **Silence Override Interval** is useful when your source database is updated frequently and the **Silence Interval** threshold cannot be reached, since the time counter is continuously reset. In this scenario, you can use the **Silence Override Interval** to force a refresh of the cache after a certain period of time, such as five minutes or ten minutes. For example, if you set the **Silence Override Interval** to ten minutes, every ten minutes Analysis Services refreshes the cache only if a change was detected.

Availability settings

Availability settings allow you to control how data is exposed to end users during cache refresh. If your cache takes several minutes to update or rebuild, you may want to consider configuring the proactive caching settings to allow users to see an older cache until the refreshed cache is ready. For example, if you configure a partition to use **Automatic MOLAP** settings (**Silence Interval** = 10 seconds and **Silence Override Interval** = 10 minutes), during cache refresh, users query the old cache until the new cache is ready. If the cache takes five hours to rebuild, users must wait five hours for cache rebuild to complete. If you always want users to view refreshed data, enable the **Bring Online Immediately** setting. With **Bring Online Immediately** enabled, during cache refresh all queries are directed to the relational source database to retrieve the latest data for end users. While this provides users with refreshed data, it can also result in reduced query performance given that Analysis Services needs to redirect queries to the relational source database. If you want finer grained control over users viewing refreshed data, you can use the **Latency** setting to define a threshold that controls when queries are redirected to the source database during a cache refresh. For example, if you set **Latency** to four hours and the cache requires five hours to rebuild, for four hours, the queries will be satisfied by the older cache. After four hours, queries are redirected to the source database until the cache has completed its refresh.

Figure 29 illustrates how the proactive caching settings impact queries during cache rebuilding.

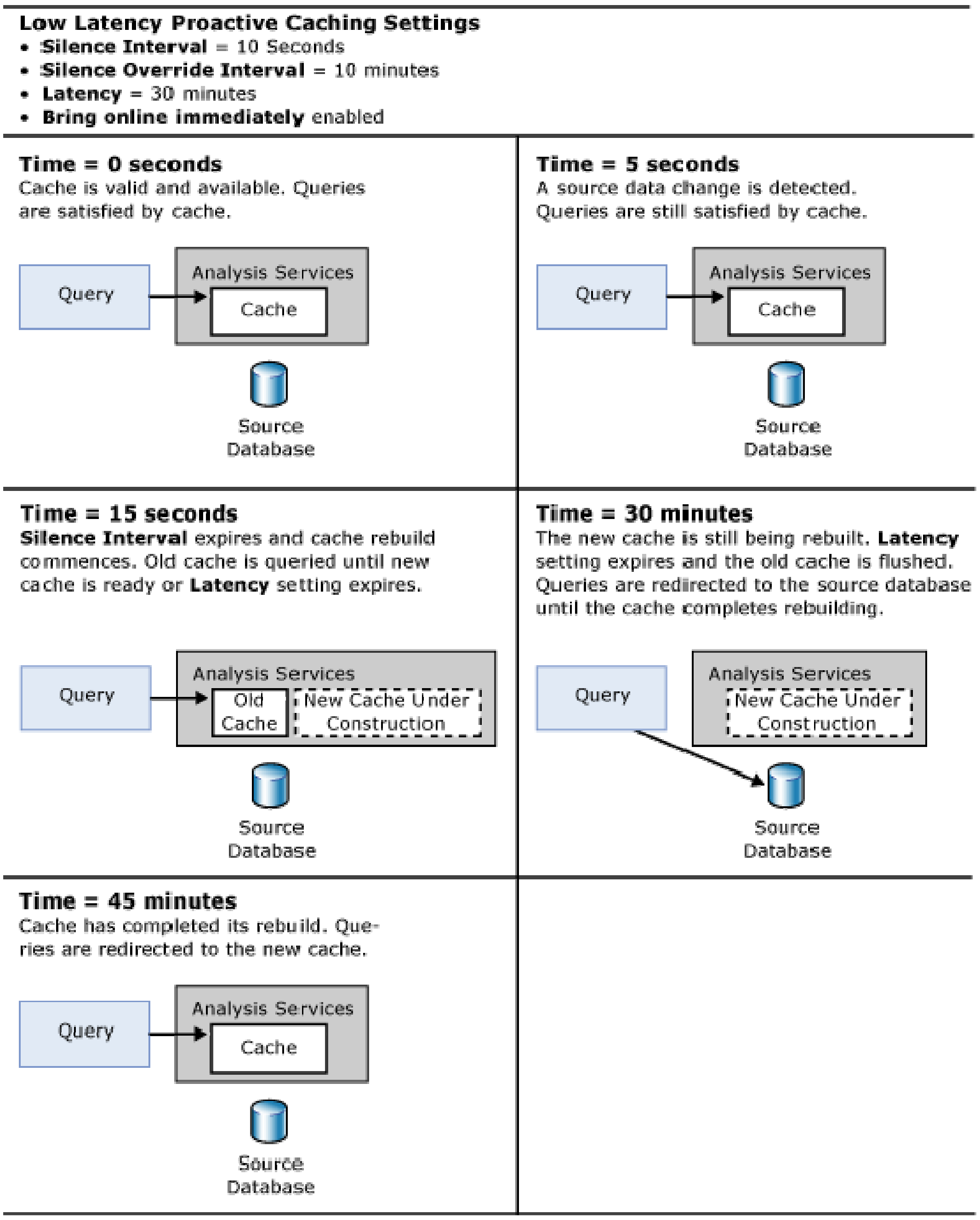

Low Latency Proactive Caching Settings
• Silence Interval = 10 Seconds
• Silence Override Interval = 10 minutes
• Latency = 30 minutes
• Bring online immediately enabled
Time = 0 seconds
Cache is valid and available. Queries are satisfied by cache.
Query
Analysis Services
Cache
Source Database
Time = 5 seconds
A source data change is detected. Queries are still satisfied by cache.
Time = 15 seconds
Silence Interval expires and cache rebuild commences. Old cache is queried until new cache is ready or Latency setting expires.
Old Cache
New Cache Under Construction
Time = 30 minutes
The new cache is still being rebuilt. Latency setting expires and the old cache is flushed. Queries are redirected to the source database until the cache completes rebuilding.
Time = 45 minutes
Cache has completed its rebuild. Queries are redirected to the new cache.

Figure 29 Proactive caching example

Generally speaking, to maximize query performance, it is a good practice to increase **Latency** where possible so that queries can continue to execute against the existing cache while data is read and processed into a new cache whenever the silence interval or silence override interval is reached. If you set the **Latency** too low, query performance may suffer as queries are continuously redirected to the relational source. Switching back and forth between the partition and the relational source can provide very unpredictable query response times for users. If you expect queries to constantly be redirected to the source database, to optimize query performance, you must ensure that Analysis Services understands the partition's data slice. Setting a partition's data slice is not necessary for traditional partitions. However, the data slice must be manually set for proactive caching partitions, as well as any ROLAP partition. In light of the potential redirection of queries to the relational source, proactive caching is generally not recommended on cubes that are based on multiple data sources.

If you enable proactive caching for your dimensions, to optimize processing, pay special attention to the **Latency** setting for each dimension and the overall impact of redirecting queries to the source database. When a dimension switches from the dimension cache to the relational database, all partitions that use the dimension need to be fully reprocessed. Therefore, where possible, it is a good idea to ensure that the **Latency** setting allows you to query the old cache until the new cache is rebuilt.

Tuning Server Resources

Query responsiveness and efficient processing require effective usage of memory, CPU, and disk resources. To control the usage of these resources, Analysis Services 2005 introduces a new memory architecture and threading model that use innovative techniques to manage resource requests during querying and processing operations.

To optimize resource usage across various server environments and workloads, for every Analysis Services instance, Analysis Services exposes a collection of server configuration properties. To provide ease-of-configuration, during installation of Analysis Services 2005, many of these server properties are dynamically assigned based on the server's physical memory and number of logical processors. Given their dynamic nature, the default values for many of the server properties are sufficient for most Analysis Services deployments. This is different behavior than previous versions of Analysis Services where server properties were typically assigned static values that required direct modification. While the Analysis Services 2005 default values apply to most deployments, there are some implementation scenarios where you may be required to fine tune server properties in order to optimize resource utilization.

Regardless of whether you need to alter the server configuration properties, it is always a best practice to acquaint yourself with how Analysis Services uses memory, CPU, and disk resources so you can evaluate how resources are being utilized in your server environment.

For each resource, this section presents two topics: 1) a topic that describes how Analysis Services uses system resources during querying and processing, and 2) a topic that organizes practical guidance on the design scenarios and data architectures that may require the tuning of additional server properties.

Understanding how Analysis Services uses memory – Making the best performance decisions about memory utilization requires understanding how the Analysis Services server manages memory overall as well as how it handles the memory demands of processing and querying operations.

Optimizing memory usage – Optimizing memory usage requires applying a series of techniques to detect whether you have sufficient memory resources and to identify those configuration properties that impact memory resource utilization and overall performance.

Understanding how Analysis Services uses CPU resources - Making the best performance decisions about CPU utilization requires understanding how the Analysis Services server uses CPU resources overall as well as how it handles the CPU demands of processing and querying operations.

Optimizing CPU usage - Optimizing CPU usage requires applying a series of techniques to detect whether you have sufficient processor resources and to identify those configuration properties that impact CPU resource utilization and overall performance.

Understanding how Analysis Services uses disk resources - Making the best performance decisions about disk resource utilization requires understanding how the Analysis Services server uses disk resources overall as well as how it handles the disk resource demands of processing and querying operations.

Optimizing disk usage – Optimizing disk usage requires applying a series of techniques to detect whether you have sufficient disk resources and to identify those configuration properties that impact disk resource utilization and overall performance.

Understanding how Analysis Services uses memory

Analysis Services 2005 introduces a new memory architecture that allocates and manages memory in a more efficient manner than previous versions of Analysis Services where the memory management of dimensions and other objects placed limits on querying and processing performance. The primary goal of memory management in Analysis Service 2005 is to effectively utilize available memory while balancing the competing demands of processing and querying operations.

To help you gain some familiarity with the distinct memory demands of these two operations, following is a high-level overview of how Analysis Services 2005 uses memory during processing and querying:

- **Querying**—During querying, memory is used at various stages of query execution to satisfy query requests. To promote fast data retrieval, the Storage Engine cache is used to store measure group data. To promote fast calculation evaluation, the Query Execution Engine cache is used to store calculation results. To efficiently retrieve dimensions, dimension data is paged into memory as needed by queries, rather than being loaded into memory at server startup, as in prior versions of Analysis Services. To improve query performance, it is essential to have sufficient memory available to cache data results, calculation results, and as-needed dimension data.
- **Processing**—During processing, memory is used to temporarily store, index, and aggregate data before writing to disk. Each processing job requests a specific amount of memory from the Analysis Services memory governor. If sufficient memory is not available to perform the job, the job is blocked. To ensure that processing proceeds in an efficient manner, it is important to verify that there is enough memory available to successfully complete all processing jobs and to optimize the calculation of aggregations in memory.

Given the distinct memory demands of querying and processing, it is important to not only understand how each operation impacts memory usage, but it is also important to understand how the Analysis Services server manages memory across all server operations. The sections that follow describe the memory management techniques of the Analysis Services server as well as how it handles the specific demands of querying and processing

Memory management

To effectively understand Analysis Services memory management techniques, you must first consider the maximum amount of memory that Analysis Services can address. Analysis Services relies on Microsoft Windows virtual memory for its memory page pool. The amount of memory it can address depends on the version of SQL Server that you are using:

- For SQL Server 2005 (32-bit), the maximum amount of virtual memory that an Analysis Services process can address is 3 gigabytes (GB). By default, an Analysis Services process can only address 2 GB; however it is possible to enable Analysis

Services to address 3 GB. For guidelines on how to optimize the addressable memory of Analysis Services, see Increasing available memory.

- SQL Server 2005 (64-bit) is not limited by a 3-GB virtual address space limit, enabling the 64-bit version of Analysis Services to use as much address space as it needs. This applies to both IA64 and X64 architectures.

To perform server operations, Analysis Services requests allocations of memory from the Windows operating system, and then returns that memory to the Windows operating system when the allocated memory is no longer needed. Analysis Services manages the amount of memory allocated to the server by using a memory range that is defined by two server properties: **Memory\TotalMemoryLimit** and **Memory\LowMemoryLimit**.

- **Memory\TotalMemoryLimit** represents the upper limit of memory that the server uses to manage all Analysis Services operations. If **Memory\TotalMemoryLimit** is set to a value between 0 and 100, it is interpreted as a percentage of total physical memory. If the property is set to a value above 100, Analysis Services interprets it as an absolute memory value in bytes. The default value for **Memory\TotalMemoryLimit** is 80, which translates to 80% of the amount of physical memory of the server. Note that this property does not define a hard limit on the amount of memory that Analysis Services uses. Rather, it is a soft limit that is used to identify situations where the server is experiencing memory pressure. For some operations, such as processing, if Analysis Services requires additional memory beyond the value of **Memory\TotalMemoryLimit**, the Analysis Services server attempts to reserve that memory regardless of the value of the property.
- **Memory\LowMemoryLimit** represents the lower limit of memory that the server uses to manage all Analysis Services operations. Like the **Memory\TotalMemoryLimit** property, a value between 0 and 100 is interpreted as a percentage of total physical memory. If the property is set to a value above 100, Analysis Services interprets it as an absolute memory value in bytes. The default value for the **Memory\LowMemoryLimit** property is 75, which translates to 75% of the amount of physical memory on the Analysis Services server.

Analysis Services uses these memory range settings to manage how memory is allocated and used at various levels of memory pressure. When the server experiences elevated levels of memory pressure, Analysis Services uses a set of cleaner threads, one cleaner thread per logical processor, to control the amount of memory allocated to various operations. Depending on the amount of memory pressure, the cleaner threads are activated in parallel to shrink memory as needed. The cleaner threads clean memory according to three general levels of memory pressure:

- **No Memory Pressure**—If the memory used by Analysis Services is below the value set in the **Memory\LowMemoryLimit** property, the cleaner does nothing.
- **Some Memory Pressure**—If the memory used by Analysis Services is between the values set in the **Memory\LowMemoryLimit** and the **Memory\TotalMemoryLimit** properties, the cleaner begins to clean memory using a cost/benefit algorithm. For more information on how the cleaner shrinks memory, see Shrinkable vs. non-shrinkable memory.
- **High Memory Pressure**—If the memory used by Analysis Services is above the value set in the **Memory\TotalMemoryLimit** property, the cleaner cleans until the memory used by Analysis Services reaches the **Memory\TotalMemoryLimit**. When

the memory used by Analysis Services exceeds the **Memory\TotalMemoryLimit**, the server goes into an aggressive mode where it cleans everything that it can. If the memory used is mostly non-shrinkable (more information on non-shrinkable memory is included in the next section), and cannot be purged, Analysis Services detects that the cleaner was unable to clean much. If it is in this aggressive mode of cleaning, it tries to cancel active requests. When this point is reached, you may see poor query performance, out of memory errors in the event log, and slow connection times.

Shrinkable vs. non-shrinkable memory

Analysis Services divides memory into two primary categories: shrinkable memory and non-shrinkable memory as displayed in Figure 30.

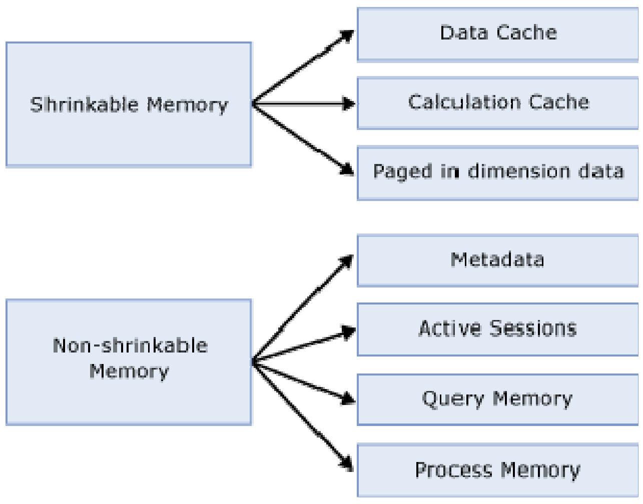


Figure 30 Shrinkable vs. non-shrinkable memory

When the cleaner is activated, it begins evicting elements of shrinkable memory, based on a cost/benefit algorithm that takes into account a variety of factors, including how frequently the entry is used, the amount of resources required to resolve the entries, and how much space is consumed by related entries. Shrinkable memory elements include the following:

- **Cached Results**—Cached results include the Storage Engine data cache and Query Execution Engine calculation cache. As stated earlier in this document, the Storage Engine data cache contains measure group data and the Query Execution Calculation Engine cache contains calculation results. While both caches can help improve query response times, the data cache provides the most benefit to query performance by storing data that has been cached from disk. In situations of memory pressure, the cleaner shrinks the memory used for cached results. With this in mind, it is a good practice to monitor the usage of memory so that you can minimize the scenarios where elevated levels of memory pressure force the removal of cached results. For more information on how to monitor memory pressure, see Monitoring memory management.
- **Paged in dimension data**—Dimension data is paged in from the dimension stores as needed. The paged-in data is kept in memory until the cleaner is under memory

pressure to remove it. Note that this is different behavior than previous versions of Analysis Services where all dimension data was resident in memory.

- **Expired Sessions**—Idle client sessions that have exceeded a longevity threshold are removed by the cleaner based on the level of memory pressure. Several server properties work together to manage the longevity of idle sessions. For more information on how to evaluate these properties, see Monitoring the timeout of idle sessions.

Non-shrinkable memory elements are not impacted by the Analysis Services cleaner. Non-shrinkable memory includes the following components:

- **Metadata** For each Analysis Services database, metadata is initialized and loaded into memory on demand. Metadata includes the definition of all objects in the database (not the data elements). The more objects in your database (including cubes, measure groups, partitions, and dimensions) and the more databases that you have on a given server, the larger the metadata overhead in memory. Note that this overhead is generally not large for most implementations. However, you can experience significant overhead if your Analysis Services server contains hundreds of databases with tens or hundreds of objects per database, such as in hosted solutions. For more information on how to monitor metadata overheard, see Minimizing metadata overhead.
- **Active Sessions**—For each active session, calculated members, named sets, connections, and other associated session information is retained as non-shrinkable memory.
- **Query Memory and Process Memory**—Analysis Services reserves specific areas of memory for temporary use during querying and processing. During the execution of a query, for example, memory may be used to materialize data sets such as during the cross joining of data. During processing, memory is used to temporarily store, index, and aggregate data before it are written to disk. These memory elements are non-shrinkable because they are only needed to complete a specific server operation. As soon as the operation is over, these elements are removed from memory.

Memory demands during querying

During querying, memory is primarily used to store cached results in the data and calculation caches. As stated previously, of the two caches, the one that provides the most significant performance benefit is the data cache. When Analysis Services first starts, the data cache is empty. Until the data cache is loaded with data from queries, Analysis Services must resolve user queries by using data stored on disk, either by scanning the fact data or by using aggregations. Once these queries are loaded into the data cache, they remain there until the cleaner thread removes them or the cache is flushed during measure group or partition processing.

You can often increase query responsiveness by preloading data into the data cache by executing a generalized set of representative user queries. This process is called *cache warming*. While cache warming can be a useful technique, cache warming should not be used as a substitute for designing and calculating an appropriate set of aggregations. For more information on cache warming, see Warming the data cache.

Memory demands during processing

During processing, memory is required to temporarily store fact data and aggregations prior to writing them to disk.

Processing Fact data

Processing uses a double-buffered scheme to read and process fact records from the source database. Analysis Services populates an initial buffer from the relational database, and then populates a second buffer from the initial buffer where the data is sorted, indexed, and written to the partition file in segments. Each segment consists of 65,536 rows; the number of bytes in each segment varies based on the size of each row.

The **OLAP\Process\BufferMemoryLimit** property controls the amount of memory that is used per processing job to store and cache data coming from a relational data source. This setting along with **OLAP\Process\BufferRecordLimit** determines the number of rows that can be processed in the buffers. The **OLAP\Process\BufferMemoryLimit** setting is interpreted as a percentage of total physical memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 60, which indicates that a maximum of 60% of the total physical memory can be used. For most deployments, the default value of **OLAP\Process\BufferMemoryLimit** provides sufficient processing performance. For more information on scenarios where it may be appropriate to change the value of this property, see Tuning memory for partition processing.

Building Aggregations

Analysis Services uses memory during the building of aggregations. Each partition has its own aggregation buffer space limit. Two properties control the size of the aggregation buffer:

- **OLAP\Process\AggregationMemoryLimitMax** is a server property that controls the maximum amount of memory that can be used for aggregation processing per partition processing job. This value is interpreted as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 80, which indicates that a maximum of 80% of the total physical memory can be used for aggregation processing.
- **OLAP\Process\AggregationMemoryLimitMin** is a server property that controls the minimum amount of memory that can be used for aggregation processing per partition processing job. This value is interpreted as a percentage of total memory if the value is less than 100, or an absolute value of bytes if the value is greater than 100. The default value is 10, which indicates that a minimum of 10% of the total physical memory will be used for aggregation processing

For a given partition, all aggregations are calculated at once. As a general best practice, it is a good idea to verify that all aggregations can fit into memory during creation; otherwise temporary files are used, which can slow down processing although the impact on performance is not as significant as in prior versions of Analysis Services. For more information on how to monitor the usage of temporary files, see Tuning memory for partition processing.

Optimizing memory usage

Optimizing memory usage for querying and processing operations requires supplying the server with adequate memory and verifying that the memory management properties are configured properly for your server environment. This section contains a summary of the guidelines that can help you optimize the memory usage of Analysis Services.

Increasing available memory

If you have one or more large or complex cubes and are using SQL Server 2005 (32-bit), use Windows Advanced Server® or Datacenter Server with SQL Server 2005 Enterprise Edition (or SQL Server 2005 Developer Edition) to enable Analysis Services to address up to 3 GB of memory. Otherwise, the maximum amount of memory that Analysis Services can address is 2 GB.

To enable Analysis Services to address more than 2 GB of physical memory with either of these editions, enable the Application Memory Tuning feature of Windows. To accomplish this, use the **/3GB** switch in the boot.ini file If you set the **/3GB** switch in the boot.ini file, the server should have at least 4 GB of memory to ensure that the Windows operating system also has sufficient memory for system services. If you run other applications on the server, you must factor in their memory requirements as well.

If you have one or more very large and complex cubes and your Analysis Services memory needs cannot be met within the 3-GB address space, SQL Server 2005 (64-bit) allows the Analysis Services process to access more than 3 GB of memory. You may also want to consider SQL Server 2005 (64-bit) in design scenarios where you have many partitions that you need to process in parallel or large dimensions that require a large amount of memory to process.

If you cannot add additional physical memory to increase performance, increasing the size of the paging files on the Analysis Services server can prevent out–of-memory errors when the amount of virtual memory allocated exceeds the amount of physical memory on the Analysis Services server.

Monitoring memory management

Given that the **Memory\TotalMemoryLimit** and **Memory\LowMemoryLimit** properties are percentages by default, they dynamically reflect the amount of physical memory on the server, even if you add new memory to the server. Using these default percentages is beneficial in deployments where Analysis Services is the only application running on your server.

If Analysis Services is installed in a shared application environment, such as if you have Analysis Services installed on the same machine as SQL Server, consider assigning static values to these properties as opposed to percentages in order to constrain Analysis Services memory usage. In shared application environments, it is also a good idea to constrain how other applications on the server use memory.

When modifying these properties, it is a good practice to keep the difference between the **Memory\LowMemoryLimit** and **Memory\TotalMemoryLimit** is at least five percent, so that the cleaner can smoothly transition across different levels of memory pressure.

You can monitor the memory management of the Analysis Services server by using the following performance counters displayed in Table 3.

Table 3 Memory management performance counters

Performance counter name	Definition
MSAS 2005:Memory\Memory Limit Low KB	Displays the Memory\LowMemoryLimit from the configuration file
MSAS 2005:Memory\Memory Limit High KB	Displays the Memory\TotalMemoryLimit from the configuration file.
MSAS 2005:Memory\Memory Usage KB	Displays the memory usage of the server process. This is the value that is compared to Memory\LowMemoryLimit and Memory\TotalMemoryLimit. Note that the value of this performance counter is the same value displayed by the Process\Private Bytes performance counter.
MSAS 2005:Memory\Cleaner Balance/sec	Shows how many times the current memory usage is compared against the settings. Memory usage is checked every 500ms, so the counter will trend towards 2 with slight deviations when the system is under high stress.
MSAS 2005:Memory\Cleaner Memory nonshrinkable KB	Displays the amount of memory, in KB, non subject to purging by the background cleaner.
MSAS 2005:Memory\Cleaner Memory shrinkable KB	Displays the amount of memory, in KB, subject to purging by the background cleaner.
MSAS 2005:Memory\Cleaner Memory KB	Displays the amount of memory, in KB, known to the background cleaner. (Cleaner memory shrinkable + Cleaner memory non-shrinkable.) Note that this counter is calculated from internal accounting information so there may be some small deviation from the memory reported by the operating system.

Minimizing metadata overhead

For each database, metadata is initialized and loaded into non-shrinkable memory. Once loaded into memory, it is not subject to purging by the cleaner thread.

To monitor the metadata overhead for each database

1. Restart the Analysis Services server.
2. Note the starting value of the **MSAS 2005:Memory\Memory Usage KB** performance counter.
3. Perform an operation that forces metadata initialization of a database, such as iterating over the list of objects in AMO browser sample application, or issuing a backup command on the database.

4. After the operation has completed, note the ending value of **MSAS 2005:Memory\Memory Usage KB.** The difference between the starting value and the ending value represents the memory overhead of the database.

For each database, you should see memory growth proportional to the number of databases you initialize. If you notice that a large amount of your server memory is associated with metadata overhead, you may want consider whether you can take steps to reduce the memory overhead of a given database. The best way to do this is to re-examine the design of the cube. An excessive number of dimensions attributes or partitions can increase the metadata overhead. Where possible you should follow the design best practices outlined in Optimizing the dimension design and Reducing attribute overhead.

Monitoring the timeout of idle sessions

Client sessions are managed in memory. In general, there is a one-to-one relationship between connections and sessions. While each connection consumes approximately 32 KB of memory, the amount of memory a given session consumes depends on the queries and calculations performed in that session. You can monitor the current number of user sessions and connections by using the **MSAS 2005:Connection\Current connections** and **MSAS 2005:Connection\Current user sessions** performance counters to evaluate the connection and session demands on your system.

As stated earlier, active sessions consume non-shrinkable memory, whereas expired sessions consume shrinkable memory. Two main properties determine when a session expires:

- The **MinIdleSessionTimeout** is the threshold of idle time in seconds after which the server can destroy a session based on the level of memory pressure. The **MinIdleSessionTimeout** is set to 2,700 seconds (45 minutes). This means that a session must be idle for 45 minutes before it is considered an expired session that the cleaner can remove when memory pressure thresholds are exceeded.
- The **MaxIdleSessionTimeout** is the time in seconds after which the server forcibly destroys an idle session regardless of memory pressure. By default, the **MaxIdleSessionTimeout** is set to zero seconds, meaning that the idle session is never forcibly removed by this setting.

In addition to the properties that manage expired sessions, there are properties that manage the longevity of sessions that lose their connection. A connectionless session is called an orphaned session.

- The **IdleOrphanSessionTimeout** is a server property of that controls the timeout of connection-less sessions. By default this property is set to 120 seconds (three minutes), meaning that if a session loses its connection and a reconnection is not made within 120 seconds, the Analysis Services server forcibly destroys this session.
- **IdleConnectionTimeout** controls the timeout of connections that have not been used for a specified amount of time. By default, this property is set to zero seconds. This means that the connection never times out. However, given that the connection cannot exist outside of a session, any idle connection will be cleaned up whenever its session is destroyed.

For most scenarios, these default settings provide adequate server management of sessions. However, there may be scenarios where you want finer-grained session management. For example, you may want to alter these settings according to the level amount of memory pressure that the Analysis Services server is experiencing. During busy periods of elevated memory pressure, you may want to destroy idle sessions after 15 minutes. At times when the server is not busy, you want the idle sessions to be destroyed after 45 minutes. To accomplish this, set the **MinIdleSessionTimeout** property to 900 seconds (15 minutes) and the **MaxIdleSessionTimeout** to 2,700 seconds.

Note that before changing these properties, it is important to understand how your client application manages sessions and connections. Some client applications, for example, have their own timeout mechanisms for connections and sessions that are managed independently of Analysis Services.

Tuning memory for partition processing

Tuning memory for partition processing involves three general techniques:

- Modifying the **OLAP\Process\BufferMemoryLimit** property as appropriate.
- Verifying that sufficient memory is available for building aggregations.
- Splitting up processing jobs in memory-constrained environments.

Modifying the OLAP\Process\BufferMemoryLimit property as appropriate

OLAP\Process\BufferMemoryLimit determines the size of the fact data buffers using during partition processing. While the default value of the **OLAP\Process\BufferMemoryLimit** is sufficient for many deployments, you may find it useful to alter the property in the following scenarios:

- If the granularity of your measure group is more summarized than the relational source fact table, generally speaking you may want to consider increasing the size of the buffers to facilitate data grouping. For example, if the source data has a granularity of day and the measure group has a granularity of month, Analysis Services must group the daily data by month before writing to disk. This grouping only occurs within a single buffer and it is flushed to disk once it is full. By increasing the size of the buffer, you decrease the number of times that the buffers are swapped to disk and also decrease the size of the fact data on disk, which can also improve query performance.
- If the OLAP measure group is of the same granularity as the source relational fact table, you may benefit from using smaller buffers. When the relational fact table and OLAP measure group are at roughly the same level of detail, there is no need to group the data, because all rows remain distinct and cannot be aggregated. In this scenario, assigning smaller buffers is helpful, allowing you to execute more processing jobs in parallel.

Verifying that sufficient memory is available for building aggregations

During processing, the aggregation buffer determines the amount of memory that is available to build aggregations for a given partition. If the aggregation buffer is too small, Analysis Services supplements the aggregation buffer with temporary files.

Temporary files are created in the **TempDir** folder when memory is filled and data is sorted and written to disk. When all necessary files are created, they are merged together to the final destination. Using temporary files can potentially result in some performance degradation during processing; however, the impact is generally not significant given that the operation is simply an external disk sort. Note that this behavior is different than in previous versions of Analysis Services.

To monitor any temporary files used during processing, review the **MSAS 2005:Proc Aggregations\Temp file bytes written/sec** or the **MSAS 2005:Proc Aggregations\Temp file rows written/sec** performance counters.

In addition, when processing multiple partitions in parallel or processing an entire cube in a single transaction, you must ensure that the total memory required does not exceed the **Memory\TotalMemoryLimit** property. If Analysis Services reaches the **Memory\TotalMemoryLimit** during processing, it does not allow the aggregation buffer to grow and may cause temporary files to be used during aggregation processing. Furthermore, if you have insufficient virtual address space for these simultaneous operations, you may receive out-of-memory errors. If you have insufficient physical memory, memory paging will occur. If processing in parallel and you have limited resources, consider doing less in parallel.

Splitting up processing jobs in memory-constrained environments

During partition processing in memory-constrained environments, you may encounter a scenario where a **ProcessFull** operation on a measure group or partition cannot proceed due to limited memory resources. What is happening in this scenario is that the Process job requests an estimated amount of memory to complete the total **ProcessFull** operation. If the Analysis Services memory governor cannot secure enough memory for the job, the job can either fail or block other jobs as it waits for more memory to become available. As an alternative to performing a **ProcessFull**, you can split the processing operation into two steps by performing two operations serially: **ProcessData** and **ProcessIndexes**. In this scenario, the memory request will be smaller for each sequential operation and is less likely to exceed the limits of the system resources.

Warming the data cache

During querying, memory is primarily used to store cached results in the data and calculation caches. To optimize the benefits of caching, you can often increase query responsiveness by preloading data into the data cache by executing a generalized set of representative user queries. This process is called *cache warming*. To do this, you can create an application that executes a set of generalized queries to simulate typical user activity in order to expedite the process of populating the query results cache. For example, if you determine that users are querying by month and by product, you can create a set of queries that request data by product and by month. If you run this query whenever you start Analysis Services, or process the measure group or one of its partitions, this will pre-load the query results cache with data used to resolve these queries before users submit these types of query. This technique substantially improves Analysis Services response times to user queries that were anticipated by this set of queries.

To determine a set of generalized queries, you can use the Analysis Services query log to determine the dimension attributes typically queried by user queries. You can use an application, such as a Microsoft Excel macro, or a script file to warm the cache whenever you have performed an operation that flushes the query results cache. For example, this application could be executed automatically at the end of the cube processing step.

Running this application under an identifiable user name enables you to exclude that user name from the Usage-Based Optimization Wizard's processing and avoid designing aggregations for the queries submitted by the cache warming application.

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. You can empty the results cache using a simple XMLA command such as the following:

```
<Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
  <ClearCache>
    <Object>
      <DatabaseID>Adventure Works DW</DatabaseID>
    </Object>
  </ClearCache>
</Batch>
```

This example XMLA command clears the cache for the Adventure Works DW database. To execute the ClearCache statement, you can either manually run the XMLA statement in SQL Server Management Studio or use the ASCMD tool command-line utility to execute any XMLA script.

Understanding how Analysis Services uses CPU resources

Analysis Services uses processor resources for both querying and processing. Increasing the number and speed of processors can significantly improve processing performance and, for cubes with a large number of users, improve query responsiveness as well.

Job architecture

Analysis Services uses a centralized job architecture to implement querying and processing operations. A job itself is a generic unit of processing or querying work. A job can have multiple levels of nested child jobs depending on the complexity of the request.

During processing operations, for example, a job is created for the object that you are processing, such as a dimension. A dimension job can then spawn several child jobs that process the attributes in the dimension. During querying, jobs are used to retrieve fact data and aggregations from the partition to satisfy query requests. For example, if you have a query that accesses multiple partitions, a parent job is generated for the query itself along with one or more child jobs per partition.

Generally speaking, executing more jobs in parallel has a positive impact on performance as long as you have enough processor resources to effectively handle the concurrent operations as well as sufficient memory and disk resources. The maximum number of jobs that can execute in parallel across all server operations (including both processing and querying) is determined by the **CoordinatorExecutionMode** property.

- A negative value for **CoordinatorExecutionMode** specifies the maximum number of parallel jobs that can start per processor.
- A value of zero enables the server to automatically determine the maximum number of parallel operations, based on the workload and available system resources.
- A positive value specifies an absolute number of parallel jobs that can start per server.

The default value for the **CoordinatorExecutionMode** is -4, which indicates that four jobs will be started in parallel per processor. This value is sufficient for most server environments. If you want to increase the level of parallelism in your server, you can increase the value of this property either by increasing the number of jobs per processor or by setting the property to an absolute value. While this globally increases the number of jobs that can execute in parallel, **CoordinatorExecutionMode** is not the only property that influences parallel operations. You must also consider the impact of other global settings such as the **MaxThreads** server properties that determine the maximum number of querying or processing threads that can execute in parallel. In addition, at a more granular level, for a given processing operation, you can specify the maximum number of processing tasks that can execute in parallel using the **MaxParallel** command. These settings are discussed in more detail in the sections that follow.

Thread pools

To effectively manage processor resources for both querying and processing operations, Analysis Services 2005 uses two thread pools:

- **Querying thread pool**—The querying thread pool controls the worker threads used by the Query Execution Engine to satisfy query requests. One thread from the querying pool is used per concurrent query. The minimum number of threads from the querying pool is determined by the value of the **ThreadPool\Query\MinThreads** property; its default setting is 1. The maximum number of worker threads maintained in the querying thread pool is determined by the value of **ThreadPool\Query\MaxThreads**; its default setting is 10.
- **Processing thread pool**—The processing thread pool controls the worker threads used by the Storage Engine during processing operations. The processing thread pool is also used during querying to control the threads used by the Storage Engine to retrieve data from disk. The **ThreadPool\Process\MinThreads** property determines the minimum number of processing threads that can be maintained at a given time. The default value of this property is 1. The **ThreadPool\Process\MaxThreads** property determines the maximum number of processing threads that can be maintained at a given time. The default value of this property is 64.

For scenarios on when these values should be changed, see Optimizing CPU usage. Before you modify these properties, it is useful to examine how these threads are used during querying and processing.

Processor demands during querying

During querying, to manage client connections, Analysis Services uses a listener thread to broker requests and create new server connections as needed. To satisfy query requests, the listener thread manages worker threads in the querying thread pool and the processing thread pool, assigns worker threads to specific requests, initiates new worker threads if there are not enough active worker threads in a given pool, and terminates idle worker threads as needed.

To satisfy a query request, the thread pools are used as follows:

1. Worker threads from the query pool check the data and calculation caches respectively for any data and/or calculations pertinent to a client request.
2. If necessary, worker threads from the processing pool are allocated to retrieve data from disk.
3. Once data is retrieved, worker threads from the querying pool store the results in the query cache to resolve future queries.
4. Worker threads from the querying pool perform necessary calculations and use a calculation cache to store calculation results.

The more threads that are available to satisfy queries, the more queries that you can execute in parallel. This is especially important in scenarios where you have a large number of users issuing queries. For more information on how to optimize processor resources during querying, see Maximize parallelism during querying.

Processor demands during processing

Where possible, Analysis Services naturally performs all processing operations in parallel. For every processing operation, you can specify the parallelism of the Analysis

Services object by using the **MaxParallel** processing command. By default, the **MaxParallel** command is configured to **Let the server decide**, which is interpreted as unlimited parallelism, constrained only by hardware and server workload. For more information on how you can change this setting, see Maximize parallelism during processing.

Of all of the processing operations, partitions place the largest demands on processor resources. Each partition is processed in two stages and each stage is a multithreaded activity.

- During the first stage of processing a partition, Analysis Services populates an initial buffer from the relational database, populates a second buffer from the initial buffer, and then writes segments to the partition file. Analysis Services utilizes multiple threads for this stage, which execute asynchronously. This means that while data is being added to the initial buffer, data is being moved from the initial buffer into the second buffer and sorted into segments. When a segment is complete, it is written to the partition file. Processor usage during this first phase depends on the speed of the data transfer from the relational tables. Generally this stage is not particularly processor-intensive, using less than one processor. Rather, this stage is generally limited by the speed of retrieving data from the relational database. The maximum size of the buffer used to store the source data is determined by the **OLAP\Process\BufferMemoryLimit** and **OLAP\Process\BufferRecordLimit** server properties. In some scenarios, it can be beneficial to modify these settings to improve processing performance. For more information on these properties, see Memory demands during processing.
- During the second stage, Analysis Services creates and computes aggregations for the data. Analysis Services utilizes multiple threads for this stage, executing these tasks asynchronously. These threads read the fact data into an aggregation buffer. If sufficient memory is allocated to the aggregation buffer, these aggregations are calculated entirely in memory. As stated previously in the Memory demands during processing section, if Analysis Services does not have sufficient memory to calculate aggregations, Analysis Services uses temporary files to supplement the aggregation buffer. This stage can be processor-intensive; Analysis Services takes advantage of multiple processors if they are available.

Optimizing CPU usage

While adding additional processor resources can improve the overall performance of Analysis Services, use the following guidelines to optimize the usage of processor resources.

Maximize parallelism during querying

As stated in the Thread pools section, **Threadpool\Query\MaxThreads** determines the maximum number of worker threads maintained in the querying thread pool. The default value of this property is 10. For servers that have more than one processor, to increase parallelism during querying, consider modifying **Threadpool\Query\MaxThreads** to be a number dependent on the number of server processors. A general recommendation is to set the **Threadpool\Query\MaxThreads** to a value of less than or equal to 2 times the number of processors on the server. For example, if you have an eight-processor machine, the general guideline is to set this value to no more than 16. In practical terms, increasing **Threadpool\Query\MaxThreads** will not significantly increase the performance of a given query. Rather, the benefit of increasing this property is that you can increase the number of queries that can be serviced concurrently.

Since querying also involves retrieving data from partitions, to improve parallel query operations, you must also consider the maximum threads available in the processing pool as specified by the **Threadpool\Process\MaxThreads** property. By default, this property has a value of 64. While partitions are naturally queried in parallel, when you have many queries that require data from multiple partitions, you can enhance data retrieval by changing the **Threadpool\Process\MaxThreads** property. When modifying this property, a general recommendation is to set the **Threadpool\Process\MaxThreads** to a value of less than or equal to 10 times the number of processors on the machine. For example, if you have an eight-processor server, the general guideline is setting this value to no more than 80. Note even though the default value is 64, if you have fewer than eight processors on a given server, you do not need to reduce the default value to throttle parallel operations. As you consider the scenarios for changing the **Threadpool\Process\MaxThreads** property, remember that changing this setting impacts the processing thread pool for both querying and processing. For more information on how this property specifically impacts processing operations, see Maximizing parallelism during processing.

While modifying the **Threadpool\Process\MaxThreads** and **Threadpool\Query\MaxThreads** properties can increase parallelism during querying, you must also take into account the additional impact of the **CoordinatorExecutionMode**. Consider the following example. If you have a four-processor server and you accept the default **CoordinatorExecutionMode** setting of -4, a total of 16 jobs can be executed at one time across all server operations. So if ten queries are executed in parallel and require a total of 20 jobs, only 16 jobs can launch at a given time (assuming that no processing operations are being performed at that time). When the job threshold has been reached, subsequent jobs wait in a queue until a new job can be created. Therefore, if the number of jobs is the bottleneck to the operation, increasing the thread counts may not necessarily improve overall performance.

In practical terms, the balancing of jobs and threads can be tricky. If you want to increase parallelism, it is important to assess your greatest bottleneck to parallelism, such as the number of concurrent jobs and/or the number of concurrent threads, or both. To help you determine this, it is helpful to monitor the following performance counters:

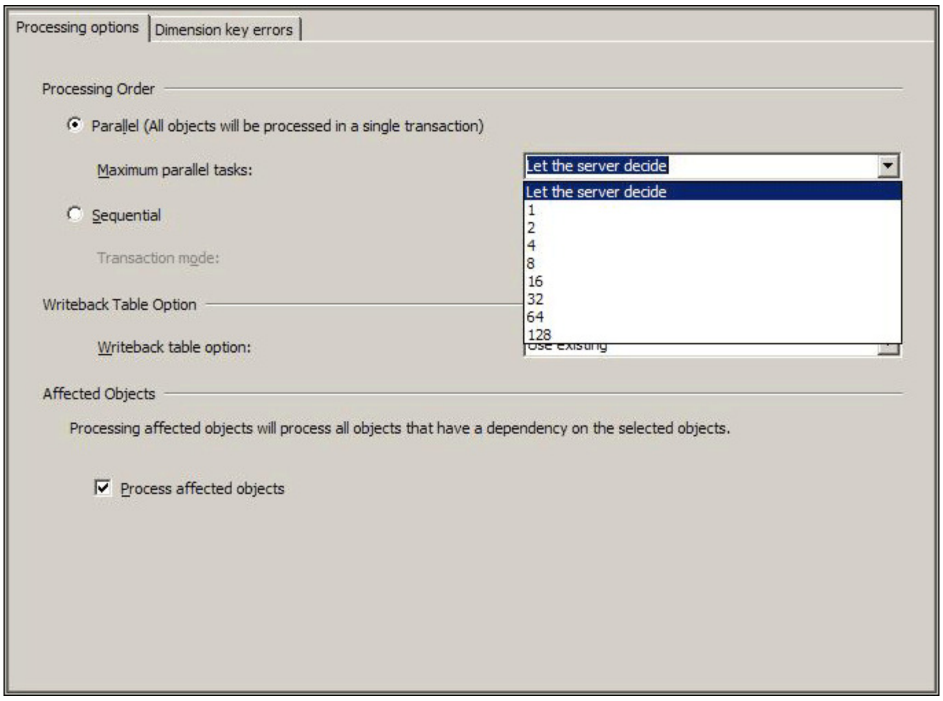
- **MSAS 2005: Threads\Query pool job queue length**—The number of jobs in the queue of the query thread pool. A non-zero value means that the number of query jobs has exceeded the number of available query threads. In this scenario, you may consider increasing the number of query threads. However, if CPU utilization is already very high, increasing the number of threads will only add to context switches and degrade performance.
- **MSAS 2005: Threads\Query pool busy threads**—The number of busy threads in the query thread pool.

- **MSAS 2005: Threads\Query pool idle threads**—The number of idle threads in the query thread pool.

Maximize parallelism during processing

For processing operations, you can use the following mechanisms to maximize parallelism:

- **CoordinatorExecutionMode**—As stated earlier, this server-wide property controls the number of parallel operations across the server. If you are performing processing at the same time as querying, it is a good practice to increase this value.
- **Threadpool\Process\MaxThreads**—Also discussed earlier in this section, this server-wide property increases the number of threads that can be used to support parallel processing operations.
- **MaxParallel** processing command—Rather than globally specifying the number of parallel operations for a given Analysis Services instance, for every processing operation, you can specify the maximum number of tasks that can operate in parallel. In many scenarios, this is the most common setting that is used to affect parallelism. You can specify the **MaxParallel** command in two ways: the **Maximum parallel tasks** option in the processing user interface or a custom XMLA script.
    - **Maximum parallel tasks** option—When you launch a processing operation from SQL Server Management Studio or Business Intelligence Development Studio, you can specify the **Maximum parallel tasks** option to change the level of parallelism for a given processing operation as displayed in Figure 31. The default value of this setting is **Let the server decide**, which is interpreted as unlimited parallelism, constrained only by hardware and server workload. The drop-down list displays a list of suggested values but you can specify any value. If you increase this value to increase parallelism, be wary of setting the property too high. Performing too many parallel operations at once can be counterproductive if it causes context switching and degrades performance.


Processing options | Dimension key errors

Processing Order

(●) Parallel (All objects will be processed in a single transaction)

Maximum parallel tasks: Let the server decide

Let the server decide
1
2
4
8
16
32
64
128

( ) Sequential

Transaction mode:

Writeback Table Option

Writeback table option: Use existing

Affected Objects

Processing affected objects will process all objects that have a dependency on the selected objects.

[✓] Process affected objects

Figure 31 Maximum parallel tasks setting

- **Custom XMLA script** —As an alternative to specifying the **MaxParallel** command in the user interface, you can write a custom XMLA script to perform a processing operation and use the **MaxParallel** element to control the number of parallel operations within the XMLA script.

When processing multiple partitions in parallel, use the guidelines displayed in Table 4 for the number of partitions that can be processed in parallel according to the number of processors. These guidelines were taken from processing tests performed using Project REAL cubes.

Table 4 Partition processing guidelines

# of Processors	# of Partitions to be processed in parallel
4	2 – 4
8	4 – 8
16	6 – 16

Note that the actual number of partitions that can be processed in parallel depends on the querying workload and design scenario. For example, if you are performing querying and processing at the same time, you may want to decrease the number of partitions processed in parallel in order to keep some free resources for querying. Alternatively, if your design contains SQL queries with many complex joins, your parallel partition processing performance could be limited by the source database. If the source database is on the same machine as Analysis Services, you may see memory and CPU interactions that limit of the benefits of parallel operations. In fact, with too much parallelism you can overload the RDBMS so much that it leads to timeout errors, which cause processing to fail. By default, the maximum number of concurrent connections, and thus queries, for a data source is limited to ten. This can be changed by altering the **Maximum Number of Connections** setting of the data source properties in either Business Intelligence Development Studio or SQL Server Management Studio.

To help you monitor the number of partitions processing in parallel, you can review the **MSAS 2005:Processing\Rows read/sec** performance counter. Generally you should expect this counter to display 40,000–60,000 rows per second for one partition. If your partition contains complex SQL joins or hundreds of source columns, you are likely to see a lower rate. Additionally, you can monitor the number of threads being used during processing by using the **MSAS 2005: Threads\Processing pool busy threads** performance counter. You can also view jobs that are waiting to execute by using the **MSAS 2005: Threads\Processing pool job queue length** performance counter.

Note that when you perform parallel processing of any object, all parallel operations are committed in one transaction. In other words, it is not possible to perform a parallel execution and then commit each transaction as it progresses. While this is not specifically a performance issue, it does impact your processing progress. If you encounter any errors during processing, the entire transaction rolls back.

Use sufficient memory

The Optimizing memory usage section describes techniques to ensure that Analysis Services has sufficient memory to perform querying and processing operations. Ensuring that Analysis Services has sufficient memory can also impact Analysis Services usage of processor resources. If the Analysis Services server has sufficient memory, the Windows operating system will not need to page memory from disk. Paging reduces processing performance and query responsiveness.

Use a load-balancing cluster

If your performance bottleneck is processor utilization on a single system as a result of a multi-user query workload, you can increase query performance by using a cluster of Analysis Services servers to service query requests. Requests can be load balanced across two Analysis Services servers, or across a larger number of Analysis Services servers to support a large number of concurrent users (this is called a *server farm*). Load-balancing clusters generally scale linearly. Both Microsoft and third-party vendors provide cluster solutions. The Microsoft load-balancing solution is Network Load Balancing (NLB), which is a feature of the Windows Server operating system. With NLB, you can create an NLB cluster of Analysis Services servers running in multiple host mode. When an NLB cluster of Analysis Services servers is running in multiple host mode, incoming requests are load balanced among the Analysis Services servers. When you use a load-balancing cluster, be aware that the data caches on each of the servers in the load-balancing cluster will be different, resulting in differences in query response times from query to query by the same client.

A load-balancing cluster can also be used to ensure availability in the event that a single Analysis Services server fails. An additional option for increasing performance with a load-balancing cluster is to distribute processing tasks to an offline server. When new data has been processed on the offline server, you can update the Analysis Services servers in the load-balancing cluster by using Analysis Services database synchronization.

If your users submit a lot of queries that require fact data scans, a load-balancing cluster may be a good solution. For example, queries that may require a large number of fact data scans include wide queries (such as top count or medians), and random queries against very complex cubes where the probability of hitting an aggregation is very low.

However, a load-balancing cluster is generally not needed to increase Analysis Services performance if aggregations are being used to resolve most queries. In other words, concentrate on good aggregation and partitioning design first. In addition, a load-balancing cluster does not solve your performance problem if processing is the bottleneck or if you are trying to improve an individual query from a single user. Note that one restriction to using a load-balancing cluster is the inability to use writeback, because there is no single server to which to write back the data.

Understanding how Analysis Services uses disk resources

Analysis Services uses disk I/O resources for both querying and processing. Increasing the speed of your disks, spreading the I/O across multiple disks, and using multiple controllers, can significantly improve processing performance. These steps also significantly improve query responsiveness when Analysis Services is required to perform fact data scans. If you have a large number of queries that require fact data scans, Analysis Services can become constrained by insufficient disk I/O when there is not enough memory to support the file system cache in addition to Analysis Services memory usage.

Disk resource demands during processing

As stated previously in the Memory demands during processing section, during processing, the aggregation buffer determines the amount of memory that is available to build aggregations for a given partition. If the aggregation buffer is too small, Analysis Services uses temporary files. Temporary files are created in the **TempDir** folder when memory is filled and data is sorted and written to disk. When all necessary files are created, they are merged together to the final destination. Using temporary files can result in some performance degradation during processing; however, the impact is generally not significant given that the operation is simply an external disk sort. Note that this behavior is different than previous versions of Analysis Services. To monitor any temporary files used during processing, review the **MSAS 2005:Proc Aggregations\Temp file bytes written/sec** or the **MSAS 2005:Proc Aggregations\Temp file rows written/sec** performance counters.

Disk resource demands during querying

During querying, Analysis Services may request arbitrary parts of the data set, depending on user query patterns. When scanning a single partition, the I/Os are essentially sequential, except that large chunks may be skipped because the indexes may indicate that they aren't needed. If commonly used portions of the cube (particularly the mapping files) fit in the file system cache, the Windows operating system may satisfy the I/O requests from memory rather than generating physical I/O. With large cubes, using a 64-bit version of the Microsoft Windows Server 2003 family increases the amount of memory that the operating system can use to cache Analysis Services requests. With sufficient memory, much of the cube can be stored in the file system cache.

Optimizing disk usage

While increasing disk I/O capacity can significantly improve the overall performance of Analysis Services, there are several steps you can take to use existing disk I/O more effectively. This section contains guidelines to help you optimize disk usage of Analysis Services.

Using sufficient memory

The Optimizing memory usage section describes techniques to ensure that Analysis Services has sufficient memory to perform querying and processing operations. Ensuring that Analysis Services has sufficient memory can also impact Analysis Services usage of disk resources. For example, if there is not enough memory to complete processing operations, Analysis Services uses temporary files, generating disk I/O.

If you cannot add sufficient physical memory to avoid memory paging, consider creating multiple paging files on different drives to spread disk I/O across multiple drives when memory paging is required.

Optimizing file locations

The following techniques can help you to optimize the data files and temporary files used during processing:

- Place the Analysis Services data files on a fast disk subsystem.

  The location of the data files is determined by the **DataDir** server property. To optimize disk access for querying and processing, place the Analysis Services Data folder on a dedicated disk subsystem (RAID 5, RAID 1+0, or RAID 0+1).

- If temporary files are used during processing, optimize temporary file disk I/O.

  The default location of the temporary files created during aggregation processing is controlled by the **TempDir** property. If a temporary file is used, you can increase processing performance by placing this temporary folder on a fast disk subsystem (such as RAID 0 or RAID 1+0) that is separate from the data disk.

Disabling unnecessary logging

Flight Recorder provides a mechanism to record Analysis Services server activity into a short-term log. Flight Recorder provides a great deal of benefit when you are trying to troubleshoot specific querying and processing problems; however, it introduces a certain amount of I/O overheard. If you are in a production environment and you do not require Flight Recorder capabilities, you can disable its logging and remove the I/O overhead.

The server property that controls whether Flight Recorder is enabled is the **Log\Flight Recorder\Enabled** property. By default, this property is set to **true**.

Conclusion

For more information:

http://www.microsoft.com/technet/prodtechnol/sql/2005/technologies/ssasvcs.mspx

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), how would you rate this paper?

Appendix A – For More Information

The following white papers might be of interest.

- Analysis Services 2005 Migration
- Real Time Business Intelligence with Analysis Services 2005
- Strategies for Partitioning Relational Data Warehouses in Microsoft SQL Server
- Analysis Services 2005 Processing Architecture
- Introduction to MDX Scripting in Microsoft SQL Server 2005
- Project REAL Monitoring and Instrumentation
- Project REAL: Analysis Services Technical Drilldown
- Project REAL: Enterprise Class Hardware Tuning for Microsoft Analysis Services

Appendix B - Partition Storage Modes

Each Analysis Services partition can be assigned a different storage mode that specifies where fact data and aggregations are stored. This appendix describes the various storage modes that Analysis Services provides: multidimensional OLAP (termed MOLAP), hybrid OLAP (HOLAP), and relational OLAP (ROLAP). Generally speaking. MOLAP provides the fastest query performance; however, it typically involves some degree of data latency.

In scenarios where you require near real-time data refreshes and the superior query performance of MOLAP, Analysis Services provides proactive caching. Proactive caching is an advanced feature that requires a special set of performance tuning techniques to ensure that it is applied effectively. For more information on the performance considerations of using proactive caching, see Near real-time data refreshes in this white paper.

Multidimensional OLAP (MOLAP)

MOLAP partitions store aggregations and a copy of the source data (fact and dimension data) in a multidimensional structure on the Analysis Services server. All partitions are stored on the Analysis Services server.

Analysis Services responds to queries faster with MOLAP than with any other storage mode for the following reasons:

- **Compression**—Analysis Services compresses the source fact data and its aggregations to approximately 30 percent of the size of the same data stored in a relational database. The actual compression ratio varies based on a variety of factors, such as the number of duplicate keys and bit encoding algorithms. This reduction in storage size enables Analysis Services to resolve a query against fact data or aggregations stored in a MOLAP structure much faster than against data and aggregations stored in a relational structure because the size of the physical data being retrieved from the hard disk is smaller.

- **Multidimensional data structures**—Analysis Services uses native multidimensional data structures to quickly find the fact data or aggregations. With ROLAP and HOLAP partitions, Analysis Services relies on the relational engine to perform potentially large table joins against fact data stored in the relational database to resolve some or all queries. Large table joins against relational structures take longer to resolve than similar queries against the MOLAP structures.
- **Data in a single service**—MOLAP partitions are generally stored on a single Analysis Services server, with the relational database frequently stored on a server separate from the Analysis Services server. When the relational database is stored on a separate server and partitions are stored using ROLAP or HOLAP, Analysis Services must query across the network whenever it needs to access the relational tables to resolve a query. The impact of querying across the network depends on the performance characteristics of the network itself. Even when the relational database is placed on the same server as Analysis Services, inter-process calls and the associated context switching are required to retrieve relational data. With a MOLAP partition, calls to the relational database, whether local or over the network, do not occur during querying.

Hybrid OLAP (HOLAP)

HOLAP partitions store aggregations in a multidimensional structure on the Analysis Services server, but leave fact data in the original relational database. As a result, whenever Analysis Services needs to resolve a query against fact data stored in a HOLAP partition, Analysis Services must query the relational database directly rather than querying a multidimensional structure stored on the Analysis Services server. Furthermore, Analysis Services must rely on the relational engine to execute these queries. Querying the relational database is slower than querying a MOLAP partition because of the large table joins generally required.

Some administrators choose HOLAP because HOLAP appears to require less total storage space while yielding excellent query performance for many queries. However, these apparent justifications for using HOLAP storage option are negated by the likelihood of excessive aggregations and additional indexes on relational tables.

- **Excessive aggregations**—Query responsiveness with HOLAP partitions relies on the existence of appropriate aggregations so that Analysis Services does not have to resolve queries against the fact table in the relational database. To ensure that a wide range of aggregations exists, administrators sometimes resort to generating excessive aggregations by increasing the performance improvement percentage in the Aggregation Design Wizard, or artificially increasing the partition row counts (and sometimes both). While these techniques increase the percentage of queries that Analysis Services can resolve using aggregations, there will always be some queries that can only be resolved against the fact data (remember the one-third rule). In addition, generating additional aggregations to improve query responsiveness comes at the cost of significantly longer processing times and increased storage requirements (which also negates the space savings).
- **Additional indexes on relational tables**—To ensure that the relational engine can quickly resolve queries that Analysis Services must resolve against the fact table in the relational database, administrators often add appropriate indexes to the fact and dimension tables. These additional indexes frequently require more space than

MOLAP requires to store the entire cube. The addition of these indexes negates the apparent savings in disk space that is sometimes used to justify HOLAP. In addition, maintaining the indexes on the relational tables slows the relational engine when adding new data to the relational tables.

From a processing perspective, there is no significant difference in processing performance between MOLAP partitions and HOLAP partitions. In both cases, all fact data is read from the relational database, and aggregations are calculated. With MOLAP, Analysis Services writes the fact data into the MOLAP structure. With HOLAP, Analysis Services does not store fact data. This difference has minimal impact on processing performance, but can have a significant impact on query performance. Because HOLAP and MOLAP processing speeds are approximately the same and MOLAP query performance is superior, MOLAP is the optimum storage choice.

Relational OLAP (ROLAP)

ROLAP partitions store aggregations in the same relational database that stores the fact data. By default, ROLAP partitions store dimensions in MOLAP on the Analysis Services server, although the dimensions can also be stored using ROLAP in the relational database (for very large dimensions). Analysis Services must rely on the relational engine to resolve all queries against the relational tables, storing both fact data and aggregations. The sheer number of queries with large table joins in large or complex cubes frequently overwhelms the relational engine.

Given the slower query performance of ROLAP, the only situation in which ROLAP storage should be used is when you require reduced data latency and you cannot use proactive caching. For more information on proactive caching, see Near real-time data refreshes in this white paper. In this case, to minimize the performance cost with ROLAP, consider creating a small near real-time ROLAP partition and create all other partitions using MOLAP. Using MOLAP for the majority of the partitions in a near real-time OLAP solution allows you to optimize the query responsiveness of Analysis Services for most queries, while obtaining the benefits of real-time OLAP.

From a processing perspective, Analysis Services can store data, create MOLAP files, and calculate aggregations faster than a relational engine can create indexes and calculate aggregations. The primary reason the relational engine is slower is due to the large table joins that the relational engine must perform during the processing of a ROLAP partition. In addition, because the relational engine performs the actual processing tasks, competing demands for resources on the computer hosting the relational tables can negatively affect processing performance for a ROLAP partition.

Appendix C – Aggregation Utility

As a part of the Analysis Services 2005 Service Pack 2 samples, the Aggregation Utility is an advanced tool that complements the Aggregation Design Wizard and the Usage-Based Optimization Wizard by allowing you to create custom aggregation designs without using the aggregation design algorithm. This is useful in scenarios where you need to override the algorithm and create a specific set of aggregations to tune your query workload. Rather than relying on the cost/benefit analysis performed by the algorithm, you must decide which aggregations are going to be most effective to improve query performance without negatively impacting processing times.

Benefits of the Aggregation Utility

The Aggregation Utility enables you to complete the following tasks.

View and modify specific aggregations in an existing aggregation design.

Using the Aggregation Utility, you can view, add, delete, and change individual aggregations in existing designs. Once you build an aggregation design using the Aggregation Design Wizard or Usage-Based Optimization Wizard, you can use the utility to view the attributes that make up each aggregation. In addition, you have the ability to modify an individual aggregation by changing the attributes that participate in the aggregation.

Create new aggregation designs.

You can either create new aggregation designs by manually selecting the attributes for the aggregations, or by using the utility to build aggregations based on the query log. Note that the Aggregation Utility's ability to build aggregations from the query log is very different than the functionality of the Usage-Based Optimization Wizard. Remember that the Usage-Based Optimization Wizard reads data from the query log and then uses the aggregation design algorithm to determine whether or not an aggregation should be built. While the Usage-Based Optimization Wizard gives greater consideration to the attributes contained in the query log, there is no absolute guarantee that they will be built.

When you use the Aggregation Utility to build new aggregations from the query log, you decide which aggregations provide the most benefit for your query performance without negatively impacting processing times. In other words, you are no longer relying on the aggregation design algorithm to select which aggregations are built. To help you make effective decisions, the utility enables you to optimize your design, including the ability to remove redundancy, eliminate duplicates, and remove large aggregations that are close to the size of the fact table.

Review whether aggregations are flexible or rigid.

A bonus of the Aggregation Utility is the ability to easily identify whether an aggregation is flexible or rigid. By default, aggregations are flexible. Remember that in a flexible aggregation, one or more attributes have flexible relationships while in a rigid aggregation, all attributes have rigid relationships. If you want change an aggregation from flexible to rigid, you must first change all of the necessary attribute relationships. Once you make these changes, you can use the utility to confirm that you have been successful as the aggregation will now be identified as rigid. Without the utility, you need to manually review the aggregation files in the operating system to determine whether they were flexible or rigid. For more information on rigid and flexible aggregations, see Evaluating rigid vs. flexible aggregations in this white paper.

How the Aggregation Utility organizes partitions

Using the Aggregation Utility, you can connect to an instance of Analysis Services and manage aggregation designs across all of the cubes and databases in that instance. For each measure group, the Aggregation Utility groups partitions by their aggregation designs. Figure 32 displays an example of this grouping.

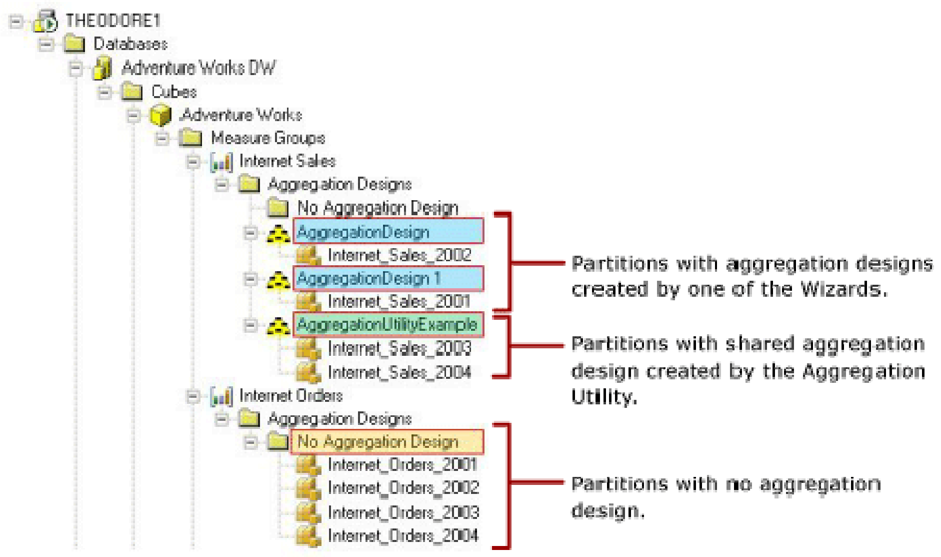


Figure 32 Aggregation display for the Internet Sales measure group

The partitions in Figure 32 are grouped as follows:

- **Aggregation design created by a wizard**—Aggregation designs created by the Aggregation Design Wizard or the Usage-Based Optimization Wizard are automatically named AggregationDesign, with an optional number suffix if more than one Wizard-created aggregation design exists per measure group. For example, in the Internet Sales measure group, the Internet_Sales_2002 partition has an aggregation design named AggregationDesign, and the Internet_Sales_2001 partition contains an aggregation design named AggregationDesign 1. Both AggregationDesign and AggregationDesign 1 were created by one of the wizards.
- **Aggregation design created by the Aggregation Utility**—The Internet_Sales_2003 and Internet_Sales_2004 partitions share an aggregation design, called AggregationUtilityExample. This aggregation design was created by the Aggregation Utility. The Aggregation Utility allows you to provide a custom name for each aggregation design.
- **No Aggregation Design**—For the Internet Orders measure group, none of the partitions in that measure group have an aggregation design yet.

How the Aggregation Utility works

The most common scenario for using the Aggregation Utility is to design aggregations based on a query log. Following is a list of steps to effectively use the Aggregation Utility to design new aggregations based on the query log.

To add new aggregations based on the query log

1. **Perform pre-requisite set up tasks**.

   Before using the Aggregation Utility, you must configure Analysis Services query logging just as you would before you use the Usage-Based Optimization Wizard. As you set up query logging, pay close attention to configure an appropriate value for the **QueryLogSampling** property. The default value of this property is set to one out of every ten queries. Depending on your query workload, you may need to increase this value in order to collect a representative set of queries in the Analysis Services query log table. Obtaining a good sampling of queries is critical to effectively using the Aggregation Utility.

2. **Add a new aggregation design based on a query log**.

   To extract data from the query log table, the Aggregation Utility provides a default query that returns a distinct list of datasets for a given partition. A dataset is the subcube that is used to satisfy query requests. An example of the default query is depicted in the query below. The values highlighted in yellow are placeholder values.

   ```
   Select distinct dataset from OLAPQueryLog
   Where MSOLAP_Database   = DatabaseName and
         MSOLAP_ObjectPath = MeasureGroupName
   ```

   Generally speaking, it is a good idea to modify the default SQL statement to apply additional filters that restrict the records based on **Duration** or **MSOLAP_User**. For example, you may only return queries where the **Duration** > 30 seconds or **MSOLAP_User** = Joe.

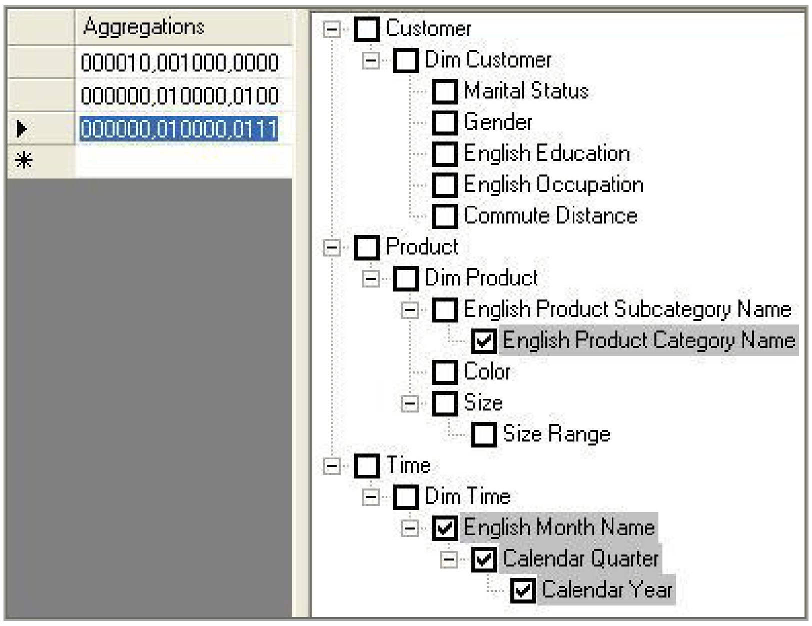
   In addition, whenever you add a new aggregation design by using the Aggregation Utility, it is a good idea to use a special naming convention to name the aggregation design as well as the aggregation prefix. This allows you to easily identify those aggregations that have been created by the utility. For example, when you use SQL Server Profiler to analyze the effectiveness of your aggregations, with easily recognizable names, you will be able to quickly identify those aggregations that have been created by the Aggregation Utility.

3. **Eliminate redundant aggregations.**

   You can optimize a new aggregation design by eliminating redundant aggregations. Redundant aggregations are those aggregations that include one or more attributes in the same attribute relationship tree.

   The aggregation highlighted in Figure 33 identifies an aggregation with attributes from two dimensions: Product and Time. From the product dimension, the aggregation includes the English Product Category Name attribute. From the Time dimension, the aggregation includes the following attributes: English Month Name, Calendar Quarter, and Calendar Year. This is a redundant aggregation since English

Month Name, Calendar Quarter, and Calendar Year are in the same attribute relationship tree.

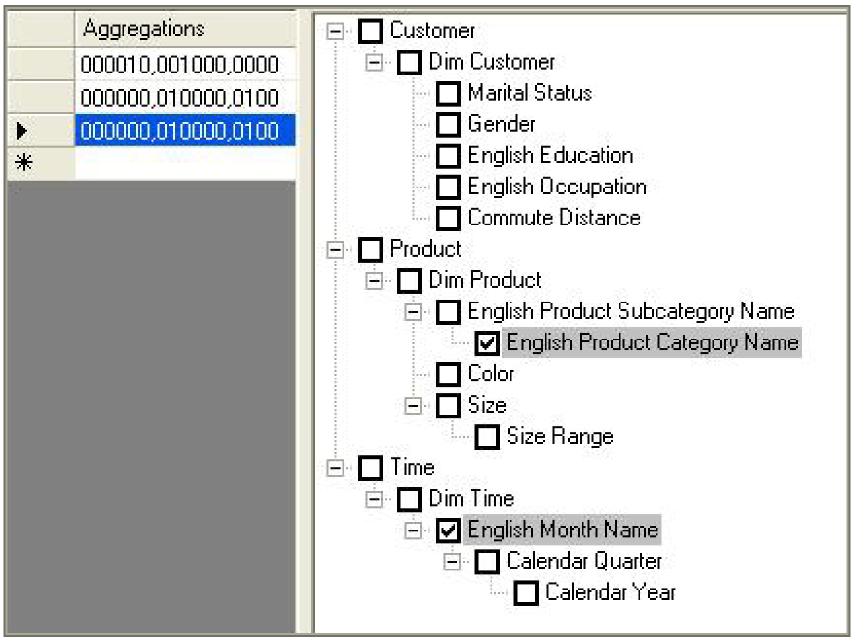


	Aggregations
	000010,001000,0000
	000000,010000,0100
▶	000000,010000,0111
*	

- [ ] Customer
  - [ ] Dim Customer
    - [ ] Marital Status
    - [ ] Gender
    - [ ] English Education
    - [ ] English Occupation
    - [ ] Commute Distance
- [ ] Product
  - [ ] Dim Product
    - [ ] English Product Subcategory Name
      - [✓] English Product Category Name
    - [ ] Color
    - [ ] Size
      - [ ] Size Range
- [ ] Time
  - [ ] Dim Time
    - [✓] English Month Name
      - [✓] Calendar Quarter
        - [✓] Calendar Year

Figure 33 Redundant aggregation example

To remove the redundancy in this aggregation, use the **Eliminate Redundancy** option in the Aggregation Utility. Figure 34 displays the aggregation after the **Eliminate Redundancy** option is applied. The aggregation now only includes the English Month Name attribute from the **Time** dimension.



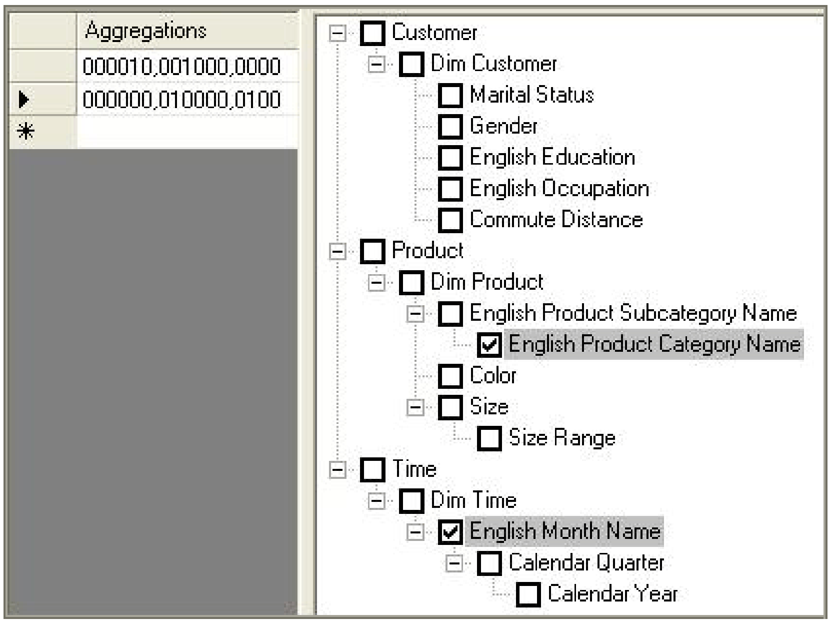
	Aggregations
	000010,001000,0000
	000000,010000,0100
▶	000000,010000,0100
*	

- [ ] Customer
  - [ ] Dim Customer
    - [ ] Marital Status
    - [ ] Gender
    - [ ] English Education
    - [ ] English Occupation
    - [ ] Commute Distance
- [ ] Product
  - [ ] Dim Product
    - [ ] English Product Subcategory Name
      - [✓] English Product Category Name
    - [ ] Color
    - [ ] Size
      - [ ] Size Range
- [ ] Time
  - [ ] Dim Time
    - [✓] English Month Name
      - [ ] Calendar Quarter
        - [ ] Calendar Year

Figure 34 Aggregations with redundancy eliminated

4. **Eliminate duplicate aggregations**.

   Duplicate aggregations are aggregations that include the exact same set of attributes. Continuing with example in Figure 34, note that the there are two identical aggregations for 0000000,010000,0100. This aggregation consists of the English Product Category Name and English Month Name attributes. After the **Eliminate Duplicates** option is applied, the duplicated aggregation is removed and the updated aggregation design is presented in Figure 35.



	Aggregations
	000010,001000,0000
▶	000000,010000,0100
✳	

```
☐ Customer
  ☐ Dim Customer
    ☐ Marital Status
    ☐ Gender
    ☐ English Education
    ☐ English Occupation
    ☐ Commute Distance
☐ Product
  ☐ Dim Product
    ☐ English Product Subcategory Name
      ☑ English Product Category Name
    ☐ Color
    ☐ Size
      ☐ Size Range
☐ Time
  ☐ Dim Time
    ☑ English Month Name
      ☐ Calendar Quarter
        ☐ Calendar Year
```

Figure 35 Aggregations with duplicates eliminated

5. **Assign the aggregation design to a partition**.

   After you assign the aggregation design to one or more partitions, the utility displays the assigned partitions under the name of the new aggregation design, as displayed in Figure 32.

6. **Save the measure group to SQL Server.**

   Your new aggregation design and partition assignment is not saved on the server until you perform an explicit save on the modified measure group. If you exit out of the utility and do not save, your changes will not be committed to the server.

7. **Process the partition**.

   Process the necessary measure group or partitions to build the aggregations for your new design. This operation needs to be performed outside of the Aggregation Utility using your normal processing techniques. Note that if you simply need to build aggregations, you can perform a **ProcessIndexes** operation on the appropriate measure group / partitions. For more information on **ProcessIndexes,** see Partition-processing commands.

8. **Evaluate the aggregation size**.

With the aggregations processed, you can use the Aggregation Utility to evaluate the relative size of each aggregation compared with the fact data for the partition. Using this information, you manually eliminate relatively large aggregations that take a long time to process and do not offer significant querying benefits. Remember that the aggregation design algorithm eliminates any aggregations that are greater than one third the size of the fact table. To apply similar logic, you can easily identify and delete any aggregations in your custom aggregation design that are significantly large.

9. **Re-save to SQL Server and reprocess.**

   After you evaluate the aggregation size and remove any large aggregations, re-save the aggregation design and then reprocess the necessary partitions. For any subsequent changes that you make over time, always remember to re-save and reprocess.

Microsoft

Microsoft® SQL Server® 2008 R2

Analysis Services Operations Guide

SQL Server White Paper

Writers: Thomas Kejser, John Sirmon, and Denny Lee

Technical Writer and Editor: Heidi Steen, Beth Inghram

Contributors and Technical Reviewers:

Kagan Arca
Akshai Mirchandani
Edward Melomed
Brad Daniels
Ashvini Sharma
Sedat Yogurtcuoglu
Alexei Khalyako
Peter Adshead (UBS)
Willfried Färber (Trivadis)
Dae Seong Han
Anne Zorner
Andrea Uggetti
Mike Vovchik
Chris Webb (Crossjoin Consulting)
Greg Galloway (Artis Consulting)
Andrew Calvett (UBS)
Alejandro Leguizamo (SolidQ)
Darren Gosbell (James & Monroe)
Marco Russo (Loader)
Alberto Ferrari (SQLBI)
Sanjay Nayyar (IM Group)
Marcel Franke (pmOne)
Thomas Ivarsson (Sigma AB)
John Desch
Didier Simon
Marius Dumitru

Published: June 2011

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

Summary: This white paper describes how operations engineers can test, monitor, capacity plan, and troubleshoot Microsoft SQL Server Analysis Services OLAP solutions in SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2.

Copyright

Contents

1 Introduction

In this guide you will find information on how to test and run Microsoft SQL Server Analysis Services in SQL Server 2005, SQL Server 2008, and SQL Server 2008 R2 in a production environment. The focus of this guide is how you can test, monitor, diagnose, and remove production issues on even the largest scaled cubes. This paper also provides guidance on how to configure the server for best possible performance.

Analysis Services cubes are a very powerful tool in the hands of the business intelligence (BI) developer. They provide an easy way to expose even large data models directly to business users. Unlike traditional, static reporting, where the query workload is known in advance, cubes support ad-hoc queries. Typically, such queries are generated by the Microsoft Excel spreadsheet software – without the business user being aware of the intricacies of the query engine. Because cubes allow such great freedom for users, the power they give to developers comes with responsibility. It is in the interaction between development and operations that the business value of the cubes is realized and where the proper steps can be taken to ensure that the power of the cubes is used responsibly.

It is the goal of this guide to make your operations processes as painless as possible, and to have you run with the best possible performance without any additional development effort to your deployed cubes. In this guide, you will learn how to get the best out of your existing data model by making changes transparent to the data model and by making configuration changes that improve the user experience of the cube.

However, no amount of operational readiness can cure a poorly designed cube. Although this guide shows you where you can make changes transparent to end users, it is important to be aware that there are cases where design change is the only viable path to good performance and reliability. Cubes do not take away the ubiquitous need for informed data modeling. Fortunately, this operations guide has a companion volume targeted at developers: the Analysis Services Performance Guide. We highly recommend that your developers read that white paper and follow the guidance in it.

Cubes do not exist in isolation – they rely on relational data sources to build their data structures. Although a full treatment of good relational data warehouse modeling out of scope for this document, it still provides some pointers on how to tune the database sources feeding the cube. Relational engines vary in their functionality; this paper focuses on guidance for SQL Server data sources. Much of the information here should apply equally to other engines and your DBA should be able to transfer the guidance here to other database systems you run.

In every IT project, preproduction testing is a crucial part of the development and deployment cycle. Even with the most careful design, testing will still be able to shake out errors and avoid production issues. Designing and running a test run of an enterprise cube is time well invested. Hence, this guide includes a description of the test methods available to you.

2 Configuring the Server

Installing Analysis Services is relatively straightforward. However, applying certain post-installation configuration options can improve performance and scalability of the cube. This section introduces these settings and provides guidance on how to configure them.

2.1 Operating System

Because Analysis Services uses the file system cache API, it partially relies on code in the Windows Server operating system for performance of its caches. Small changes are made to the file system cache in most versions of Windows Server, and this can have an effect on Analysis Services performance. We have found that patches related to the following software should be applied for best performance.

Windows Server 2008:

- KB 961719 – Applications that perform asynchronous cached I/O read requests and that use a disk array that has multiple spindles may encounter a low performance issue in Windows Server 2008, in Windows Essential Business Server 2008, or in Windows Vista SP1

Windows Server 2008 R2:

- KB 979149 – A computer that is running Windows 7 or Windows Server 2008 R2 becomes unresponsive when you run a large application
- KB 976700 – An application stops responding, experiences low performance, or experiences high privileged CPU usage if many large I/O operations are performed in Windows 7 or Windows Server 2008 R2
- KB 982383 – You encounter a decrease in I/O performance under a heavy disk I/O load on a Windows Server 2008 R2-based or Windows 7-based computer

Kerberos-enabled systems:

For Kerberos-enabled systems, you may need the following additional patches:

- http://support.microsoft.com/kb/969083/
- http://support.microsoft.com/kb/2285736/

References:

- Configure Monitoring Server for Kerberos delegation - http://technet.microsoft.com/en-us/library/bb838742(office.12).aspx
- How to configure SQL Server 2008 Analysis Services and SQL Server 2005 Analysis Services to use Kerberos authentication - http://support.microsoft.com/kb/917409
- Manage Kerberos Authentication Issues in a Reporting Services Environment http://msdn.microsoft.com/en-us/library/ff679930.aspx

2.2 msmdsrv.ini

You can control the behavior of the Analysis Services engine to a great degree from the **msmdsrv.ini** file. This XML file is found in the **<instance dir>/OLAP/Config** folder. Many of the settings in **msmdsrv.ini** should not be changed without the assistance of Microsoft Product Support, but there are still a large number of configuration options that you can control yourself. It is the aim of this guide to provide you wth the guidance you need to properly configure these settings. The following table provides an overview of the settings available to you and can serve as starting point for exploring reconfiguration options.

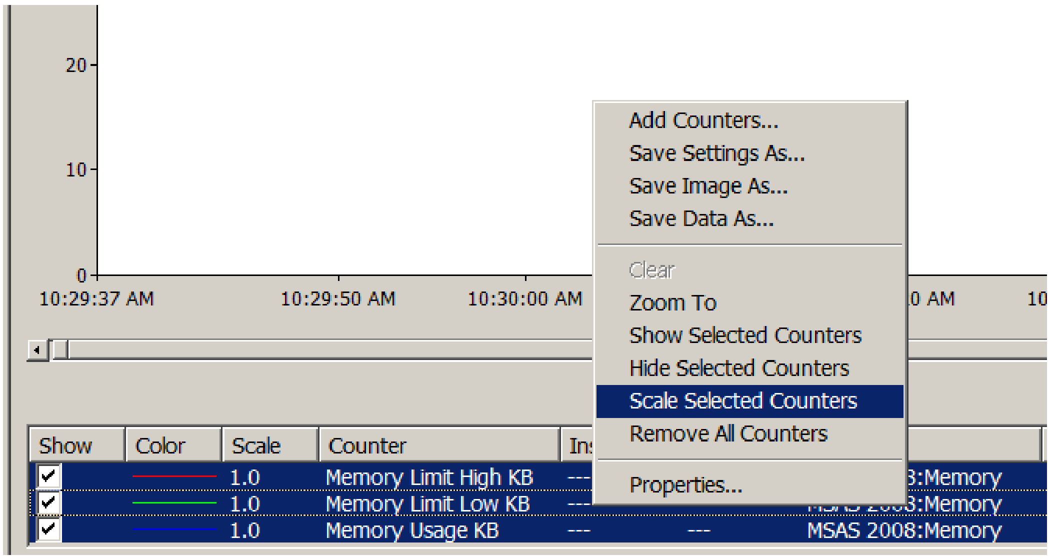
Setting	Used for	Described in
<MemoryHeapType> **<HeapTypeForObjects>**	Increasing throughput of high concurrency system	Section 2.3.2.4
<HardMemoryLimit>	Preventing Analysis Services from consuming too much memory	Section 2.3.2
<LowMemoryLimt>	Reserving memory for the Analysis Services process	Section 2.3.2
<PreAllocate>	Locking Analysis Services memory allocation, and improving performance on Windows Server 2003 and potentially Windows Server 2008	Section 2.3.2.1
<TotalMemoryLimit>	Controlling when Analysis Services starts trimming working set	Section 2.3.2
<CoordinatorExecutionMode> **<CoordinatorQueryMaxThreads>**	Setting concurrency of queries in thread pool during query execution	Section 2.4.2
<CoordinatorBuildMaxThreads>	Increasing processing speeds	Section 7.2.1
<AggregationMemoryMin> **<AggregationMemoryMax>**	Increasing parallelism of partition processing	Section 7.2.2
<CoordinatorQueryBalancingFactor> **<CoordinatorQueryBoostPriorityLevel>**	Preventing single users from monopolizing the system	Section 2.4.3
<LimitSystemFileCacheSizeMB>	Controlling the size of the file system cache	Section 2.3.2
<ThreadPool> (and subsections)	Increasing thread pools for high concurrency systems	Section 2.4
<BufferMemoryLimit> **<BufferRecordLimit>**	Increasing compression during processing, but consuming more memory	Section 2.3.2.3
<DataStorePageSize> **<DataStoreHashPageSize>**	Increasing concurrency on a large machine	Section 2.3.2.4
<MinIdleSessionTimeout> **<MaxIdleSessionTimeout>**	Cleaning up idle sessions faster on systems that have many	Section 2.3.2.5

<IdleOrphanSessionTimeout> **<IdleConnectionTimeout>**	connect/disconnects	
<Port>	Changing the port that Analysis Services is listening on	Section 5.1
<AllowedBrowsingFolders>	Controlling which folders are viewable by the server administrator	Section 2.5 and Section 5.5
<TempDir>	Controlling where disk spills go	Section 2.5
<BuiltinAdminsAreServerAdmins>	Controlling segregation-of-duties scenarios	Section 5
<ServiceAccountIsServerAdmin>	Controlling segregation-of-duties scenarios	Section 5
<ForceCommitTimeout> **<CommitTimeout>**	Killing queries that are blocking a processing operation or killing the processing operation	Section 7.4.4.4

2.2.1 A Warning About Configuration Errors

One of the most common mistakes encountered when tuning Analysis Services is misconfiguration of the **msmdsrv.ini** file. Because of this, changing your **msmdsrv.ini** file settings should be done with extreme care. If you inherit the managing responsibilities of an existing Analysis Services server, one of the first things you should do is to run the **Windif.exe** utility, which comes with Windows Server, against the current **msmdsrv.ini** file with the default **msmdsrv.ini** file. Also, if you upgraded to SQL Server 2008 or SQL Server 2008 R2 from SQL Server 2005, you will likely want to compare the current **msmdsrv.ini** file settings with the default settings for SQL Server 2008 or SQL Server 2008 R2. As described in the Thread Pool section, for optimization, some settings were changed in SQL Server 2008 and SQL Server 2008 R2 from their default values in SQL Server 2005.

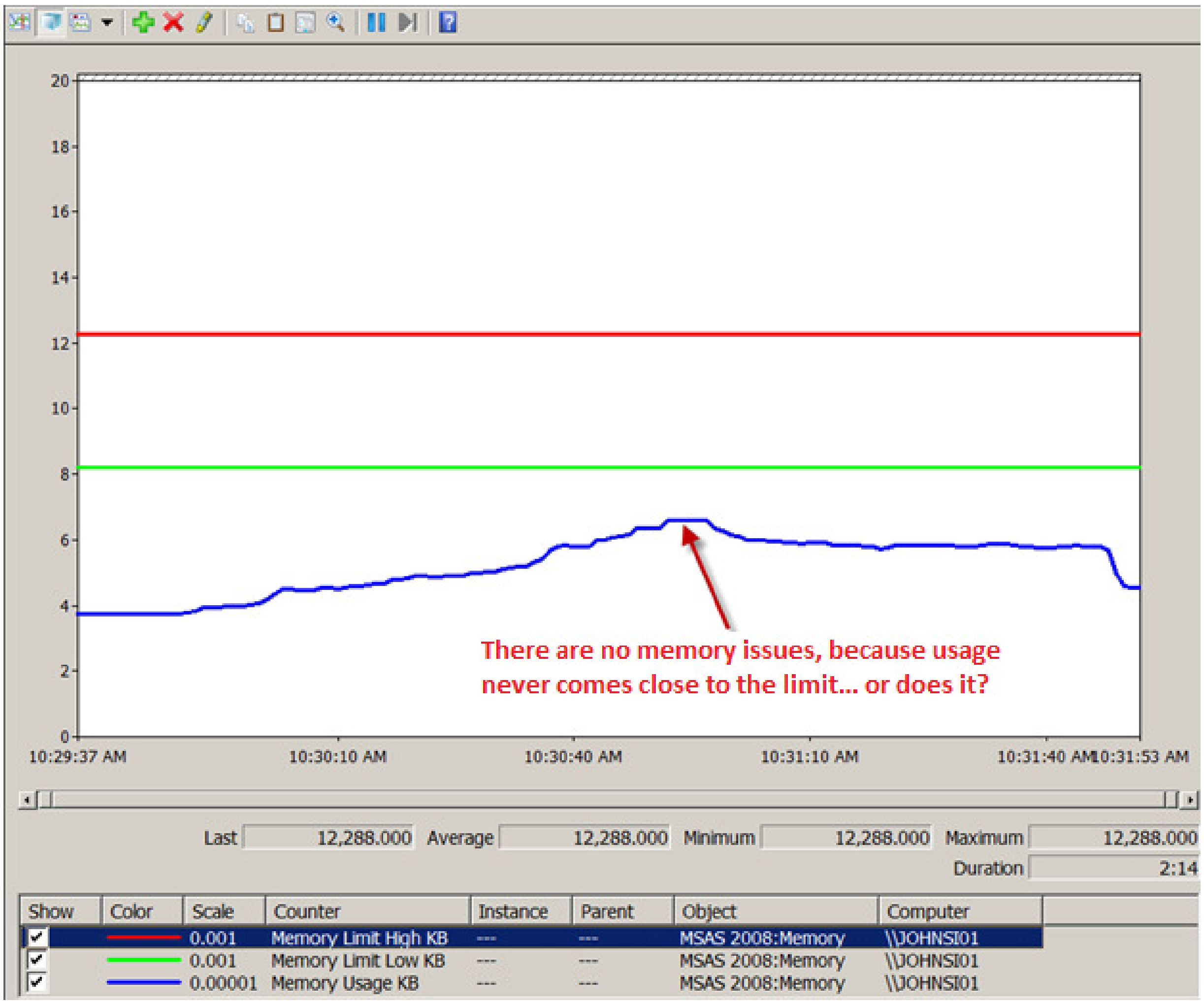
Take the following example from a customer configuration. The customer was experiencing extremely poor query/processing performance. When we looked at the memory counters, we saw this.



Show	Color	Scale	Counter	Ins		
✓	🟥	1.0	Memory Limit High KB	---		:Memory
✓	🟩	1.0	Memory Limit Low KB	---		:Memory
✓	🟦	1.0	Memory Usage KB	---	---	MSAS 2008:Memory

Figure 1 - Scaling Counters

10

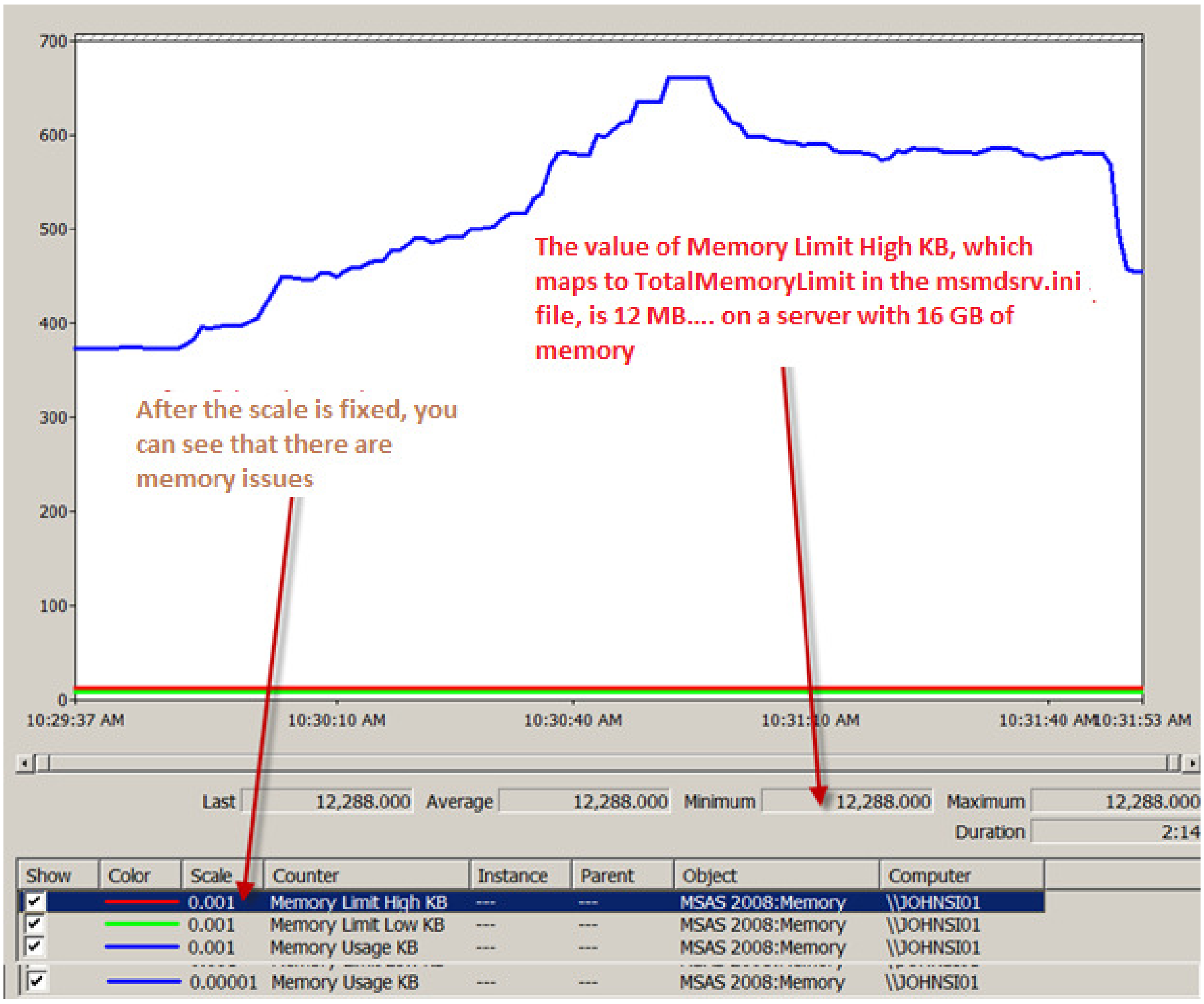

20
18
16
14
12
10
8
6
4
2
0
There are no memory issues, because usage never comes close to the limit... or does it?
10:29:37 AM
10:30:10 AM
10:30:40 AM
10:31:10 AM
10:31:40 AM
10:31:53 AM

Last	Average	Minimum	Maximum	Duration
12,288.000	12,288.000	12,288.000	12,288.000	2:14

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	0.001	Memory Limit High KB	---	---	MSAS 2008:Memory	\\JOHNSI01
✓	🟩	0.001	Memory Limit Low KB	---	---	MSAS 2008:Memory	\\JOHNSI01
✓	🟦	0.00001	Memory Usage KB	---	---	MSAS 2008:Memory	\\JOHNSI01

Figure 2 - Memory measurements with scaled counters

At first glance, this screenshot seems to indicate that everything is OK; there is no memory issue. However, in this case, because the customer used the **Scale Selected Counters** option in Windows Performance Monitor incorrectly, it was not possible to compare the three counters. A closer look, after the scale of the counters was corrected, shows this.



The value of Memory Limit High KB, which maps to TotalMemoryLimit in the msmdsrv.ini file, is 12 MB.... on a server with 16 GB of memory

After the scale is fixed, you can see that there are memory issues

Last 12,288.000 Average 12,288.000 Minimum 12,288.000 Maximum 12,288.000
Duration 2:14

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	0.001	Memory Limit High KB	---	---	MSAS 2008:Memory	\\JOHNSI01
✓	🟩	0.001	Memory Limit Low KB	---	---	MSAS 2008:Memory	\\JOHNSI01
✓	🟦	0.001	Memory Usage KB	---	---	MSAS 2008:Memory	\\JOHNSI01
✓	🟦	0.00001	Memory Usage KB	---	---	MSAS 2008:Memory	\\JOHNSI01

Figure 3 - Properly scaled counters

Now the fact that there is a serious memory problem is clear. Take a look at the actual **Memory Limit High KB** value. In a Performance Monitor log this and **Memory Limit Low KB** will always be constant values that reflect the value of **TotalMemoryLimit** and **LowMemoryLimit** respectively in the **msmdsrv.ini** file. So it appears that someone has modified the **TotalMemoryLimit** from the default 80 percent to an absolute value of 12,288 *bytes*, probably thinking the setting was in megabytes when in reality, the setting is in bytes. As you can see, the results of an incorrectly configured .ini file setting can be disastrous.

The moral of this story is this: Always be extremely careful when you modify your **msmdsrv.ini** file settings. One of the first things the Microsoft Customer Service and Support team does when working on an Analysis Services issue is grab the **msmdsrv.ini** file and compare it with the default .ini file for the customer's version of Analysis Services. There are numerous file comparison tools you can use, the most obvious being **Windiff.exe**, which comes with Windows Server.

References:

12

- How to Use the Windiff.exe Utility - http://support.microsoft.com/kb/159214

2.3 Memory Configuration

This section describes the memory model of Analysis Services and provides guidance on how to initially configure memory settings for a server. As will become clear in the Monitoring and Tuning section, it is often a good idea to readjust the memory settings as you learn more about the workload that is running on your server.

To understand the tradeoffs you will make during server configuration, it is useful to understand a bit more about how Analysis Services uses memory.

2.3.1 Understanding Memory Usage

Apart from the executable itself, Analysis Services has several data structures that consume physical memory, including caches that boost the performance of user queries. Memory is allocated using the standard Windows memory API (with the exception of a special case described later) and is part of the Msmdsrv.exe process private bytes and working set. Analysis Services does not use the AWE API to allocate memory, which means that on 32-bit systems you are restricted to either 2 GB or 3 GB of memory for Analysis Services. If you need more memory, we recommend that you move to a 64-bit system. The total memory used by Analysis Services can be monitored in Performance Monitor using either: **MSOLAP:Memory\Memory Usage Kb** or **Process\Private Bytes – msmdsrv.exe.**

Memory allocated by Analysis Services falls into two broad categories: shrinkable and non-shrinkable memory.

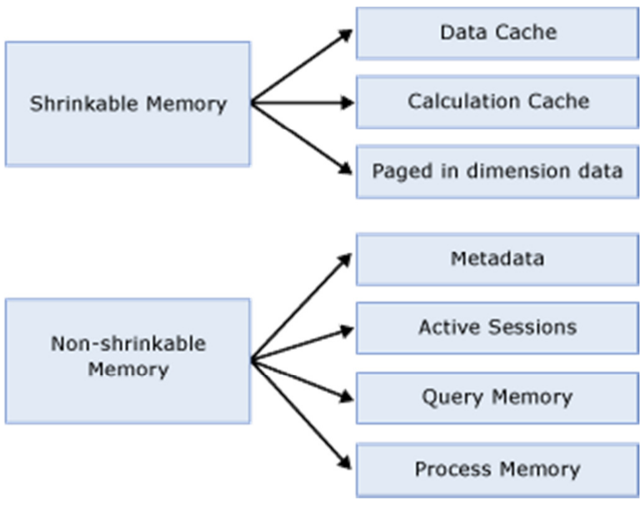


Figure 4 - Memory Structures

Non-shrinkable memory includes all the structures that are required to keep the server running: active user sessions, working memory of running queries, metadata about the objects on the server, and the process itself. Analysis Services does not release memory in the non-shrinkable memory spaces – though this memory can see still be paged out by the operating system (but see the section on PreAllocate).

Shrinkable memory includes all the caches that gradually build up to increase performance: The formula engine has a calculation cache that tries to keep frequently accessed and calculated cells in memory. Dimensions that are accessed by users are also kept in cache, to speed up access to dimensions, attributes, and hierarchies. The storage engine also caches certain accessed subcubes. (Not all subcubes are cached. For example, the ones used by arbitrary shapes are not kept.) Analysis Services gradually trims its working set if the shrinkable memory caches are not used frequently. This means that the total memory usage of Analysis Services may fluctuate up and down, depending on the server state – this is expected behavior.

Outside of the Analysis Services process space, you have to consider the memory used by the operation system itself, other services, and the memory consumed by the file system cache. Ideally, you want to avoid paging important memory – and as we will see, this may require some configuration tweaks.

References:

- Russinovich, Mark and David Solomon: *Windows Internals, 5th edition*. http://technet.microsoft.com/en-us/sysinternals/bb963901
- Webb, Chris, Marco Russo, and Alberto Ferrari: *Expert Cube Development with Microsoft SQL Server 2008 Analysis Services* – Chapter 11
- Arbitrary Shapes in AS 2005 - https://kejserbi.wordpress.com/2006/11/16/arbitrary-shapes-in-as-2005/

2.3.2 Memory Setting for the Analysis Service Process

The Analysis Services memory behavior can be controlled by using parameters available in the **msmdsrv.ini** file. Except for the **LimitFileSystemCacheMB** setting, all the memory settings behave like this:

- If the setting has a value below 100, the running value is that percent of total RAM on the machine.
- If the setting has a value above 100, the running value is that amount of KB.

For ease of use, we recommend that you use the percentage values. The total memory allocations of Analysis Services process are controlled by the following settings.

LowMemoryLimit – This is the amount of memory that Analysis Services will hold on to, without trimming its working set. After memory goes above this value, Analysis Services begins to slowly deallocate memory that is not being used. The configuration value can be read from the Performance Monitor counter: **MSOLAP:Memory\LowMemoryLimit.** Analysis Services does not allocate this memory at startup; however, after the memory is allocated, Analysis Services does not release it.

PreAllocate – This optional parameter (with a default of 0) enables you to preallocate memory at service startup. It is described in more detail later in this guide. **PreAllocate** should be set to a value less than or equal to **LowMemoryLimit**.

TotalMemoryLimit –When Analysis Services exceeds this memory value, it starts deallocating memory aggressively from shrinkable memory and trimming its working set. Note that this setting is not an upper memory limit for the service; if a large query consumes a lot of resources, the service can still consume memory above this value. This value can also be read from the Performance Monitor counter: **MSOLAP:Memory\TotalMemoryLimit**.

HardMemoryLimit – This setting is only available in SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services – not SQL Server 2005 Analysis Services. It is a more aggressive version of **TotalMemoryLimit**. If Analysis Services goes above **HardMemoryLimit**, user sessions are selectively killed to force memory consumption down.

LimitSystemFileCacheMB – The Windows file system cache (described later) is actively used by Analysis Services to store frequently used blocks on disk. For some scan-intensive workloads, the file system cache can grow so large that Analysis Services is forced to trim its working set. To avoid this, Analysis Services exposes the Windows API to limit the size of the file system cache with a configuration setting in Msmdsrv.ini. If you choose to use this setting, note that it limits the file system cache for the entire server, not just for the Analysis Services files. This also means that if you run more than one instance of Analysis Services on the server, you should use the same value for **LimitSystemFileCacheMB** for each instance.

2.3.2.1 PreAllocate and Locked Pages

The **PreAllocate** setting found in **msmdsrv.ini** can be used to reserve virtual and/or physical memory for Analysis Services. For installations where Analysis Services coexists with other services on the same machine, setting **PreAllocate** can provide a more stable memory configuration and system throughput.

Note that if the service account used to run Analysis Services also has the **Lock pages in Memory** privilege, **PreAllocate** causes Analysis Services to use large memory pages. Large memory pages are more efficient on a big memory system, but they take longer to allocate. **Lock pages in Memory** is set using Gpedit.msc. Bear in mind that large memory pages cannot be swapped out to the page file. While this can be an advantage from a performance perspective, a high number of allocated large pages can cause the system to become unresponsive.

Important: PreAllocate has the largest impact on the Windows Server 2003 operating system. With the introduction of Windows Server 2008, memory management has been much improved. We have been testing this setting on Windows Server 2008, but have not measured any significant performance benefits of using **PreAllocate** on this platform. However, you may want still want to make use of the locked pages functionality in Window Server 2008.

To learn more about the effects of **PreAllocate**, see the following technical note:

- Running Microsoft SQL Server 2008 Analysis Services on Windows Server 2008 vs. Windows Server 2003 and Memory Preallocation: Lessons Learned - (http://sqlcat.com/technicalnotes/archive/2008/07/16/running-microsoft-sql-server-2008-analysis-services-on-windows-server-2008-vs-windows-server-2003-and-memory-preallocation-lessons-learned.aspx)

References:

- How to: Enable the Lock Pages in Memory Option (Windows) - http://msdn.microsoft.com/en-us/library/ms190730.aspx

2.3.2.2 AggregationMemory Max/Min

Under the server properties of SQL Server Management Studio and in **msmdsrv.ini**, you will find the following settings:

- **OLAP\Process\AggregationMemoryLimitMin**
- **OLAP\Process\AggregationMemoryLimitMax**

These two settings determine how much memory is allocated for the creation of aggregations and indexes in each partition. When Analysis Services starts partition processing, parallelism is throttled based on the **AggregationMemoryMin/Max** setting. The setting is per partition. For example, if you start five concurrent partition processing jobs with **AggregationMemoryMin** = 10, an estimated 50 percent (5 x 10%) of reserved memory is allocated for processing. If memory runs out, new partition processing jobs are blocked while they wait for memory to become available. On a large memory system, allocating 10 percent of available memory per partition may be too much. In addition, Analysis Services may sometimes misestimate the maximum memory required for the creation of aggregates and indexes. If you process many partitions in parallel on a large memory system, lowering the value of **AggregationMemoryLimitMin** and **AggregationMemoryMax** may increase processing speed. This works because you can drive a higher degree of parallelism during the process index phase.

Like the other Analysis Services memory settings, if this setting has a value greater than 100 it is interpreted as a fixed amount of kilobytes, and if is lower than 100, it is interpreted as a percentage of the memory available to Analysis Services. For machines with large amounts of memory and many partitions, using an absolute kilobyte value for these settings may provide a better control of memory than using a percentage value.

2.3.2.3 BufferMemoryLimit and BufferRecordLimit

OLAP\Process\BufferMemoryLimit determines the size of the fact data buffers used during partition data processing. While the default value of the **OLAP\Process\BufferMemoryLimit** is sufficient for most deployments, you may find it useful to alter the property in the following scenario.

If the granularity of your measure group is more summarized than the relational source fact table, you may want to consider increasing the size of the buffers to facilitate data grouping. For example, if the source data has a granularity of day and the measure group has a granularity of month; Analysis Services must group the daily data by month before writing to disk. This grouping occurs within a single buffer and it is flushed to disk after it is full. By increasing the size of the buffer, you decrease the number of times that the buffers are swapped to disk. Because the increased buffer size supports a higher compression ratio, the size of the fact data on disk is decreased, which provides higher performance. However, be aware that high values for the **BufferMemoryLimit** use more memory. If memory runs out, parallelism is decreased.

You can use another configuration setting to control this behavior: **BufferRecordLimit**. This setting is expressed in received records from the data source instead of a **Memory %/Kb** size. The lower of the two takes precedence. For example, if **BufferMemoryLimit** is set to 10 percent of a 32-GB memory space and **BufferRecordLimit** is set to 10 million rows, either 3.2 GB or 10,000,000 times the rowsize is allocated for the merge buffer, whichever is smaller.

2.3.2.4 Heap Settings, DataStorePageSize, and DataStoreHashPageSize

During query execution, Analysis Services generally touches a lot of data and performs many memory allocations. Analysis Services has historically relied on its own heap implementation to achieve the best performance. However, since Windows Server 2003, advances in the Windows Server operating system mean that memory can now be managed more efficiently by the operating system. This turns out to be especially true for multi-user scenarios. Because of this, servers that have many users should generally apply the following changes to the **msmdsrv.ini** file.

Setting	Default	Multi-user, faster heap
<MemoryHeapType>	1	2
<HeapTypeForObjects>	1	0

It is also possible to increase the page size that is used for managing the hash tables Analysis Services uses to look up values. Especially on modern hardware, we have seen the following change yield a significant increase in throughput during query execution.

Setting	Default	Bigger pages
<DataStorePageSize>	8192	65536
<DataStoreHashPageSize>	8192	65536

References:

- **KB2004668** - You experience poor performance during indexing and aggregation operations when using SQL Server 2008 Analysis Services - http://support.microsoft.com/kb/2004668

2.3.2.5 Clean Up Idle Sessions

Client tools may not always clean up sessions opened on the server. The memory consumed by active sessions is non-shrinkable, and hence is not returned the Analysis Services or operating system process for other purposes. After a session has been idle for some time, Analysis Services considers the session expired and move the memory used to the shrinkable memory space. But the session still consumes memory until it is cleaned up.

Because idle sessions consume valuable memory, you may want to configure the server to be more aggressive about cleaning up idle sessions. There are some **msmdsrv.ini** settings that control how Analysis Services behaves with respect to idle sessions. Note that a value of zero for any of the following settings means that the sessions or connection is kept alive indefinitely.

Setting	Description
<MinIdleSessionTimeOut>	This is the minimum amount of time a session is allowed to be idle before Analysis Services considers it ready to destroy. Sessions are destroyed only if there is memory pressure.
<MaxIdleSessionTimeout>	This is the time after which the server forcibly destroys the session, regardless of memory pressure.
<IdleOrphanSessionTimeout>	This is the timeout that controls sessions that no longer have a connection associated with them. Examples of these are users running a query and then disconnecting from the server.
<IdleConnectionTimeout>	This timeout controls how long a connection can be kept open and idle until Analysis Services destroys it. Note that if the connection has no active sessions, **MaxIdleSessionTimeout** eventually marks the session for cleaning and the connection is cleaned with it.

If your server is under memory pressure and has many users connected, but few executing queries, you should consider lowering **MinIdleSessionTimeOut** and **IdleOrphanSessionTimeout** to clean up idle sessions faster.

2.4 Thread Pool and CPU Configuration

Analysis Services uses thread pools to manage the threads used for queries and processing. The thread management subsystem that Analysis Services uses enables you to fine-tune the number of threads that are created. Tuning the thread pool is a balancing act between CPU overutilization and underutilization: If too many threads are created, unnecessary context switches and contention for system resources lower performance, and too few threads can lead to CPU and disk underutilization, which means that performance is not optimal on the hardware allocated to the process. When you tune your thread pools, it is essential that you benchmark your performance before and after your configuration changes; misconfiguration of the thread pool can often cause unforeseen performance issues.

2.4.1 Thread Pool Sizes

This section discusses the settings that control thread pool sizes.

ThreadPool\Process\MaxThreads determines the maximum number of available threads to Analysis Services during processing and for accessing the I/O system during queries. On large, multiple-CPU machines, the default value in SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services of 64 or 10 multiplied by the number of CPU cores (whichever is higher) may be too low to take advantage of all CPU cores. In SQL Server 2005 Analysis Services the settings for process thread pool were set to the static value of 64. If you are running an installation of SQL Server 2005 Analysis Services, or if you upgraded from SQL Server 2005 Analysis Services to SQL Server 2008 Analysis Services or SQL Server 2008 R2 Analysis Services, it might be a good idea to increase the thread pool.

ThreadPool\Query\MaxThreads determines the maximum number of threads available to the Analysis Services formula engine for answering queries. In SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services, the default is 2 multiplied by the number of logical CPU or 10, whichever is higher. In SQL Server 2005 Analysis Services, the default value was fixed at 10. Again, if you are running on SQL Server 2005 Analysis Services or an upgraded SQL Server 2005 Analysis Services, you may want to use the new settings.

2.4.2 CoordinatorExecutionMode and CoordinatorQueryMaxThreads

Analysis Services uses a centralized storage engine job architecture for both querying and processing operations. When a subcube request or processing command is executed, a coordinator thread is responsible for gathering the data needed to satisfy the request.

The value of **CoordinatorExecutionMode** limits the total number of coordinator jobs that can be executed in parallel by a subcube request in the storage engine. A negative value specifies the maximum number of parallel coordinator jobs that can start per processing core per subcube request. By default **CoordinatorExecutionMode** is set to -4, which means the server is limited to 4 jobs in parallel per core. For example, on a 16-core machine with the default values of **CoordinatorExecutionMode** = -4, a total of 64 concurrent threads can execute per subcube request.

When a subcube is requested, a coordinator thread starts up to satisfy the request. The coordinator first queues up one job for each partition that must be touched. Each of those jobs then continues to queue up more jobs, depending on the total number of segments that must be scanned in the partition. The value of **CoordinatorQueryMaxThreads**, with a default of 16, limits the number of partition jobs that can be executed in parallel for each partition. For example, if the formula engine requests a subcube that requires two partitions to be scanned, the storage engine is limited to a maximum of 32 threads for scanning. Also note that both the coordinator jobs and the scan threads are limited by the maximum number of threads configured in **ThreadPool/Processing/MaxThreads.** The following diagram illustrates the relationship between the different thread pools and the coordinator settings.

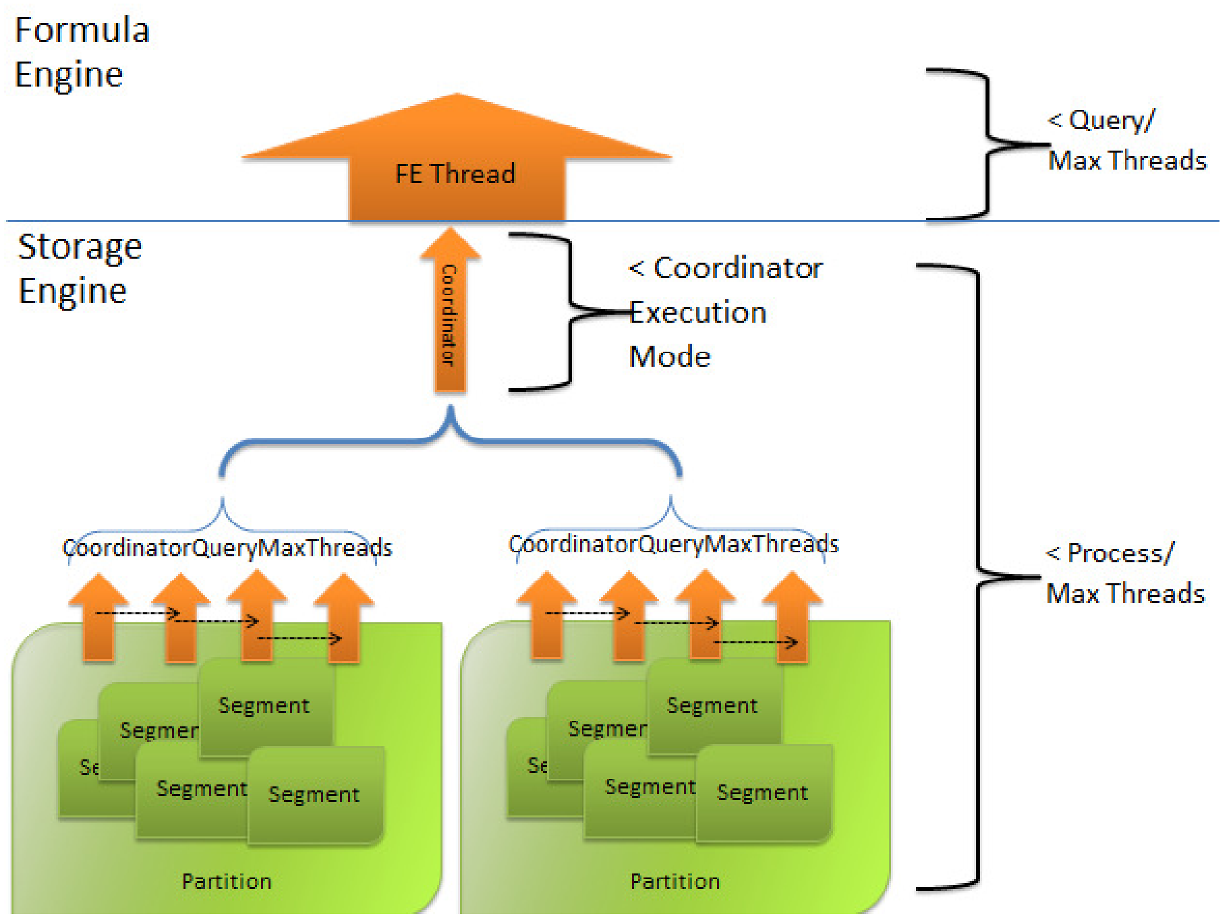


Figure 5 - Coordinator Queries

If you increase the processing thread pool, you should make sure that the **CoordinatorExecutionMode** settings, as well as the **CoordinatorQueryMaxThreads** settings, have values that enable you to make full use of the thread pool.

If the typical query in your system touches many partitions, you should be careful with the **CoordinatorQueryMaxThreads**. For example, if every query touches 10 partitions, a total 160 threads can be used just to answer that query. It will not take many queries to run the thread pool dry under those conditions. In such a case, consider lowering the setting of **CoordinatorQueryMaxThreads**.

2.4.3 Multi-User Process Pool Settings

In multi-user scenarios, long-running queries can starve other queries; specifically they can consume all available threads, blocking execution of other queries until the longer-running queries complete.

You can reduce the aggressiveness of how each coordinator job allocates threads per segment by modifying **CoordinatorQueryBalancingFactor** and **CoordinatorQueryBoostPriorityLevel** as follows.

Setting	Default	Multi-user nonblocking settings

CoordinatorQueryBalancingFactor	-1	1
CoordinatorQueryBoostPriorityLevel	3	0

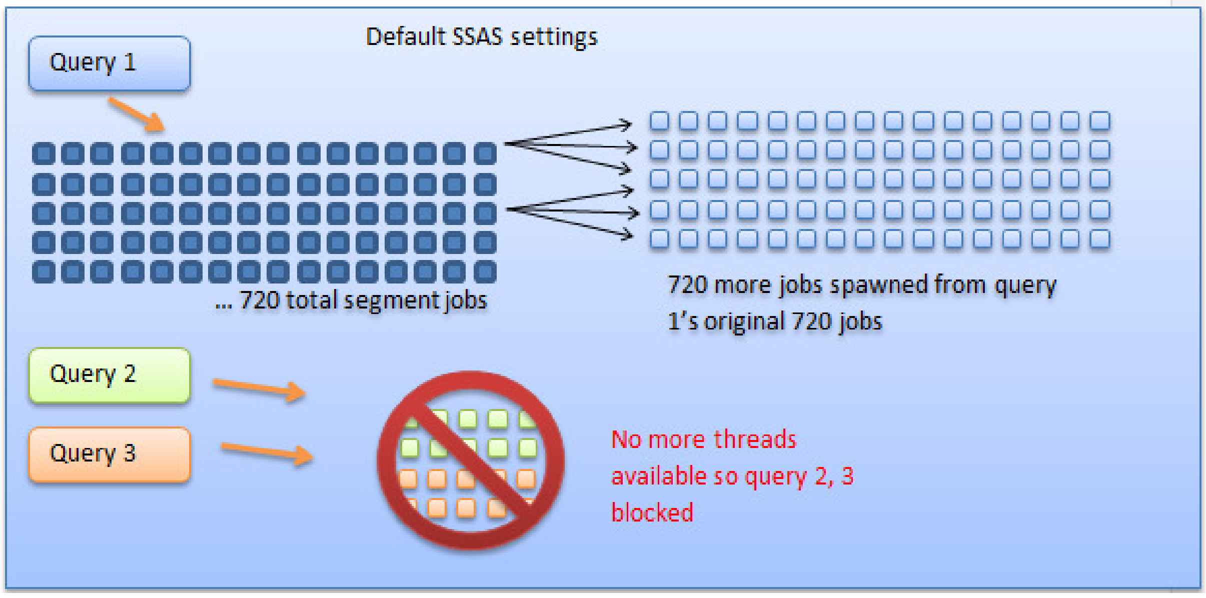
If you want to understand exactly what these settings do, you need to know a little about the internals of Analysis Services. First, a word of warning: The remainder of this section looks at Analysis Services at a very detailed level. It is perfectly acceptable to take a query workload and test with the default settings and then with the multi-user settings to decide if it is worth making these changes.

With the disclaimer out of the way, let’s look at an example to explain this behavior. On a 45-core Windows Server 2008 server with default .ini file settings, you have a long running query that appears to be blocking many of the other queries being executed by other users. Behind the scenes in Analysis Services, multiple segment jobs (different from coordinator jobs) are created to query the respective segments. A segment of data in Analysis Services is composed of roughly 64,000 records, which are subdivided into pages. There are 256 pages in a segment and 256 records in a page. These records are brought into memory in chunks upon request by queries. Analysis Services determines which pages need to be scanned to retrieve the relevant records for the data requested. These jobs are chained, meaning Job 1 queues Job 2 to the thread pool, Job 1 performs its own job, Job 2 queues Job 3 to the thread pool, Job 2 performs its own job, and so on. Each job has its own thread, or *segment job*.

In our example, using the default settings the first query fires off 720 jobs, scanning a lot of data. The second query fires off 1 job. The first 720 jobs fire off their own 720 jobs, using up all of the threads. This prevents the second query from executing, because no threads are available. This behavior causes blocking of the second and subsequent queries that need threads from the process pool. The multi-user settings (**CoordinatorQueryBalancingFactor**=1, **CoordinatorQueryBoostPriorityLevel**=0) prevent all of the threads from being allocated so the second and third queries can execute their jobs.

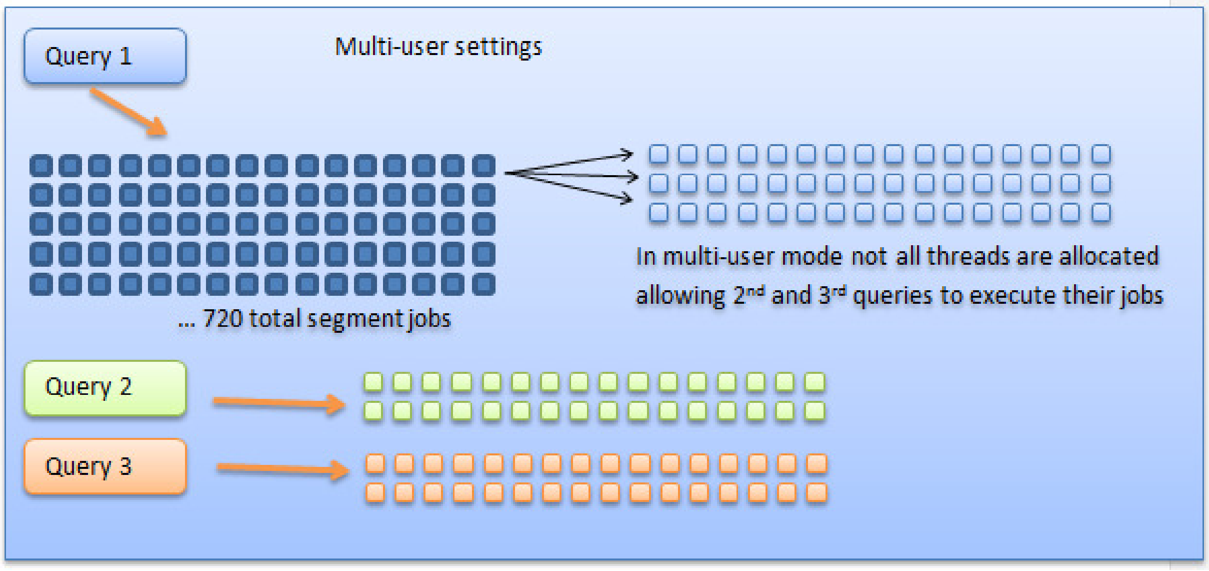
Again, each segment job uses one thread. If the long-running query requires scanning of multiple segments, Analysis Services creates the necessary number of threads. In single-user mode, the first query fires off 720 jobs, and the second query fires off 1 job. Each segment job is immediately queued up before the prior segment job begins scanning data. The first 720 jobs fire off their own 720 jobs, preventing second query from executing. In multi-user mode, not all threads are allocated, allowing second query (and third) to execute their jobs.

Be careful modifying these settings; although these settings reduce or stop the blocking of shorter-running queries by longer-running ones, it may slow the response times of individual queries. (In the figure, SSAS stands for SQL Server Analysis Services.)



Default SSAS settings
Query 1
... 720 total segment jobs
720 more jobs spawned from query 1's original 720 jobs
Query 2
Query 3
No more threads available so query 2, 3 blocked

Figure 6 - Default settings



Multi-user settings
Query 1
... 720 total segment jobs
In multi-user mode not all threads are allocated allowing 2nd and 3rd queries to execute their jobs
Query 2
Query 3

Figure 7 - Multi-user settings

2.4.4 Hyperthreading

We receive numerous questions from customers around hyperthreading. The Analysis Services development team has no official recommendation on hyperthreading; it is included in this white paper only because our customers ask about it frequently. With that said, we have made the following observations in installations in which hyperthreading is used with Analysis Services:

- If the load on your server is more CPU-bound, turning on hyperthreading offers no improvement, in our experience.
- If the load on your server is more I/O-bound, there may be some benefit to turning on hyperthreading.
- Processors are constantly changing, and many characteristics of behavior and performance related to hyperthreading are going to be specific to the processor.

2.5 Partitioning, Storage Provisioning and Striping

When you configure the storage of a server you are often presented with a series of LUNs for data storage. These are provisioned from either your SAN, internal drives, or NAND memory. NAND memory, in some configuration also known as Solid State Disks/Devices (SSD) is typically a good fit for large cubes, because the latency and I/O pattern favored by these drives are a good match for the Analysis Services storage engine workload.

The question is how to make the best use of the storage for your Analysis Services instance.

Apart from performance and capacity, the following factors must also be considered when designing the disk volumes for Analysis Services:

- Is clustering of the Analysis Services instance required?
- Will you be building a scaled-out environment?

Consult the subsections that follow for guidance in these specific setups. However, the following general guidance applies.

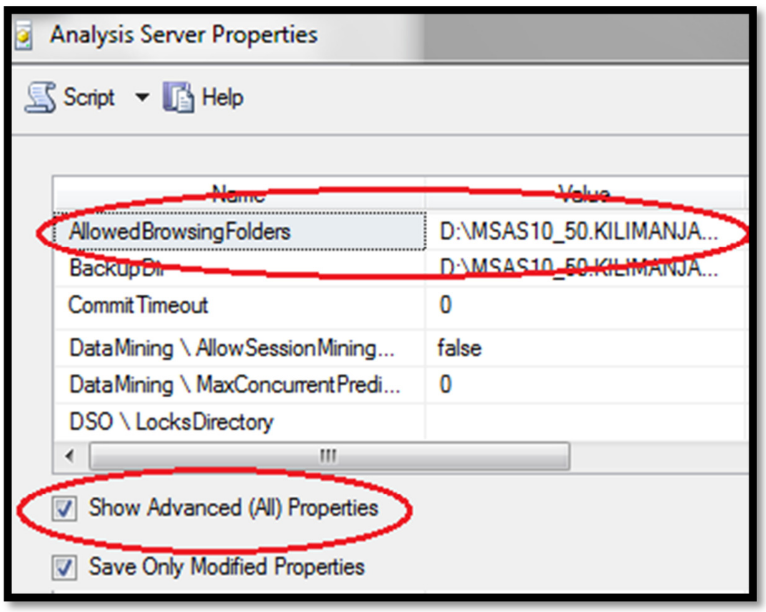
SAN Mega LUNs: If you are using a SAN, your storage administrator may be able to provision you a single, large LUN for your cubes. Unless you plan to build a consolidation environment, having such a single, mega-LUN is probably the easiest and most manageable way to provision your storage. First of all, it provides the IOPS as a general resource to the server. Secondly, and additional advantage of a mega-LUN is that you can easily disk align this in Windows Server 2003. For Windows Server 2008 you do not need to worry about disk alignment on newly created volumes.

Windows Server dynamic disk stripes: Using Disk Manager it is possible to combine multiple LUNs into a single Windows volume. This is a very simple way to combine multiple, similarly sized, similarly performing LUNs into a single mount point or drive letter. We have tested dynamic disk stripes in Windows Server 2008 R2 all the way up to 100.000 IOPS – and the performance overhead to that level is negligible.

> **Note:** You *cannot* use dynamic disk stripes in a cluster. This is discussed in greater detail later in this guide.

Drive letters versus mount points: Both drive letters and mount points will work with Analysis Services cubes. If the server is dedicated to a single Analysis Services instance, a drive letter may be the simplest solution. Choosing drive letters versus mount points is often a matter of personal preference, or it can be driven by the standards of your operations team. From the perspective of performance, one is not superior to the other.

AllowedBrowsingFolders: Remember that in order for a directory to be visible to administrators of Analysis Services in SQL Server Management Studio, it must be listed in **AllowedBrowsingFolders**, which is available in SQL Server Management Studio by clicking **Server Properties** and then **Advanced Properties**.



Name	Value
AllowedBrowsingFolders	D:\MSAS10_50.KILIMANJA...
BackupDir	D:\MSAS10_50.KILIMANJA...
CommitTimeout	0
DataMining \ AllowSessionMining...	false
DataMining \ MaxConcurrentPredi...	0
DSO \ LocksDirectory	

[✓] Show Advanced (All) Properties

[✓] Save Only Modified Properties

Figure 8 – AllowedBrowsingFolders

The Analysis Services account must also have permission to both read and write to these folders. Just adding them to the **AllowedBrowsingFolders** list is not enough – you must also assign file system permissions.

MFT versus GPT disk: If you expect your cubes to be larger than 2 terabytes, you should use a GPT disk. The default, an MFT disk, only allows 2-terabyte partitions.

TempDir Folder: The **TempDir** folder, configured in the Advanced Properties of the server, should be moved to the fastest volume you have. This may mean you will share TempDir with cube data, which is fine if you plan capacity accordingly. For more information, see the section about the **TempDir** folder.

NTFS Allocation Unit Size (AUS): With careful optimization, it is sometimes possible to get slightly better performance by smartly choosing between 32K or 64K (a few percent). But, unless you are hunting for benchmark performance, just go with 32K. If you have standardized on 64K for other SQL Server services, that will work too.

Don't suboptimize storage: There are a few cases where it makes sense to split your data into multiple volumes – for example if you have different storage types attached to the server (such as NAND for latest data and SATA drives for historical data). However, it is generally *not* a good idea to suboptimize the storage layout of Analysis Services by creating complicated data distributions that span different storage types. The rule of thumb for optimal disk layout is to utilize all disk drives for all cube partitions. Create large pools of disk, presented as single, large volumes.

Exclude Analysis Services folders from virus scanners: If you are running a virus scanner on the server, make sure the **Data** folder, **TempDir**, and backup folders are not being scanned or touched by the filter drivers of the antivirus tool. There are no executable files in these Analysis Services folders, and enabling virus scanners on the folder may slow down the disk access speeds significantly.

Consider Defragmenting the Data Folder: Analysis Services cubes get very fragmented over time. We have seen cases where defragmenting cube data folders, using the standard disk defragment utility, yielded a substantial performance benefit. Note that defragmenting a drive also has impact on the performance of that drive as it moves the files around. If you choose to defragment an Analysis Services drive, do so in batch window where the service can be taken offline while the defragmentation happens, or plan disk speeds accordingly to make sure the performance impact is acceptable.

References:

- White paper: Configuring Dynamic Volumes - http://technet.microsoft.com/en-us/library/bb727122.aspx

2.5.1 I/O Pattern

Because Analysis Services uses bitmap indexes to quickly locate fact data, the I/O generated is mostly random reads. I/O sizes will typically average around 32-KB block sizes.

As with all SQL Server databases systems, we recommend that you test your I/O subsystem before deploying the database. This allows you to measure how close the production system is to your maximum potential throughput. Not running preproduction I/O testing of an Analysis Services server is the equivalent of not knowing how much memory or how many CPU cores your server has.

Due to the threading architecture of the storage engine, the I/O pattern will also be highly multi threaded. Because the NTFS file system cache issues synchronous I/O, each thread will have only one or two outstanding I/O requests. Incidentally, this means that cubes will often run very well on NAND storage that favor this type of I/O pattern.

The Analysis Services I/O pattern can be easily simulated and tested using SQLIO.exe. The following command-line parameter will give you a good indication of the expected performance:

```
sqlio -b32 -frandom -o1 -s30 -t256 -kR <path of file>
```

Make sure you run on a sufficiently large test file. For more information, refer to the links in the References section.

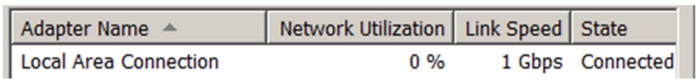
References:

- SQLIO download: http://www.microsoft.com/downloads/en/details.aspx?familyid=9a8b005b-84e4-4f24-8d65-cb53442d9e19&displaylang=en
- White paper: Analyzing I/O characteristics and sizing storage for SQL Server Database Applications – http://sqlcat.com/whitepapers/archive/2010/05/10/analyzing-i-o-characteristics-and-sizing-storage-systems-for-sql-server-database-applications.aspx
- Blog: The Memory Shell Game – http://blogs.msdn.com/b/ntdebugging/archive/2007/10/10/the-memory-shell-game.aspx

2.6 Network Configuration

During the **ProcessData** phase of Analysis Services processing, rows are transferred from the relational database specified in the data source to Analysis Services. If your data source is on a remote server, the data should be retrieved over TCP/IP. It is important to make sure all networking components are configured to support the optimal throughput. If your Ethernet throughput is consistently close to 80 percent of your maximum capacity, adding more network capacity typically speeds up **ProcessData**.

Creating a high-speed network is fairly easy with the networking hardware available today. Specifically most servers come with a Gigabit NICs out of the box. Infiniband and 10-gigabit NICs are also available for even more throughput. With that said, your overall throughput can be limited by routers and other hardware in your network that don't support the speed of your NIC. You can use the **Networking** tab in Task Manager to quickly determine your link speed. In the following screenshot you can see that the NIC and switch support a maximum of 1Gbps.



Adapter Name	Network Utilization	Link Speed	State
Local Area Connection	0 %	1 Gbps	Connected

Figure 9 - Viewing NIC speed

If you have a 1-Gbps NIC, but only a 100-Mbps switch, Task Manager displays 100 Mbps. Depending on your network topology there may be more to determining your link speed than this, but using Task Manager is a simple way to get a rough idea of the configuration.

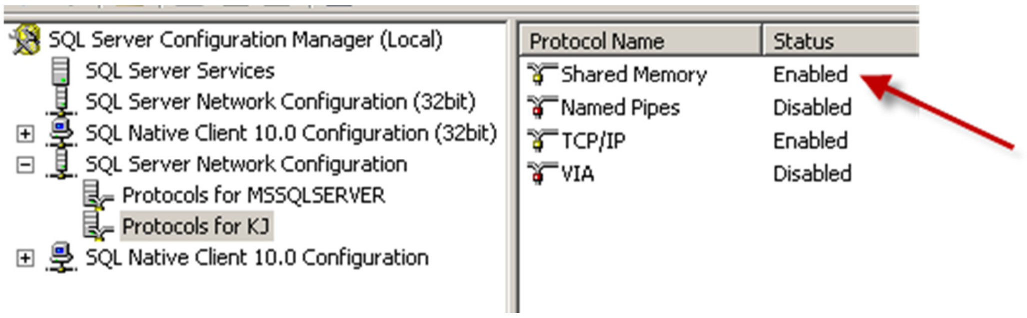
In addition to creating a high-speed network, there are some additional configurations you can change to further speed up network traffic.

2.6.1 Use Shared Memory for Local SQL Server Data Sources

If Analysis Services is on the same physical machine as the data source, and the data source is SQL Server, you should make sure you are exchanging data over the shared memory protocol. Shared memory is much faster than TCP/IP, as it avoids the overhead of the translation layers in the network stack. Shared memory is only possible if the SQL Server database is on the same physical machine as Analysis Services. If you cannot get a high speed network set up in your organization, you can sometimes benefit from running SQL Server and Analysis Services on the same physical machine.

To modify your data source connection to specify shared memory:

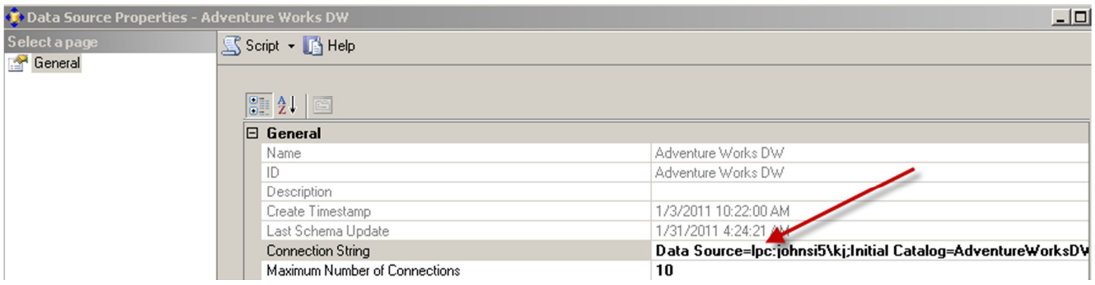
1. First, make sure that the Shared Memory protocol is enabled in SQL Server Configuration Manager.



SQL Server Configuration Manager (Local)
- SQL Server Services
- SQL Server Network Configuration (32bit)
- SQL Native Client 10.0 Configuration (32bit)
- SQL Server Network Configuration
  - Protocols for MSSQLSERVER
  - Protocols for KJ
- SQL Native Client 10.0 Configuration

Protocol Name	Status
Shared Memory	Enabled
Named Pipes	Disabled
TCP/IP	Enabled
VIA	Disabled

2. Next, add lpc: to your data source in the connection string to force shared memory.



General

Name	Adventure Works DW
ID	Adventure Works DW
Description	
Create Timestamp	1/3/2011 10:22:00 AM
Last Schema Update	1/31/2011 4:24:21 AM
Connection String	Data Source=lpc:johnsi5\kj;Initial Catalog=AdventureWorksDW
Maximum Number of Connections	10

3. After you start processing, you can verify your connection is using shared memory by executing the following SELECT statement.

```
SELECT session_id, net_transport, net_packet_size
FROM sys.dm_exec_connections
```

The **net_transport** for the Analysis Services SPID should show: **Shared memory**.

For more information about shared memory connections, see “Creating a Valid Connection String Using Shared Memory Protocol” (http://msdn.microsoft.com/en-us/library/ms187662.aspx).

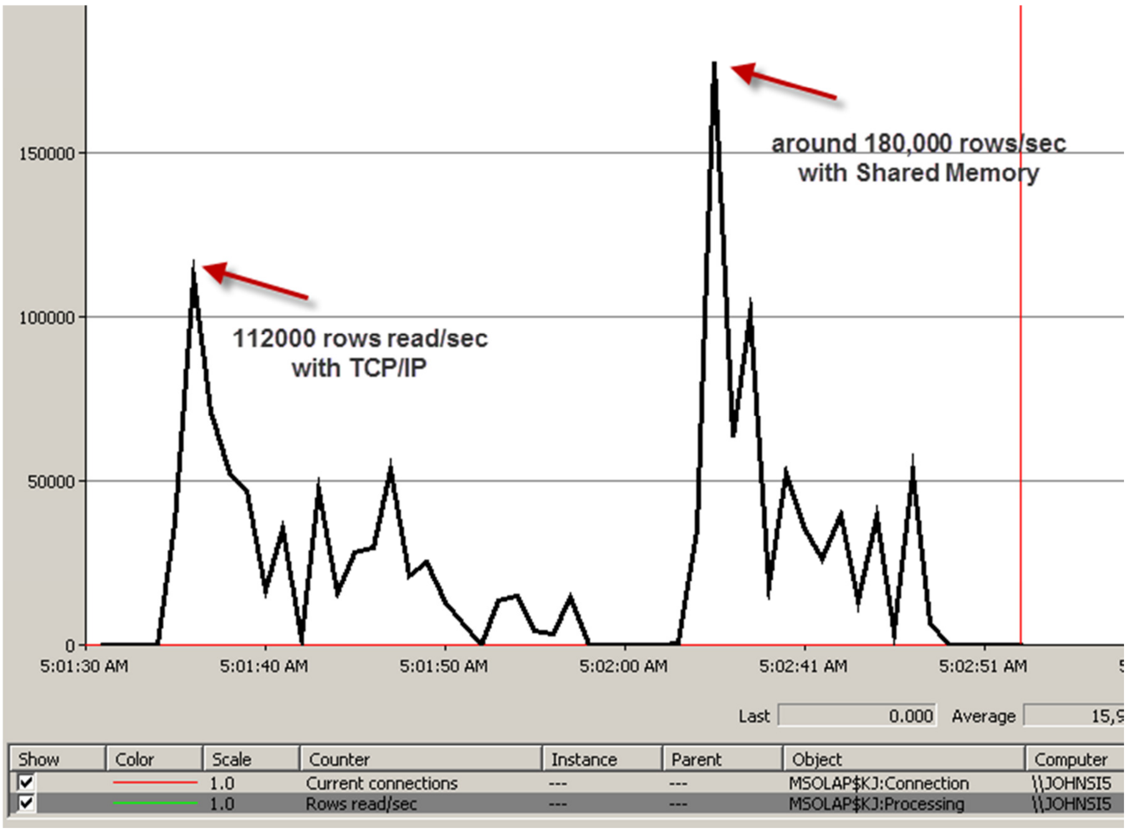
To compare TCP/IP and shared memory, we ran the following two processing commands.

```
<!--SQL Server Native Client with TCP-->
<Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
  <Parallel>
  <Process …
    <Object>
      <DatabaseID>Adventure Works DW 2008R2</DatabaseID>
      <CubeID>Adventure Works</CubeID>
    </Object>
    <Type>ProcessFull</Type>
    <DataSource xsi:type="RelationalDataSource">
      <ID>Adventure Works DW</ID>
      <Name>Adventure Works DW</Name>
      <ConnectionString>
      Provider=SQLNCLI10.1;Data Source=tcp:johnsi5\kj;
```

27

```
      Integrated Security=SSPI;Initial Catalog=AdventureWorksDW2008R2;
      </ConnectionString>
      <ImpersonationInfo>
        <ImpersonationMode>ImpersonateCurrentUser</ImpersonationMode>
      </ImpersonationInfo>
      <Timeout>PT0S</Timeout>
    </DataSource>
  </Process>
    </Parallel>
</Batch>

<!--SQL Native Client with shared memory-->
<Batch xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
  <Parallel>
    <Process …
    <Object>
      <DatabaseID>Adventure Works DW 2008R2</DatabaseID>
      <CubeID>Adventure Works</CubeID>
    </Object>
    <Type>ProcessFull</Type>
    <DataSource xsi:type="RelationalDataSource">
      <ID>Adventure Works DW</ID>
      <Name>Adventure Works DW</Name>
      <ConnectionString>
      Provider=SQLNCLI10.1;Data Source=lpc:johnsi5\kj;
      Integrated Security=SSPI;Initial Catalog=AdventureWorksDW2008R2;
      </ConnectionString>
      <ImpersonationInfo>
        <ImpersonationMode>ImpersonateCurrentUser</ImpersonationMode>
      </ImpersonationInfo>
      <Timeout>PT0S</Timeout>
    </DataSource>
  </Process>
    </Parallel>
</Batch>
```

The first command using TCP/IP maxed out at 112,000 rows per second. Because the data source for the cube was on the same machine as Analysis Services, we were able to use shared memory in the second processing command and get much better throughput: 180,000 max rows/sec.



Last 0.000 Average 15,9

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1.0	Current connections	---	---	MSOLAP$KJ:Connection	\\JOHNSI5
✓	🟩	1.0	Rows read/sec	---	---	MSOLAP$KJ:Processing	\\JOHNSI5

Figure 10 - Comparing rows/sec throughput

2.6.2 High-Speed Networking Features

Windows Server 2008 R2 has numerous improvements in network virtualization support that enable enhanced networking support. Windows Server 2008 R2 has also enhanced the support of jumbo packets and TCP offloading that was introduced in Windows Server 2008. Additionally Virtual Machine Queue (VMQ) support was added to allow network routing and data copy processing to be offloaded to a physical NIC. These features were introduced to take advantage of the capabilities found in the 10GbE server NICs.

2.6.2.1 TCP Chimney

TCP Chimney is a networking technology that transfers TCP/IP protocol processing from the CPU to a network adapter during the network data transfer. There have been some issues with enabling TCP Chimney in the past, and for that reason many people recommended turning it off. The technology has matured and many of the issues reported were specific to the NIC manufacturer. Applications that have long-lived connections transferring a lot of data benefit the most from the TCP Chimney feature. Processing Analysis Services data from a remote server falls into this category, so we recommend that you make sure that TCP Chimney is enabled and configured correctly. TCP Chimney can be enabled and disabled in the operating system and in the advanced properties of the network adapter.

To enable TCP Chimney you need to perform the following:

1. Enable TCP chimney in the operating system using the **netsh** commands.
2. Ensure the physical network adapter supports TCP Chimney offload, and then enable it for the adapter in the network driver.

TCP chimney has three modes of operation: Automatic (new in Windows Server 2008 R2), Enabled, and Disabled. The default mode in Windows Server 2008 R2, Automatic, checks to make sure the connections considered for offloading meet the following criteria:

- 10Gbps Ethernet NIC installed and connection established through 10GbE adapter
- Mean round trip link latency is less than 20 milliseconds
- Connection has exchanged at least 130 KB of data

To determine whether TCP Chimney is enabled, run the following from an elevated command prompt.

```
netsh int tcp show global
```

In the results, check the Chimney Offload State setting.

```
TCP Global Parameters
----------------------------------------------
Receive-Side Scaling State                  : enabled
Chimney Offload State                       : automatic
NetDMA State                                : enabled
Direct Cache Access (DCA)                   : disabled
Receive Window Auto-Tuning Level            : normal
Add-On Congestion Control Provider          : none
ECN Capability                              : disabled
RFC 1323 Timestamps                         : disabled
```

In this example, the results show that TCP Chimney offloading is set to automatic. This means that it is enabled as long as the requirements mentioned earlier are met.

After you verify that TCP Chimney is enabled at the operating system level, check the NIC settings in device manager:

1. Go to start | run and type **devmgmt.msc**.
2. In Device Manager, expand Network Adapters, and right-click the name of the physical NIC adapter, and then click **Properties**.
3. On the **Advanced** tab, under **Property**, click **TCP Chimney Offload** or **TCP Connection Offload**, and then under **Value**, verify that **Enabled** is displayed. You may need to do this for both IPv4 and IPv6. Note that **TCP Checksum Offload** is not the same as **TCP Chimney Offload**.

If the system has IPsec enabled, no TCP connections are offloaded, and TCP Chimney provides no benefit. There are numerous different configuration options for TCP Chimney offloading that our outside the scope of this white paper. See the references section for a deeper treatment.

References:

- Windows Server 2008 R2 Networking Deployment Guide: Deploying High-Speed Networking Features http://download.microsoft.com/download/8/E/D/8EDE21BC-0E3B-4E14-AAEA-9E2B03917A09/HSN_Deployment_Guide.doc
- Windows Server 2008 R2 Networking Deployment Guide http://download.microsoft.com/download/8/E/D/8EDE21BC-0E3B-4E14-AAEA-9E2B03917A09/HSN_Deployment_Guide.doc

2.6.2.2 Jumbo Frames

Jumbo frames are Ethernet frames with more than 1,500 and up to 9,000 bytes of payload. Jumbo frames have been shown to yield a significant throughput improvement during Analysis Services processing, specifically Process Data. Network throughput is increased while CPU usage is minimized. Jumbo frames are only available on gigabit networks, and all devices in the network path must support them (switches, routers, NICs, and so on).

Most default Ethernet set ups are configured to have MTU sizes of up to 1,500 bytes. Jumbo frames allow you to go up to 9,000 bytes in a single transfer. To enable jumbo frames:

1. Configure all routers to support jumbo frames.
2. Configure the NICs on the Analysis Services machine and the data source machine to support jumbo frames.
    a) Click the **Start** button, point to **Run**, and then type **ncpa.cpl**.
    b) Right-click your network connection, and then click **Properties**.
    c) Click **Configure,** and then click the **Advanced** tab. Under **Property**, click **Jumbo Frame**, and then under **Value**, change the value from **Disabled** to **9kb MTU** or **9014**, depending on the NIC.
    d) Click **OK** on all the dialog boxes. After you make the change, the NIC loses network connectivity for a few seconds and you should reboot.

3. After you configure jumbo frames, you can easily test the change by pinging the server with a large transfer:

   Ping <servername> -f –l 9000

   You should only measure one network packet per ping request in Network Monitor.

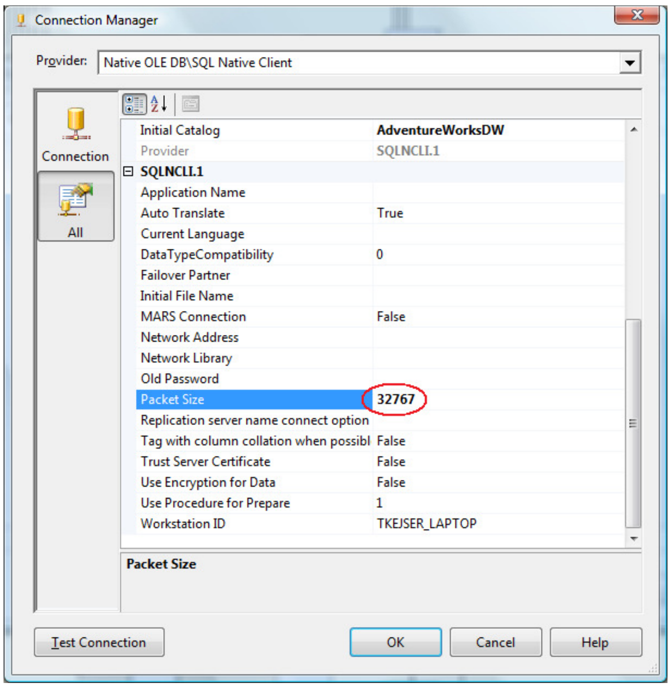
2.6.3 Hyper-V and Networking

If you are running Analysis Services and SQL Server in a Hyper-V virtual machine, there are a few things you should be aware of specific to networking.

When you enable Hyper-V and create an external virtual network, both the guest and the host machine go through a virtual network adapter to connect to the physical network. All traffic goes through your virtual network adapter, and there is additional latency overhead associated with this.

2.6.4 Increase Network Packet Size

Under the properties of your data source, increasing the network packet size for SQL Server minimizes the protocol overhead require to build many, small packages. The default value for SQL Server 2008 is 4096. With a data warehouse load, a packet size of 32K (in SQL Server, this means assigning the value 32767) can benefit processing. Don’t change the value in SQL Server using **sp_configure**; instead override it in your data source. This can be set whether you are using TCP/IP or Shared Memory.



Provider: Native OLE DB\SQL Native Client

Initial Catalog	AdventureWorksDW
Provider	SQLNCLI.1
⊟ SQLNCLI.1	
Application Name	
Auto Translate	True
Current Language	
DataTypeCompatibility	0
Failover Partner	
Initial File Name	
MARS Connection	False
Network Address	
Network Library	
Old Password	
Packet Size	32767
Replication server name connect option	
Tag with column collation when possibl	False
Trust Server Certificate	False
Use Encryption for Data	False
Use Procedure for Prepare	1
Workstation ID	TKEJSER_LAPTOP

Packet Size

Figure 11 Tuning network packet size

2.6.5 Using Multiple NICs

If you are hitting a network bottleneck when you process your cube, you can add additional NICs to increase throughput. There are two basic configurations for using multiple NICs on a server. The first and default option is to use the NICs separately. The second option is to use NIC teaming.

Multiple NICs can be used individually to concurrently run many multipartition processing commands. Each partition in Analysis Services can refer to a different data source, and it is this feature you make use of. To use multiple NICs, follow these steps:

1. Set up NICs with different IP numbers.
2. Add new data sources.
3. Change your partition setup to reference the new data sources.

First, set up multiple NICs in the source and the target, each with its own IP number. Match each NIC on the cube server with a corresponding NIC on the relational database. Set up the subnet mask so only the desired NIC on the source can be reached from its partner on the cube server. If you are limited in bandwidth on the switches or want to create a very simple setup, you can use this technique with a crossover cable, which we have done successfully

Second, add a data source for each NIC on the data source, and set up each data source to point to its own source NIC.

Third, configure partitions in the cube so that they are balanced across all data sources. For example, if you have 16 partitions and 4 data sources per NIC – point 4 partitions to each data source.

Schematically, the setup looks like this.

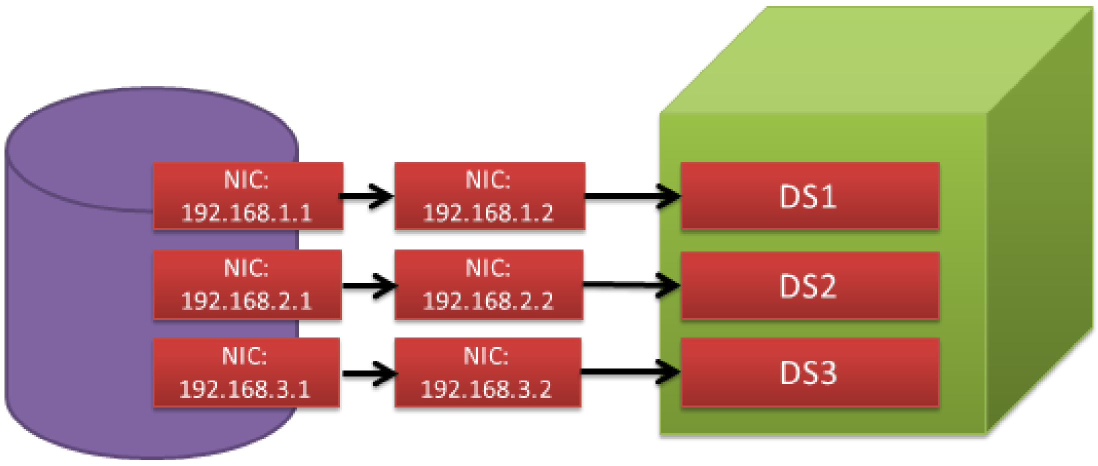


Figure 12 - Using multiple NICs for processing

NIC teaming: Another option is to team multiple NICs so they appear as one NIC to Windows Server. It is difficult to make specific recommendations around using teaming, because performance is specific to the hardware and drivers used for teaming. To determine whether NIC teaming would be beneficial to your processing workload, test with it both enabled and disabled, and measure your performance results with both settings.

2.7 Disabling Flight Recorder

Flight Recorder provides a mechanism to record Analysis Services server activity into a short-term log. Flight Recorder provides a great deal of benefit when you are trying to troubleshoot specific querying and processing problems by logging snapshots of common DMV into the log file. However, it introduces an amount of I/O overheard on the server itself.

If you are in a production environment and you are already monitoring the Analysis Services instance using this guide, you do not need Flight Recorder capabilities and you can disable its logging and remove the I/O overhead. The server property that controls whether Flight Recorder is enabled is the **Log\Flight Recorder\Enabled** property. By default, this property is set to **true**.

3 Monitoring and Tuning the Server

As part of healthy operations management, you must collect data that allows both reactive and proactive behaviors that increase system stability, performance, and integrity. The temptation is to collect a lot of data in the belief that with more knowledge comes better decisions. This is not always the case, because you may either overload the server with data collection overhead or collect data points that you cannot take any action on.

Hence, for every data point you collect, you should have an idea about what that data collection helps you achieve. This guide provides you with the knowledge you need to interpret the measurements you configure and how to take action on them. For ease of reference, this section summarizes the data collection required. You can use it as a checklist for data collection.

The tool used to collect the data does not matter much; it is often a question of preference and operational procedures. What this guide provides is the source of the data points – how you aggregate and collect them will depend on your organization.

3.1 Performance Monitor

On an Analysis Services server, the minimal collection of data is this.

Performance object	Counter	Description
Logical Disk	All / All	Collects data about disk load and utilization.
Process	Private Bytes – msmdsrv.exe	The memory consumed by the Analysis Services process.
MSOLAP:Memory	Memory Usage Kb	Alternative to **Process**.
MSOLAP:Connection	Current Connections	Measures the number of open connections to gauge concurrency.
MSOLAP:Threads	*	Allows tuning of thread pools.
Memory	Cache Bytes Standby Cache Normal Priority Bytes Standby Cache Core Bytes Standby Cache Reserve Bytes	Estimates the size of the file system cache.
Memory	Page Faults / sec	Tests for excessive paging of the server.
Memory	Available Bytes	Used to tune memory settings.
MSOLAP: Proc Aggregations	temp file bytes written	Measures the spill from processing operations. Ideally, cube and hardware should be balanced to make this 0.
MSOLAP: Proc Indexes	Current Partitions Rows/sec	Measures speed and concurrency of process index.
MSOLAP: Processing	Rows read/sec Rows written/sec	Measures speed of relation read and efficiency of merge buffers.
MSOLAP:MDX	*	Used by cube tuners to determine whether calculation scripts or MDX queries can be

		improved.
System	File Read Bytes/sec	Measure bytes read from the file system cache.
System	File Read Operations/sec	Measures IOPS from file system cache
Network Interface	Bytes Received/sec Bytes Sent/sec	Contains the capacity plan to follow if NIC speed slows. See section 2.6.5.
TCPv4 and **TCPv6**	Segments / sec Segments Retransmitted/sec	Discovers unstable connections.

In most cases, you can collect this information every 15 seconds without causing measurable impact to the system.

If you are using Systems Center Operations Manager (SCOM) to monitor servers, it is a good idea to add this data collection to your monitoring. Alternatively, you can use the built in performance monitor in Windows Server, but that will require that you harvest the counters from the servers yourself.

3.2 Dynamic Management Views

Starting with SQL Server 2008 Analysis Services, there is a new set of monitoring tools available to you: dynamic management views (DMVs). These views provide information about the operations of the service that you cannot find in SQL Server Profiler or Performance Monitor.

The following table lists the most useful DMVs collect on a regular basis from the server.

DMV	Description
$system.discover_commands	Lists all currently running commands on the server, including commands run by the server itself.
$system.discover_sessions	List all sessions on the server. It is used to map commands and connections together.
$system.discover_connections	Lists current open connections.
$system.discover_memoryusage	Lists all memory allocations in the server.
$system.discover_locks	Lists currently held and requested locks.

Your capture rate of the data depends on the system you are running and how quickly your operations team response to events. **$system.discover_locks** and **$system.discover_memoryusage** are expensive DMVs to query – and gathering them too often consumes significant CPU resources on the server. Capturing the DMVs once every few minutes is probably enough for most operations management purposes. If you choose to capture them more often, measure the impact on your server, which will depend on the concurrency of executing sessions, memory sizes, and cube design.

Depending on how often you query the DMV, you can generate a lot of data. It is often a good idea to keep the recent data at a high granularity and aggregate older data. Using a tool like Microsoft StreamInsight enables you to perform such historical aggregation in real time.

Capturing the DMV data enables you to monitor the progress of queries and alert you to heavy resource consumers early. Here are some examples of issues you can identify when you use DMVs:

- Queries that consume a lot of memory
- Queries that have been blocked for a long time
- Locks that are held for a long time
- Sessions that have run for a long time, or consumed a lot of I/O
- Connections that are transmitting a large amount of data over the network

One way to collect this data is to use a linked server from the SQL Server Database Engine to Analysis Services and use the Database Engine as the storage for the data you collect. You may also be able to configure your favorite monitoring tool to do the same. Unfortunately, Analysis Services does not ship with a fully automated tool for this data collection.

The following example collection scripts can get you started with a basic data collection framework:

```
select SESSION_SPID /* Key in commands */
/* Monitor for large values and large different with below */
, COMMAND_ELAPSED_TIME_MS
, COMMAND_CPU_TIME_MS /* Monitor for large values */
, COMMAND_READS    /* Monitor for large values */
, COMMAND_WRITES /* Monitor for large values */
, COMMAND_TEXT /* Track any problems to the query */
from $system.discover_commands

select SESSION_SPID     /* Join to SPID in Sessions */
, SESSION_CONNECTION_ID /* Join to connection_id in Connections */
, SESSION_USER_NAME /* Finds the authenticated user */
, SESSION_CURRENT_DATABASE
from $system.discover_sessions

select CONNECTION_ID    /* Key in connections */
, CONNECTION_HOST_NAME /* Find the machine the user is coming from */
, CONNECTION_ELAPSED_TIME_MS
, CONNECTION_IDLE_TIME_MS /* Find clients not closing connections */
/* Monitor the below for large values */
, CONNECTION_DATA_BYTES_SENT
, CONNECTION_DATA_BYTES_RECEIVED
from $system.discover_connections

select SPID
, MemoryName
, MemoryAllocated
, MemoryUsed
, Shrinkable
, ObjectParentPath
, ObjectId
from $system.discover_memoryusage
where SPID <> 0
```

References:

36

- MSDN library reference on Analysis Services DMVs: http://msdn.microsoft.com/en-us/library/ee301466.aspx
- CodePlex ResMon tool, which captures detailed information about cubes: http://sqlsrvanalysissrvcs.codeplex.com/wikipage?title=ResMon%20Cube%20Documentation

3.3 Profiler Traces

Analysis Services exposes a large number of events through the profiler API. Collecting every single one during normal operations not only generates high data collection volumes, it also consumes significant CPU and network traffic. While it is possible to dig into great detail of the server, you should measure to impact on the system while running the trace. If every event is traced, a lot of trace information is generated: Make sure you either clean up historical records or have enough space to hold the trace data. Very detailed SQL Server Profiler traces are best reserved for situations where you need to do diagnostics on the server or where you have enough CPU and storage capacity to collect lots of details.

References:

- ASTrace – a tool to collect profiler traces into SQL Server for further analysis: http://msftasprodsamples.codeplex.com/wikipage?title=SS2005%21Readme%20for%20ASTrace%20Utility%20Sample&referringTitle=Home

3.4 Data Collection Checklist

You can use the following checklist to make sure you have covered the basic data collection needs of the server:

- ✓ Windows Server-specific Performance Monitor counters
- ✓ Analysis Services-specific Performance Monitor counters
- ✓ Data Management Views collected:
    - $system.discover_commands
    - $system.discover_sessions
    - $system.discover_connections
    - $system.discover_memoryusage
    - $system.discover_locks

3.5 Monitoring and Configuring Memory

To discover the best memory configuration for your Analysis Services instance, you need to collect some data about the typical usage of the system.

First of all, you should start up the server with Analysis Services disabled and make sure that all the other services you need for normal operations in a state that is typical for your standard server configuration. What typical is varies by environment –you are looking to measure how much memory will be available to Analysis Services after the operating system and other services (such as virus scanners and monitoring tools) have taken their share. Note down the value of the Performance Monitor counter **Memory/Available Bytes**.

Secondly, start Analysis Services and run a typical user workload on the server. You can for example use queries from your test harness. For this test, also make sure **PreAllocate** is set to 0. While the workload is running, run the following query.

```
SELECT * FROM $SYSTEM.DISCOVER_MEMORYUSAGE
```

If you have any long-running, high-memory consuming queries, you should also measure how much memory they consume while they execute.

Copy the data into an Excel spreadsheet for further analysis. You can also use the CodePlex project ResMon cube (see reference section) to periodically log snapshots of this DMV and browse memory usage trends summarized in a cube.

With the data collected in the previous section, you can define some values needed to set the memory:

[Physical RAM] = The total physical memory on the box

[Available RAM] = The value of the Performance Monitor counter **Memory/Available** Bytes as noted down earlier.

[Non Shrinkable Memory] = The sum of the **MemoryUsed** column from $SYSTEM.DISCOVER_MEMORYUSAGE where the column **Shrinkable** is **False.**

[Valuable Objects] = The sum of all objects that you want to reserve memory for from $SYSTEM.DISCOVER_MEMORYUSAGE where column **Shrinkable** is **True**. For example, you will most likely want to reserve memory for all dimensions and attributes. If you have a small cube, this value may simply be everything that is marked as shrinkable.

[Worst Queries Memory] = The amount of memory used by the all the most demanding queries you expect will run concurrently. You can use the DMV **$system.discover_memoryusage** to measure this on a known workload.

With the preceding values you can calculate the memory configuration for Analysis Services:

LowMemoryLimit = [Non Shrinkable Memory] + [Valuable Objects] / [Total RAM] But keep a gap of at least 20 percent between this value and **HighMemoryLimit** to allow the memory cleaners to release memory at a good rate.

HighMemoryLimit = ([Available RAM] - [Worst Queries Memory]) / [Total RAM]

HardMemoryLimit = [Available RAM] / [Total RAM]

LimitSystemFileCacheMB = [Available RAM] - LowMemoryLimit

Note that it may not always be possible to measure all the components that make up these equations. In such case, your best bet is to guesstimate them. The idea behind this method is that Analysis Services will always hold on to enough memory to keep the valuable objects in memory. What is valuable depends on your particular setup and query set. The rest of the available memory is shared between the file system cache and the Analysis Services caches; if you use the operating systems memory usage optimizations, the ideal balance between Analysis Services and the file system cache is adjusted dynamically.

If Analysis Services coexists with other services on the machine, take their maximum memory consumption into consideration. When you calculate **[Available RAM]**, subtract the maximum memory use by other large memory consumers, such as the SQL Server relational engine. Also, make sure those other memory consumers have their maximum memory settings adjusted to allow space for Analysis Services.

The following diagram illustrates the different uses of memory.

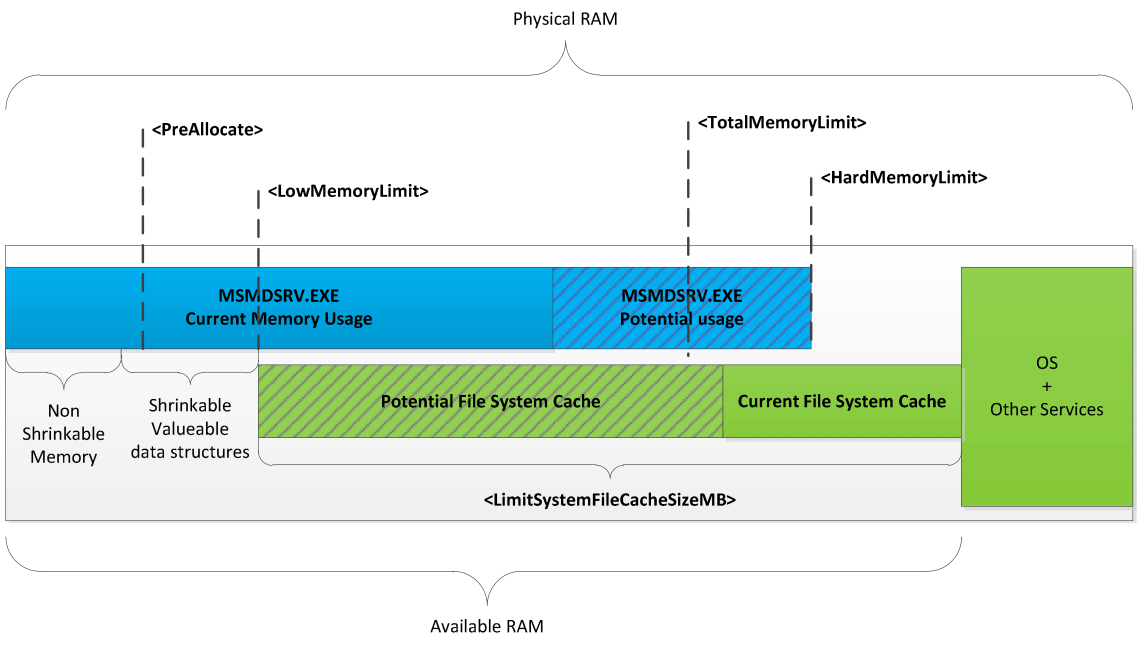


Figure 13 - Memory Settings

References:

- CodePlex ResMon project - http://sqlsrvanalysissrvcs.codeplex.com/wikipage?title=ResMon%20Cube%20Documentation

3.6 Monitoring and Tuning I/O and File System Caches

To tune the Analysis Services I/O subsystem, it is helpful to understand how a user query is translated into I/O requests. The following diagram illustrates the caching and I/O architecture of Analysis Services.

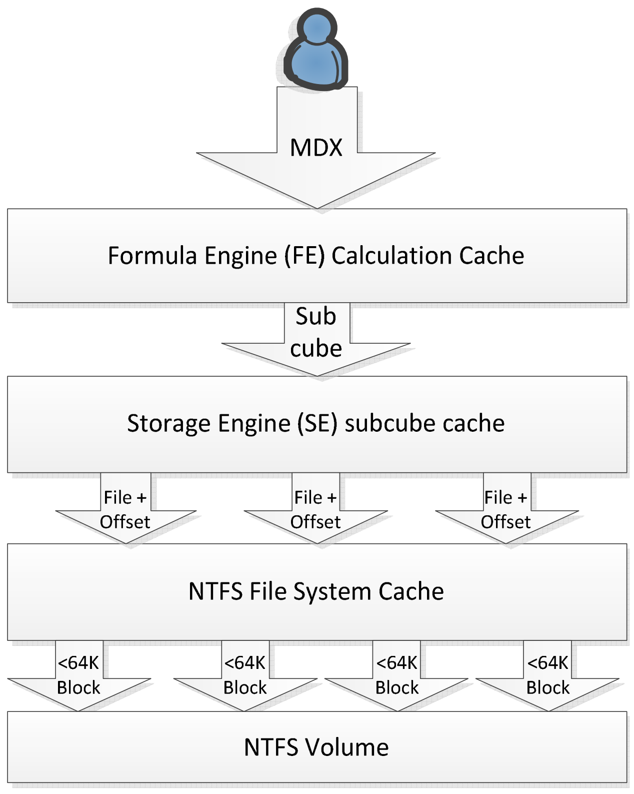


Figure 14 - I/O stack

From the perspective of the I/O system, the storage engine requests data from an offset in a file. The building blocks of the storage used by Analysis Services are the files used to store dimension and fact data. Dimension data is typically cached by the storage engine. Hence, the most frequently requested files from the storage system are measure group data, and they have the following file names: ***.fact.data, *.aggs.rigid.data** and ***.agg.flex.data**. Because Analysis Services uses buffered I/O, frequently requested blocks in files are typically retained by the NTFS file system cache. Note that this means that the memory used for the file system caching of data comes from outside the Analysis Services working set.

When a block from a file is requested by Analysis Services, the path taken by the operating system depends on the state of the cache.

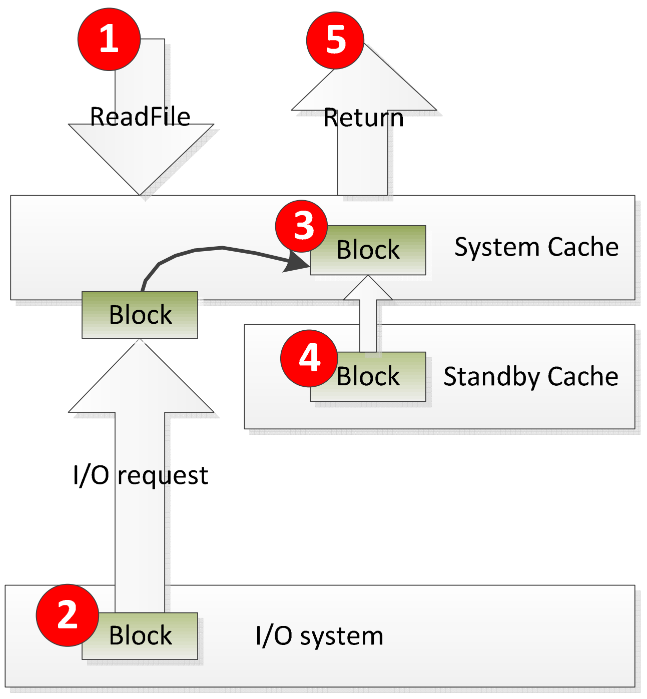


Figure 15 - Accessing a file in the NTFS cache

In the preceding illustration, Analysis Services executes the ReadFileEx call to the Windows API.

- If the block is not in the cache, an I/O request is issued (2), the file is put in the cache (3), and the data is returned to Analysis Services (5).
- If he block is already in the cache (3), it is simply returned to Analysis Services (5).
- If a block is not frequently accessed or if there is memory pressure, the NTFS file system cache may choose to place the block in the Standby Cache (4) and eventually remove the file from memory. If the file is requested before it is removed, it is brought back into the System Cache (3) and then returned to Analysis Services (5).
    - Note that in Windows Server 2003, the file is simply removed from the system cache, bypassing the standby cache.

3.6.1 System Cache vs. Physical I/O

Because Analysis Services uses the NTFS cache, not every I/O request reaches the I/O subsystem. This means that even with a clear storage engine and formula engine cache, query performance will still vary depending on the state of the file system cache. In the worst case, a query will run on a clean file system cache and every I/O request will hit the physical disk. In the best case, every I/O request will be served by the memory cache. This depends on the amount of data requested by the query.

It is useful to know the ratio between the I/O issues to the disk and the I/O served by the file system cache. This helps you capacity plan for growing cube. Small cubes, less than the size of the server memory, are typically served mostly from the file system cache. However, as cubes grow larger and exceed the file system cache size, you will eventually have to assist cube performance with a fast disk system. Measuring the current cache hit ratio helps shed light on this.

To measure the I/O served by the file system cache, use Performance Monitor to measure **System:File Read Bytes/sec** and **System File Read Operations/sec**. These counters will give you an estimate of the number of I/O operations and file bytes that are served from memory. By comparing this with **Logical Disk / Disk reads /sec** you can measure the ratio between memory buffered and physical I/O. As the amount of data in the cube grows, you will often see that the amount of disk access begins to grow in proportion to memory access. By extrapolating from historical trends, you can then decide what the best ratio between IOPS and RAM sizes should be going forward.

To measure the total memory consumption of Analysis Services, you will also have to take the file system cache into account. The total memory used by analysis services is the sum of:

- **Process – msmdsrv.exe / Private Bytes** – The memory used by Analysis Services itself
- **Memory – Cache Bytes** – The files currently live in cache
- **Memory – Standby Cache Normal Priority Bytes** – The files that can be freed from the cache

However, note that the memory counters also measure other files' caches in the NTFS file system cache. If you are running Analysis Services together with other services on the machine, the file system counter may not accurately reflect the caches used by Analysis Services.

3.6.2 TempDir Folder

When memory is scarce on the server, Analysis Services spills operations to disk. Examples of such operation are:

- Dimension processing
    - ByTable processing commands that don't fit memory
    - Processing of large dimension where the hash tables created exceed available memory
- Aggregation processing
- ROLAP dimension attribute stores

Dimension processing: To optimize dimension processing, we recommend that you use the techniques described later in this document to avoid spills and speed up processing. **ByTable** should generally only be used if you can keep the dimension data in memory during processing.

Aggregation processing: During cube processing, aggregation buffers described in configuration section determine the amount of memory that is available to build aggregations for a given partition. If the aggregation buffers are too small, Analysis Services supplements the aggregation buffers with temporary files. To monitor any temporary files used during aggregation, review **MSOLAP:Proc Aggregations\Temp file bytes written/sec**. You should try to design your cube in such a way that memory spills to temp files does not occur, that is, keep the temp bytes written at zero. There are several techniques available to avoid memory spills during aggregation processing:

- Only create aggregations that are small enough to fit in memory during processing.
- Process fewer partitions in parallel.
- Add more memory to the machine or allocating more memory to Analysis Services.

It is generally a good idea to split **ProcessData** and **ProcessIndex** operations into two different processing jobs. **ProcessData** typically consumes less memory than **ProcessIndex** and you run many **ProcessData** operations in parallel. During **ProcessIndex**, you can then decrease concurrency if you are short on memory, to avoid the disk spill.

ROLAP dimensions: In general, you should try to avoid ROLAP dimensions; MOLAP stores are much faster for dimension access. However, the requirement for ROLAP dimensions is often driven by a lack of sufficient memory to hold the attribute stores requires for drillthrough actions, which returns data using a degenerate ROLAP dimension. If this is your scenario, you may not be able to avoid spills to the **TempDir** folder.

If you cannot design the cube to fit all operations in memory, spilling to **TempDir** may be your only option. If that is the case, we recommend that you place the **TempDir** folder on a fast LUN. You can either use a LUN backed by caches (for example in a SAN) or one that sits on fast storage – for example, NAND devices.

3.7 Monitoring the Network

One way to easily monitor network throughput is through a performance monitor trace. Windows Performance Monitor requests the current value of performance counters at specified time intervals and logs them into a trace log (.blg or .csv).

The following counters will help you isolate any problems in the network layer.

Processing: Rows read/sec – this is the rate of rows read from your data source. This is one of the most important counters to monitor when you measure your throughput from Analysis Services to your relational data source. When you monitor this counter, you will want to view the trace using a line chart, as it gives you a better idea of your throughput relative to time. It is reasonable to expect tens of thousands of rows per second per network connection to a SQL Server data source. Third-party sources may experience significantly slower throughput.

Bytes received/sec and **Bytes sent/sec** - These two counters enable you to measure how far you are from the theoretical NIC speeds. It allows capacity planning for faster NIC.

TCPv4 and TCPv6 Segments/sec and **Segments retransmitted/sec** – These counters enable you to discover unstable network connections. The ratio between segments retransmitted and segments for both TCPv4 and TCPv6 should not exceed 3-4 percent on a stable network.

4 Testing Analysis Services Cubes

As you prepare for user acceptance and preproduction testing of a cube, you should first consider what a cube is and what that means for user queries. Depending on your background and role in the development and deployment cycle, there are different ways to look at this.

As a database administrator, you can think of a cube as a database that can accept *any* query from users, and where the response time from any such query is expected to be “reasonable” – a term that is often vaguely defined. In many cases, you can optimize response time for specific queries using aggregates (which for relational DBAs is similar to index tuning), and testing should give you an early idea of good aggregation candidates. But even with aggregates, you must also consider the worst case: You should expect to see leaf-level scan queries. Such queries , which can be easily expressed by tools like Excel, may end up touching every piece of data the cube. Depending on your design, this can be a significant amount of data. You should consider what you want to do with such queries. There are multiple options: For example, you may choose to scale your hardware to handle them in a decent response time, or you may simply choose to cancel them. In either case, as you prepare for testing, make sure such queries are part the test suite and that you observe what happens to the Analysis Services instance when they run. You should also understand what a “reasonable” response time is for your end user and make that part of the test suite.

As a BI developer, you can look at the cube as your description of the multidimensional space in which queries can be expressed. A part of this space will be instantiated with data structures on disk supporting it: dimensions, measure groups, and their partitions. However, some of this multidimensional space will be served by calculations or ad-hoc memory structures, for example: MDX calculations, many-to-many dimensions, and custom rollups. Whenever queries are not served directly by instantiated data, there is a potential, query-time calculation price to be paid, this may show up as bad response time. As the cube developer, you should make sure that the testing covers these cases. Hence, as the BI-developer, you should make sure the test queries also stress noninstantiated data. This is a valuable exercise, because you can use it to measure the impact on users of complex calculations and then adjust the data model accordingly.

Your approach to testing will depend on which situation you find yourself in. If you are developing a new system, you can work directly towards the test goals driven by business requirements. However, if you are trying to improve an existing system, it is an advantage to acquire a test baseline first.

4.1 Testing Goals

Before you design a test harness, you should decide what your testing goals will be. Testing not only allows you to find functional bugs in the system, it also helps quantify the scalability and potential bottlenecks that may be hard to diagnose and fix in a busy production environment. If you do not know what your scale-barrier is or where they system might break, it becomes hard to act with confidence when you tune the final production system.

Consider what characteristics are reasonable to expect from the BI systems and document these expectations as part of your test plan. The definition of *reasonable* depends to a large extent on your familiarity with similar systems, the skills of your cube designers, and the hardware you run on. It is often a good idea to get a second opinion on what test values are reachable – for example from a neutral third party that has experience with similar systems. This helps you set expectations properly with both developers and business users.

BI systems vary in the characteristics organizations require of them – not everyone needs scalability to thousands of users, tens of terabytes, near-zero downtime, and guaranteed subsecond response time. While all these goals can be achieved for most cases, it is not always cheap to acquire the skills required to design a system to support them. Consider what your system needs to do for your organization, and avoid overdesigning in the areas where you don't need the highest requirements. For example: you may decide that you need very fast response times, but that you also want a very low-cost server that can run in a shared storage environment. For such a scenario, you may want to reduce the data in the cube to a size that will fit in memory, eliminating the need for the majority of I/O operations and providing fast scan times even for poorly filtered queries.

Here is a table of potential test goals you should consider. If they are relevant for your organization's requirements, you should tailor them to reflect those requirements.

Test Goal	Description	Example goal
Scalability	How many concurrent users should be supported by the system?	"Must support 10,000 concurrently connected users, of which 1,000 run queries simultaneously."
Performance/ throughput	How fast should queries return to the client? This may require you to classify queries into different complexities. Not all queries can be answered quickly and it will often be wise to consult an expert cube designer to liaise with users to understand what query patterns can be expected and what the complexity of answering these queries will be. Another way to look at this test goal is to measure the throughput in queries answered per second in a mixed workload.	"Simple queries returning a single product group for a given year should return in less than 1 second – even at full user concurrency." "Queries that touch no more than 20% of the fact rows should run in less than 30 seconds. Most other queries touching a small part of the cube should return in around 10 seconds. With our workload, we expect throughput to be around 50 queries returned per second." "User queries requesting the end-of-month currency rate conversion should return in no more than 20 seconds. Queries that do not require currency conversion should return in less than 5 seconds."

Data Sizes	What is the granularity of each dimension attribute? How much data will each measure group contain? Note that the cube designers will often have been considering this and may already know the answer.	“The largest customer dimension will contain 30 million rows and have 10 attributes and two user hierarchies. The largest non-key attribute will have 1 million members.” “The largest measure group is sales, with 1 billion rows. The second largest is purchases, with 100 million rows. All other measure groups are trivial in size.”
Target Server Platforms	Which server model do you want to run on? It is often a good idea to test on both that server and an even bigger server class. This enables you to quantify the benefits of upgrading.	“Must run on 2-socket 6-core Nehalem machine with 32 GB of RAM.” “Must be able to scale to 4-socket Nehalem 8-core machine with 256 GB of RAM.
Target I/O system	Which I/O system do you want to use? What characteristics will that system have?	“Must run on corporate SAN and use no more than 1,000 random IOPS at 32,000 block sizes at 6ms latency.” “Will run on dedicated NAND devices that support 80,000 IOPS at 100 µs latency.”
Target network infrastructure	Which network connectivity will be available between users and Analysis Services, and between Analysis Services and the data sources? Note that you may have to simulate these network conditions in a lab.	“In the worst case scenario, users will connect over a 100ms latency WAN link with a maximum bandwidth of 10Mbit/sec.” “There will be a 10Gbit dedicated network available between the data source and the cube.”
Processing Speeds	How fast should rows be brought into the cube and how often?	“Dimensions should be fully processed every night within 30 minutes.” “Two times during the day, 100,000,000 rows should be added to the sales measure group. This should take no longer than 15 minutes.”

4.2 Test Scenarios

Based on the considerations from the previous section you should be able to create a user workload that represents typical user behavior and that enables you to measure whether you are meeting your testing goals.

Typical user behavior and well-written queries are unfortunately not the only queries you will receive in most systems. As part of the test phase, you should also try to flush out potential production issues before they arise. We recommend that you make sure your test workload contains the following types of queries and tests them thoroughly :

- ✓ Queries that touch several dimensions at the same time
- ✓ Enough queries to test every MDX expression in the calculation script
- ✓ Queries that exercise many-to-many dimensions
- ✓ Queries that exercise custom rollups
- ✓ Queries that exercise parent/child dimensions
- ✓ Queries that exercise distinct count measure groups
- ✓ Queries that crossjoin attributes from dimensions that contain more than 100,000 members
- ✓ Queries that touch every single partition in the database
- ✓ Queries that touch a large subset of partitions in the database (for example, current year)
- ✓ Queries that return a lot of data to the client (for example, more than 100,000 rows)
- ✓ Queries that use cube security versus queries that do not use it
- ✓ Queries executing concurrently with processing operations – if this is part of your design

You should test on the full dataset for the production cube. If you don't, the results will not be representative of real-life operations. This is especially true for cubes that are larger than the memory on the machine they will eventually run on.

Of course, you should still make sure that you have plenty of queries in the test scenarios that represent typical user behaviors – running on a workload that only showcases the slowest-performing parts of the cube will not represent a real production environment (unless of course, the entire cube is poorly designed).

As you run the tests, you will discover that certain queries are more disruptive than others. One goal of testing is to discover what such queries look like, so that you can either scale the system to deal with them or provide guidance for users so that they can avoid exercising the cube in this way if possible.

Part of your test scenarios should also aim to observe the cubes behavior as user concurrency grows. You should work with BI developers and business users to understand what the worst-case scenario for user concurrency is. Testing at that concurrency will shake out poorly scalable designs and help you configure the cube and hardware for best performance and stability.

4.3 Load Generation

It is hard to create a load that actually looks like the expected production load – it requires significant experience and communication with end users to come up with a fully representative set of queries. But after you have a set of queries that match user behavior, you can feed them into a test harness. Analysis Services does not ship with a test harness out of the box, but there are several solutions available that help you get started:

ascmd – You can use this command-line tool to run a set of queries against Analysis Services. It ships with the Analysis Services samples and is maintained on CodePlex.

Visual Studio – You can configure Microsoft Visual Studio to generate load against Analysis Services, and you can also use Visual Studio to visually analyze that load.

Third-party tools – You can use tools such as HP LoadRunner to generate high-concurrency load. Note that Analysis Services also supports an HTTP-based interface, which means it may be possible to use web stress tools to generate load.

Roll your own: We have seen customers write their own test harnesses using .NET and the ADOMD.NET interface to Analysis Services. Using the .NET threading libraries, it is possible to generate a lot of user load from a single load client.

No matter which load tool you use, you should make sure you collect the runtime of all queries and the Performance Monitor counters for all runs. This data enables you to measure the effect of any changes you make during your test runs. When you generate user workload there are also some other factors to consider.

First of all, you should test both a sequential run and a parallel run of queries. The sequential run gives you the best possible run time of the query while no other users are on the system. The parallel run enables you to shake out issues with the cube that are the result of many users running concurrently.

Second, you should make sure the test scenarios contain a sufficient number of queries so that you will be able to run the test scenario for some time. To stress the server properly, and to avoid querying the same hotspot values over and over again, queries should touch a variety of data in the cube. If all your queries touch a small set of dimension values, it will hardly represent a real production run. One way to spread queries over a wider set of cells in the cube is to use query templates. Each template can be used to generate a set of queries that are all variants of the same general user behavior.

Third, your test harness should be able to create reproducible tests. If you are using code that generates many queries from a small set of templates, – make sure that it generates the same queries on every test run. If not, you introduce an element of randomness in the test that makes it hard to compare different runs.

References:

48

- Ascmd.exe on MSDN - http://msdn.microsoft.com/en-us/library/ms365187%28v=sql.100%29.aspx
- Analysis Services Community samples - http://sqlsrvanalysissrvcs.codeplex.com/
    - Describes how to use **ascmd** for load generation
    - Contains Visual Studio sample code that achieves a similar effect
- HP LoadRunner - https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17^8_4000_100

4.4 Clearing Caches

To make test runs reproducible, it is important that each run start with the server in the same state as the previous run. To do this, you must clear out any caches created by earlier runs.

There are three caches in Analysis Services that you should be aware of:

- The formula engine cache
- The storage engine cache
- The file system cache

Clearing formula engine and storage engine caches: The first two caches can be cleared with the XMLA **ClearCache** command. This command can be executed using the **ascmd** command-line utility:

```
<ClearCache
      xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
  <Object>
    <DatabaseID><database name></DatabaseID>
  </Object>
</ClearCache>
```

Clearing file system caches: The file system cache is a bit harder to get rid of because it resides inside Windows itself.

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.

Alternatively, you can use the utility **RAMMap** from sysinternals. This utility 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.

There is currently a CodePlex project called ASStoredProcedures 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 users connected to the cube. Also note that neither **RAMMap** or ASStoredProcedures is supported by Microsoft.

4.5 Actions During Testing for Operations Management

While your test team is creating and running the test harness, your operations team can also take steps to prepare for deployment.

Test your data collection setup: Test runs give you a unique chance to try out your data collection procedures before you go into production. The data collection you perform can also be used to drive early feedback to the development team.

Understand server utilization: While you test, you can get an early insight into server utilization. You will be able to measure the memory usage of the cube and the way the number of users maps to I/O load and CPU utilization. If the cube is larger than memory, you can also measure the effect of concurrency and leaf level scan on the I/O subsystem. Remember to measure the worst-case examples described earlier to understand what the impact on the system is.

Early thread tuning: During testing, you can discover threading bottlenecks, as described in this guide. This enables you to go into production with pretuned settings that improve user experience, scalability, and hardware utilization of the solution.

4.6 Actions After Testing

When testing is complete, you have reports that describe the run time of each query, and you also have a greater understanding of the server utilization. This is a good time to review the design of the cube with the BI developers using the numbers you collected during testing. Often, some easy wins can be harvested at this point.

When you put a cube into production, it is important to understand the long-term effects of users building spreadsheets and reports referencing it. Consider the dependencies that are generated as the cube is successfully deployed in ad-hoc data structures across the organization. The data model created and exposed by the cube will be linked into spreadsheets and reports, and it becomes hard to make data model changes without disturbing users. From an operational perspective, preproduction testing is typically your last chance to request cheap data model changes from your BI developers before business users inevitably lock themselves into the data structures that unlock their data. Notice that this is different from typical, static reporting and development cycles. With static reports, the BI developers are in control of the dependencies ;if you give users Excel or other ad-hoc access to cubes, that control is lost. Explorative data power comes at a price.

5 Security and Auditing

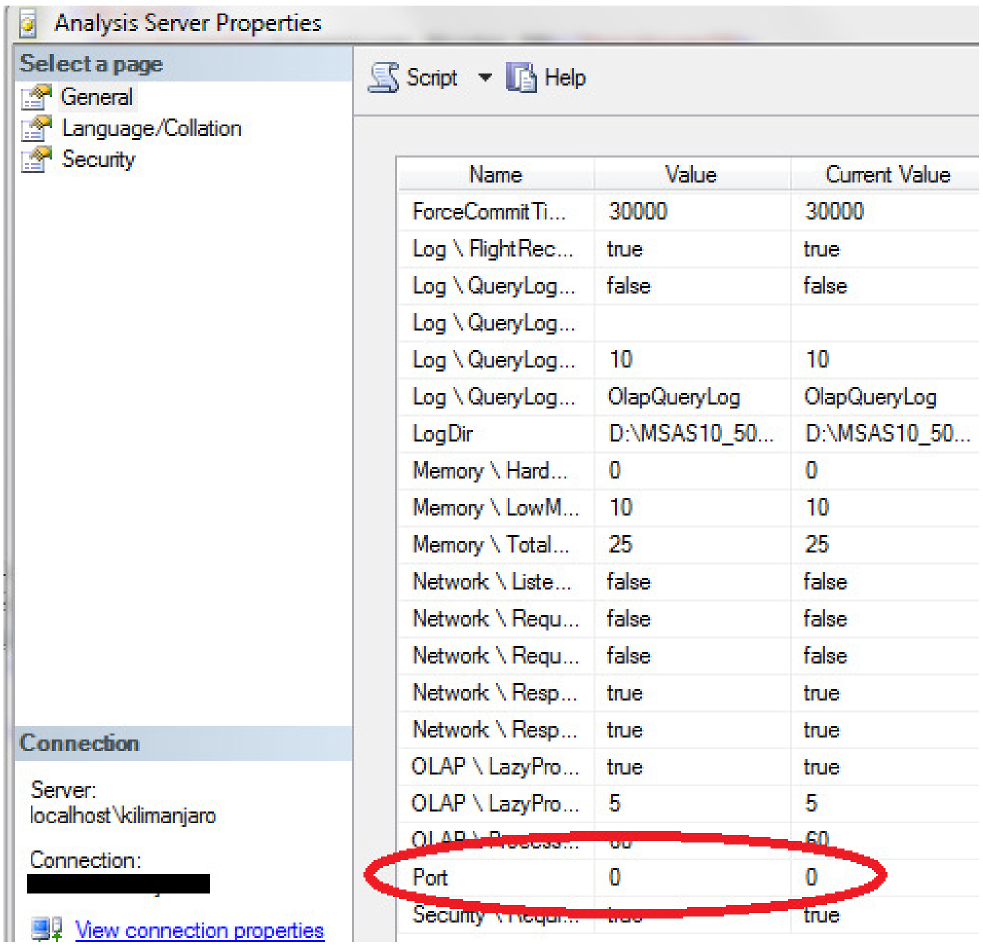
Cubes often contain the very heart of your business data – the source of your decision making. That means that you have to consider potential attack vectors that a malicious user or intruder could use to acquire access to your data. The following table provides an overview of the potential vulnerabilities and the countermeasures you can take. Note that most environments will not need all of these countermeasures – it depends on the data security and on the attack vectors that are possible on your network.

Attack vector	Countermeasure
Other services listening on the server	Firewall all ports other than those used by Analysis Services
Sniff TCP/IP packets to client **Sniff TCP/IP packets during processing**	Configure IPsec or SSL encryption Configure SQL Server Protocol Encryption
Steal physical media containing cubes	Encrypt file system used to store cubes
Compromise the service account	Configure minimum privileges to the service account Require a strong password
Access the file system as a logged-in user	Secure file system with minimal privileges

A full treatment of security configuration is outside the scope of this guide, but the following sections provide references that serve as a starting point for further reading and give you an overview of the options available to you.

5.1 Firewalling the Server

By default, Analysis Services communicates with clients on port 2383. If you want to change this, access the properties of the server.



Server:
localhost\kilimanjaro

Connection:
[redacted]

Name	Value	Current Value
ForceCommitTi...	30000	30000
Log \ FlightRec...	true	true
Log \ QueryLog...	false	false
Log \ QueryLog...		
Log \ QueryLog...	10	10
Log \ QueryLog...	OlapQueryLog	OlapQueryLog
LogDir	D:\MSAS10_50...	D:\MSAS10_50...
Memory \ Hard...	0	0
Memory \ LowM...	10	10
Memory \ Total...	25	25
Network \ Liste...	false	false
Network \ Requ...	false	false
Network \ Requ...	false	false
Network \ Resp...	true	true
Network \ Resp...	true	true
OLAP \ LazyPro...	true	true
OLAP \ LazyPro...	5	5
OLAP \ Process...	60	60
Port	0	0
Security \ Requ...	true	true

Figure 16 - Changing the port used for Analysis Services

Bear in mind that you will need to open the port assigned here in your firewall for TCP/IP traffic. If you leave the default value of 0, Analysis Services will use port 2382.

If you are using named instances, your client application may also need to access the SQL Server browser service. The browser service allows clients to resolve instance names to port numbers, and it listens on TCP port 2382. Note that it is possible to configure the connection string for Analysis Services in such a way that you will not need the browser service port open. To connect directly to the port that Analysis Services is listening on, use this format **[Server name}:[Port].** For example, to connect to an instance listening on port 2384 on server MyServer, use **MyServer:2384**.

Analysis Services can be set up to use HTTP to communicate with clients. If you choose this option, follow the guidelines for configuring Microsoft Internet Information Services. In this case, you will typically need to open either port 80 or port 443.

References:

- How to: Configure Windows Firewall for Analysis Services Access - http://msdn.microsoft.com/en-us/library/ms174937.aspx
- Resolving Common Connectivity Issues in SQL Server 2005 Analysis Services Connectivity Scenarios - http://msdn.microsoft.com/en-us/library/cc917670.aspx
    - Also applies to SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services
- Configuring HTTP Access to SQL Server 2005 Analysis Services on Microsoft Windows 2003 - http://technet.microsoft.com/en-us/library/cc917711.aspx
    - Also applies to SQL Server 2008 Analysis Services, SQL Server 2008 R2 Analysis Services, Windows Server 2008, and Windows Server 2008 R2
- Analysis Services 2005 protocol - XMLA over TCP/IP - http://www.mosha.com/msolap/articles/as2005_protocol.htm

5.2 Encrypting Transmissions on the Network

Analysis Services communicates in a compressed and encrypted format on the network. You may still want to use IPsec to restrict which machines can listen in on the network traffic, but the communication channel itself is already encrypted by Analysis Services.

If you have configured Analysis Services to communicate over HTTP, the communication can be secured using the SSL protocol. However, be aware that you may have to acquire a certificate to use SSL encryption over public networks. Also, note that SSL encryption normally uses port 443, not port 80, to communicate. This difference may require changes in the firewall configuration. Using the HTTP protocol also allows you to run secured lines to parties outside the corporate network – for example, in an extranet setup.

Depending on your network configuration, you may also be concerned about network packets getting sniffed during cube processing. To avoid this, you again have to encrypt traffic. Again, you can use IPsec to achieve this. Another option is to use protocol encryption in SQL Server, which is described in the references.

References:

- How to configure SQL Server 2008 Analysis Services and SQL Server 2005 Analysis Services to use Kerberos authentication - http://support.microsoft.com/kb/917409
- Windows Firewall with Advanced Security: Step-by-Step Guide: Deploying Windows Firewall and IPsec Policies - http://www.microsoft.com/downloads/en/details.aspx?FamilyID=0b937897-ce39-498e-bb37-751c00f197d9&displaylang=en
- How To Configure IPsec Tunneling in Windows Server 2003 - http://support.microsoft.com/kb/816514
- How to enable SSL encryption for an instance of SQL Server by using Microsoft Management Console - http://support.microsoft.com/kb/316898

5.3 Encrypting Cube Files

Some security standards require you to secure the media that the data is stored on, to prevent intruders from physically stealing the data. If you have MOLAP cubes on the server, media security may be a concern to you. Because Analysis Services, unlike the relational engine, does not ship with native encryption of MOLAP cube data, you must rely on encryption outside the engine. You can use Windows File System encryption (Windows Server 2003) or BitLocker (on Windows Server 2008 and Windows Server 2008 R2) to encrypt the drive used to store cubes. Be careful, though; encrypting MOLAP data can have a big impact on performance, especially during processing and schema modification of the cube. We have seen performance drops of up to 50 percent when processing dimensions on encrypted drives – though less so for fact processing. Weigh the requirement to encrypt data carefully against the desired performance characteristics of the cube.

References:

- Encrypting File System in Windows XP and Windows Server 2003 - http://technet.microsoft.com/en-us/library/bb457065.aspx
- BitLocker Drive Encryption - http://technet.microsoft.com/en-us/library/cc731549%28WS.10%29.aspx

5.4 Securing the Service Account and Segregation of Duties

To secure the service account for Analysis Services, it is useful to first understand the different security roles that exist at a server level.

The service account is the account that runs the msmdsrv.exe file. It is configured during installation and can be changed using SQL Server Configuration Manager. The service account is the account used to access all files required by Analysis Services, including MOLAP stores. For your convenience, a local group **SQLServerMSASUser$<Instance Name>** is created that has the right privileges on the binaries required to run Analysis Services. If you configure the service account during installation, the account will automatically be added to this group. If you later choose to change the service account in SQL Server Configuration Manager, you must manually update the group membership.

The server administrator role has privileges to change server settings, and to create, back up, restore, and delete databases. Members of this account can also control security on all databases in the instance. It is the DBA role of the instance and almost equivalent to the **sysadmin** role for SQL Server. Membership in the server administrator role is configured in the properties of the server in SQL Server Management Studio.

In a secure environment, you should run Analysis Services under a dedicated service account. You can configure this during installation of the service.

If your environment requires segregation of duties between those who configure the service account and those who administer the server, you need to make some changes to the msmdsrv.ini file:

- By default, local administrators are members of the server administrator role. To remove this association, set **<BuiltinAdminsAreServerAdmins>** to 0.
- By default, the service account for Analysis Services is a member of the server administrator role. Remove this association by setting **<ServiceAccountIsServerAdmin>** to 0.

However, keep in mind that a local administrator could still access the msmdsrv.ini file and alter the changes you have made, so you should audit for this possibility.

5.5 File System Permissions

As mentioned earlier, the Analysis Services installer will create the Windows NT group **SQLServerMSASUser$<Instance Name>** and add the service account to this group. At install time, this group is also granted access to the **Data**, backup, log, and **TempDir** folders.

If you at a later time add more folders to the instance to hold data, backups, or log files, you will need to grant the **SQLServerMSASUser$<Instance Name>** group read and write access to the new folders. If you move the TempDir folder, you will also need to assign the group read/write permission to the new location. No other users need permissions on these folders for Analysis Services to operate.

In the configuration of the server you will also find the **AllowedBrowsingFolders** setting. This setting, a pipe-separated list of directories, limits the visible folders that server administrators can see when they configure storage locations Analysis Services data. **AllowedBrowsingFolders** is *not* a security feature, because server administrator can change the values in it to reflect anything that the service account can see. However, it does serve as a convenient filter to display only a subset of the folders that the service account can access. Note that server administrators cannot directly access the data visible through the **AllowedBrowsingFolders** setting, but they can write backups in the folders, restore form the folders, move **TempDir** there, and set the storage location of databases, dimensions, and partitions to those folders.

6 High Availability and Disaster Recovery

High availability and disaster recovery are not robustly integrated as part of Analysis Services for large-scale cubes. In this section, you learn about the combination of methods you can use to achieve these goals within an enterprise environment.

To ensure disaster recovery, use the built-in Backup/Restore method, Analysis Services Synch, or Robocopy to ensure that you have multiple copies of the database. You can achieve high availability by deploying your Analysis Services environment onto clustered servers. Also note that by using a combination of multiple copies, clustering, and scale out (section 7.4), you can achieve both high availability and disaster recovery for your Analysis Services environment. In a large-scale environment, the scale-out method generally provides the best of use hardware resources while also providing both backup and high availability.

6.1 Backup/Restore

Cubes are data structures on disk and as such, they contain information that you may want to back up.

Analysis Services backup – Analysis Services has a built-in backup functionality that generates a single backup file from a cube database. Analysis Services backup speeds have been significantly improved in SQL Server 2008 and for most solutions, you can simply use this built-in backup and restore functionality. As with all backup solutions, you should of course test the restore speed.

SAN based backup – If you use a SAN, you can often make backups of a LUN using the storage system itself. This backup process is transparent to Analysis Services and typically operates on the LUN level. You should coordinate with your SAN administrator to make sure the correct LUNs are backed up, including all relevant data folders used to store the cube. You should also make sure that the SAN backup utility uses a VDI/VSS compliant tool to call to Windows before the backup is taken. If you do not use a VDI/VSS tool, you risk getting chkdsk errors when the LUN is restored.

File based backup copy – In SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services, you can attach database if you have a copy of the data folder the database resides in. This means that a detached copy of files in a stale cube can serve as a backup of the database. This option is available only with SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services. You can use the same tools that you use for scale-out cubes (for example, Robocopy or NiceCopy). Restoring in this case means copying the files back to the server and attaching the database.

Don't back up – While this may sound like a silly option, it does have merit in some disaster recovery scenarios. Cubes are built on relational data sources. If these sources are guaranteed to be available and securely backed up, it may be faster to simply reprocess the cube than to restore it from backup. If you use this option, make sure that no data resides in the cubes that cannot be re-created from the relational source (for example, data loaded via the SQL Server Integration Services Analysis Services destination). Of course, you should also make sure that the cube structure itself is available, including all aggregation and partition designs that have been changed on the production server from the standard deployment script.

This is particularly true for ROLAP cubes. In this case (as in all the previous scenarios), you should *always* maintain an updated backup of your Analysis Services project that allows for a redeployment of the solution (with the subsequent required processing), in case your backup media suffers any kind of physical corruption.

6.1.1 Synchronization as Backup

If you are backing up small or medium-size databases, the Analysis Services synchronization feature is an operationally easy method. It synchronizes databases from source to target Analysis Service servers. The process scans the differences between the two databases and transfers only the files that have been modified. There is overhead associated with scanning and verifying the metadata between the two different databases, which can increase the time it takes to synchronize. This overhead becomes increasingly apparent in relation to the size and number of partitions of the databases.

Here are some important factors to consider when you work with synchronization:

- At the end of the synchronization process, a Write lock must be applied to the target server. If this lock is queued up behind a long running query, it can prevent users from querying the target database.
- During synchronization, a read commit lock is applied to the source server, which prevents users from processing new data but allows multiple synchronizations to occur from the same source server.
- For enterprise-size databases, the synchronization method can be expensive and low-performing. Because some operations are single-threaded (such as the delete operation), having high-performance servers and disk subsystems may not result in faster synchronization times. Because of this, we recommended that for enterprise size databases you use alternate methods, such as Robocopy (discussed later in this guide) or hardware-based solutions (for example, SAN clones and snapshots).
- When executing multiple synchronization operations against a single server, you may get the best performance (by minimizing lock contentions) by queuing up the synchronization requests and then running them serially.
- For the same locking contention reasons, plan your synchronizations for down times, when querying and processing are not occurring on the affected servers. While there are locks in place to prevent overwrites, processes such as long-running queries and processing may prevent the synchronization from completing in a timely fashion.

For more information, see the “Analysis Services Synch Method” section in Analysis Services Synchronization Best Practices technical note (http://sqlcat.com/technicalnotes/archive/2008/03/16/analysis-services-synchronization-best-practices.aspx).

6.1.2 Robocopy

The basic principle behind the Robocopy method is to use a fast copy utility, such as Robocopy, to copy the OLAP data folder from one server to another. By following the sample script noted in the technical note, Sample Robocopy Script to customer synchronize Analysis Services databases (http://sqlcat.com/technicalnotes/archive/2008/01/17/sample-robocopy-script-to-customer-synchronize-analysis-services-databases.aspx), you can perform delta copies (that is, copy only the data that has changed) of the OLAP data folder from one source to another target source in parallel. This method is often employed in enterprise environments because the key factor here is fast, robust data file transfer.

However, the key disadvantages of using this approach include:

- You must stop and then restart (in SQL Server 2005 Analysis Services) or detach (in SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services) your Analysis Services servers when you use a fast copy utility.
- You cannot use the synchronization feature and Robocopy together.
- This approach makes the assumption that there is only one database on that instance; this is usually okay because of its size.

- Some functionality is lost if you use this method, including, but not limited to, writeback, ROLAP, and real-time updates.

Nevertheless, this is an especially effective method for query server / processing server architectures that involve only one database per server.

References:

- Scale-Out Querying with Analysis Services (http://sqlcat.com/whitepapers/archive/2007/12/16/scale-out-querying-with-analysis-services.aspx)

6.2 Clustered Servers

Analysis Services can participate in a Windows failover cluster using a shared disk subsystem, such as a SAN. When provisioning storage for a cluster, there are some things you should be aware of.

It is not possible to use dynamic disk stripes. In this case, you have two options for spreading the cubes over all available LUN:

- Have your SAN administrator configure a mega-LUN.
- Selectively place partitions on different LUNs and then manually balance the load between these LUNs.

Mega LUN: Most SANs today can stripe multiple, smaller LUNs into a large mega-LUN. Talk to your SAN administrator about this option.

Selective placement: If you need to use multiple LUNs for a single cube, you should try to manually balance I/O traffic between these LUNs. One way to achieve this is to partition the cube into roughly equal-sized slices based on a dimension key. Very often, the only way to achieve roughly equal balance like this is to implement a two-layer partitioning on both date and the secondary, balancing key. For more information, see the Repartitioning section in this guide.

7 Diagnosing and Optimizing

This section discusses how to troubleshoot problems and implement changes that can be made transparently in the cube structures to improve performance. Many of these changes are already documented in the Analysis Services Performance Guide. But some additional considerations apply from an operations perspective.

7.1 Tuning Processing Data

During **ProcessData** operations, Analysis Services uses the processing thread pool for worker threads used by the storage engine.

ProcessData operations use three parallel threads to perform the following tasks:

- Query the data source and extract data
- Look up dimension keys in dimension stores and populate the processing buffer
- Write the processing buffer to disk when it is full

You can usually increase throughput of **ProcessData** by tuning the network stack and processing more partitions in parallel. However, you may still benefit from tuning the process pool.

To optimize these settings for the **ProcessData** phase, check your Performance Monitor counter on the object **MSOLAP: Threads** and use the following table for guidance.

Situation	Action
Processing pool job queue length > 0 and **Processing pool idle threads** = 0 for longer periods during processing.	Increase **ThreadPool\Process\MaxThreads** and then retest.
Both **Processing pool job queue length** > 0 and **Processing pool idle threads** > 0 at same time during processing.	Decrease **CoordinatorExecutionMode** (for example, change it from -4 to -8) and then retest.

You can use the **Processor –% Processor Time – Total** counter as a rough indicator of how much you should change these settings. For example, if your CPU load is 50 percent, you can double **ThreadPool\Process\MaxThreads** to see whether this also doubles your CPU usage. It is possible to get to 100 percent CPU load in a system without bottlenecks, though you may want to leave some headroom for growth. Keep in mind that increased parallelism of processing also has an effect on queries running at the system. Ideally, use a separate processing server or a dedicated processing time window where no one is querying Analysis Services. If this is not an option, as you dedicate more CPU power and threads to processing, less CPU will be used for query responses. Because processing consumes threads from the same pool as query subcube requests, you should also be careful that you don't run the process thread pool dry if you process and query at the same time. If you are processing the cubes during a set processing window with no users on the box, this will of course not be an issue.

References:

- SQL Server 2005 Analysis Services (SSAS) Server Properties (http://technet.microsoft.com/en-us/library/cc966526.aspx)

7.1.1 Optimizing the Relational Engine

In addition to looking at Analysis Services configurations, and settings, you can also look at the relational engine when you are planning improvements to your Analysis Services installation. This section focuses on working with relational data from SQL Server 2005, SQL Server 2008, or SQL Server 2008 R2. Although Analysis Services can be used with any OLE DB or .NET driver enabled database (such as Oracle or Teradata), the advice here may not apply to such third-party environments. However, if you are a third-party DBA, you may be able to translate the techniques discussed here to similar ones in your own environment.

7.1.1.1 Relational Indexing for Partition Processing

While you generally want each cube partition to touch at most one relational partition, the reverse is not true. It is perfectly viable to have to have more than one cube partition accessing the same relational partition. As an example, a relational source that is partitioned by year with a cube that is partitioned by month can still provide optimal processing performance.

If you do not have a one-to-one relationship between relational and cube partitions, you generally want an index to support the fact processing query. The best choice of index for this purpose is a clustered index; if your load strategy allows you to maintain such an index, this is what you should aim for.

When a partition processing query is supported by an index the plan should look like this.

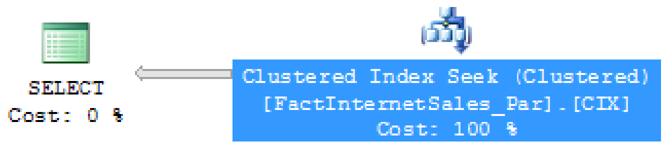

SELECT
Cost: 0 %
Clustered Index Seek (Clustered)
[FactInternetSales_Par].[CIX]
Cost: 100 %

Figure 24 Supporting Measure Group processing with an index

References:

- Top 10 SQL Server 2005 Performance Issues for Data Warehouse and Reporting Applications (http://sqlcat.com/top10lists/archive/2007/11/21/top-10-sql-server-2005-performance-issues-for-data-warehouse-and-reporting-applications.aspx)
- Ben-Gan, Itzik and Lubor Kollar, *Inside Microsoft SQL Server 2005: T-SQL Querying.* Redmond, Washington: Microsoft Press, 2006.

7.1.1.2 Relational Indexing for Dimension Processing

If you follow a dimensional star schema design (which we recommend for large cubes), most dimension processing queries should run relatively fast and take only a tiny portion of the total cube processing time. But if you have very large dimensions with millions of rows or dimensions with lots of attributes, some performance can be still be gained by indexing the relational dimension table. To understand the best indexing strategy, it is useful to know which form dimensions processing queries take. The number of queries generated by Analysis Services depends on the attribute relationships defined in the dimension. For each attribute in the dimension, the following query is generated during processing.

```
SELECT DISTINCT <attribute>,  [<related attribute>  […n] ]
FROM <dimension table>
```

Consider the following example dimension, with **CustomerID** as the key attribute.

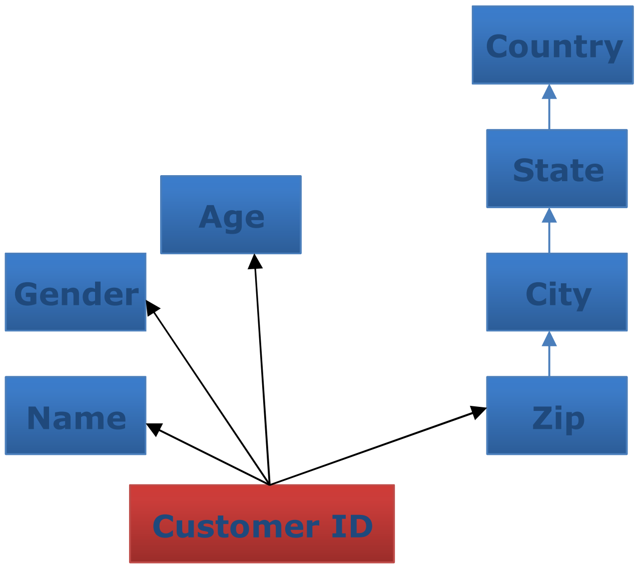


Figure 17 – Example Customer Dimension – Attribute relationships

The following queries are generated during dimension processing.

```
SELECT DISTINCT Country FROM Dim.Customer
SELECT DISTINCT State, Country FROM Dim.Customer
SELECT DISTINCT City, State FROM Dim.Customer
SELECT DISTINCT Zip, City FROM Dim.Customer
SELECT DISTINCT Name FROM Dim.Customer
SELECT DISTINCT Gender FROM Dim.Customer
SELECT DISTINCT Age FROM Dim.Customer
SELECT DISTINCT CustomerID, Name, Gender, Age, Zip FROM Dim.Customer
```

The indexing strategy you apply depends on which attribute you are trying to optimize for. To illustrate a tuning process, this section walks through some typical cases.

Key attribute: the key attribute in a dimension, in this example **CustomerID,** can be optimized by creating a unique, clustered index on the key. Typically, using such a clustered index is also the best strategy for relational user access to the table – so your DBA will be happy if you do this. In this example, the following index helps with key processing.

```
CREATE UNIQUE CLUSTERED INDEX CIX_CustomerID ON Dim.Customer (CustomerID)
```

This will create the following, optimal query plan.

61

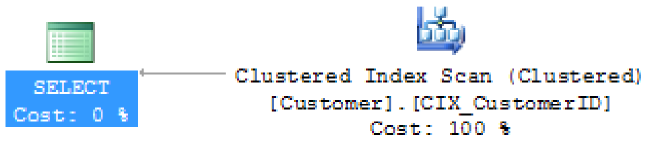


Figure 18 - A good key processing plan

High cardinality attributes: For high cardinality attributes, like **Name**, you need a nonclustered index. Notice the DISTINCT in the SELECT query generated by Analysis Services.

```
SELECT DISTINCT Name FROM Dim.Customer
```

DISTINCT generally forces the relational engine to perform a sort to remove duplicates in the returned dataset. The sort operation results in a plan that looks like this.

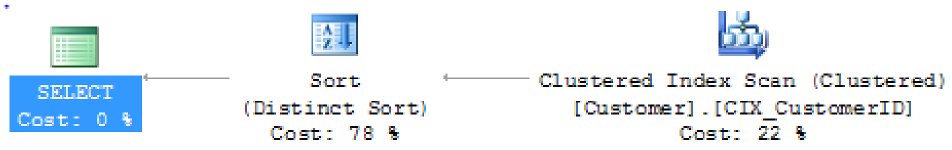


Figure 19 - Expensive sort plan during dimension processing

If this plan takes a long time to run, which could be the case for large dimension, consider creating an index that helps remove the sort. In this example, you can create this index.

```
CREATE INDEX IX_Customer_Name ON Dim.Customer (Name)
```

This index generates the following, much better plan.

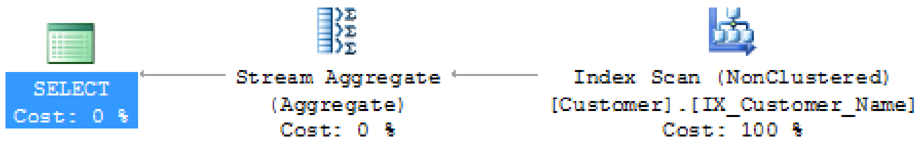


Figure 20 - A fast high cardinality attribute processing plan

Low cardinality attributes: For attributes that are part of a large dimension but low granularity, even the preceding index optimization may result in an expensive plan. Consider the **City** attribute in the customer dimension example. There are very few cities compared to the total number of customers in the dimension. For example, you want to optimize for the following query.

```
SELECT DISTINCT City, State FROM Dim.Customer
```

Creating a multi-column index on **City** and **State** removes the sort operation required to return DISTINCT rows – resulting in a plan very similar to the optimization performed earlier with the **Name** attribute. This is better than running the query with no indexed access. But it still results in touching one row per customer – which is far from optimal considering that there are very few cities in the table.

If you have SQL Server Enterprise, you can optimize the SELECT query even further by creating an indexed view like this.

```
CREATE VIEW Dim.Customer_CityState
WITH SCHEMABINDING
AS
SELECT City, State, COUNT_BIG(*) AS NumRows FROM Dim.Customer
GROUP BY City, State

GO

CREATE UNIQUE CLUSTERED INDEX CIX_CityState
  ON Dim.Customer_CityState (City, State)
```

SQL Server maintains this aggregate view on disk, and it stores only the distinct values of **City** and **State** – exactly the data structure you are looking for. Running the dimension processing query now results in the following optimal plan.

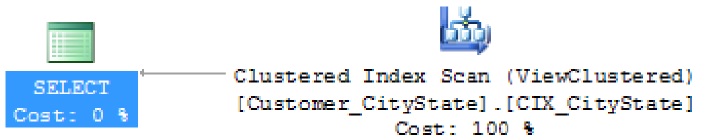

SELECT
Cost: 0 %
Clustered Index Scan (ViewClustered)
[Customer_CityState].[CIX_CityState]
Cost: 100 %

Figure 21 - Using indexed views to optimize for low-cardinality attributes

7.1.1.3 Overoptimizing and Wasting Time

It is possible to tune the relational engine to cut down time on processing significantly, especially for partition processing and large dimensions. However, bear in mind that every time you add an index to a table, you add one more data structure that must be maintained when users modify rows in that table. Relational indexing, like aggregation design in a cube and much of BI and data warehousing, is tradeoff between data modification speed and user query performance. There is typically a sweet spot in this space that will depend on your workload. Different people have different skills, and the perception of where that sweet spot lies will change with experience. As you get closer to the optimal solution, increased the tuning effort will often reach a point of diminishing returns where the speed of the system moves asymptotically towards the optimum balance. Monitor your own tuning efforts and try to understand when you are getting close to that flatline behavior. As tempting as full tuning exercises can be to the technically savvy, not every system needs benchmark performance.

7.1.1.4 Using Index FILLFACTOR = 100 and Data Compression

If page splitting occurs in an index, the pages of the index may end up less than 100 percent full. The effect is that SQL Server will be reading more database pages than necessary when scanning the index.

You can check for index pages are not full by querying the SQL Server DMV **sys.dm_db_index_physical_stats**. If the column **avg_page_space_used_in_percent** is significantly lower than 100 percent, a FILLFACTOR 100 rebuild of the index may be in order. It is not always possible to rebuild the index like this, but this trick has the ability to reduce I/O. For stale data, rebuilding the indexes on the table is often a good idea before you mark the data as read-only.

In SQL Server 2008 you can use either row or page compression to further reduce the amount of I/O required by the relational database to serve the fact process query. Compression has a CPU overhead, but reduction in I/O operations is often worth it.

References:

- Data Compression: Strategy, Capacity Planning and Best Practices - http://msdn.microsoft.com/en-us/library/dd894051%28v=sql.100%29.aspx

7.1.1.5 Eliminating Database Locking Overhead

When SQL Server scans an index or table, page locks are acquired while the rows are being read. This ensures that many users can access the table concurrently. However, for data warehouse workloads, this page level locking is not always the optimal strategy – especially when large data retrieval queries like fact processing access the data.

By measuring the Perfmon counter **MSSQL:Locks – Lock Requests / Sec** and looking for **LCK** events in **sys.dm_os_wait_stats**, you can see how much locking overhead you are incurring during processing.

To eliminate this locking overhead, you have three options:

- Option 1: Set the relational database in Read Only mode before processing.
- Option 2: Build the fact indexes with `ALLOW_PAGE_LOCKS = OFF` and `ALLOW_ROW_LOCKS = OFF`.
- Option 3: Process through a view, specifying the `WITH (NOLOCK)` or `WITH (TABLOCK)` query hint.

Option 1 may not always fit your scenario, because setting the database to read-only mode requires exclusive access to the database. However, it is a quick and easy way to completely remove any lock waits you may have.

Option 2 is often a good strategy for data warehouses. Because SQL Server Read locks (S-locks) are compatible with other S-locks, two readers can access the same table twice, without requiring the fine granularity of page and row locking. If insert operations are only done during batch time, relying solely on table locks may be a viable option. To disable row and page locking on a table and index, rebuild ALL by using a statement like this one.

```
ALTER INDEX ALL ON FactInternetSales REBUILD
WITH (ALLOW_PAGE_LOCKS = OFF, ALLOW_ROW_LOCKS = OFF)
```

Option 3 is a very useful technique. Processing through a view provides you with an extra layer of abstraction on top of the database –a good design strategy. In the view definition you can add a NOLOCK or TABLOCK hint to remove database locking overhead during processing. This has the advantage of making your locking elimination independent of how indexes are built and managed.

```
CREATE VIEW vFactInternetSales
AS
SELECT [ProductKey], [OrderDateKey], [DueDateKey]
      ,[ShipDateKey], [CustomerKey], [PromotionKey]
      ,[CurrencyKey], [SalesTerritoryKey], [SalesOrderNumber]
      ,[SalesOrderLineNumber], [RevisionNumber], [OrderQuantity]
      ,[UnitPrice], [ExtendedAmount], [UnitPriceDiscountPct]
      ,[DiscountAmount], [ProductStandardCost], [TotalProductCost]
      ,[SalesAmount], [TaxAmt], [Freight]
      ,[CarrierTrackingNumber] ,[CustomerPONumber]
FROM [dbo].[FactInternetSales] WITH (NOLOCK)
```

If you use the **NOLOCK** hint, beware of the dirty reads that can occur. For more information about locking behaviors, see SET TRANSACTION ISOLATION LEVEL (http://technet.microsoft.com/en-us/library/ms173763.aspx) in SQL Server Books Online.

7.1.1.6 Forcing Degree of Parallelism in SQL Server

During **ProcessData**, a partition processing task is limited by the speed achievable from a single network connection the data source. These speeds can vary between a few thousand rows per second for legacy data sources, to around 100,000 rows per second from SQL Server. Perhaps that is not fast enough for you, and you have followed the guidance in this document and the Analysis Services Performance Guide to allow multiple partitions to process in parallel – scaling **ProcessData** nearly linearly.

We have seen customer reach 6.5 million rows per second into a cube by processing 64 partitions concurrently. Bear in mind what it takes to transport millions of rows out of a relational database every second. Each processing query will be selecting data from a big table, aggressively fetching data as fast as the network stack and I/O subsystem can deliver them. When SQL Server receives just a single request for all rows in a large fact table, it will spawn multiple threads inside the database engine to serve that it as fast as possible – utilizing all server resources. But what the DBA of the relational engine may not know, is that in a few milliseconds, your cube design is set up to ask for *another* one of those large tables – concurrently. This presents the relational engine with a problem: how many threads each query should be assigned to optimize the total throughput of the system, without sacrificing overall performance of individual processing commands. It is easy to see that race conditions may create all sorts on interesting situations to further complicate this. If parallelism overloads the Database Engine, SQL Server must resort to context switching between the active tasks, continuously redistributing the scarce CPU resources and wasting CPU time with scheduling. You can measure this happening as a high **SOS_SCHEDULER_YIELD** wait in **sys.dm_os_wait_stats**.

Fortunately, you can defend yourself against excessive parallelism by working together with the cube designer to understand how many partitions are expected to process at the same time. You can then assign a smaller subset of the CPU cores in the relational database for each partition. Carefully designing for this parallelism can have a large impact. We have seen cube process data speeds more than double when the assigned CPU resources are carefully controlled.

You have one or two ways to partition server resources for optimal process data speeds, depending on which version of SQL Server you run.

Instance reconfiguration – Using **sp_configure**, you can limit the number of CPU resources a single query can consume. For example, consider a cube that processes eight partitions in parallel from a computer running SQL Server with 16 cores. To distribute the processing tasks equally across cores, you would configure like this.

```
EXEC sp_configure 'show advanced options', 1
RECONFIGURE
EXEC sp_configure 'max degree of parallelism', 2
RECONFIGURE
```

Unfortunately, this is a brute-force approach, which has the side effect of changing the behavior of the entire instance of SQL Server. It may not be a problem if the instance is dedicated for cube processing, but it is still a crude technique. Unfortunately, this is the only option available to you on SQL Server 2005.

Resource Governor – If you run SQL Server 2008 or SQL Server 2008 R2, you can use Resource Governor to control processing queries. This is the most elegant solution, because it can operate on each data source view individually.

The first step is to create a resource pool and a workload group to control the Analysis Services connections. For example, to the following statement limits each ProcessData task to 2 CPU cores and a maximum memory grant of 10 percent per query.

```
CREATE RESOURCE POOL [cube_process] WITH(
  min_cpu_percent=0
  , max_cpu_percent=100
  , min_memory_percent=0
  , max_memory_percent=100)
GO
```

```
CREATE WORKLOAD GROUP [process_group] WITH(
    group_max_requests=0
  , importance=Medium
  , request_max_cpu_time_sec=0
  , request_max_memory_grant_percent=10
  , request_memory_grant_timeout_sec=0
  , max_dop=2)
USING [cube_process]
GO
```

The next step is to design a classifier function that recognizes the incoming cube process requests. There are several ways to recognize cube connection. One is to use the host name. Another is to use application names (which you can set in the connection string in the data source view in the cube). The following example recognizes all Analysis Services connections that use the default values in the connection string.

```
USE [master]
GO
CREATE FUNCTION fnClassifier()
RETURNS sysname
WITH SCHEMABINDING
AS
BEGIN
  DECLARE @group SYSNAME
  IF APP_NAME() LIKE '%Analysis Services%' BEGIN
    SET @group= 'process_group'
  END
   ELSE BEGIN
    SET @group = 'default'
  END
  RETURN (@group)
END
```

Make sure you test the classifier function before applying it in production. After you are certain that the function works, enable it.

```
ALTER RESOURCE GOVERNOR WITH (CLASSIFIER_FUNCTION = [dbo].[fnClassifier]);
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

7.1.1.7 Loading from Oracle

Analysis Services is commonly deployed in heterogeneous environments, with Oracle being one of the main data sources. Because cubes often need to read a lot of data from the source system during processing, it is important that you use high speed drivers to extract this data. We have found that the native Oracle drivers provide a reasonable performance, especially if many partitions are processed in parallel.

However, we have also found that data can be extracted from a SQL Server data source at 5-10 times the speed of the same Oracle source. The SQL Server driver SQLNLCI is optimized for very high extraction speeds – sometimes reaching up to 80,000-100,000 rows per second from a single TCP/IP connection, depending on the source schema). We have tested processing speeds on top of SQL Server all the way to 6.1 million rows per second.

Consider that SQL Server Integration Services, part of the same SKU as Analysis Services, has a high-speed Oracle driver available for download. This driver is optimized for high-speed extraction from Oracle. We have found that the following architecture often provides faster processing performance than processing directly on top of Oracle.

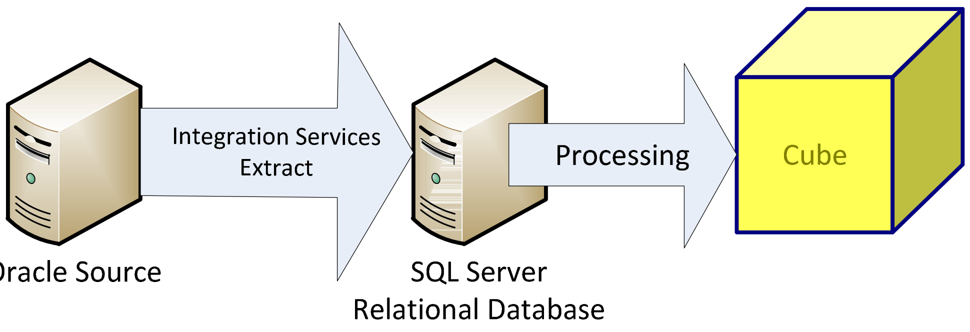


Figure 22 - Fast Processing on Oracle

References:

- Microsoft Connectors Version 1.1 for Oracle and Teradata by Attunity - http://www.microsoft.com/downloads/en/details.aspx?FamilyID=6732934c-2eea-4a7f-85a8-8ba102e6b631
- The Data Loading Performance Guide - http://msdn.microsoft.com/en-us/library/dd425070.aspx
  - Describes how to move lots of data into SQL Server

7.2 Tuning Process Index

As with **ProcessData** workloads, you can often increase speed of **ProcessIndex** by running more partitions in parallel. However, if this option is not available to you, the thread settings can provide an extra benefit. When you measure CPU utilization with the counter **Processor –% Processor Time – Total** and you find utilization is less than 100 percent with no I/O bottlenecks, there is a good chance that you can increase the speed of the **ProcessIndex** phase further by using the techniques in this section.

7.2.1 CoordinatorBuildMaxThreads

When a single partition is scanned, the amount of threads used to scan extents is limited by the **CoordinatorBuildMaxThreads** setting. The setting determines the maximum number of threads allocated per aggregation processing job. It is the absolute number of threads that can be run by an aggregation processing job. Keep in mind that you are still limited by the number of threads in the process thread pool when processing, so increasing this value may require increasing the threads available to the process thread pool too.

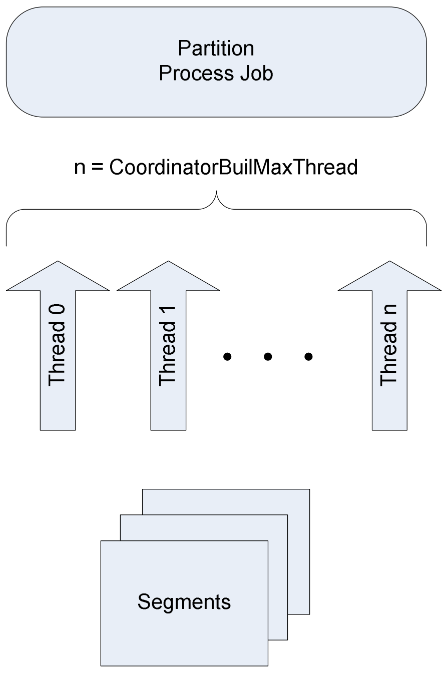


Figure 29 CoordinatorBuildMaxThreads

If you are not able to drive high parallelism by using more partitions, you can change the **CoordinatorBuildMaxThreads** value. Increasing this allows you to use more threads per partition while building aggregations for each partition during the **ProcessIndex** phase

7.2.2 AggregationMemoryMin and Max

As you increase parallelism of **ProcessIndex**, you may run into memory bottlenecks. On a server with more than around ten **Process ndex** jobs running in parallel, you may need to adjust **AggregationMemoryMin** and **AggregationMemoryMax** to get optimal results. For more information, see the Memory section of this guide.

7.3 Tuning Queries

Query optimization is mostly covered in the Analysis Services Performance Guide and will generally involve design or application level changes. However, there are some optimizations that can be made in a production cube that are transparent to users and BI developers. These are described here.

7.3.1 Thread Pool Tuning

Analysis Services uses the parsing thread pools, the query thread pool, and the process thread pool for query workloads on Analysis Services. A listener thread listens for a client request on the TCP/IP port specified in the Analysis Services properties. When a query comes in, the listener thread brokers the request to one of the parsing thread pools. The parsing thread pools either execute the request immediately or send the request off to the query or process thread pool.

Worker threads from the query pool check the data and calculation caches to see whether the request can be served from cache. If the request contains calculations that need to be handled by the formula engine, worker threads in the query pool are use to perform the calculation and store the results. If the query needs to go to disk to retrieve aggregations or scan a partition, worker threads from the processing pool are used to satisfy the request and store the results in cache.

There are settings in the **msmdsrv.ini** file that allow users to tune the behavior of the thread pools involved in query workloads. Guidance on using them is provided in this section.

7.3.1.1 Parsing Thread Pools

The Analysis Services protocol uses Simple Object Analysis Protocol (SOAP) and XMLA for Analysis (XMLA) with TCP/IP or HTTP/HTTPS as the underlying transport mechanism. After a client connects to Analysis Services and establishes a connection, commands are then forwarded from the connection's input buffers to one of the two parsing thread pools where the XMLA parser begins parsing the XMLA while analyzing the SOAP headers to maintain the session state of the command.

There are two parsing thread pools: the short-command parsing pool and the long-command parsing pool. The short-command parsing pool is used for commands that can be executed immediately and the long-command parsing pool is used for commands that require more system resources and generally take longer to complete. Requests longer than one package are dispatched to the long-parsing thread pool, and one-package requests go to the short-parsing pool. If a request is a quick command, like a DISCOVER, the parsing thread is used to execute it. If the request is a larger operation, like an MDX query or an MDSCHEMA_MEMBERS, it is queued up to the query thread pool.

For most Analysis Services configurations you should not modify any of the settings in the short-parsing or long-parsing thread pools. However, with the information provided here, you can imagine a workload where either the short-parsing or long-parsing thread pools run dry. We have only seen one such workload, at very high concurrency, so it is unlikely that you will need to tune the parsing thread pool. If you do have a problem with one the parsing thread pools, it will show up as values consistently higher than zero in **MSOLAP/Thread - Short parsing job queue length** or **MSOLAP/Thread – Long parsing job queue length.**

7.3.1.2 Query Thread Pool Settings

Although modifying the **ThreadPool\Process\MaxThreads** and **ThreadPool\Query\MaxThreads** properties can increase parallelism during querying, you must also take into account the additional impact of **CoordinatorExecutionMode** as described in the configuration section.

In practical terms, the balancing of jobs and threads can be tricky. If you want to increase parallelism, it is important to first narrow down parallelism as the bottleneck. To help you determine whether this is the case, it is helpful to monitor the following performance counters:

- **Threads\Query pool job queue length**—The number of jobs in the queue of the query thread pool. A nonzero value means that the number of query jobs has exceeded the number of available query threads. In this scenario, you may consider increasing the number of query threads. However, if CPU utilization is already very high, increasing the number of threads only adds to context switches and degrades performance.

- **Threads\Query pool busy threads**—The number of busy threads in the query thread pool. Note that this counter is broken in some versions of Analysis Services and does not display the correct value. The value can also be derived from the size of the thread pool minus the **Threads\Query pool idle threads** counter.
- **Threads\Query pool idle threads**—The number of idle threads in the query thread pool.

Any indication of queues in any of these thread pools with a CPU load less than 100 percent indicate a potential option for tuning the thread pool.

References

- Analysis Services Query Performance Top 10 Best Practices (http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/ssasqptb.mspx)

7.3.2 Aggregations

Aggregations behave very much like indexes in a relational database. They are physical data structures that are used to answer frequently asked queries in the database. Adding aggregations to a cube enables you to speed up the overall throughput, at the cost of more disk space and increased processing times.

Note that just as with relational indexing, not every single, potentially helpful aggregate should be created. When the space of potential aggregates grows large, one aggregate being read may push another out of memory and case disk thrashing. Analysis Services may also struggle to find the best aggregate among many matching a given query.

A good rule of thumb is that no measure group should have more than 30 percent of its storage space dedicated to aggregates. To discover which aggregates are most useful, you should collect the **query subcube verbose** event using SQL Server Profiler while running a representative workload. By summing the run time of each unique subcube you can get a good overview of the most valuable aggregations to create. For more information about how to create aggregates, see the Analysis Services Performance Guide.

There are some aggregates that you generally always want to create, namely the ones that are used directly by sets defined in the calculation script. You can identify those by tracing queries going to the cube after a process event. The first user to run a query or connect to the cube after a processing event may trigger **query subcube verbose** events that have SPID = 0 – this is the calculation script itself requesting data to instantiate any named sets defined there.

References:

- Analysis Services 2005 Aggregation Design Strategy - http://sqlcat.com/technicalnotes/archive/2007/09/11/analysis-services-2005-aggregation-design-strategy.aspx
    - Also applies to SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services
- Aggregation Manager on CodePlex - http://bidshelper.codeplex.com/wikipage?title=Aggregation%20Manager&referringTitle=Home
    - Makes the task of designing aggregations easier
- BIDS Helper - http://bidshelper.codeplex
    - Assists with many common tasks, including aggregation design

7.3.3 Optimizing Dimensions

Good dimension design is key to good performance of large cubes. Your developers should design dimensions that fit the needs of the business users. However, there are some minor changes you can often make on a live system that can yield a significant performance benefit. The changes are not 100 percent transparent to business users, but they may well be acceptable or even improve the behavior of the cube. Be sure to coordinate with your BI developers to understand whether you can make any of the changes discussed here in the cube.

Removing the (All) Level – In some dimensions, it does not make sense to ask for the (All) level in a query. The classic example is a date dimension. If the (All) level is present, and no default has been set, a user connecting to the cube may inadvertently ask for the sum of all years. This number is rarely a useful value and just wastes resources at the server.

Setting default members – Whenever a user issues an MDX statement, every hierarchy not explicitly mentioned in the query uses its default member value. By default, this member is the (All) level in the hierarchy. The (All) level may not be the typical usage scenario, which will cause the user to reissue the query with a new value. It is sometimes useful to set another default value in the dimension that more accurately reflects the most common usage scenario. This can be done with a simple modification to the calculation script of the cube. For example, the following command sets a new default member in the Date dimension.

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

Scope the All Level to NULL –Removing the (All) level and setting default members can be confusing to users of Excel. Another option is to force the (All) level to be NULL for the dimensions where querying that level makes no sense. You can use a SCOPE statement for that. For more information, see the performance guide.

AttributeHierarchyEnabled – This property, when set to **false**, makes the property invisible as an attribute hierarchy to users browsing the cube. This reduces the metadata overhead of the cube. Not all attributes can be disabled like this, but you may be able to remove some of them after working with the users.

Note that there are many other optimizations that can be done on dimensions, and the Analysis Services Performance Guide contains more detailed information.

7.3.4 Calculation Script Changes

The calculation script of a cube contains the MDX statements that make up the nonmaterialized part of the cube. Optimizing the calculation script is a very large topic that is outside the scope of this document. What you should know it that such changes can often be made transparently to the user and that the gains, depending on the initial design, can often be substantial.

You should generally collect the performance counters under **MSOLAP:MDX** because these can be used by cube developers to determine whether there are potential gains to be had. A typical indicator of a slow calculation script is a large ratio between the **MSOLAP:MDX/Number of cell-by-cell evaluation nodes** and **MSOLAP:MDX/Number of bulk-mode evaluation nodes.**

7.3.5 Repartitioning

Partitioning of cubes can be used to both increase processing and query speeds. For example, if you struggle with process data speeds, splitting up the nightly batch into multiple partitions can increase concurrency of the processing operation. This technique is documented in the Analysis Services Performance Guide.

Partitions can also be used to selectively move data to different I/O subsystems. An example of this is a customer that keeps the latest data in the cube on NAND devices and moves old and infrequently accessed data to SATA disks. You can move partition around using SQL Server Management Studio in the properties pane of the partition. Alternatively, you can use XML or scripting to move partition data by executing a query like this.

```
<Alter ObjectExpansion="ExpandFull"
   xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
 <Object>
  <DatabaseID>Adventure Works DW</DatabaseID>
  <CubeID>Adventure Works DW</CubeID>
  <MeasureGroupID>Fact Reseller Sales</MeasureGroupID>
  <PartitionID>Reseller_Sales_2001</PartitionID>
 </Object>
 <ObjectDefinition>
  <Partition>
   <ID>Reseller_Sales_2001</ID>
   <Name>Reseller_Sales_2001</Name>
   <StorageLocation>D:\MSAS10_50.KILIMANJARO\OLAP\2001\</StorageLocation>
  </Partition>
 </ObjectDefinition>
</Alter>
```

Be aware that after you have changed the storage location of the partition that marks the partition as unprocessed and empty, you must reprocess the partition to physically move the data. Note that Analysis Services creates a folder under the path you specify, and this folder is named using a GUID – not easy to decode for a human. To keep track of where your moved partitions are, it is therefore an advantage to precreate folders with human-readable names to hold the data.

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. 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.

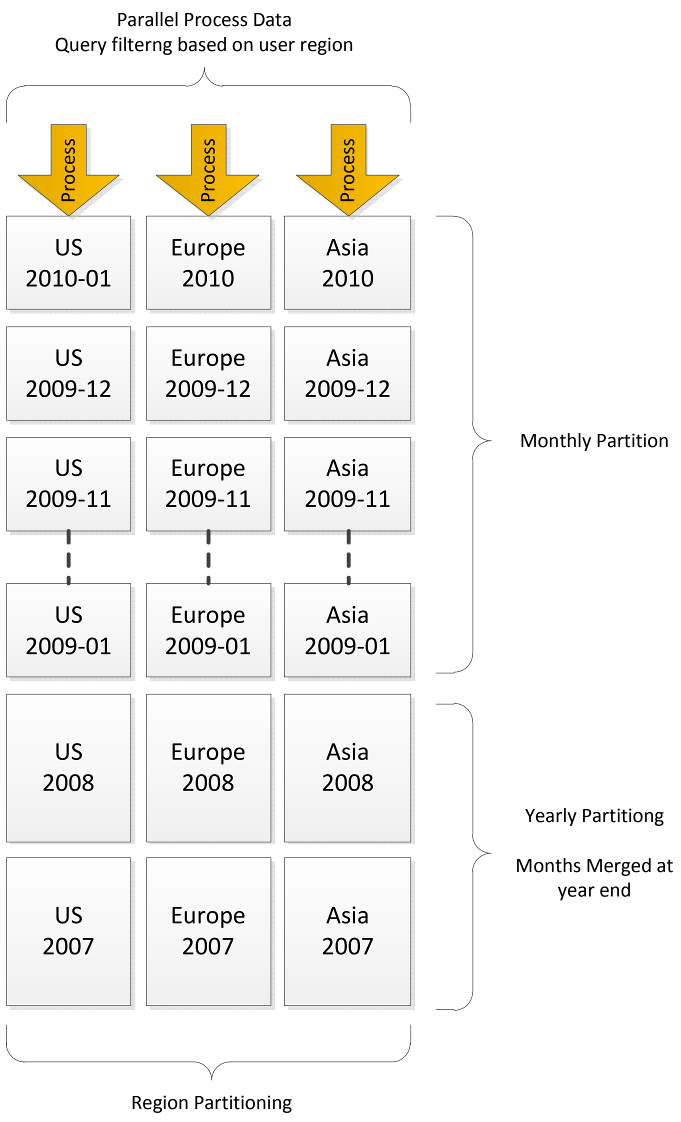


Figure 23 - Example of matrix partitioning

74

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 or 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

Sometimes it is not possible to come up with a good distribution of the keys across the partitions. Perhaps you just don't have a good key candidate that fits the description in the previous paragraph, 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. Users will have to touch all the hash buckets as they query the cube, but at least you can perform parallel loading. If you expect every query to touch many partitions, it is important that you pay special attention to the **CoordinatorQueryBalancingFactor** described earlier.

As you add more partitions, the metadata overhead of managing the cube grows exponentially. As a rule of thumb, you should therefore seek to keep the number of partitions in the cube in the low thousands. This affects **ProcessUpdate** and **ProcessAdd** operations on dimensions, which have to traverse the metadata dependencies to update the cube when dimensions change. For large cubes, prefer larger partitions over creating too many partitions. This also means that you can safely ignore the Analysis Management Objects warning (AMO) in Visual Studio that partition sizes should not exceed 20 million rows. We have measured the effect of large partition sizes, and found that they show negligible performance differences compared to smaller partition sizes. Therefore, the reduction in partition management overhead justifies relaxing the guidelines on partition sizes, particularly when you have large numbers of partitions.

7.3.5.1 Adding and Managing Partitions

Managing partitions on a large cube quickly becomes a large administrative task if done manually through SQL Server Management Studio. We recommend that you create scripts to help you manage partitioning of the cube in an automated manner instead.

The best way to manage a large cube is to use a relational database to hold the metadata about the desired partitioning scheme of the cube and use XMLA to keep the cube structure in sync with the metadata in the relational source. Every time you request metadata from a cube, you have to run a DISCOVER command - some of these commands are expensive on a large cube with many partitions. You should design partition management code to extract the metadata from the cube in a single bulk operation instead of multiple small DISCOVER commands. Note that when you use AMO (and the SQL Server Integration Services task) to access the cube, some amount of error handling is built into the .NET library. This error handling code executes DISCOVER commands and be quite chatty with the server. A good example of this is the partition add function in AMO. This code will first check to see whether the partition already exists (it raises an error if it does) using a DISCOVER command. After that it will issue the CREATE command, creating the partition in the cube. This technique turns out to be a highly inefficient way to add many new partitions on cube with thousands of existing partitions. Instead, it is faster to first read all the partitions using a single XMLA command, discover which partitions are missing ,and then run an XMLA command to create each missing partition directly – avoiding the error checking that AMO.NET does.

In summary, design partition management code carefully and test it using SQL Server Profiler to trace the commands you generate as the code runs. Because it provides so much feedback to the server, partition management code is very expensive, and it is often surprising to customers how easy it is to create inefficient operations on a big cube. That is the price of the safety net that ADOMD.NET gives you. Using XMLA to carefully manage partitions is often a better solution for a large cube.

7.4 Locking and Blocking

Locks are used to manage concurrent operations. In an Analysis Services deployment, you'll find that locks are used pervasively in discovery and execution to ensure the stability of underlying data structures while a query or process is running.

Occasionally, locks taken for one operation can block other processes or queries from running. A common example is when a query takes a long time to execute, and another session issues a write operation involving the same objects. The query holds locks, causing the write operation to wait for the query to complete. Additional new queries issued on other sessions are then blocked as they wait for the original query operation to complete, potentially causing the entire service to stop handling commands until the original long-running query completes.

This section takes a closer look at how locks are used in different scenarios, and how to diagnose and address any performance problems or service disruptions that arise from locking behaviors. One of the tools at your disposal is the new SQL Server Profiler lock events that were introduced in SQL Server 2008 R2 Service Pack 1 (SP1). Finally, this section looks at deadlocks and provides some recommendations for resolving them if they begin to occur too frequently.

7.4.1 Lock Types

In Analysis Services, the most commonly used lock types are Read, Write, Commit_Read, and Commit_Write. These locks are used for discovery and execution on the server. SQL Server Profiler, queries, and dynamic management views (DMVs) use them to synchronize their work. Other lock types (such as LOCK_NONE, LOCK_SESSION_LOCK, LOCK_ABORTABLE, LOCK_INPROGRESS, LOCK_INVALID) are used peripherally as helper functions for concurrency management. Because they are peripheral to this discussion, this section focuses on the main lock types instead.

The following table lists these locks and briefly describes how they are used.

Lock type	Object	Usage
Read	All	Read locks are used for reading metadata objects, ensuring that these objects cannot be modified while they are being used. Read locks are shared, meaning that multiple transactions can take Read locks on the same object.
Write	All	Write locks are used for create, alter, and update operations. Write locks are exclusive. Exclusive locks prevent concurrent transactions from taking Read or Write locks on the object at the same time.
Commit_Read	Database, Server, Server Proxy	Commit_Read locks are shared, but they block write operations that are waiting to commit changes to disk. Database Objects: Commit_Read locks are used for the following: - Query operations to ensure that no commit transactions overwrite any of the metadata objects used in the query. The lock is held for the duration of a query. - During processing, while acquiring Read and Write locks on other objects. - At the beginning of a session to calculate user permissions for a given database. - During some discover operations, such as those that read from a database object. Server Objects: Commit_Read locks are used at the beginning of a session to compute session security. Commit_Read is also used at the start of a transaction to read the master version map. Server Proxy Objects: Commit_Read locks are used at the beginning of SQL, DMX, and MDX queries to prevent changes to assemblies or administrative role while the objects are retrieved.
Commit_Write	Database, Server, Server Proxy	Commit_Write locks are exclusive locks on an object, taken to prevent access to that object while it is being updated. Commit_Write is the primary mechanism for ensuring “one version of the truth” for queries.

	Database Objects: A Commit_Write is used in a commit transaction that creates, updates, or deletes DDL structures in the database. In a commit transaction, Commit_Write locks can be taken on multiple databases at the same time, assuming the transaction includes them (for example, deleting multiple databases in one transaction). Server Object: A Commit_Write is used to update the master version map. More information about the master version map is in the next section on server locks. Server Proxy Object: A Commit_Write is taken whenever there are changes to assemblies or membership of the server role.

7.4.2 Server Locks

In Analysis Services, there are two server lock objects, the Server and the Server Proxy, each used for very specific purposes.

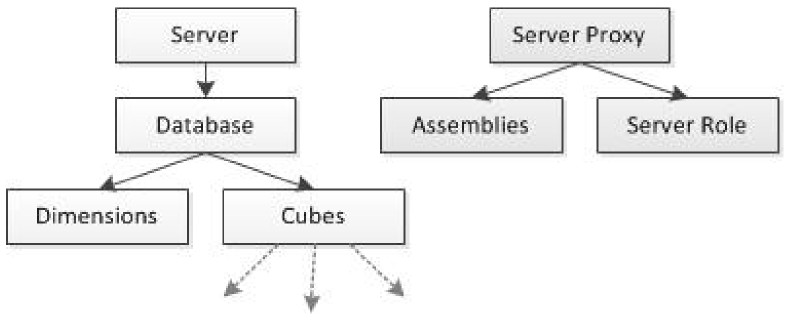


Figure 24 - Server Lock Hierarchies

The Server object takes a commit lock for operations such as processing or querying that traverse the object hierarchy. Typically, commit locks on the server are very brief, taken primarily whenever the server reads the master version map (Master.vmp) file or updates it so that it contains newer versions of objects changed by a transaction.

The master version map is a list of object identifiers and current version number for all of the major objects recognized by the server (that is, Database, Cube, Measure Groups, Partitions, Dimensions and Assemblies). Minor objects, such as hierarchy levels or attributes, do not appear in the list. The master version map identifies which object version to use at the start of a query or process. The version number of a major object changes each time you process or update it. Only the Server object reads and writes to the Master.vmp file.

When reading the master version map, the server takes a Commit_Read to protect against changes to the file while it is being read. A Commit_Write is taken on the Server object to update the master version map, by merging newer version information that was created in a transaction.

The Server Proxy object is used for administrative locks, for example when you make changes to the membership of the Server role or to assemblies used by the database.

7.4.3 Lock Fundamentals

This section explores some of the foundational concepts that describe locking behavior in Analysis Services. Like all locking mechanisms, the purpose of locks in Analysis Services is to protect metadata or data from the effects of concurrency.

The rules or principles of how locks are used can be distilled into the following points:

- Locks protect metadata; latches protect data. Latches are lightweight locks that are rapidly acquired and released. They are used for atomic operations, like reading a data value out of system data.
- Read locks are taken for objects that feed into a transaction; Write locks are taken for objects that are changed by the transaction. Throughout the rest of this section, we'll see how this principle is applied in different operations.

Consider the case of dimension processing. Objects like data sources and data source views that provide data to the dimension (that is, objects that the dimension depends on) take a Read lock. Objects that are changed by the operation, such as partitions, take a Write lock. When a dimension is processed, the partitions need to be unprocessed and then reprocessed, so a Write lock is taken on the partitions to update the dimension-related data within each partition.

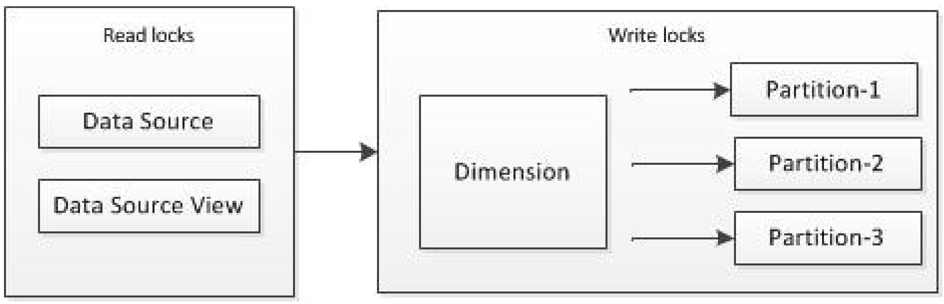


Figure 25 - Locks during dimension processing

Compare the preceding illustration to the following one, which shows only Read locks used for partition processing. Because no major objects depend on a partition, the only Write lock in play is on the partition itself.

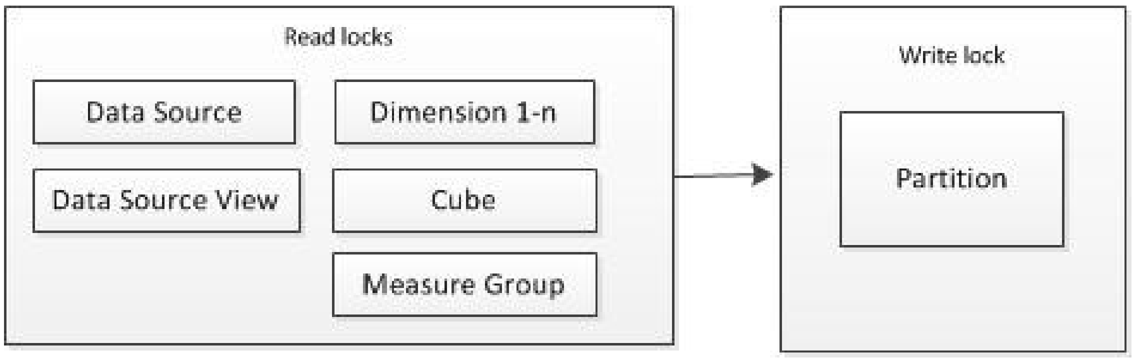


Figure 26 - Locks during partition processing

7.4.4 Locks by Operation

This section explains the locks taken by different operations on the server. This list of locks is not exhaustive:

- Begin Session
- Queries
- Discover
- Processing
- DDL operations
- Rollback

Other operations such as lazy processing, sessions, and proactive caching pose interesting challenges in terms of understanding locks. The reason is discussed at the end of this section, but these other operations are not discussed in detail. Also missing from the list is writeback, Because writeback is a hybrid of query and process operations, which are covered individually.

7.4.4.1 Begin Session

When a session starts, Analysis Services determines which databases the user has permission to use. Performing this task requires taking a Commit_Read lock on the database. If a database is not specified, session security is computed for each database on the server until a database is found that the session has read access to. Locks are released after session security is computed. Occasionally, the commit lock is acquired and released so quickly that it never shows up in a trace.

7.4.4.2 Queries

All queries run on a database and in a session. A Commit_Read lock is taken simultaneously on the Server Proxy object and database when the query starts. The lock is held for the duration of the query.

The Commit_Read lock on a database held by query is sometimes blocks processing operations.

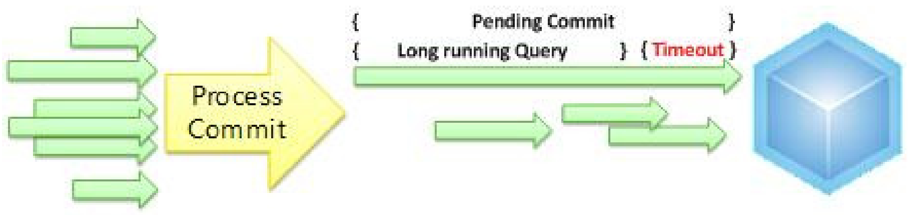


Figure 27 - Write locks blocking queries

In the preceding example, the long-running query prevents the Commit_Write lock required by the processing operation (yellow arrow) from being taken. New readers requiring the Commit_Read lock on the database then become blocked behind the processing command. It is this combination of events that can turn a three-second query into a three-minute (or longer) query.

7.4.4.3 Discovers

There are two types of discover operations that use locks: those that request metadata about instances or objects (such as MDSCHEMA_CUBE), and those that are part of a dynamic management view (DMV) query.

The Discover method on metadata objects behaves very similarly to a query in that it takes a Commit_Read lock on the database.

Discovers issued for DMV operations don't take Read locks. Instead, DMVs use interlocking methods to synchronize access to system data that is created and maintained by the server. This approach is sufficient because DMV discovers do not read from the database structure. As such, the protection offered by locks is overkill. Consider DISCOVER_OBJECT_MEMORY_USAGE. It is a DMV query that returns shrinkable and unshrinkable memory used by all objects on the server at given point in time. Because it is a fast read of system data directly on the server, DISCOVER_OBJECT_MEMORY_USAGE uses interlocked access over memory objects to retrieve this information.

7.4.4.4 Processing

For processing, locks are acquired in two phases, first during object acquisition and again during schedule processing. After these first two phases, the data can be processed and the changes committed.

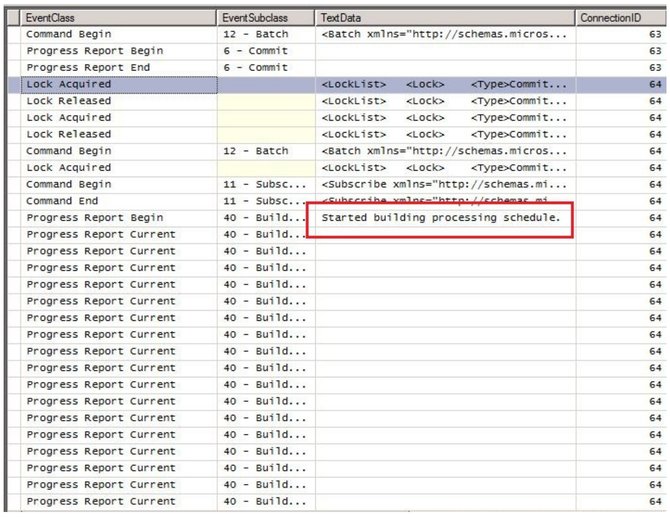
Phase 1: Object Acquisition

1. A Commit_Read lock is acquired on the database while the objects are retrieved.
2. The objects used in processing are looked up.
3. The objects are identified, and then the Commit_Read lock on the database is released.

Phase 2: Schedule Processing

This phase finds the objects on which the dimension or partition depends, as well as those objects that depend on it. In practice, building the schedule is an iterative process to ensure that all of the dependencies are identified. After all of the dependencies are understood, the execution workflow moves forward to the processing phase. The following list summarizes the events in schedule processing.

1. Scheduler builds a dependency graph that identifies all of the objects involved in the processing command. Based on this graph, it builds a schedule of operations.



EventClass	EventSubclass	TextData	ConnectionID
Command Begin	12 - Batch	<Batch xmlns="http://schemas.micros...	63
Progress Report Begin	6 - Commit		63
Progress Report End	6 - Commit		63
Lock Acquired		<LockList> <Lock> <Type>Commit...	64
Lock Released		<LockList> <Lock> <Type>Commit...	64
Lock Acquired		<LockList> <Lock> <Type>Commit...	64
Lock Released		<LockList> <Lock> <Type>Commit...	64
Command Begin	12 - Batch	<Batch xmlns="http://schemas.micros...	64
Lock Acquired		<LockList> <Lock> <Type>Commit...	64
Command Begin	11 - Subsc...	<Subscribe xmlns="http://schemas.mi...	64
Command End	11 - Subsc...	<Subscribe xmlns="http://schemas.mi	64
Progress Report Begin	40 - Build...	Started building processing schedule.	64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64
Progress Report Current	40 - Build...		64

2. Scheduler acquires a Commit_Read on the database, and then it acquires Read and Write locks on the objects.
3. Scheduler generates the jobs and is now free to start executing the jobs.

Phase 3: Process

After scheduling, there are no Commit_Read locks, just Write locks on the affected objects, and read locks on objects. Only Read and Write locks are used to do the work. Read and Write locks prevent other transactions from updating any objects used in the transaction. At the same time, objects that are related but not included in the current transaction can be updated. For example, suppose two different dimensions are being processed in parallel. If each dimension is referenced by different cubes and if each dimension is fully independent of the other, there is no conflict when the transaction is committed. Had a Commit_Write lock been held on the database, this type of parallel processing would not be possible.

Phase 4: Commit

When a Commit transaction starts, a Commit_Write lock is taken on the database and held for the duration of the transaction. No new Commit_Read locks are accepted while Commit_Write is pending. Any new Begin Sessions may encounter a connection timeout. New queries are queued or canceled, depending on timeout settings.

The Commit transaction waits for the query queue to drain (that is, it waits for Commit_Read locks to be released). At this point, reading from the property settings defined on the server, it might use one of the following properties to cancel a query if it is taking too long:

ForceCommitTimeout cancels transactions that hold Commit_Read locks. By default, this property is set to 30 seconds. However, it is important to remember that cancelations are not always instantaneous. Sometimes it takes several minutes to release commit locks.

CommitTimeout cancels transactions requesting Commit_Write locks, in effect prioritizing queries over processing. The server uses whichever timeout occurs first. If **ForceCommitTimeout** occurs sooner than **CommitTimeout**, cancelation is called on long-running queries instead of the write operation.

Note: If the client application has its own retry logic, it can reissue a command in response to a connection timeout and the connection will succeed if there are available threads.

A commit transaction also takes a Commit_Write lock on the server. This is very short and occurs when the transaction version map is merged into the master version map that keeps track of which object versions are the master versions. Because this is a Write lock, it must wait for Read locks to clear – this wait time is controlled by **ForceCommitTimeout** and **CommitTimeout**. A commit transaction creates new versions of the master objects, ensuring consistent results for all queries.

After the Master.vmp file is updated, the server deletes unused version data files. At this point, the Commit_Write lock is released.

7.4.4.5 DDL Operations

For operations that create, alter, or delete metadata objects, locks are acquired once. It is similar to the processing workflow but without the scheduling phase. It finds the objects that the object depends on and locks them using Read locks, and Write locks are taken on the objects that depend on the object that is being modified. Write locks are also acquired on the objects that are created, updated, or deleted in the transaction.

7.4.4.6 Rollback

For rollback, there is a server latch that protects Master.vmp, but a rollback by itself does not take a Commit_Read or Commit_Write lock on the database. In a rollback, nothing is changing, so no commit locks are required. However, any Read and Write locks that were taken by the transaction are still held during the rollback, but they are released after the rollback has been completed. Rollback deletes the unused versions of any objects created by the original transaction.

7.4.4.7 Session Transactions, Proactive Caching, and Lazy Processing

In terms of locking, sessions, proactive caching and lazy processing are tricky because they have breakable locks. Pending a write operation from an administrator, Lock Manager cancels Write locks in a session, proactive caching, or lazy processing, and it lets the Write Commit prevail.

Sessions can have Write locks that don't conflict. Snapshots and checkpoints are used to manage write operations for session cubes. The classic example is the grouping behavior in Excel where new dimensions are created in-session, on the fly, to support ad-hoc data structures. The new dimensions are always rolled back eventually, but they can be problematic during their lifetime. If an error occurs in session, the server might perform a partial rollback to achieve a stable state; sometimes these rollback operations have unintended consequences.

7.4.4.8 Synchronization, Backup and Restore, and Attach

The locking mechanism for synchronization, restore, and attach are already documented in the "Analysis Services Synchronization Best Practices" article on the SQLCAT web site (http://sqlcat.com). Restated from that article, the basic workflow of lock acquisition and release for synchronization is as follows.

During synchronization, a Write lock is applied before the merge of the files on the target server, and a read commit lock is applied to the source database while files are transferred.

- The Write lock is applied on the target server (when a target database exists), preventing users from querying and/or writing in database. The lock is released from the target database after the metadata is validated, the merge has been performed and the transaction is committed.
- The read commit lock is taken on the source server to prevent processing from committing new data, but it allows queries to run, including other source synchronization. Because of this, multiple servers can be synchronized at the same time from the same source. The lock is released at about the same moment as the Write lock, as it is taken in this distributed transaction.

For synchronization only, there is also an additional Commit_Write lock while the databases are merged. This lock can be held for a long period of time while the database is created and overwritten.

For backup, there is only a Commit_Read on the database.

A restore operation uses a more complex series of locks. It acquires a Write lock on the database that is being restored, ensuring the integrity of the database as its being restored, but blocking queries while Restore is being processed. Best practice recommendations suggest using two databases with different names so that you can minimize query downtime.

The basic workflow for all of these operations is as follows:

1. Write locks are taken on the database that is being synchronized, restored, or attached.
2. Files are extracted and changes committed while processing.
3. The locks are released.

7.4.5 Investigating Locking and Blocking on the Server

This section describes the tools and techniques for monitoring locks and removing locking problems. One of the tools discussed is the new lock events that are introduced in SQL Server 2008 R2 SP1. The other tool is DMVs (and the DISCOVER_LOCKS schema rowset in particular), which are discussed with emphasis on how it fits into a troubleshooting scenario.

Locking and blocking problems manifest themselves in a server environment in different ways. As with all performance problems, this one is a matter of degree. You might have a blocking issue in your environment that resolves so quickly, it never registers as a problem that requires a solution. At the other end of the spectrum, blocking can become so severe that users are locked out of the system. Connection requests fail; queries either time out or fail to start altogether.

In these extreme scenarios, it is difficult to clearly diagnose the problem if you do not have an available thread to connect SQL Server Profiler or if you are unable to run a DMV query that tells you what is going on. In this case, the only way to confirm a locking problem is to work with a Microsoft support engineer to analyze a memory dump. The engineer can tell you whether your server unavailability is due to locks, indicated by **PCLockManager::Wait** on multiple threads.

7.4.5.1 Using DMV Queries and XMLA to Cancel a Blocked Transaction

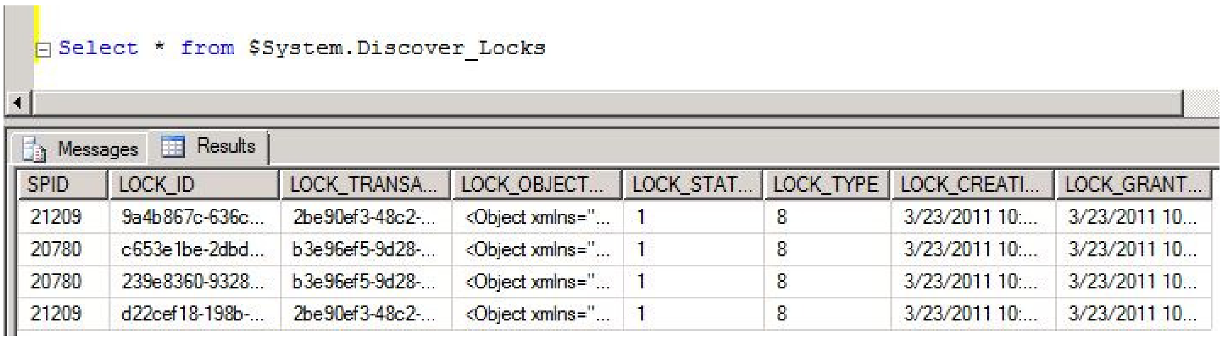
If your situation is less dire, you can use a DMV query and XML for Analysis (XMLA) to unblock your server. Given a free thread for processing a new connection request, you can connect to the server using an administrator account, run a DMV query to get the list of locks, and then try to kill the blocking SPID by running this XMLA command.

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

      <SPID>nnnn</SPID>

</Cancel>
```

To get the SPID, you can use SQL Server Management Studio to connect to the server and then issue a DISCOVER_LOCKS statement as an MDX query.

```
      Select * from $SYSTEM.DISCOVER_LOCKS
```

This DMV query returns a snapshot of the locks used at specific point in time. DISCOVER_LOCKS returns a rowset directly from Lock Manager. As such, the data you get back might not be as clear or easy to follow as the trace events in SQL Server Profiler. The following example contains a snapshot of the locks held during a query but no information about the query itself.



```
Select * from $System.Discover_Locks
```

SPID	LOCK_ID	LOCK_TRANSA...	LOCK_OBJECT...	LOCK_STAT...	LOCK_TYPE	LOCK_CREATI...	LOCK_GRANT...
21209	9a4b867c-636c...	2be90ef3-48c2-...	<Object xmlns="...	1	8	3/23/2011 10:...	3/23/2011 10...
20780	c653e1be-2dbd...	b3e96ef5-9d28-...	<Object xmlns="...	1	8	3/23/2011 10:...	3/23/2011 10...
20780	239e8360-9328...	b3e96ef5-9d28-...	<Object xmlns="...	1	8	3/23/2011 10:...	3/23/2011 10...
21209	d22cef18-198b-...	2be90ef3-48c2-...	<Object xmlns="...	1	8	3/23/2011 10:...	3/23/2011 10...

Figure 28 - Output of $system.discover_locks

Unfortunately, there is no mitigation for this. You cannot run DISCOVER_LOCKS in conjunction with other DMV statements to get additional insight into the timing of the lock acquisition-release lifecycle relative to the transactions running on your server. You can only run the MDX SELECT statement and then act on the information it provides.

7.4.5.2 Using SQL Server Profiler to Analyze Lock Contention

SQL Server Profiler offers numerous advantages over DMV queries in terms of depth and breadth of information, but it comes at a cost. It is often not feasible to run SQL Server Profiler in a production environment because of the additional resource demands it places on server. But if you can use it, and if you are running SQL Server 2008 R2 SP1, you can add the new Lock Acquired, Lock Released, and Lock Waiting events to a trace to understand the locking activity on your server.

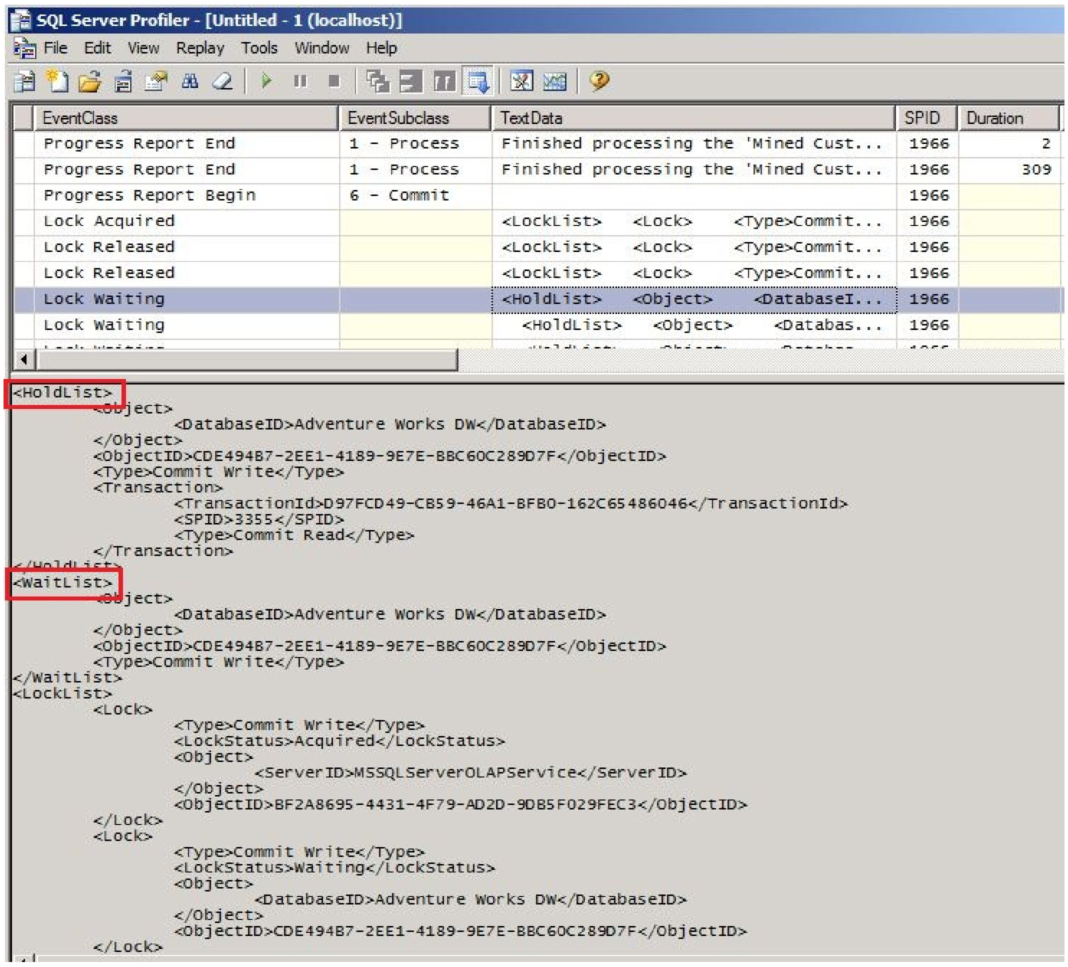
SQL Server 2008 R2 SP1adds the following events to the Locks event category in SQL Server Profiler: Lock Acquired, Lock Released, Lock Waiting.

Like other lock events, they are not enabled by default. You must select the **Show all columns** check box before you can select events in the Locks category. To get an idea of how locks are acquired and released in the course of a transaction be sure to add Command Begin and Command End to the trace. If you want to view the MDX executed, you should also include the Query Begin and Query End.

In Analysis Services, it is normal to see a large number of acquired and released lock events. Every transaction that includes Read or Write operations on a major object requests a lock to perform that action. Lock Waiting is less common, but by itself is not symptomatic of a problem. It merely indicates that a queue has formed for transactions that are requesting the same object. If the queue is not long and operations complete quickly, the queue drains and query or processing tasks proceed with only a small delay.

The following illustration shows a Lock Waiting event and the XML structures that identify which transaction currently holds a lock on the object, and which transactions are waiting for that same object. In the illustration:

- **<HoldList>** shows which transaction is currently holding a Commit_Write lock on the database.
- **<WaitList>** indicates that another transaction is waiting for a Commit_Write lock on the same database.


SQL Server Profiler - [Untitled - 1 (localhost)]
File Edit View Replay Tools Window Help

EventClass	EventSubclass	TextData	SPID	Duration
Progress Report End	1 - Process	Finished processing the 'Mined Cust...	1966	2
Progress Report End	1 - Process	Finished processing the 'Mined Cust...	1966	309
Progress Report Begin	6 - Commit		1966	
Lock Acquired		<LockList> <Lock> <Type>Commit...	1966	
Lock Released		<LockList> <Lock> <Type>Commit...	1966	
Lock Released		<LockList> <Lock> <Type>Commit...	1966	
Lock Waiting		<HoldList> <Object> <DatabaseI...	1966	
Lock Waiting		<HoldList> <Object> <Databas...	1966	

```
<HoldList>
        <Object>
                <DatabaseID>Adventure Works DW</DatabaseID>
        </Object>
        <ObjectID>CDE494B7-2EE1-4189-9E7E-BBC60C289D7F</ObjectID>
        <Type>Commit Write</Type>
        <Transaction>
                <TransactionId>D97FCD49-CB59-46A1-BFB0-162C65486046</TransactionId>
                <SPID>3355</SPID>
                <Type>Commit Read</Type>
        </Transaction>
</HoldList>
<WaitList>
        <Object>
                <DatabaseID>Adventure Works DW</DatabaseID>
        </Object>
        <ObjectID>CDE494B7-2EE1-4189-9E7E-BBC60C289D7F</ObjectID>
        <Type>Commit Write</Type>
</WaitList>
<LockList>
        <Lock>
                <Type>Commit Write</Type>
                <LockStatus>Acquired</LockStatus>
                <Object>
                        <ServerID>MSSQLServerOLAPService</ServerID>
                </Object>
                <ObjectID>BF2A8695-4431-4F79-AD2D-9DB5F029FEC3</ObjectID>
        </Lock>
        <Lock>
                <Type>Commit Write</Type>
                <LockStatus>Waiting</LockStatus>
                <Object>
                        <DatabaseID>Adventure Works DW</DatabaseID>
                </Object>
                <ObjectID>CDE494B7-2EE1-4189-9E7E-BBC60C289D7F</ObjectID>
        </Lock>
```

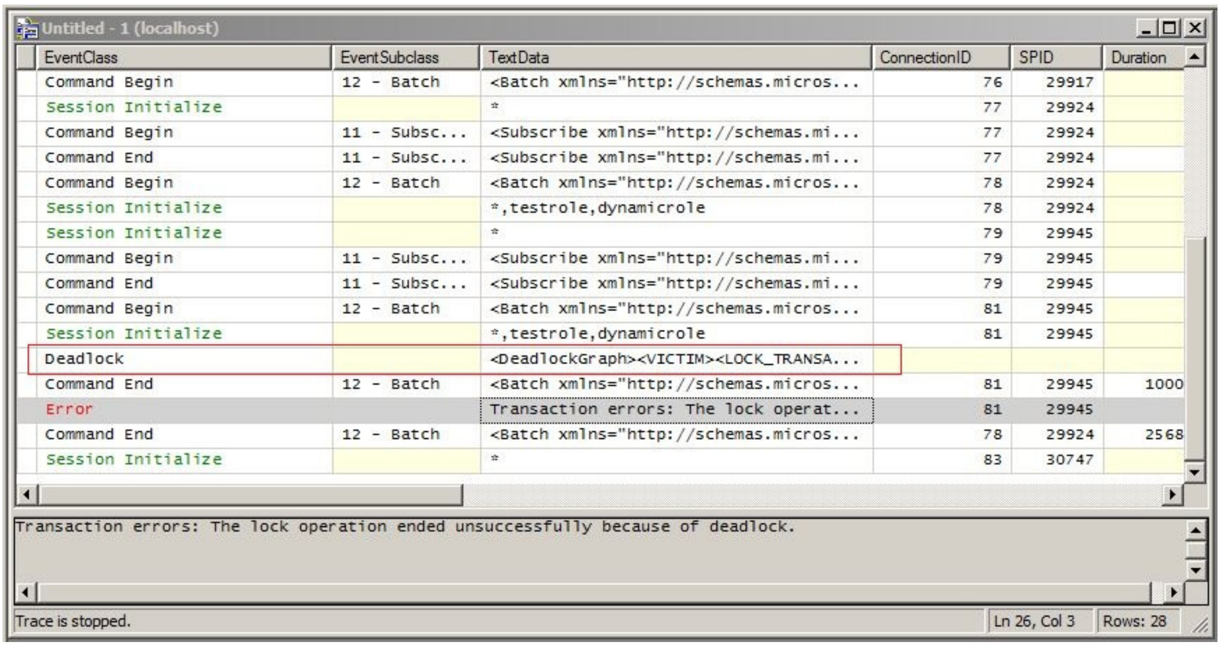
Figure 29 – SQL Server Profiler output of lock events

7.4.6 Deadlocks

A deadlock in Analysis Services is lock contention between two or more sessions, where operations in either session can't move forward because each is waiting to acquire a lock held by the other session.

Analysis Services has deadlock detection but it only works for locks (Read, Write, Commit_Read, Commit_Write). It won't detect deadlocks for latches. When a deadlock is detected, Analysis Services stops the transaction in one of the sessions so that the other transaction can complete.

Deadlocks are typically a boundary case. If you can reproduce it, you can run a SQL Server Profiler trace and use the Deadlock event to determine the point of contention. In SQL Server Profiler, a deadlock looks like the following:


Untitled - 1 (localhost)

EventClass	EventSubclass	TextData	ConnectionID	SPID	Duration
Command Begin	12 - Batch	<Batch xmlns="http://schemas.micros...	76	29917	
Session Initialize		*	77	29924	
Command Begin	11 - Subsc...	<Subscribe xmlns="http://schemas.mi...	77	29924	
Command End	11 - Subsc...	<Subscribe xmlns="http://schemas.mi...	77	29924	
Command Begin	12 - Batch	<Batch xmlns="http://schemas.micros...	78	29924	
Session Initialize		*,testrole,dynamicrole	78	29924	
Session Initialize		*	79	29945	
Command Begin	11 - Subsc...	<Subscribe xmlns="http://schemas.mi...	79	29945	
Command End	11 - Subsc...	<Subscribe xmlns="http://schemas.mi...	79	29945	
Command Begin	12 - Batch	<Batch xmlns="http://schemas.micros...	81	29945	
Session Initialize		*,testrole,dynamicrole	81	29945	
Deadlock		<DeadlockGraph><VICTIM><LOCK_TRANSA...			
Command End	12 - Batch	<Batch xmlns="http://schemas.micros...	81	29945	1000
Error		Transaction errors: The lock operat...	81	29945	
Command End	12 - Batch	<Batch xmlns="http://schemas.micros...	78	29924	2568
Session Initialize		*	83	30747	

```
Transaction errors: The lock operation ended unsuccessfully because of deadlock.
```

Trace is stopped. Ln 26, Col 3 Rows: 28

Figure 30 - Deadlock events in profiler

A deadlock event uses an XML structure called a deadlockgraph to report on which sessions, transactions, and objects created the event. The <VICTIM> node in the first few lines identifies which transaction was sacrificed for the purpose of ending the deadlock. The remainder of the graph enumerates the lock type and status for each object. In the following example, you can see which objects are requested and whether the lock was granted or waiting.

```
<DeadlockGraph>
 <VICTIM>
   <LOCK_TRANSACTION_ID>E0BF8927-F827-4814-83C1-98CA4C7F5413</LOCK_TRANSACTION_ID>
   <SPID>29945</SPID>
 </VICTIM>
<LOCKS>

  <Lock>s
    <LOCK_OBJECT_ID><Object><DatabaseID>FoodMart
2008</DatabaseID><DimensionID>Promotion</DimensionID></Object></LOCK_OBJECT_ID>
    <LOCK_ID>326321DF-4C08-43AA-9AEC-6C73440814F4</LOCK_ID>
    <LOCK_TRANSACTION_ID>E0BF8927-F827-4814-83C1-98CA4C7F5413</LOCK_TRANSACTION_ID>
    <SPID>29945</SPID>
    <LOCK_TYPE>2</LOCK_TYPE>      ---- Lock_Type 2 is a Read lock
    <LOCK_STATUS>1</LOCK_STATUS>  ---- Lock_Status 1 is 'acquired'
  </Lock>

  <Lock>
    <LOCK_OBJECT_ID><Object><DatabaseID>FoodMart 2008</DatabaseID>
<DimensionID>Product</DimensionID></Object></LOCK_OBJECT_ID>
    <LOCK_ID>21C7722B-C759-461A-8195-1C4F5A88C227</LOCK_ID>
    <LOCK_TRANSACTION_ID>E0BF8927-F827-4814-83C1-98CA4C7F5413</LOCK_TRANSACTION_ID>
    <SPID>29945</SPID>
    <LOCK_TYPE>2</LOCK_TYPE>      ---- Lock_Read
    <LOCK_STATUS>0</LOCK_STATUS>  ---- waiting on Product, which is locked by 29924
  </Lock>
  <Lock>
    <LOCK_OBJECT_ID><Object><DatabaseID>FoodMart 2008</DatabaseID>
<DimensionID>Promotion</DimensionID></Object></LOCK_OBJECT_ID>
    <LOCK_ID>7F15875F-4CCB-4717-AE11-5F8DD48229D0</LOCK_ID>
    <LOCK_TRANSACTION_ID>1D3C42F3-E875-409E-96A0-B4911355675D</LOCK_TRANSACTION_ID>
    <SPID>29924</SPID>
    <LOCK_TYPE>4</LOCK_TYPE>      ---- Lock_Write
    <LOCK_STATUS>0</LOCK_STATUS>  ---- waiting on Promotion, which is locked by 29945
  </Lock>

  <Lock><LOCK_OBJECT_ID><Object><DatabaseID>FoodMart 2008</DatabaseID>
<DimensionID>Product</DimensionID></Object></LOCK_OBJECT_ID>
    <LOCK_ID>96B09D08-F0AE-4FD2-8703-D33CAD6B90F1</LOCK_ID>
    <LOCK_TRANSACTION_ID>1D3C42F3-E875-409E-96A0-B4911355675D</LOCK_TRANSACTION_ID>
    <SPID>29924</SPID>
    <LOCK_TYPE>4</LOCK_TYPE>      ---- Lock_Write
    <LOCK_STATUS>1</LOCK_STATUS>  ---- granted
  </Lock><
/LOCKS>
</DeadlockGraph>
```

Deadlocks should be rare events, if they occur at all. Persistent deadlocks indicate a need for redesigning your processing strategy. You might need to speed up processing by making greater use of partitions, or you might need to take a closer look at other processing options or schedules to see whether you can eliminate the conflict. For more information about these recommendations, see the Analysis Services Performance Guide (http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3be0488d-e7aa-4078-a050-ae39912d2e43&displaylang=en).

References:

- Analysis Services Synchronization Best Practices - http://sqlcat.com/technicalnotes/archive/2008/03/16/analysis-services-synchronization-best-practices.aspx
- Deadlock Troubleshooting in SQL Server Analysis Services (SSAS) - http://blogs.msdn.com/b/sql_pfe_blog/archive/2009/08/27/deadlock-troubleshooting-in-sql-server-analysis-services-ssas.aspx
- SSAS: Processing, ForceCommitTimeout and "the operation has been cancelled" - http://geekswithblogs.net/darrengosbell/archive/2007/04/24/SSAS-Processing-ForceCommitTimeout-and-quotthe-operation-has-been-cancelledquot.aspx
- Locking and Unlocking Databases (XMLA) - http://msdn.microsoft.com/en-us/library/ms186690.aspx

7.5 Scale Out

Analysis Services currently supports up to 64 cores in a scale-up configuration. If you want to go beyond that scale, you will have to design for scale-out. There are also other drivers for scale-out; for example, it may simply be cheaper to use multiple, smaller machines to achieve high user concurrency. Another consideration is processing and query workloads. If you expect to spend a lot of time processing or if you are designing for real time, it is often useful to scale-out the processing on a different server than the query servers.

Scale-out architectures can also be used to achieve high availability. If you have multiple query servers in a scale-out farm, some of them can fail but the system will remain online.

Although the full details of designing a scale-out Analysis Services farm is outside the scope of this guide, it is useful to understand the tradeoffs and potential architectures that can be applied.

7.5.1 Provisioning in a Scaled Out Environment

In a scale-out environment you can use either read-only LUN and the attach and detach functionality of SQL Server 2008 Analysis Services or SAN snapshots (which can also be used in SQL Server 2005 Analysis Services) to host multiple copies of the same database on many machines.

When you update a scale-out read-only farm, a Windows volume has to be dismounted and mounted every time you update an Analysis Services database. This means that no matter which disk technology you use the scale out, the smallest unit you can update without disturbing other cubes is a Windows volume. If you have multiple databases in the scale-out environment, it is therefore an advantage to have each database live on its own Windows volume. This separation allows you to update the databases independently of each other if you are running SQL Server 2008 Analysis Services and SQL Server 2008 R2 Analysis Services.

7.5.2 Scale-out Processing Architectures

There are basically three different architectures you can use in a scale-out configuration:

- Dedicated processing architecture
- Query/processing flipping architecture
- ROLAP

The three architectures have different tradeoffs that you have to consider, which this section describes. Note: It is also possible to combine these architectures in different hybrids, but that is outside the scope of this document. Understanding the tradeoffs for each will help you make the right design decisions.

7.5.2.1 Dedicated Processing Architecture

In the dedicated processing architecture, an instance of Analysis Services is reserved to process all new, incoming data. After processing is done, the result is copied to query servers. The advantage of this architecture is that the query servers can respond to the queries without being affected by the processing operation. A lock is required only when data is updated or added to the cube.

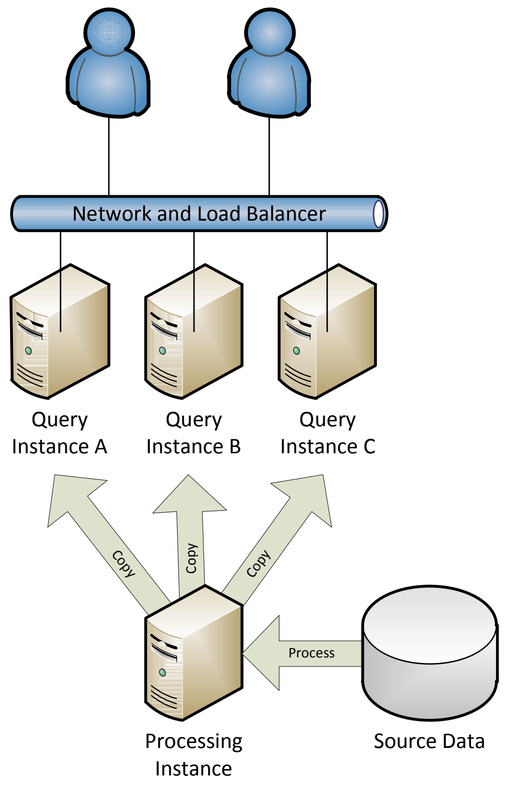


Figure 31 - Dedicated Processing Architecture

In a dedicated processing architecture, consider how to get the data from the processing instance to the query servers. There are several ways to achieve this.

Analysis Services Cube Synchronization: By using this built-in Analysis Services functionality, you can move the delta data directly to the query servers.

Robocopy or NiceCopy: By using a high-speed copying program, you can quickly synchronize each query instance with its own copy of the changed data. This method is generally faster than cube synchronization, but it requires you to set up your own copy scripts.

SAN Snapshots or Storage Mirrors: Using SAN technology, it is possible to automatically maintain copies of the data LUN on the processing servers. These copies can then be mounted on the query servers when the data is updated.

SAN Read -Only LUN: Using this technique, which is available only in SQL Server 2008 and SQL Server 2008 R2, you can use read-only LUN to move the data from the processing instance to the query servers. A read-only LUN can be shared between multiple servers, and hence enables you to use more than query server on the same physical disk.

Both SAN snapshots and read-only LUN strategies may require careful design of the storage system bandwidth. If your cube is small enough to fit in memory, you will not see much I/O activity and this technique will work very well out of the box. However, if the cube is large and cannot fit in memory, Analysis Services will have to do I/O operations. As you add more and more query servers to the same SAN, you may end up creating a bottleneck in the storage processors on the SAN to serve all this I/O. You should make sure that the SAN is capable of supporting the required throughput. If the I/O throughput is not sufficient, you may end up with a scale-out solution that performs worse than a scale-up.

If you are worried about I/O bandwidth, the Robocopy/NiceCopy solution or the cube synchronization solution may work better for you. In these solutions you can have dedicated storage on each query server . However, you have to make sure there is enough bandwidth on the network to run multiple copies over the network. You may have to use dedicated network cards for such a setup.

The dedicated processing architectures can also be used to achieve high availability. However, you need a way to protect the processing server, to avoid a single point of failure. You can have either a standby processing server or an extra (disabled) instance on one of the query servers that can be used to take over the role of processing service in the case of hardware failure. Another alternative is to use clustering on the processing server ,although this may waste hardware resources on the passive node.

References:

- Scale-Out Querying for Analysis Services with Read-Only Databases - http://sqlcat.com/whitepapers/archive/2010/06/08/scale-out-querying-for-analysis-services-with-read-only-databases.aspx
- Sample Robocopy Script to Synchronize Analysis Services Databases - http://sqlcat.com/technicalnotes/archive/2008/01/17/sample-robocopy-script-to-customer-synchronize-analysis-services-databases.aspx
- Scale-Out Querying with Analysis Services - http://sqlcat.com/whitepapers/archive/2007/12/16/scale-out-querying-with-analysis-services.aspx
- Scale-Out Querying with Analysis Services Using SAN Snapshots - http://sqlcat.com/whitepapers/archive/2007/11/19/scale-out-querying-with-analysis-services-using-san-snapshots.aspx
- Analysis Services Synchronization Best Practices - http://sqlcat.com/technicalnotes/archive/2008/03/16/analysis-services-synchronization-best-practices.aspx
- SQL Velocity – Scalable Shared Data base - http://sqlvelocity.typepad.com/blog/2010/09/scalable-shared-data-base-part-1.html

7.5.2.2 Query/Processing Flipping Architecture

The dedicated processing server architecture solves most scale-out cases. However, the time required to move the data from the processing server to the query servers may be restrictive if updates happen at intervals shorter than a few hours. Even with SAN snapshots or read-only LUN, it will still take some time to dismount the LUN, set it online on the query server, and finally mount the updated cube on the query servers. In the query/processing flipping architecture, each instance of Analysis Services performs its own processing, as illustrated in the following figure.

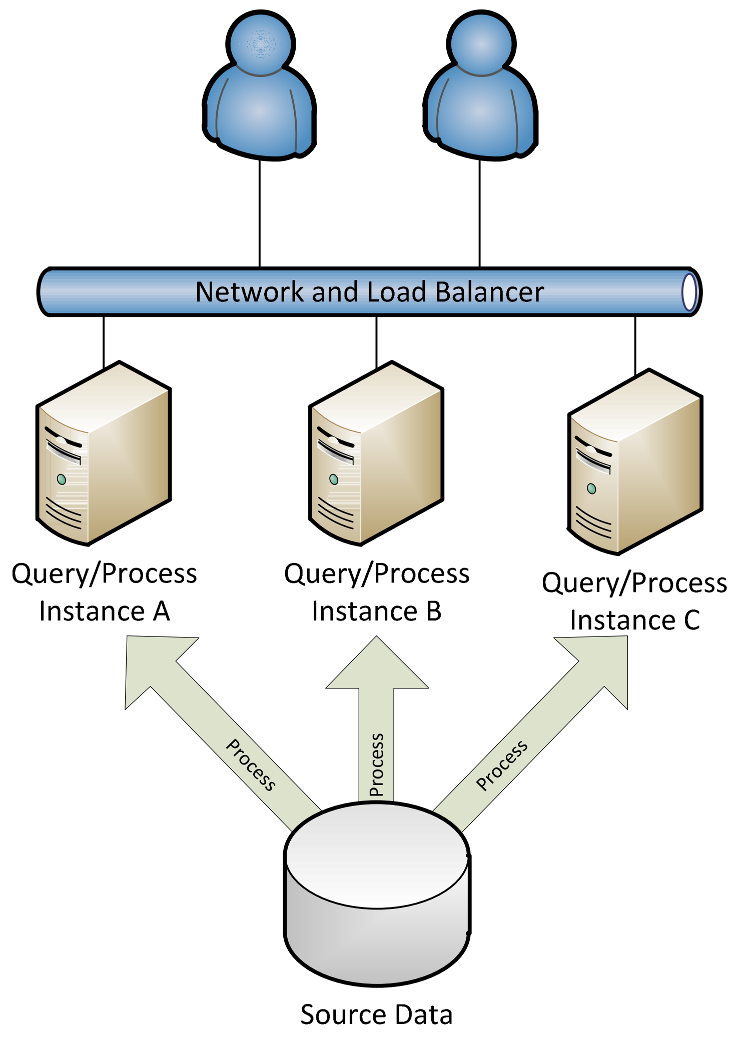


Figure 32 - Query and processing flipping

Because each server does it own processing, it is possible that some servers will have more recent data than others. This means that one user executing a query may get a later version of the data than another use executing the same query concurrently. However, for many scenarios where you are going near-real time, such a state of loose synchronization may is an acceptable tradeoff. If the tradeoff is not acceptable, you can work around it with careful load balancing – at the price of adding some extra latency to the updates.

In the preceding diagram, you can see that the source system receives more processing requests than in the dedicated processing architecture. You should scale the source accordingly and consider the network bandwidth required to read the source data more than once.

93

The query/processing flipping architecture also has a build in high availability solution. If one of the servers fails, the system can remain online but with additional load on the rest of the servers.

7.5.3 Query Load Balancing

In any scale-out architecture with more than one query server, you need to have a load balancing mechanism in place. The load balancing mechanism serves two purposes. First, it enables you to distribute queries equally across all query servers, achieving the scale-out effect. Second, it enables you to selectively take query servers offline, gracefully draining them, while they are being refreshed with new data.

When you use any load-balancing solution, be aware that the data caches on each of the servers in the load-balancing architecture will be in different states depending on the clients it is currently serving. This results in differences in response times for the same query, depending on where it executes.

There are several load balancing strategies to consider. These are treated in the following subsections. As you choose the load balancer, bear in mind the granularity of the load balancing and how this affects the process to query server switching. This is especially important if you use a dedicated processing architecture. For example, the Windows Network Load Balancing solution balances users between each Analysis Services Instance in the scale-out farm. This means that when you have to drain users from a query server and update the server with the latest version of the cube, the entire instance has to be drained. If you host more than one database per instance, this means that if one database is updated the other databases in the same instance must also be taken offline. Client load balancing and Analysis Services Load Balancer may be better solutions for you if you want to load-balance databases individually.

7.5.3.1 Client Load Balancing

In the client load balancing, each client knows which query server it will use. Implementing this strategy requires client-side code that can intelligently choose the right query server and then modify the connection string accordingly. Excel add-ins are an example of this type of client-side code. Note that you will have to develop your own load balancer to achieve this.

7.5.3.2 Hardware-Level Load Balancing

Using technologies such as load balancers from F5, it is possible to implement load balancing directly in the network layer of the architecture. This makes the load balancing transparent to both Analysis Services and the client application. If you choose to go down this route, make sure that the load-balance appliance enables you to affinitize client connections. When clients create session objects, state is stored on Analysis Services. If a later client request, relying on the same session state, is redirected to a different server, the OLE DB provider throws an error. However, even if you run affinity, you may still have to force clients off the server when processing needs to commit. For more information about the **ForceCommitTimeout** setting, see the locking section.

7.5.3.3 Windows Network Load Balancing

The Microsoft load-balancing solution is Network Load Balancing (NLB), which is a feature of the Windows Server operating system. With NLB, you can create an NLB cluster of Analysis Services servers running in multiple-host mode. When an NLB cluster of Analysis Services servers is running in multiple-host mode, incoming requests are load balanced among the Analysis Services servers.

7.5.3.4 Analysis Services Load Balancer

Analysis Services is used extensively inside Microsoft to serve our business users with data. As part of the initiative to scale out our Analysis Services farms a new load balancing solution was built. The advantages of this solution are that you can load balance on the database level and that you use a web API to control each database and the users connected to it. This customized Analysis Services load balancing solution also allows fine control over the load balancing algorithm used. Be aware that moving large datasets over the web API has a bandwidth overhead, depending on how much data is requested by user queries. Measure this bandwidth overhead as part of the cube test phase.

References:

- Analysis Services Load Balancing Solution - http://sqlcat.com/technicalnotes/archive/2010/02/08/aslb-setup.aspx

7.5.4 ROLAP Scale Out

With the query/processing architecture you can get the update latency of cubes down to around 30 minutes, depending on workload. But if you want to refresh data faster than that, you have to either go fully ROLAP or use a hybrid of one of the strategies discussed earlier and ROLAP partitions.

In a pure ROLAP setup, you only process the dimensions and redirect all measure group queries directly to a relational database. The following diagram illustrates this.

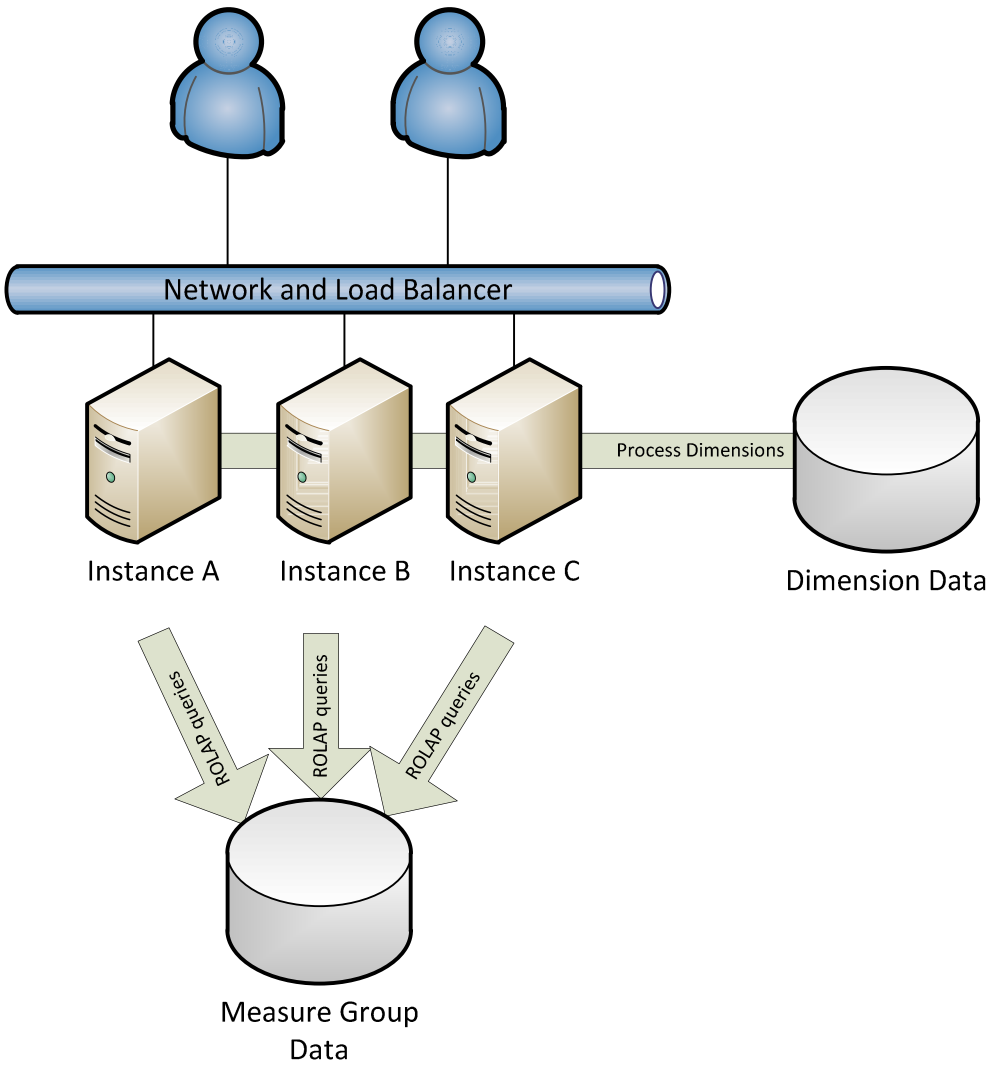


Figure 33 - ROLAP scale-out

In a ROLAP system like this, you have to consider the special requirements mentioned later in this document. You should also make sure that your relational data store is scaled to support multiple Analysis Services query servers.

96

An interesting hybrid between MOLAP and ROLAP can be built by combining the ROLAP scale-out with either the dedicated processing architecture or the query/processing flipping architecture. You can store data that changes less frequently in MOLAP partitions, which you either process or copy to the query servers. Data that changes frequently can be stored in ROLAP partitions, redirecting queries directly to the relational source. Such a setup can achieve very low update latencies, all the way down to a few seconds, while maintaining the benefits of MOLAP compression.

References:

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

8 Server Maintenance

When you move an Analysis Services instance to production, there are some regular maintenance tasks you should configure. This section describes those tasks.

8.1 Clearing Log Files and Dumps

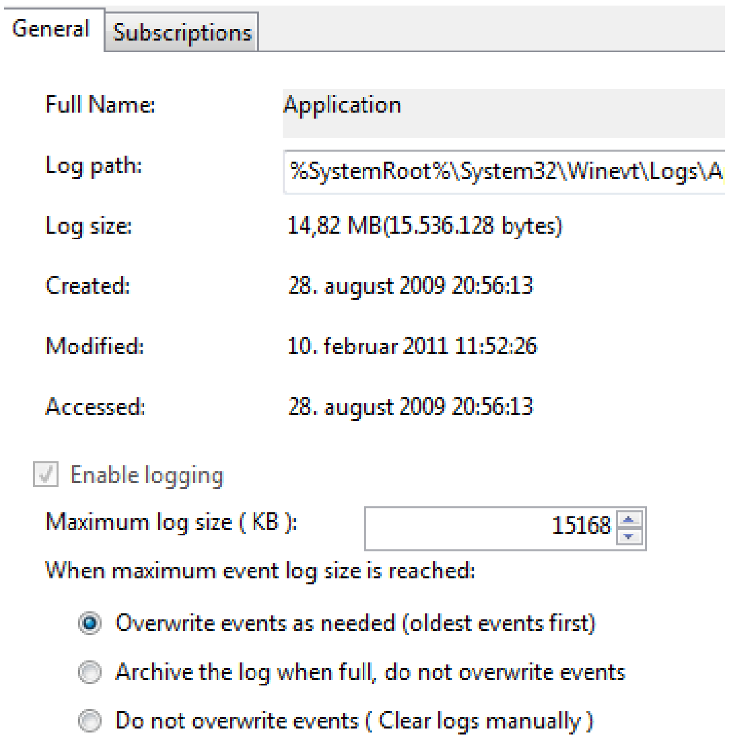
During server operations, Analysis Services generates a log file containing data about the operation. This log file is located in the folder described by the LogDir in Msmdsrv.ini – the default location being <Install dir>\OLAP\Log. This log grows extremely slowly and you should generally not need to clean it up. If you *do* need to reclaim the disk spaced used by the log file, you have to stop the service to delete it.

In the <Install Dir>\OLAP\log folder, you may also find files with extension *.mdmp. These are minidump files generated by the Analysis Services and are typically a few megabytes each. These files get generated when undetected deadlocks happen inside the process or when there is a problem with the service. The files are used by Microsoft Support to investigate stability issues and errors in the server. If you are experiencing any such behavior, you should collect these minidump files for use during case investigation. Periodically check for these files, and clean them up after any Microsoft Support case you have open is resolved.

8.2 Windows Event Log

Analysis Services uses the Windows event logs to report server errors, warnings, and information. The Application Log is used for most messages, but the System Log is also used for events that are related to Service Manager.

Depending on your server configuration, event logs may be configured to be cleaned manually. Make sure that this is a regular part of your maintenance. Alternatively, you can configure the event log to overwrite older events when the log is close to full. In both cases, make sure you have enough disk space to hold the full event log. The following illustration shows how to configure the event log to overwrite older events.



Full Name: Application

Log path: %SystemRoot%\System32\Winevt\Logs\A

Log size: 14,82 MB(15.536.128 bytes)

Created: 28. august 2009 20:56:13

Modified: 10. februar 2011 11:52:26

Accessed: 28. august 2009 20:56:13

✓ Enable logging

Maximum log size (KB): 15168

When maximum event log size is reached:

- [✓ Overwrite events as needed (oldest events first)]
- Archive the log when full, do not overwrite events
- Do not overwrite events (Clear logs manually)

Figure 34 - Recycling the event log

8.3 Defragmenting the File System

As described in the I/O section, there can be an advantage in defragmenting the files storing a cube, especially after a lot of changes to the partitions and dimensions. Running disk defrag will have a measurable impact on your disk subsystem performance, and depending on the hardware you run on, this may affect user response time. You can consider running defragmentation on the server during off-peak hours or in batch windows.

Note that the defrag utility retains the work done, even when it does not run to completion. This means that you can do partial defragmentation spread over time.

8.4 Running Disk Checks

Running disk checks (using ChkDsk.exe) on the Analysis Services volume gives you the confidence that no undetected I/O corruption has occurred. How often you want to do this depends on how often you expect the I/O subsystem to create such errors without detecting them – this varies by vendor and disk model.

Note that ChkDsk.exe can run for a very long time on a large disk volume, and that it will have an impact on your I/O speeds. Because of this, you may want to use a SAN snapshot of the LUN and run ChkDsk.exe on another machine that mounts the snapshot.

In both cases, you should be able to detect disk corruption without touching the live system. If you detect irreparable corruption, you should consider restoring the backups as per the previous section.

References:

- Chkdsk (http://technet.microsoft.com/en-us/library/bb491051.aspx)

9 Special Considerations

Using certain features of Analysis Services cubes can lead you down some design paths that require extra attention to succeed. This section describes these special scenarios and the considerations that apply when you encounter them.

9.1 ROLAP and Real Time

This section deals with issues that are specific to BI environments that do not have clearly defined batch windows for loading data. If you have a cube that updates data at the same time that ETL jobs are running on the source data or while users are connected, you need to pay special attention to certain configuration parameters.

As described in the Locking and Blocking section, processing operations generally take an instance-wide lock. This will typically prevent you from designing MOLAP systems that are updated more frequently than approximately every 30 minutes . Such a refresh frequency may not be enough, and if this is your scenario, ROLAP is the path forward, and you should be aware of the special considerations that apply. You should also be aware that a ROLAP partition can put significant load on the underlying relational source, which means you should involve the DBA function to understand this workload and tune for it.

9.1.1 Cache Coherency

By default, the storage engine of Analysis Services caches ROLAP subcubes in the same way it caches MOLAP subcubes. If the relational data changes frequently, this means that queries that touch ROLAP partitions may use a combination of the relational source and the storage engine cache to generate the response. If the relational source has changed since the cache entry was added to the storage engine, – this combination of source data can lead to results that are transactionally inconsistent from the perspective of the user because they represent an intermediate state of the system. There are of course ways to resolve this coherency issue, depending on your scenario.

Changing the connection string– It is possible to add the parameter **Real Time OLAP=true** to the connection string when the cube is accessed. Setting this parameter to **true** causes all relevant storage caches to be refreshed for every query run on that connection – including the caches generated by MOLAP queries. Note that this change can cause a significant impact on both query performance and concurrency. You should test it carefully. However, it gives you the most up-to-date results possible from the cube, because Analysis Services is essentially used as a thin MDX wrapper on top of the relational source in this mode.

Blowing away caches at regular intervals – you can either use an XMLA script to clear the cache or you can use query notifications (set in the ProActive caching properties of the partition). This allows you to clear the storage engine caches at regular intervals. Assuming you time this cache clearing with relational data loads, this gives users a consistent view that is updated every time the cache is cleared. Although this does not give you the same refresh frequency as the **Real Time OLAP=true** setting, it does have a smaller impact on user query performance and concurrency.

As you can see from these two options, going towards real-time cubes requires you to carefully consider tradeoffs between refresh frequencies and performance. Full coherency is possible but expensive. However, you can get a loose coherency that is much cheaper. Analysis Services supports both paradigms.

9.1.2 Manually Set Partition Slicers on ROLAP Partitions

When Analysis Services processes the index on a MOLAP partition, it collects data about the attributes in that partition. Assuming the data matches only one attribute value, an automatic slicer is set on the partition, eliminating it from scans that do not include that attribute value. For example, if you process a partition that has data from December 2008 only, Analysis Services detects this slicing and only accesses that partition when queries request data in that time range.

Because ROLAP data resides outside of Analysis Services, the automatic slicer functionality is not used. Unless you set the slicer manually (which can be done from both Visual Studio and SQL Server Management Studio) every query has to touch every ROLAP partition. It is therefore a good practice to always set slices on ROLAP partitions when they are available.

9.1.3 UDM Design

When you design for ROLAP access, it is generally a good idea to keep the UDM as simple as possible. The gives the relational engine the best possible conditions for optimizing for the query workload. The following table lists some optimizations you should consider when switching a cube to ROLAP mode.

Existing feature usage	ROLAP redesign
Reference dimensions	Switch to a pure star schema to eliminate unnecessary joins and provide relational engine with optimal conditions for query execution.
Parent/child dimensions	Normalize the parent-child dimension (for more information about how to do this, see the references for this section).
Many-to-many dimensions	Reduce intermediate table sizes using matrix compression.
Query binding of partitions	Switch to table binding. Consider binding to a view if queries are needed.
Query binding of dimensions	Implement the result of the query in the relational source instead.
Aggregates	Consider reducing the number of aggregates. Be aware of conditions and features in the relational engine so that you fully understand the tradeoffs. For example, in SQL Server 2008 R2, it is often a good idea to focus on aggregates that are targeted only at the leaf level and/or **[all]** level of attributes).
MDX calculations	Optimize carefully, and avoid cell-by-cell operations in large ROLAP partitions.
ROLAP dimensions	If at all possible, use MOLAP dimensions. MOLAP dimensions have much better performance than ROLAP dimensions and if you run regular **ProcessAdd** operations, you can keep them up to date at short refresh intervals.

You should work closely with the BI developers when troubleshooting ROLAP cubes. It is imperative to get the design right and follow the guidance in the Analysis Services Performance Guide.

References:

- Analysis Services Parent-Child Dimension Naturalizer on CodePlex – http://pcdimnaturalize.codeplex.com/
    - Also available from BIDS helper: http://bidshelper.codeplex.com
- Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques – http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3494E712-C90B-4A4E-AD45-01009C15C665&displaylang=en
- Analysis Services ROLAP for SQL Server Data Warehouses- http://sqlcat.com/whitepapers/archive/2010/08/23/analysis-services-rolap-for-sql-server-data-warehouses.aspx

9.1.4 Dimension Processing

Real-time dimension processing can present a special challenge. In a batch-style warehouse, you are in control of when inserts and updates happen – which means you can typically process the dimension after the relational source is done refreshing data. But if you are designing a cube on top of a real-time source, the relational data may change while you are processing a dimension. Dimension processing is by default executed as many concurrent SQL Server queries, as described in the Optimizing Processing section. Consider this sequence of events:

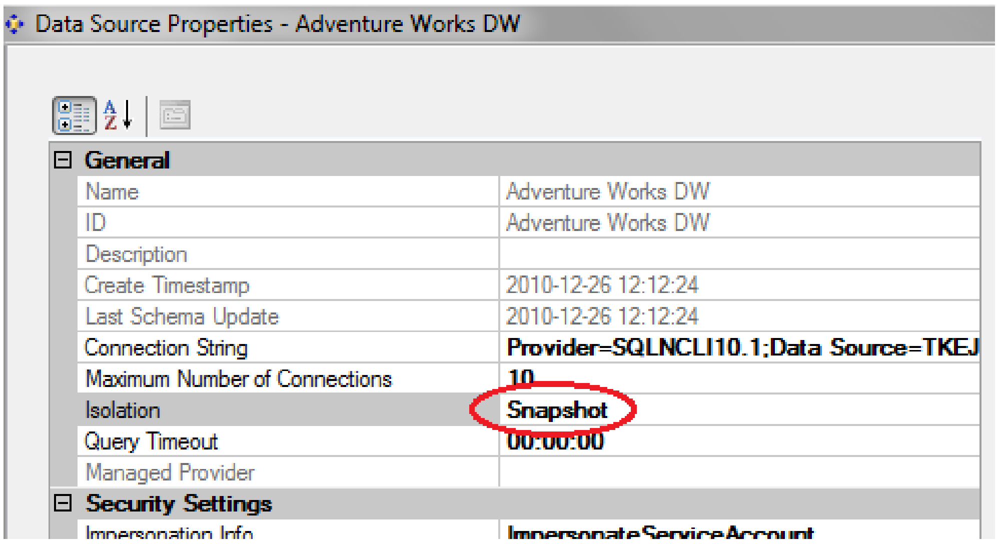
1. The customer dimension contains customers from all of the United States, but no customers from the rest of the world.
2. Dimension **ProcessAdd** starts.
3. Analysis Services sends query **SELECT DISTINCT Zip, City FROM Dimension** to the relational source. This query reads all current attribute **Zip** and **City** values.
4. The relational source inserts a new row, **City** = **Copenhagen**, in **Country = Denmark.**
5. Analysis Services, reading the next level of the hierarchy, sends the query **SELECT DISTINCT City, Country**.
6. The **City** member **Copenhagen** Is returned in the second query, but because it was not returned in the first, Analysis Services throws an error.

While this scenario may sound uncommon, we have seen it at several customers that design real-time systems. There are some ways to avoid these conditions.

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 will instead request the entire table with one query. This allows you to get a consistent view of the dimension, even under concurrency in the relational source. However, this setting has a drawback, namely that you will need to keep all the source dimension data in memory while the dimension is processing. If the server is under memory while this happens, paging can occur, which may cause a slowdown of the entire system.

MARS and Snapshot - If you are processing on top of a SQL Server data source, you can use Multiple Active Result Sets (MARS) and snapshot isolation to process the dimension and get a consistent view of the data even under updates.

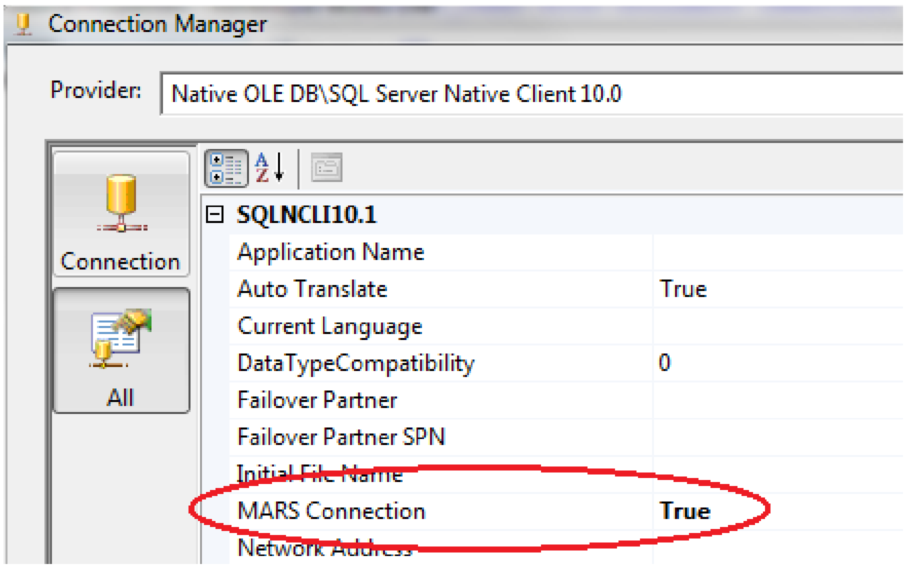
Configuring MARS and Snapshot processing requires a few configuration changes to the data source view and relational database. First, in the data source properties, change the data source view to use snapshot isolation.


Data Source Properties - Adventure Works DW

General	
Name	Adventure Works DW
ID	Adventure Works DW
Description	
Create Timestamp	2010-12-26 12:12:24
Last Schema Update	2010-12-26 12:12:24
Connection String	Provider=SQLNCLI10.1;Data Source=TKEJ
Maximum Number of Connections	10
Isolation	Snapshot
Query Timeout	00:00:00
Managed Provider	
Security Settings	
Impersonation Info	ImpersonateServiceAccount

Figure 35 - Setting the Data Source to Snapshot

Second, enable MARS in the connection string of the data source view.


Connection Manager

Provider: Native OLE DB\SQL Server Native Client 10.0

SQLNCLI10.1	
Application Name	
Auto Translate	True
Current Language	
DataTypeCompatibility	0
Failover Partner	
Failover Partner SPN	
Initial File Name	
MARS Connection	True
Network Address	

Figure 36 - Setting MARS in a DSV

And finally, enable either snapshot or read committed snapshot isolation in the SQL Server database.

```
ALTER DATABASE [Database]
SET READ_COMMITTED_SNAPSHOT ON
```

Processing now uses MARS, and snapshots generate a consistent view of the dimension during processing.

Understand that maintaining the snapshot during processing, as well as streaming the data through MARS, does not provide the same performance as the default processing option.

104

Maintaining consistency relationally – If you want to both maintain the processing speed and avoid memory consumption in the Analysis Services service, you have to design your data model to support real-time processing. There are several ways to do this, including the following:

- Add a timestamp to the rows in the dimension table that shows when they are inserted. During processing, only read the rows higher than a certain timestamp.
- Create a database snapshot of the relational source before processing.
- Manually create a copy of the source table before processing on top of the copy. The original can then be updated while the copy is being accessed by the cube.

References:

- Multiple Active Result Sets (MARS) - http://msdn.microsoft.com/en-us/library/ms345109(v=sql.90).aspx
- Using ByAttribute or ByTable Processing Group Property with Analysis Services 2005 - http://blogs.msdn.com/b/sqlcat/archive/2007/10/19/using-byattribute-or-bytable-processing-group-property-with-analysis-services-2005.aspx?wa=wsignin1.0

9.2 Distinct Count

Distinct count measures behave differently than other measures. Because data in a distinct count measure is not additive, a lot more information must be stored on disk. According to best practice, a measure group that has a distinct count measure should only have that single measure and no others. While additive measures compress very well, the same is not true for distinct count measures. This means that leaf-level data of the measure group takes up more disk space.

Targeting good aggregates for distinct count measures can also be difficult. Although aggregates for additive measures can be used by queries at higher granularities than the aggregate, the same does not apply for distinct count measures.

The combined effects of big measure groups and less useful aggregates means that queries that run against distinct count data often cause a significant amount of I/O operations and simply run longer than other queries. This is expected and part of the nature of distinct count data. However, there are some optimizations you can make that can greatly speed up both queries and processing of distinct count measures.

9.2.1 Partitioning for Distinct Count

Recall that for additive measures, it is generally recommended that you partition by time and sometimes by another dimension. This partition strategy is described in the Nonbreaking Cube Changes section. Distinct count measures are an exception to this rule of thumb. When it comes to distinct count, it is often a good idea to partition by the values of the distinct count measure itself. Analysis Services keeps track of the measure values in each partition, and assuming the intervals are not overlapping, it can benefit from some parallelism optimizations. The basic idea is to create partitions, typically one per CPU core in the machine, that each contain an equal-sized, nonoverlapping interval of measure values. The following picture illustrates these partitions.

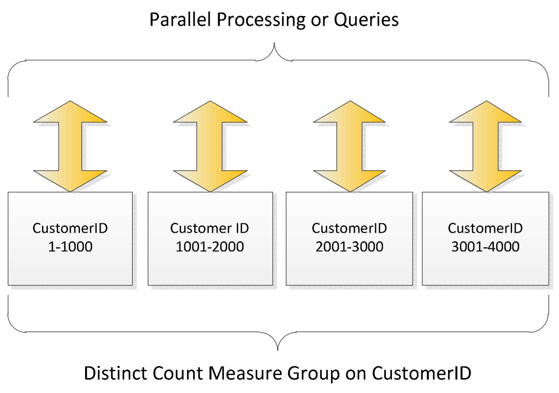


Figure 37 - Distinct Count Partitioning on a 4-Core Server

You can still apply a date based partition schema in addition to the distinct count partitioning. But if you do, make sure that queries do not cross the granularity level of this date range, or you lose part of the optimization. For example, if you do not have queries across years, you may benefit by partitioning by both year and the distinct count measure. Conversely, if you have queries that ask for data at the year level, you should not partition by month and the distinct count measure.

The white paper in the References section describes the partition strategy for distinct count measures in much more detail.

References:

- Analysis Services Distinct Count Optimization - http://www.microsoft.com/downloads/en/details.aspx?FamilyID=65df6ebf-9d1c-405f-84b1-08f492af52dd&displaylang=en
    - Describes the partition strategy that speeds up queries and processing for Distinct Count Measure groups

9.2.2 Optimizing Relational Indexes for Distinct Count

Analysis Services adds an ORDER BY clause to the distinct count processing queries have. For example, if you create a distinct count measure on **CustomerPONumber** in **FactInternetSales,** you get this query while processing.

```
SELECT … FROM FactInternetSales
ORDER BY [CustomerPONumber]
```

If your partition contains a large amount of rows, ordering the data can take a long time. Without supporting indexes, the query plan looks something like this.

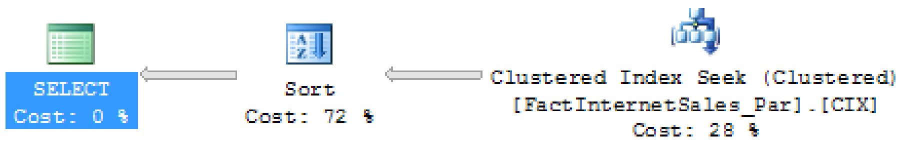

SELECT
Cost: 0 %
Sort
Cost: 72 %
Clustered Index Seek (Clustered)
[FactInternetSales_Par].[CIX]
Cost: 28 %

Figure 25 Relational sorting caused by distinct count

Notice the long time spent on the Sort operation? By creating a clustered index sorted on the distinct count column (in this case **CustomerPONumber**), you can eliminate this sort operation and get a query plan that looks like this.

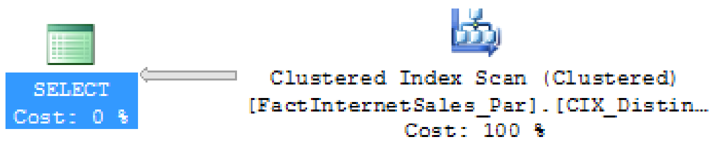

SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[FactInternetSales_Par].[CIX_Distin...
Cost: 100 %

Figure 26 Distinct count query supported by a good index

Of course, this index needs to be maintained. But having it in place speeds up the processing queries.

9.3 Many-to-Many Dimensions

Many-to-many dimensions are a powerful feature of Analysis Services cubes. They enable easy solutions for some complex, yet common scenarios in dimensional modeling. When cubes resolve many-to-many queries, the join with the intermediate table is done in Analysis Services memory and during query time. If the intermediate table is large, especially if it larger than memory, these queries can take a long time to respond. We recommend that you use many-to-many dimensions only if the intermediate table fits in memory. The links in the References section describe some techniques that enable you to reduce the memory consumption of the intermediate table.

References:

- Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques – http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3494E712-C90B-4A4E-AD45-01009C15C665&displaylang=en
- BIDS Helper has tools to estimate benefits of the Many-to-many compression described in the section:
    - http://bidshelper.codeplex.com/wikipage?title=Many-to-Many%20Matrix%20Compression

107

- Many-to-many project - http://www.sqlbi.com/manytomany.aspx
    - Design patterns for many-to-many dimensions

10 Conclusion

This document provides the means to monitor, capacity plan, and diagnose SQL Server 2008 Analysis Services operations issues.

For more information, see:

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

http://www.microsoft.com/sqlserver/: SQL Server website

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, Denny Lee at dennyl@microsoft.com, and John Sirmon at johnsi@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.

Analysis Services Performance Guide

SQL Server Technical Article

Writers: Richard Tkachuk and Thomas Kejser

Contributors and Technical Reviewers:

T.K. Anand
Marius Dumitru
Greg Galloway
Siva Harinath
Denny Lee
Edward Melomed
Akshai Mirchandani
Mosha Pasumansky
Carl Rabeler
Elizabeth Vitt
Sedat Yogurtcuoglu
Anne Zorner

Published: October 2008

Applies to: SQL Server 2008

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

Copyright

Contents

1 Introduction

Since Microsoft® SQL Server® Analysis Services query and processing performance tuning is a fairly broad subject, this white paper organizes performance tuning techniques into the following three segments.

Enhancing 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. In addition, you can improve query performance by optimizing the design of your dimension attributes, cubes, and Multidimensional Expressions (MDX) queries.

Enhancing 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 efficient dimension design, effective aggregations, partitions, and an economical processing strategy (for example, incremental vs. full refresh vs. proactive caching).

Tuning Server Resources – There are several engine settings that can be tuned that affect both querying and processing performance.

2 Understanding the 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. Figure 1 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.

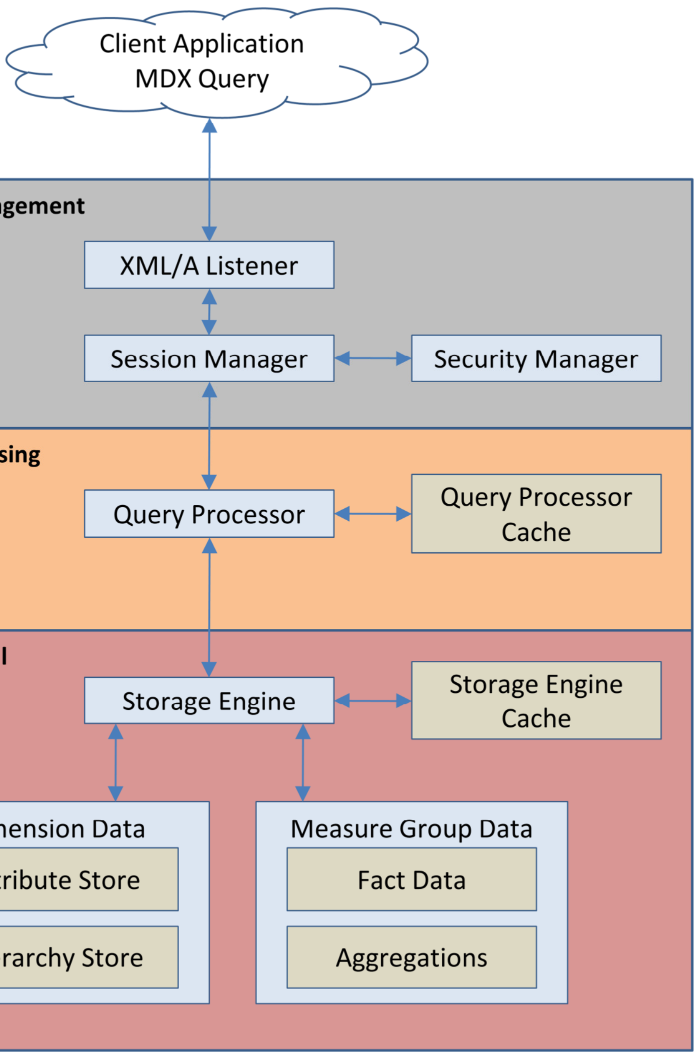


Figure 1 Analysis Services query processor architecture

2.1 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 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.

2.2 Job Architecture

Analysis Services uses a centralized job architecture to implement querying and processing operations. A *job* is a generic unit of processing or querying work. A job can have multiple levels of nested child jobs depending on the complexity of the request.

During processing operations, for example, a job is created for the object that you are processing, such as a dimension. A dimension job can then spawn several child jobs that process the attributes in the dimension. During querying, jobs are used to retrieve fact data and aggregations from the partition to satisfy query requests. For example, if you have a query that accesses multiple partitions, a parent or coordinator job is generated for the query itself along with one or more child jobs per partition.

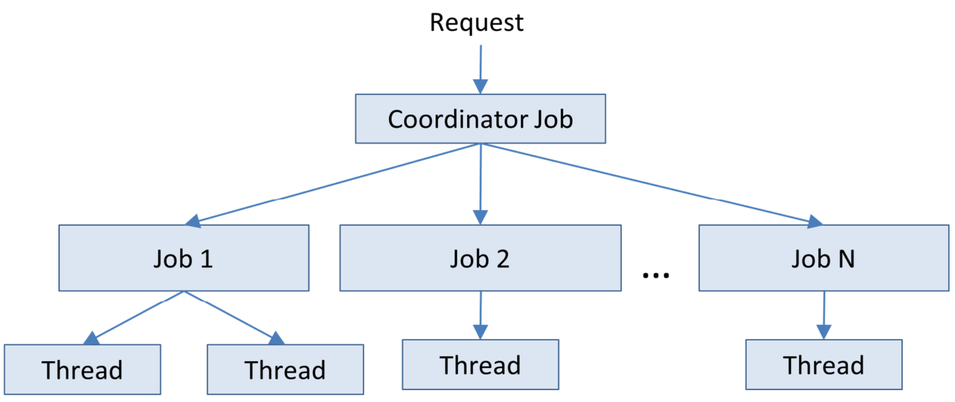


Figure 2 Job architecture

Generally speaking, executing more jobs in parallel has a positive impact on performance as long as you have enough processor resources to effectively handle the concurrent operations as well as sufficient memory and disk resources. The maximum number of jobs that can execute in parallel for the current operation operations (including both processing and querying) is determined by the **CoordinatorExecutionMode** property:

- A negative specifies the maximum number of parallel jobs that can start per core per operation.
- A value of zero indicates no limit.
- A positive value specifies an absolute number of parallel jobs that can start per server.

The default value for the **CoordinatorExecutionMode** is -4, which indicates that four jobs will be started in parallel per core. This value is sufficient for most server environments. If you want to increase the level of parallelism in your server, you can increase the value of this property either by increasing the number of jobs per processor or by setting the property to an absolute value.

While this globally increases the number of jobs that can execute in parallel, **CoordinatorExecutionMode** is not the only property that influences parallel operations. You must also consider the impact of other global settings such as the **MaxThreads** server properties that determine the maximum number of querying or processing threads that can execute in parallel (see Improving Multiple-User Performance for more information about thread settings). In addition, at a more granular level, for a given processing operation, you can specify the maximum number of processing tasks that can execute in parallel using the **MaxParallel** command. These settings are discussed in more detail in the sections that follow.

2.3 Query Processor

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, and 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. One or more members from a single level are also sometimes called a *single grain* or *single granularity*. 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 reusage of calculation results across users (with the same security roles). To optimize cache reusage, the query processor manages three cache layers that determine the level of cache reusability: global, session, and query.

2.3.1 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 any 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 any 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 any 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 reusage. 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 reusage across multiple sessions and users.

Query Context
Session Context
Global Context

Figure 3 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 can only 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, nothing will be cached globally or in the session cache. Rather, the calculations will be stored in the query cache. In addition, the following scenarios determine how the query processor caches calculation results:

- Use of cell security, any of the **UserName**, **StToSet**, or **LookupCube** functions in the MDX script or in the dimension or cell security definition disable the global cache (this means that just one expression using these functions 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.

Partial expressions (that is, a piece of a calculation that may be used more than once in the expression) and cell properties are not cached. Consider creating a separate calculated member to allow the query processor to cache results when first evaluated and reuse the results in subsequent references. For more information, see Cache Partial Expressions and Cell Properties).

2.3.2 Query Processor Internals

There are several changes to query processor intervals in SQL Server 2008 Analysis Services. In this section, these changes are discussed before specific optimization techniques are introduced.

2.3.2.1 Subspace Computation

The key idea behind subspace computation is best introduced by contrasting it with a naïve or cell-by-cell evaluation of a 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 would then proceed as represented in Figure 4.

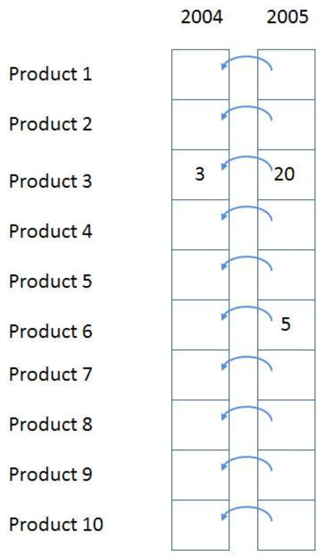


Figure 4 Cell-by -cell evaluation

The 10 cells for [2005, All Products] would each be evaluated in turn. For each, we would navigate to the previous year, obtain the sales value, and add it 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), then cells are calculated even though they are bound to return a null value. In the example above, 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. Firstly, we can consider that we work our way down an execution tree determining what spaces need to be filled. Given the query, we need to compute the space

[Product.*, 2005, RollingSum]

(where * means every member of the attribute hierarchy). Given the calculation, this means we must first compute the space

[Product.*, 2004, Sales]

followed by the space

[Product.*, 2005, Sales]

and then apply the + operator to those two spaces.

If Sales were itself covered by calculations, then the spaces necessary to calculate Sales would be determined and the tree would be expanded. In this case Sales is a base measure, so we simply obtain the storage engine data to fill the two spaces at the leaves, and then work up the tree, applying the operator 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 result in Figure 5.

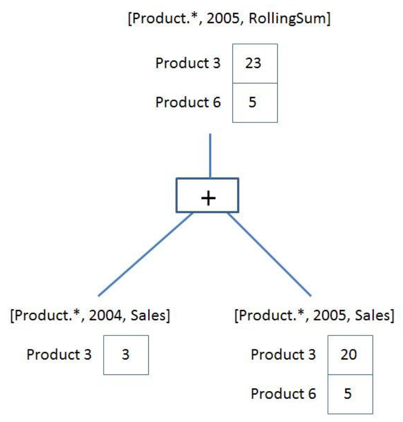


Figure 5 Execution plan

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 we are only ever operating on data that could contribute to the result. There is no notion of the theoretical space over which we must perform the calculation.

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 and the engine falls back to cell-by-cell mode. But even when evaluating an expression in cell-by-cell mode, the engine can return to subspace mode.

2.3.2.2 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.

2.3.2.3 Expression Sparsity

An expression's *sparsity* refers to the number of cells with nonnull values compared to the total number of cells. 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 if an expression is dense or sparse? Consider a simple noncalculated measure – is it dense or sparse? In OLAP, base fact measures are sparse. This means that 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. There are some other simple rules for popular expressions below.

Expression	Sparse/dense
Regular measure	Sparse
Constant Value	Dense (excluding constant null values, true/false values)
Scalar expression; e.g., count, .properties	Dense
<exp1>+<exp2> **<exp1>-<exp2>**	Sparse if both exp1 and exp1 are sparse; otherwise dense
<exp1>*<exp2>	Sparse if either exp1 or exp1 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**)

2.3.2.4 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 (blank or empty in the Microsoft Excel® spreadsheet software) 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 need only compute 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 will evaluate 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.

2.3.2.5 Varying Attributes

Cell values mostly depend on attribute coordinates. But some calculations do not depend on every attribute. For example, the expression

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

depends only on the Customer attribute in the customer dimension. When this expression is evaluated over a subspace involving other attributes, any attributes the expression doesn't depend on can be eliminated, the expression 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.

2.3.2.6 Query Processor Internals Wrap-up

Query plans, expression sparsity, default values and varying attributes are core internal concepts behind the query processor behavior – we'll be returning to these concepts as we discuss optimizing query performance.

2.4 Data Retrieval

When you query a cube, the query processor decomposes 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 partitions.

Each partition is divided in groups of 64K records called a segment.

A coordinator job is created for each subcube request. It creates as many jobs as there are partitions. (This is true where the query requests data within the partition slice. For more information, see Partition Slicing.). Each of these jobs:

- Queues up another job for the next segment (if the current segment is not the last).
- Uses the bitmap indexes to determine if there is data in the segment corresponding to the subcube request.
- Scans the segment, if there is data.

For a single partition, the job structure looks like this after each segment job is queued up.

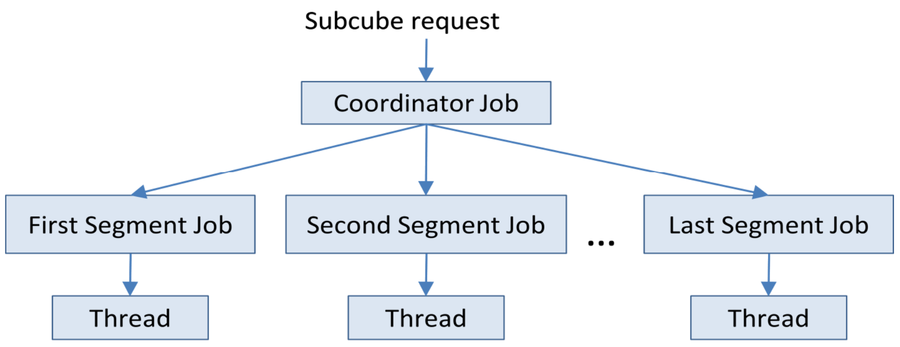


Figure 6 Partition scan job structure

Immediately after a segment job is queued, it kicks off other segment jobs, and there are as many jobs as there are segments. Should the indexes reveal that no data corresponding to the subcube is contained in the segment, the job ends.

3 Enhancing Query Performance

To improve query performance, one must first 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.

Much time can be expended pursuing dead ends – it is important to first understand the nature of the problem before applying specific techniques.

3.1 Baselining Query Speeds

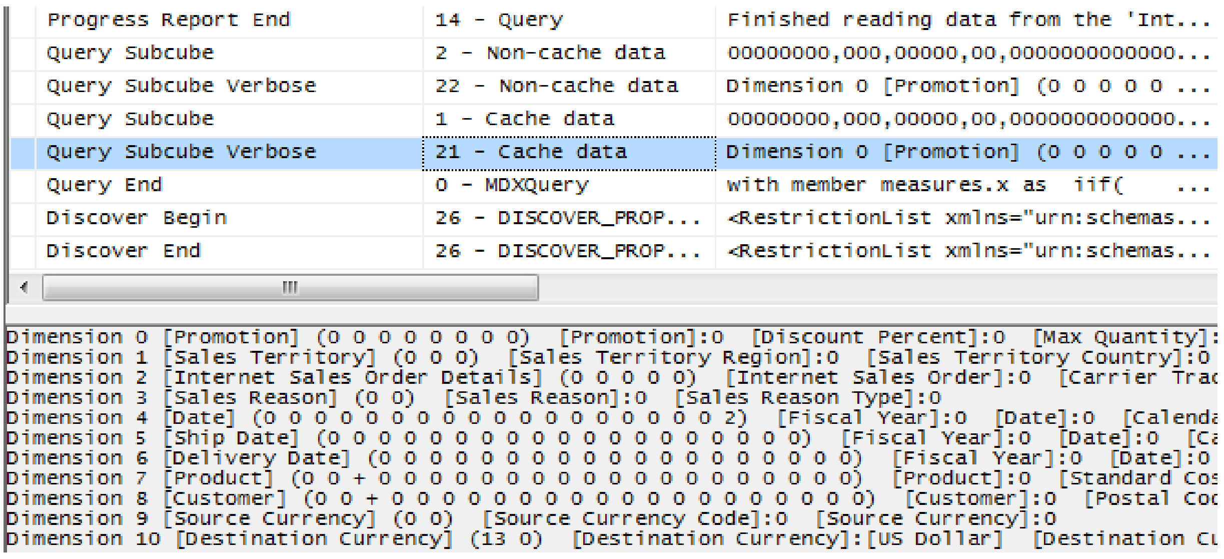
Before beginning optimization, you need a reproducible baseline. Take a measurement on a *cold* (that is, unpopulated) storage engine and query processor caches and a *warm* operating system cache. To do this, execute the query, empty the formula and storage engine caches, and then initialize the calc script by executing a query that returns and caches nothing, as follows.

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

Execute the query a second time. When the query is executed the second time, use SQL Server Profiler to take a trace with the additional events enabled:

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

The trace contains important information.



```
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 7 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.
- <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).

Save the trace – it contains important timing information, and it indicates events described later.

To empty the storage and query processor caches, use the ClearCache command.

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

```
  <Object>

    <DatabaseID>Adventure Works DW</DatabaseID>

  </Object>

</ClearCache>
```

The operating system file cache is affected by everything else on the hardware – try to reduce or eliminate other activity. This can be particularly difficult if the cube is stored on a storage area network (SAN) used by other applications.

SQL Server Management Studio reveals query times, 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 can rival the time it took the server to generate it. A SQL Server Profiler trace provides not only insight into where the time is being spent but also the precise engine duration.

3.2 Diagnosing Query Performance Issues

When performance is not what you expect, the source can be in a number of areas. Figure 8 illustrates how the source of the problem can be diagnosed.

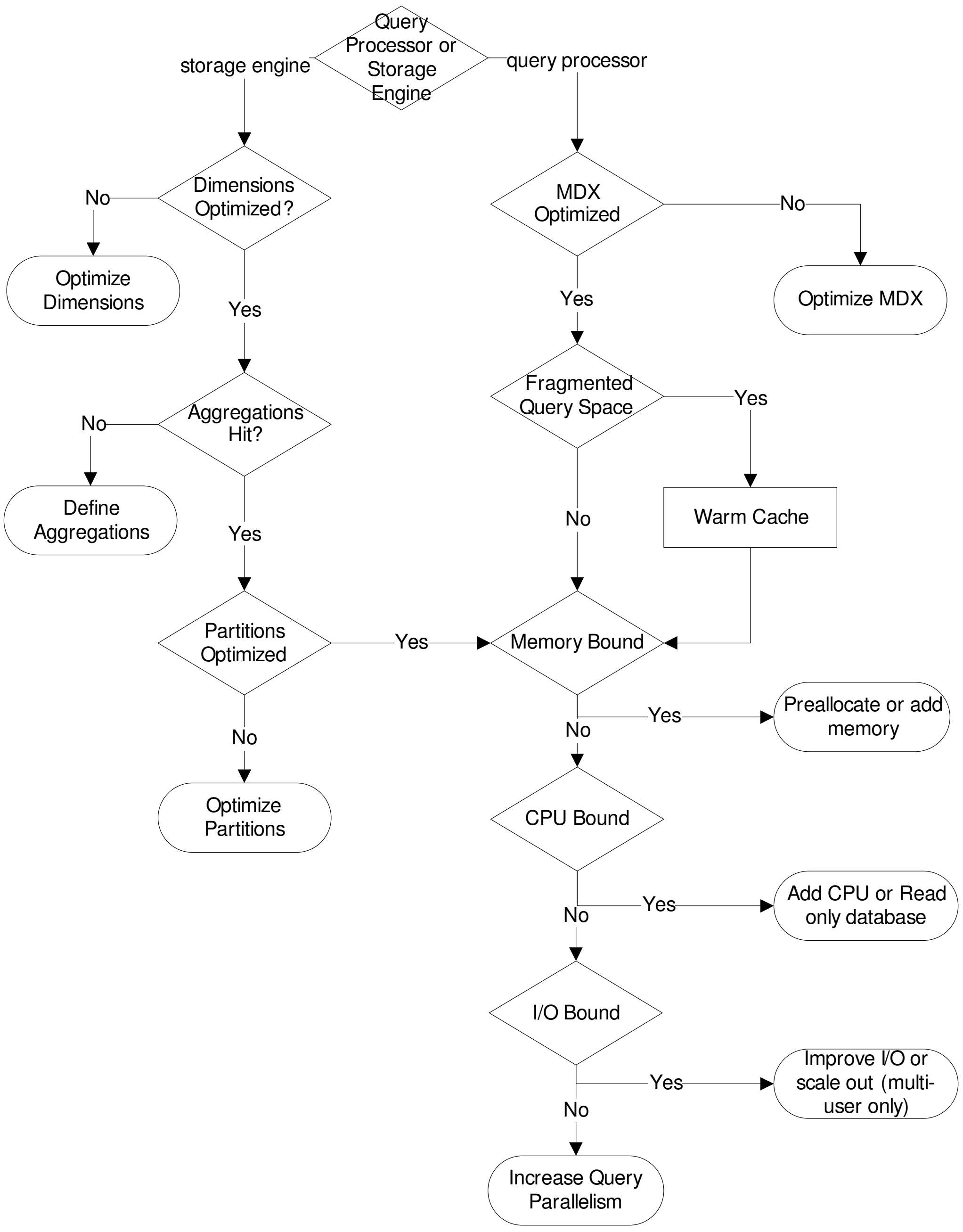
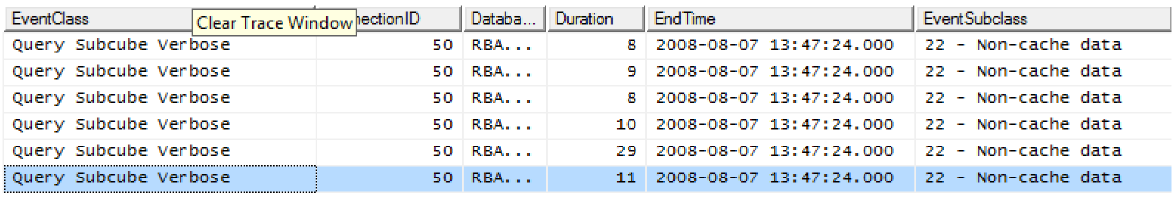


Figure 8 Query performance tuning flow chart

The first step is to determine whether the problem lies in the query processor or storage engine. To determine the amount of time the engine is scanning data, use SQL Server Profiler to create a trace. 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 Figure 9.



EventClass	ectionID	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 9 Determining time spent scanning partitions

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. Consider optimizing dimension design, designing aggregations, or using partitions to improve query performance. If the majority of time is not spent in the storage engine but in the query processor, focus on optimizing MDX.

The problem can involve both the formula and storage engines. Fragmented query space can be diagnosed with profiler where many query subcube events are generated. Each request may not take long, but the sum of them may. If this is the case, consider warming the cache to reduce the I/O thrashing that this may engender.

Some multiple-user performance issues can be resolved by addressing single-user queries, but certainly not all. Some configuration settings custom to multiple-user environments are described in the section Improving Multiple-User Performance.

If the cube is optimized, CPU and memory resource utilization can be optimized. How to increase the number of threads for single and multiple-user scenarios is described in the section Increasing Query Parallelism. The same technique can be used for reserving memory for improving query and processing performance and is included in the processing section entitled Using PreAllocate.

Performance can generally improved by scaling up with CPU, memory, or I/O. Such recommendations are out of the scope of this document. There are other techniques available to scale out with clusters or read-only databases. These are only described briefly in later sections to determine whether such a path might be the right direction to take.

Monitoring memory usage is discussed in a separate section, Monitoring and Adjusting Server Memory.

3.3 Optimizing Dimensions

A well-tuned dimension design is one of the most critical success factors of a high-performing Analysis Services solution. One of the first steps to improve cube performance is to step through the dimensions and study attribute relationships. The two most important techniques that you can use to optimize your dimension design for query performance are:

- Identifying attribute relationships.
- Using user hierarchies effectively.

3.3.1 Identifying Attribute Relationships

Attribute relationships define functional 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. More specifically, given an attribute relationship City → State, if the current city is Seattle, then we know the State must be Washington.

Oftentimes 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.

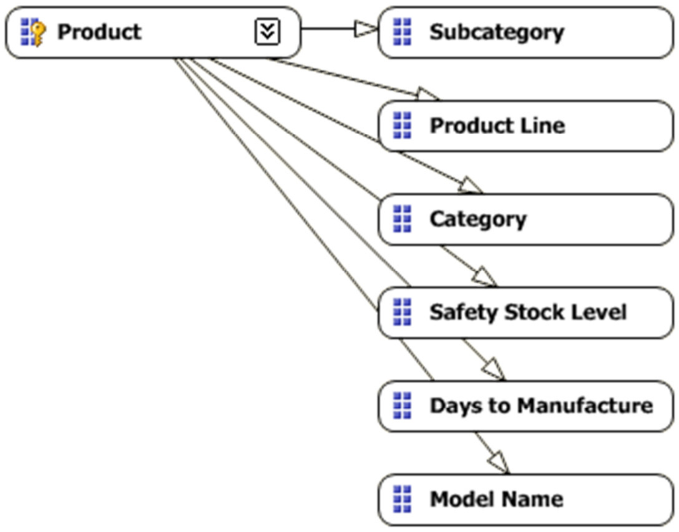


Figure 10 Bushy attribute relationships

You can optimize performance by defining 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). After redefining the relationships in the attribute relationship editor, we have the following.

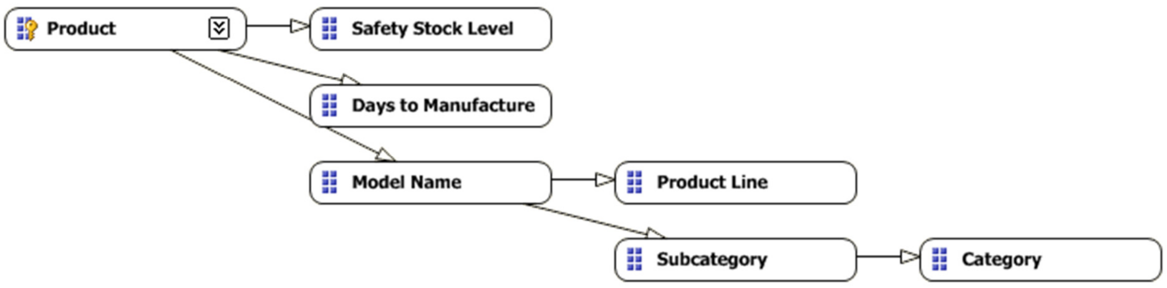


Figure 11 Redefined attribute relationships

22

Attribute relationships help performance in two significant ways:

- Indexes are built and cross products need not go through the key attribute.
- Aggregations built on attributes can be reused for queries on related attributes.

Consider the cross-product between Subcategory and Category in the two figures above. 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 nontrivially sized dimensions, this can take time. If the attribute relationship is defined, then the Analysis Services engine knows beforehand which category each subcategory belongs to via indexes built at process time.

When defining the attribute relationship, consider the relationship type as 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. (As a side note, defining an aggregation to be flexible or rigid has no impact on query performance.)

3.3.2 Using Hierarchies Effectively

Attributes only exposed in attribute hierarchies are not automatically considered for aggregation by the Aggregation Design Wizard. 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 detailed information on 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.

Analysis Services enables you to build two types of user hierarchies: natural and unnatural hierarchies, each with 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.

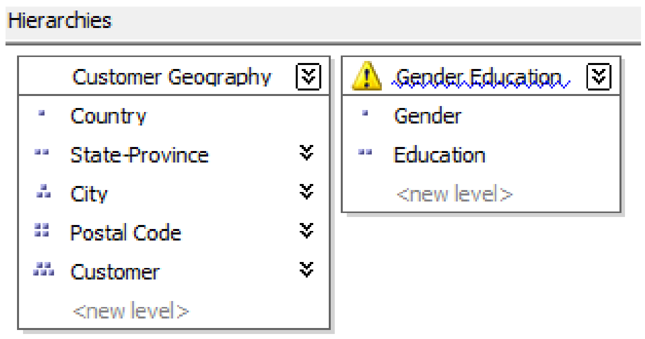


Figure 12 Natural and unnatural hierarchies

From a performance perspective, natural hierarchies behave very differently than unnatural hierarchies. 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 participating in the hierarchy.

3.4 Maximizing the Value of Aggregations

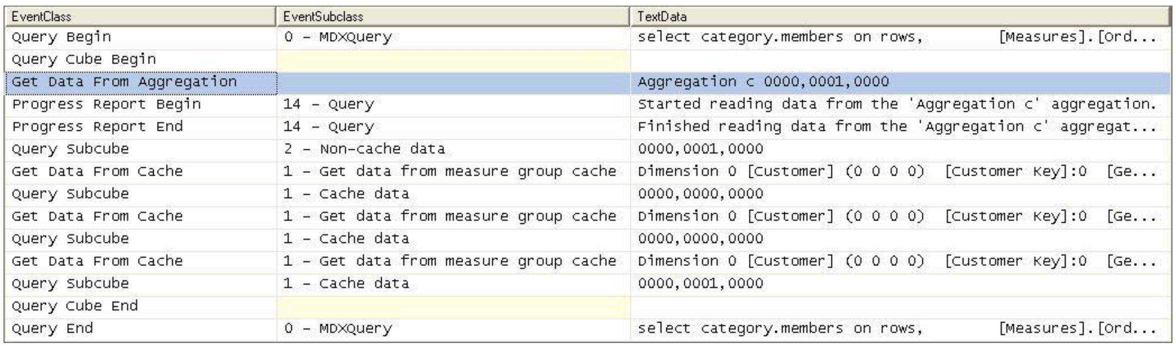
An *aggregation* is a precalculated summary of data that Analysis Services uses to enhance query performance.

Designing aggregations is the process of selecting the most effective aggregations for your querying workload. 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 on how you can improve query performance using multiple partitions, see Using Partitions to Enhance Query Performance.

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...

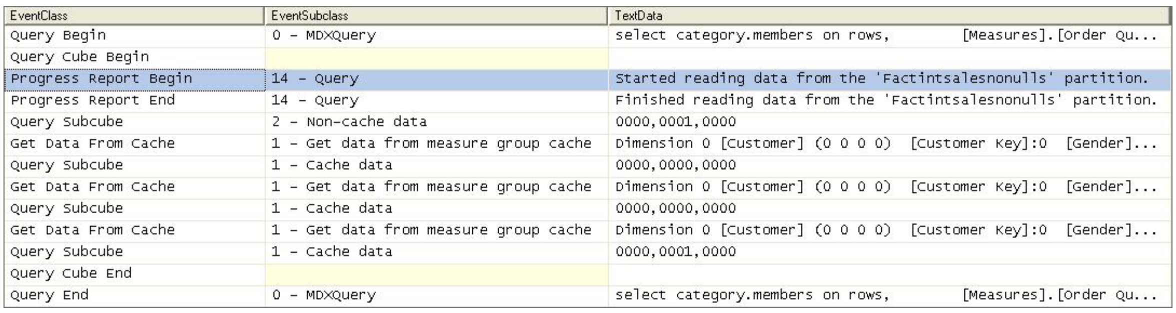
Figure 13 Scenario 1: SQL Server Profiler trace for cube with an aggregation hit

Figure 13 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 13 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.

Figure 14 displays a SQL Server Profiler trace for the same query against the same cube, but this time, the cube has no aggregations that can satisfy the query request.



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 14 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 cube. To find the order of dimensions in the cube, use one of the following two techniques. With the cube opened in SQL Server Business Intelligence Development Studio, you can review the order of dimensions in a cube on the **Cube Structure** tab. The order of dimensions in the cube is displayed in the Dimensions pane. As an alternative, you can 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 following 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.

3.4.3 Building Aggregations

To help Analysis Services successfully apply the aggregation design algorithm, you can perform the following optimization techniques to influence and enhance the aggregation design. (The sections that follow describe each of these techniques in more detail).

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.

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.

Usage-based optimization – To focus aggregations on particular usage pattern, execute the queries and launch the Usage-Based Optimization Wizard.

3.4.3.1 Suggesting Aggregation Candidates

When Analysis Services designs aggregations, the aggregation design algorithm does not automatically consider every attribute for aggregation. 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**.

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**.

3.4.3.2 Influencing Aggregation Candidates

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.

The aggregation design algorithm evaluates the cost/benefit of each aggregation based member counts and fact table record counts. Ensuring that your metadata is up-to-date can improve the effectiveness of your aggregation design. You can define the fact table source record count in the **EstimatedRows** property of each measure group, and you can define attribute member count in the **EstimatedCount** property of each attribute.

3.4.3.3 Usage-Based Optimization
The Usage-Based Optimization Wizard reviews the queries in the query log (something you must set up beforehand) and designs aggregations that cover 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. 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.

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.

3.4.3.4 Aggregations and Parent-Child Hierarchies
In parent-child hierarchies, aggregations are created only for the key attribute and the top attribute, i.e., the All attribute unless it is disabled. 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 will degrade linearly with the number of members.) Additionally, limit the number of parent-child hierarchies in your cube.

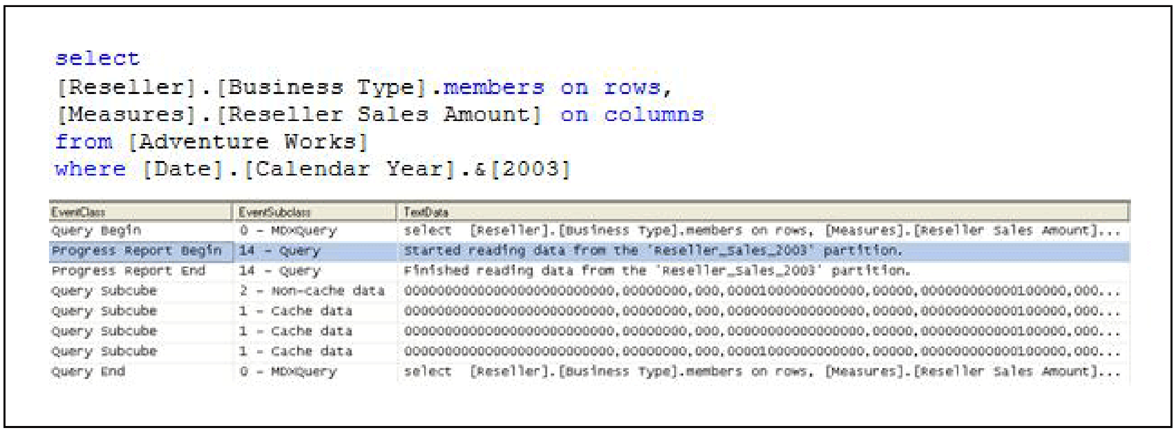
If you are in a design scenario with a large parent-child hierarchy, consider altering the source schema to reorganize part or all of the hierarchy into a regular hierarchy with a fixed number of levels. After the data has been reorganized into the user hierarchy, you can use the **Hide Member If** property of each level to hide the redundant or missing members.

3.5 Using Partitions to Enhance Query Performance
Partitions separate measure group data into physical 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 balance the benefits and costs between query and processing performance before you finalize your partitioning strategy.

3.5.1 Introduction
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 slicing: Partitions not containing data in the subcube are not queried at all, thus avoiding the cost of reading the index (or scanning the table in ROLAP mode, where there are no MOLAP indexes).

- 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	00000000000000000000000000,00000000,000,00001000000000000,00000,0000000000000100000,000...
Query Subcube	1 - Cache data	00000000000000000000000000,00000000,000,00000000000000000,00000,0000000000000100000,000...
Query Subcube	1 - Cache data	00000000000000000000000000,00000000,000,00000000000000000,00000,0000000000000100000,000...
Query Subcube	1 - Cache data	00000000000000000000000000,00000000,000,00001000000000000,00000,0000000000000100000,000...
Query End	0 - MDXQuery	select [Reseller].[Business Type].members on rows, [Measures].[Reseller Sales Amount]...

Figure 15 Intelligent querying by partitions

Figure 15 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.

3.5.2 Partition Slicing

Partitions are bound to a source table, view, or source query. For MOLAP partitions, during processing 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. Knowing this information, the storage engine can optimize which partitions it scans during querying by only choosing those partitions that are relevant to the query. For ROLAP and proactive caching partitions, you must manually identify the slice in the properties of the partition.

The Min and Max DataIDs can specify a single member or a range. 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. If they are out of order in the dimension table, then the internal sequence of DataIDs can differ from attribute keys. This can cause unnecessary partition reads. For this reason, there may be a benefit to define 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.

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 accurately reflect the specific data in the partition and not the data across the entire measure group.

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

3.5.3 Aggregation Considerations for Multiple Partitions

When you define your partitions, remember that they do not have to contain uniform datasets nor 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. The value of using heterogeneous partitions with different levels of detail is that you can more easily manage the loading of new data without disturbing existing partitions (more on this in the processing section) and you can design aggregations for groups of partitions that share the same level of detail.

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.

The newest aggregation design can also be copied to a *base partition*. This base partition holds no data—it serves only to hold the current aggregation design. When it is time to add a new partition (for example, at the start of a new month), the base partition can be cloned to a new partition. When the slice is set on the new partition, it is ready to take data as the current partition. Following an initial full process, the current partition can be incrementally updated for the remainder of the period.

3.5.4 Distinct Count Partition Design

Distinct count partitions are special. 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 to 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 between 20 to 300 percent!

For more information about optimizations for distinct count, see the white paper "Analysis Services Distinct Count", which is available from the following link:
http://www.microsoft.com/downloads/details.aspx?FamilyID=65df6ebf-9d1c-405f-84b1-08f492af52dd&displaylang=en

3.5.5 Partition Sizing

For nondistinct count measure groups, tests with partition sizes in the range of 200 megabytes (MB) to up to 3 gigabytes (GB) indicate that partition size alone does not have a substantial impact on query speeds. 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
- Support for different aggregation designs

3.6 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.

3.6.1 Diagnosing the Problem

Diagnosing the problem may be straightforward if a simple query calls out a specific calculation (in which case 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 simplest expression possible that continues to reproduce the performance issue. With some client applications, the query itself can be problem, should it demand large data volumes, push down to unnecessarily low granularities (bypassing aggregations), or contain query calculations that bypass the global and session query processor caches.

If the issue is confirmed to be in the cube itself, remove or comment out all calculations from the cube. This includes the following:

- Custom member formulas
- Unary operators
- MDX scripts (except the calculate statement, which should be left intact)

Rerun the query. It might have to be altered to account for missing members. Bring back the calculations until the problem is reproduced.

3.6.2 Calculation Best Practices

This section contains a series of best practices to apply to get the best query performance from your cube.

3.6.2.1 Cell-by-Cell Mode vs. Subspace Mode

Almost always, performance obtained by using subspace mode is superior to that obtained by using cell-by-cell mode. For more information, including the list of functions supported in subspace mode, see “Performance Improvements for MDX in SQL Server 2008 Analysis Services” in SQL Server Books Online, available at the following link:

http://msdn.microsoft.com/en-us/library/bb934106(SQL.100).aspx

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`
	
	but almost the same query ,where we replace the set 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

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	Popular Microsoft Visual Basic® for Applications (VBA) and Excel functions are natively supported in MDX. User-defined stored procedures are evaluated in cell-by-cell mode.
LookupCube	Linked measure groups are often a viable alternative.

3.6.2.2 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 its 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 – that 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. 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.

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.

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, then 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, so it is sparse. Therefore the default value of the condition is true and therefore 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]]
)
```

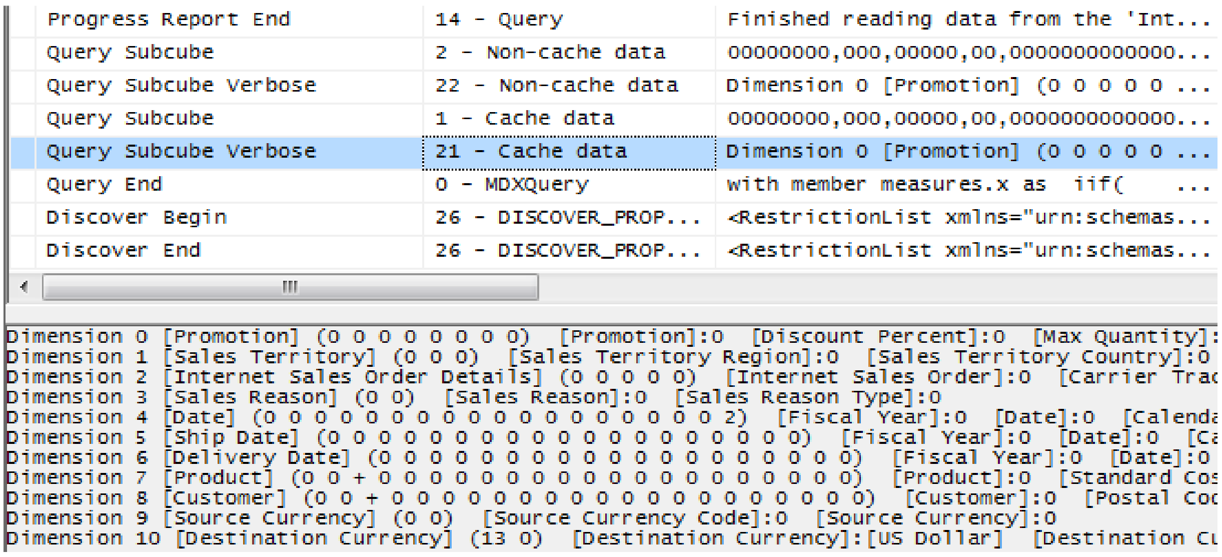
When would you want to override the default behavior? Here are the most common scenarios where you might want to change the default behavior:

- 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 simple expression below taking 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 16.



```
Progress Report End       14 - Query              Finished reading data from the 'Int...
Query Subcube             2 - Non-cache data      00000000,000,00000,00,0000000000000...
Query Subcube Verbose     22 - Non-cache data     Dimension 0 [Promotion] (0 0 0 0 0 ...
Query Subcube             1 - Cache data          00000000,000,00000,00,0000000000000...
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
```

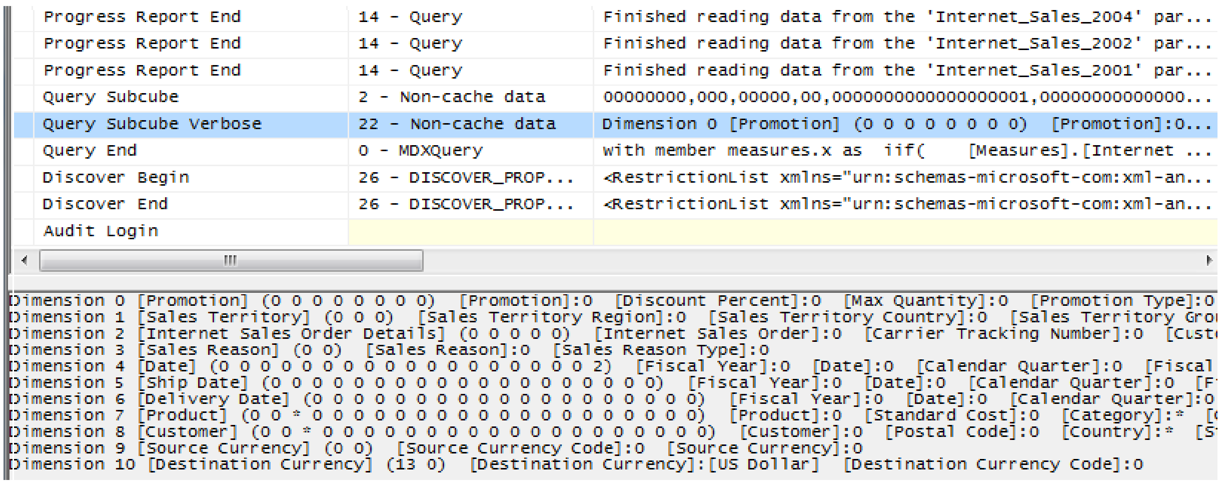
Figure 16 Default IIf query trace

36

Note the subcube definition for the Product and Customer dimension (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 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 17 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.

37

3.6.2.3 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/<complex 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, instead of

```
create member currentcube.measures.[Value] as <exp> , backgroundColor=<complex
expression>;
```

do this

```
create member currentcube.measures.MyCellPrope as <complex expression> , visible=0;

create member currentcube.measures.[Value] as <exp> , backgroundColor=<complex
expression>;
```

3.6.2.4 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, distributive unary operators (that is, one whose member order does not matter, such as +, -, and ~) are generally twice as fast as trying to mimic their capabilities with assignments.

There are rare exceptions. For example, one 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.

3.6.2.5 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]
```

This takes 2:29 on a laptop – quite a while. 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). We 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, root([Customer]))

select measures.AvgSalesWithOverWrite on 0,

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

from [Adventure Works]
```

This takes less than a second – a substantial change in performance.

3.6.2.6 Avoid Assigning Nonnull Values to Otherwise Nonempty Cells

The Analysis Services engine is very efficient at eliminating empty rows. Adding calculations with nonempty values replacing null values does not allow AS to eliminate these rows. For example, this query replaces null values with the dash, and the **non empty** key word 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
```

non empty operates on cell values and not on formatted values. In rare cases we 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 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 Reporting Services supports the fourth argument in the format_string. For more information about using the format_string calculation property, see "FORMAT_STRING Contents (MDX)" in SQL Server Books Online, which is available at the following link:
http://msdn.microsoft.com/en-us/library/ms146084.aspx

3.6.2.7 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 if 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;
```

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

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

When writing expressions as products of two other expressions, place the sparser one on the left-hand side.

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 exchanging the order of the expressions in the product of the cell calculation. 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])*
```

42

```
[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])
```

3.6.2.9 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
```

Do this.

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

3.6.2.10 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, instead of this

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

<set expression> )>0
```

Do this

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

3.6.2.11 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 when you're encountering performance issues, do consider how the calculation can be resolved in the source database or 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 leaves and aggregate with calculations. Computing this result in the source database or in the DSV will result in superior performance.

3.6.2.12 NON_EMTPY_BEHAVIOR

In some situations, it is expensive to compute the result of an expression, even though we know it will be null beforehand based on the value of some indicator tuple. The NONEMPTY_BEHAVIOR property was sometimes helpful for these kinds of calculations. When this property evaluated to null, the expression was guaranteed to be null and (most of the time) vice versa.

This property oftentimes resulted in substantial performance improvements in past releases. In 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.

44

```
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 will more often than not ignore this property and deduce 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 (i.e., <e2> is much sparser than <e1>), improved performance might be achieved 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.

3.7 Cache Warming

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. This process is called *cache warming*.

The two mechanisms are similar although the create cache statement has the advantage of not returning cell values and generally executes faster because the query processor is bypassed.

Discovering what needs to be cached can be difficult. One approach is to run a trace during query execution and examining 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, you can create an application that executes a set of generalized queries to simulate typical user activity in order to expedite the process of populating the cache. For example, if you determine that users are querying by month and by product, you can create a set of queries that request data by product and by month. If you run this query whenever you start Analysis Services or process the measure group or one of its partitions, this will preload the query results cache with data used to resolve these queries before users submit these types of query. This technique substantially improves Analysis Services response times to user queries that were anticipated by this set of queries.

To determine a set of generalized queries, you can use the Analysis Services query log to determine the dimension attributes typically queried by user queries. You can use an application, such as a Microsoft Excel macro, or a script file to warm the cache whenever you have performed an operation that flushes the query results cache. For example, this application could be executed automatically at the end of the cube processing step.

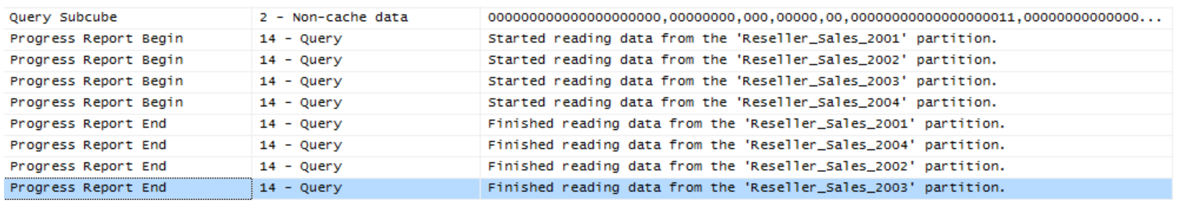
When testing the effectiveness of different cache-warming queries, you should empty the query results cache between each test to ensure the validity of your testing.

Note that the cached results can be pushed out by other query results. It may be necessary to refresh the cache results according to some schedule. Also, limit cache warming to what can fit in memory leaving enough for other queries to be cached.

Aggressive Data Scanning

It is possible that 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   00000000000000000000,00000000,000,00000,00,000000000000000000011,0000000000000...
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 18 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.

46

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, contact Microsoft Customer Service and Support for further advice.

3.8 Improving Multiple-User Performance

In many cases, poor multiple-user performance can be traced to poor single-user performance. But this isn't always true. In some cases, Analysis Services does not exploit all the resources on the computer when scaling up the number of users. There are a few options to improve performance.

3.8.1 Increasing Query Parallelism

During querying, to manage client connections, Analysis Services uses a listener thread to broker requests and create new server connections as needed. To satisfy query requests, the listener thread manages worker threads in the querying thread pool and the processing thread pool, assigns worker threads to specific requests, initiates new worker threads if there are not enough active worker threads in a given pool, and terminates idle worker threads as needed.

To satisfy a query request, the thread pools are used as follows:

- Worker threads from the query pool check the data and calculation caches respectively for any data and/or calculations pertinent to a client request.
- If necessary, worker threads from the processing pool are allocated to retrieve data from disk.
- After data is retrieved, worker threads from the querying pool store the results in the query cache to resolve future queries.
- Worker threads from the querying pool perform necessary calculations and use a calculation cache to store calculation results.

The more threads that are available to satisfy queries, the more queries that you can execute in parallel. This is especially important in scenarios where you have a large number of users issuing queries.

Threadpool\Query\MaxThreads determines the maximum number of worker threads maintained in the querying thread pool. The default value of this property is either 10 or 2x the number of cores (this is different from SQL Server 2005, so check this value for upgraded instances). Increasing **Threadpool\Query\MaxThreads** will not significantly increase the performance of a given query. Rather, the benefit of increasing this property is that you can increase the number of queries that can be serviced concurrently.

Because querying also involves retrieving data from partitions, you must also consider the maximum threads available in the processing pool as specified by the **Threadpool\Process\MaxThreads** property. By default, this property has a value of 64 in SQL Server 2005. This changed to either 64 or 10 times the number of cores, whichever is greater, in SQL Server 2008 (and this is the currently recommended value). As you consider the scenarios for changing the **Threadpool\Process\MaxThreads** property, remember that changing this setting impacts the processing thread pool for both querying and processing. For more information about how this property specifically impacts processing operations, see Adjusting Thread Settings.

While modifying the **Threadpool\Process\MaxThreads** and **Threadpool\Query\MaxThreads** properties can increase parallelism during querying, you must also take into account the additional impact of **CoordinatorExecutionMode**. Consider the following example. If you have a four-processor server and you accept the default **CoordinatorExecutionMode** setting of -4, a total of 16 jobs can be executed at one time across all server operations. So if ten queries are executed in parallel and require a total of 20 jobs, only 16 jobs can launch at a given time (assuming that no processing operations are being performed at that time). When the job threshold has been reached, subsequent jobs wait in a queue until a new job can be created. Therefore, if the number of jobs is the bottleneck to the operation, increasing the thread counts may not necessarily improve overall performance.

In practical terms, the balancing of jobs and threads can be tricky. If you want to increase parallelism, it is important to assess your greatest bottleneck to parallelism, such as the number of concurrent jobs and/or the number of concurrent threads, or both. To help you determine this, it is helpful to monitor the following performance counters:

- **Threads\Query pool job queue length**—The number of jobs in the queue of the query thread pool. A nonzero value means that the number of query jobs has exceeded the number of available query threads. In this scenario, you may consider increasing the number of query threads. However, if CPU utilization is already very high, increasing the number of threads will only add to context switches and degrade performance.
- **Threads\Query pool busy threads**—The number of busy threads in the query thread pool.
- **Threads\Query pool idle threads**—The number of idle threads in the query thread pool.

More information is available at
http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/ssasqptb.mspx.

3.8.2 Memory Heap Type

Multiple-user throughput can be improved by using the Windows heap instead of the Analysis Services heap. This increase in throughput can come at a small but measurable cost to single-user queries, but multiple-user throughput has been measured to improve by up to 100% (that is, twice as many queries being managed in the same amount of time). The significant benefits of multiple-user throughput generally outweigh the single-user performance cost. To use the NTLFH heap manager instead of the OLAP heap manager, change the following parameters.

Setting	Default	Multiple-user
MemoryHeapType	1	2
HeapTypeForObjects	1	0

3.8.3 Blocking Long-Running Queries

In multiple-user scenarios, long-running queries can starve other queries – even shorter-running queries – by consuming all available threads, and they can block execution of other queries until the longer-running query has completed. You can reduce the aggressiveness of how each coordinator job consumes queries by changing how each segment job is queued up. As discussed in Data Retrieval, each segment job is immediately queued up before the prior segment job begins scanning data. You can change this behavior to serialize segment jobs by using the following settings.

Setting	**Default**	**Multiple-user nonblocking settings**
CoordinatorQueryBalancingFactor	-1	1
CoordinatorQueryBoostPriorityLevel	3	0

But be careful – although these settings reduce or stop the blocking of shorter-running queries by longer-running ones, it lessens the overall throughput. If you change these settings and still see blocking queries, contact Microsoft Customer Service and Support.

3.8.4 Network Load Balancing and Read-Only Databases

Although they are beyond the scope of this document, fundamental design changes can be brought to bear to address query issues and are briefly described here.

3.8.4.1 Network Load Balancing

If your performance bottleneck is processor utilization on a single system as a result of a multiple-user query workload, you can increase query performance by using a cluster of Analysis Services servers to service query requests. Requests can be load balanced across two Analysis Services servers, or across a larger number of Analysis Services servers to support a large number of concurrent users (this is called a *server farm*). Load-balancing clusters generally scale linearly. Both Microsoft and third-party vendors provide cluster solutions. The Microsoft load-balancing solution is Network Load Balancing (NLB), which is a feature of the Windows Server® operating system. With NLB, you can create an NLB cluster of Analysis Services servers running in multiple host mode. When an NLB cluster of Analysis Services servers is running in multiple host mode, incoming requests are load balanced among the Analysis Services servers. When you use a load-balancing cluster, be aware that the data caches on each of the servers in the load-balancing cluster will be different, resulting in differences in query response times from query to query by the same client.

A load-balancing cluster can also be used to ensure availability in the event that a single Analysis Services server fails. An additional option for increasing performance with a load-balancing cluster is to distribute processing tasks to an offline server. When new data has been processed on the offline server, you can update the Analysis Services servers in the load-balancing cluster by using Analysis Services database synchronization.

If your users submit a lot of queries that require fact data scans, a load-balancing cluster may be a good solution. For example, queries that may require a large number of fact data scans include wide queries (such as top count or medians), and random queries against very complex cubes where the probability of hitting an aggregation is very low.

However, a load-balancing cluster is generally not needed to increase Analysis Services performance if aggregations are being used to resolve most queries. In other words, concentrate on good aggregation and partitioning design first. In addition, a load-balancing cluster does not solve your performance problem if processing is the bottleneck or if you are trying to improve an individual query from a single user. Note that one restriction to using a load-balancing cluster is the inability to use writeback, because there is no single server to which to write back the data.

3.8.4.2 Read-Only Databases

New in SQL Server 2008, a database can be marked as read-only and used by multiple instances of Analysis Services; that is, multiple instances of Analysis Services can share a single data directory (typically located on a SAN). This option should be considered if multiple-user workload is light on storage engine requirements but heavy on query processor. While Analysis Services supports multiple instances pointing to the same data folder, it is up to the application to manage user sessions across these instances.

4 Understanding and Measuring Processing

In the following sections we will provide guidance on tuning processing of cubes. Processing is the operation that loads data from one or more data sources into one or more Analysis Services objects. While OLAP systems are not generally judged by how fast they process data, processing performance impacts how quickly new data is available for querying. While every application has different data refresh requirements, ranging from monthly updates to near real-time data refreshes, 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.

4.1 Processing Job Overview

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.

Additional background information on processing can be found in the technical note Analysis Services 2005 Processing Architecture.

4.2 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.

4.2.1 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**
- **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 if you are getting closer to your performance goal. Also note the total time used by processing. The use and interpretation of the individual counters will be explained in the sections below.

4.2.2 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 for help on tuning the database. In the following we will assume that you use SQL Server as the relational foundation for Analysis Services.

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.

4.3 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. Because dimensions are processed before partitions, you can easily measure how much time you spend on dimensions.

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.

5 Enhancing Dimension Processing Performance

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:

- Optimizing SQL source queries.
- Reducing attribute overhead.

This section also includes information about dimension processing architecture.

5.1 Understanding 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 Figure 19.

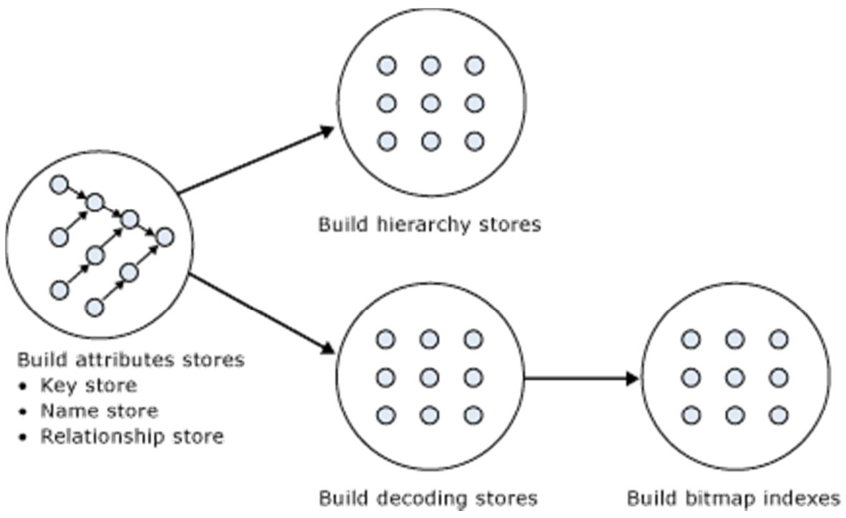


Figure 19 Dimension processing jobs

53

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.

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.

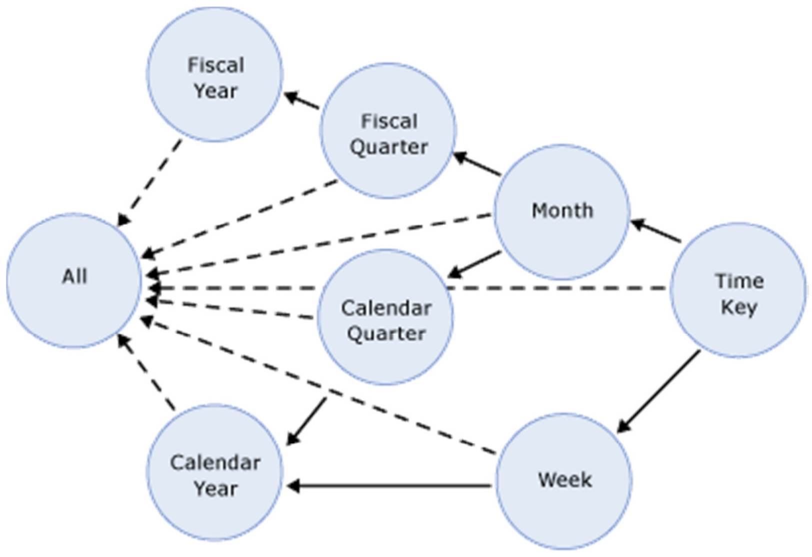


Figure 20 Execution tree example

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 primary key attribute always being processed last, since 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, since it has the most members and all other jobs (hierarchy, decoding, bitmap indexes) are waiting for it to complete. Attribute relationships will lower the memory requirement during processing. When an attribute is processed, all dependent attributes must be kept in memory. If you had no attribute relationships, all attributes would have to be kept in memory while the key attribute was being processed. This may cause out-of-memory conditions.

For more information about the importance of using cascading attribute relationships, see Identifying Attribute Relationships.

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.

5.2 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:

- **ProcessFull**
- **ProcessData**
- **ProcessIndexes**
- **ProcessUpdate**
- **ProcessAdd**

A **ProcessFull** command discards all storage contents of the dimension and rebuilds 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.

ProcessData discards all storage contents of the dimension and rebuilds only the attribute and hierarchy stores and also clears partitions. **ProcessData** is the first component executed by a **ProcessFull** operation.

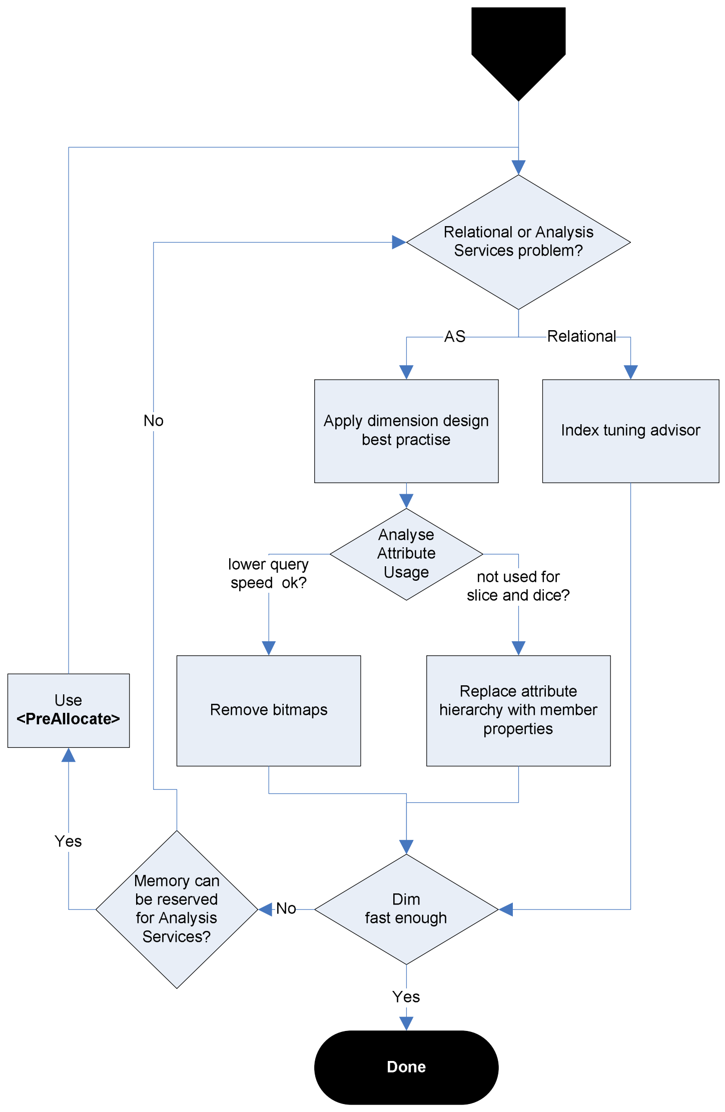
ProcessIndexes requires that a dimension already has attribute and hierarchy stores built it preserves the data in these stores and then rebuilds the bitmap indexes. **ProcessIndexes** is the second component of the **ProcessFull** operation.

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

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).

Note: ProcessAdd is only available as an XMLA command.

5.3 Dimension Processing Tuning Flow Chart



Relational or Analysis Services problem?
AS
Relational
Apply dimension design best practise
Index tuning advisor
No
Analyse Attribute Usage
lower query speed ok?
not used for slice and dice?
Use <PreAllocate>
Remove bitmaps
Replace attribute hierarchy with member properties
Yes
Memory can be reserved for Analysis Services?
No
Dim fast enough
Yes
Done

For information about SQL Server wait statistics and how to track them, see sys.dm_os_wait_stats in SQL Server Books Online.

5.4 Dimension Processing Performance Best Practices

There are some general, good design practices that are simple to implement and which provide some quick wins for performance of dimension. You should seek to incorporate these in your best practices when designing dimensions.

In SQL Server 2008 Analysis Services, the Analysis Management Objects (AMO) warnings are provided by Business Intelligence Development Studio to assist you with designing best practices.

5.4.1 Use SQL Views to Implement Query Binding for Dimensions

While query binding for dimensions does not exist in SQL Server 2008 Analysis Services, 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.

It is generally a good idea to build your Unified Dimensional Model (UDM) on top of database views. Not only can you apply relational tuning, you can also use the NOLOCK hint in the view definition. This hint removes locking overhead from the database, which can benefit performance even further.

Views provide easy of debugging. You can issue SQL queries directly on views to compare the relational data with the cube. Hence, views provide a good way to encapsulate business logic that you would normally implement as query binding in the UDM. While the UDM syntax is similar to the SQL view syntax, you cannot issue SQL statements against the UDM.

5.4.2 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 key 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.

5.4.3 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.

5.4.4 Use 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 **ValueColumn** and **NameColumn** eliminates the need for extraneous attributes. This reduces the total number of attributes in your design, making it more efficient.

Analysis Services provides the ability to source the **KeyColumns**, **ValueColumn**, and **NameColumn** properties from different source columns. This is useful when you have a single entity like a product that is identified by two different attributes: a surrogate key and a descriptive product name. When users want to slice data by products, they may find that the surrogate key lacks business relevance and will choose to use the product name instead.

It is a best practice to assign a numeric source field to the **KeyColumns** property rather than a string property. Not only does this reduce processing time, in also reduces the size of the dimension. This is especially true for attributes that have a large number of members, i.e., greater than one million members.

5.4.5 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, since 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. To avoid spending time building unnecessary bitmap indexes for the social security number attribute, it is possible to disable its bitmap indexes by setting the **AttributeHierarchyOptimizedState** property to **Not Optimized**.

5.4.6 Turn Off the Attribute Hierarchy and Use Member Properties

As an alternative to attribute hierarchies, member properties provide a different mechanism to expose dimension information. For a given attribute, member properties are automatically created for every 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 Business Intelligence Design Studio in the same manner as attribute hierarchies and user hierarchies. To query a member property, you must query the properties of the attribute that contains the 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, querying member properties can be slower than querying attribute hierarchies, because member properties are not indexed and do not participate in aggregations. The actual impact to query performance depends on how you are going to 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.

5.5 Tuning the Relational Dimension Processing Query

Unlike fact partitions, which only send one query to the server, dimension process operations will send multiple queries. Dimensions tend to be small, complex tables with very few changes, compared to facts. Tables with such characteristics 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.

The easiest way to tune the relational queries used for dimension processing is to 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.

6 Enhancing Partition Processing Performance

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, using multiple partitions, tuning I/O, optimizing networking speeds, and tuning thread and concurrency settings.

6.1 Understanding the Partition Processing Architecture

During partition processing, source data is extracted and stored on disk using the series of jobs displayed In Figure 21.

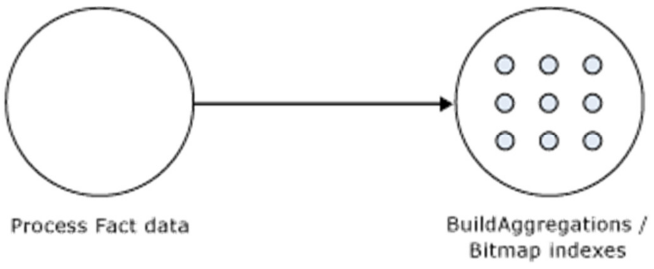


Figure 21 Partition processing jobs

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. While 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.

6.2 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**
- **ProcessClearIndex**

ProcessFull discards the storage contents of the partition and rebuilds them. Behind the scenes, a**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.

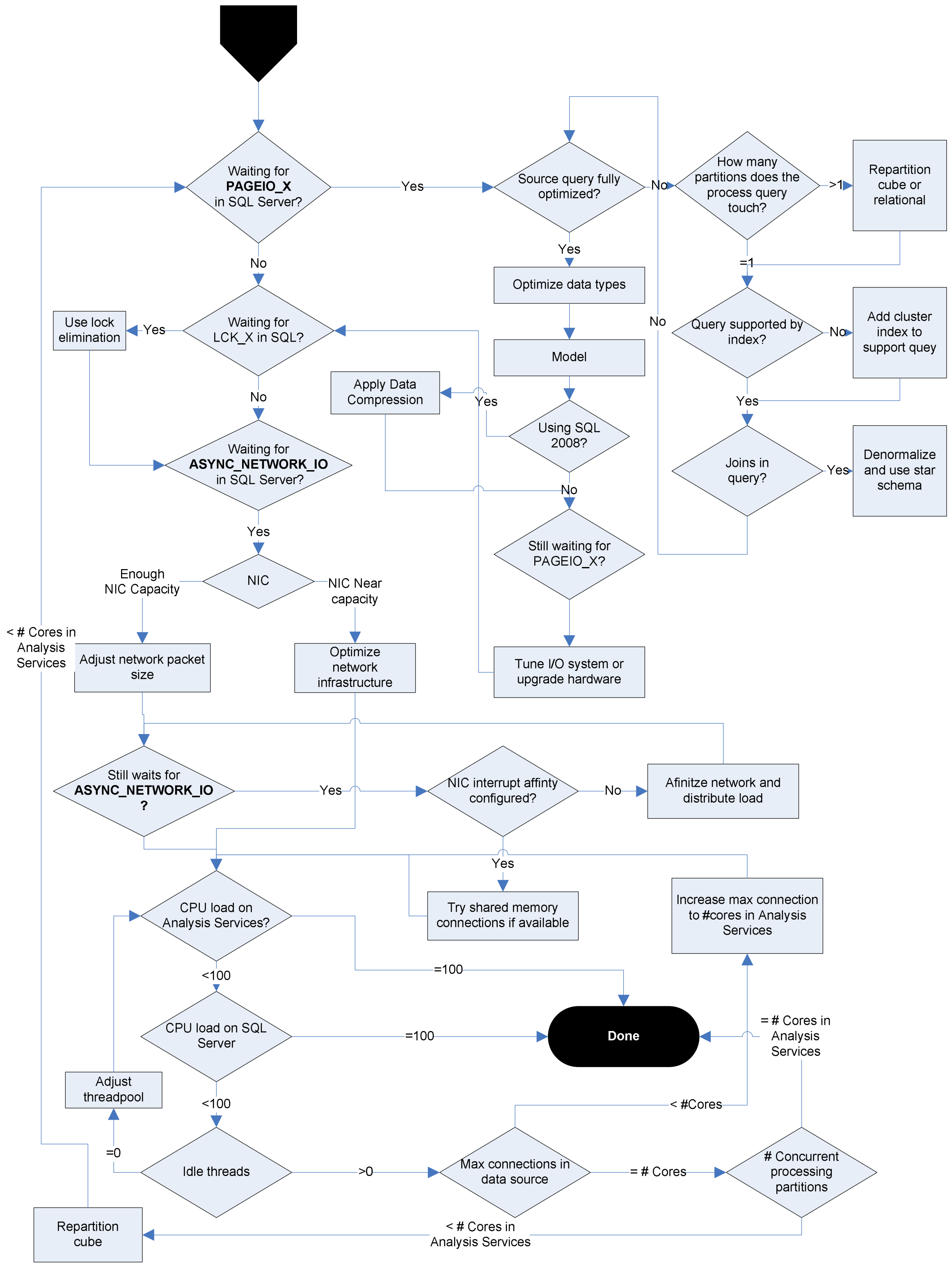
ProcessAdd internally creates a temporary partition, processes it with the target fact data, and then merges it with the existing partition. Note that **ProcessAdd** is the name of the XMLA command, in Business Intelligence Development Studio and SQL Server Management Studio this is exposed as **ProcessIncremental.**

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.

6.3 Partition Processing Tuning Flow Chart

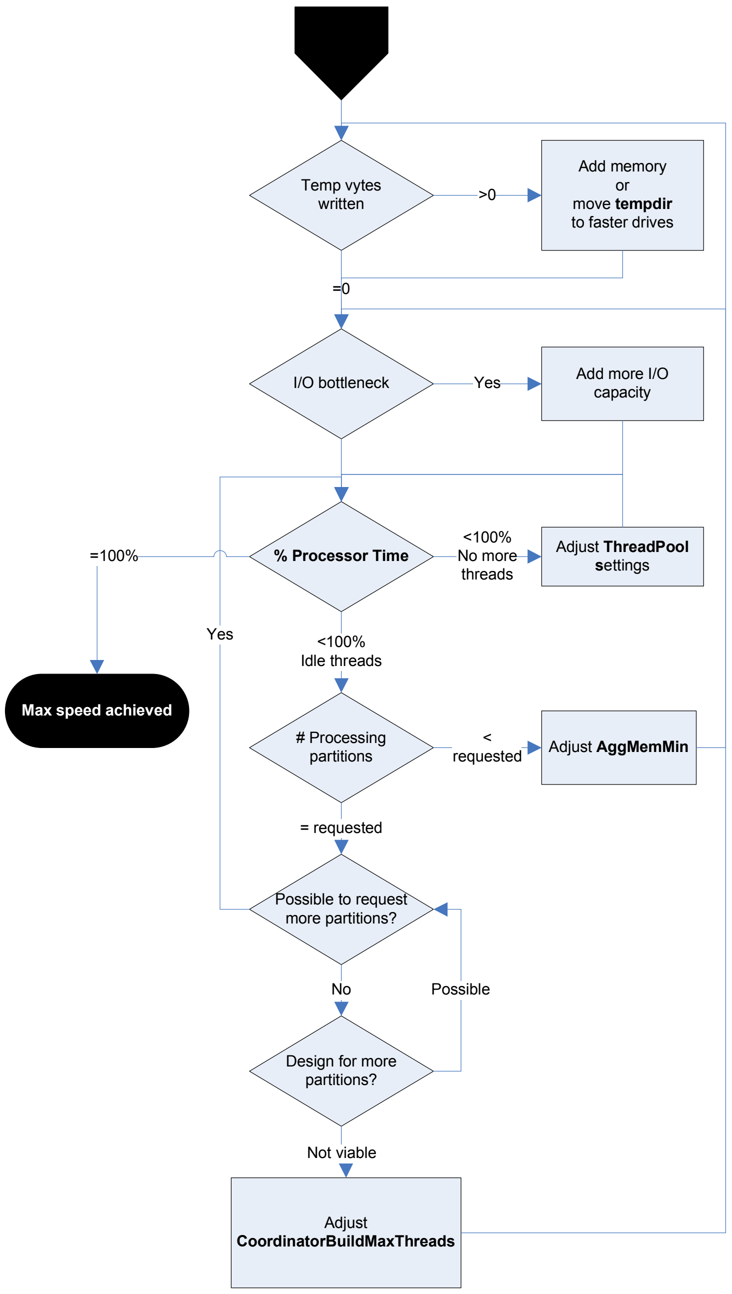
The following flowchart describes the tuning process of **ProcessData**.



Waiting for PAGEIO_X in SQL Server?
Yes
Source query fully optimized?
No
How many partitions does the process query touch?
>1
Repartition cube or relational
=1
Yes
Optimize data types
Model
Using SQL 2008?
Yes
Apply Data Compression
No
Still waiting for PAGEIO_X?
Tune I/O system or upgrade hardware
Query supported by index?
No
Add cluster index to support quey
Yes
Joins in query?
Yes
Denormalize and use star schema
No
No
Waiting for LCK_X in SQL?
Yes
Use lock elimination
No
Waiting for ASYNC_NETWORK_IO in SQL Server?
Yes
NIC
Enough NIC Capacity
Adjust network packet size
NIC Near capacity
Optimize network infrastructure
< # Cores in Analysis Services
Still waits for ASYNC_NETWORK_IO ?
Yes
NIC interrupt affinty configured?
No
Afinitze network and distribute load
Yes
Try shared memory connections if available
Increase max connection to #cores in Analysis Services
CPU load on Analysis Services?
=100
<100
CPU load on SQL Server
=100
Done
<100
Adjust threadpool
=0
Idle threads
>0
Max connections in data source
= # Cores
# Concurrent processing partitions
= # Cores in Analysis Services
< #Cores
< # Cores in Analysis Services
Repartition cube

For **ProcessIndexes**, the flowchart below applies.

63



6.4 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
- Analysis Services Processing Best Practices

6.4.1 Optimizing Data Inserts, Updates, and Deletes

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

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

- **ProcessFull**—Perform a **ProcessFull** operation for the existing partition. During the **ProcessFull** operation, the cube remains available for browsing with the existing data while a separate set of data files are created to contain the new data. When the processing is complete, the new partition data is available for browsing. Note that **ProcessFull** is technically not necessary, given that you are only doing inserts. To optimize processing for insert operations, you can use **ProcessAdd**.
- **ProcessAdd**—Use this operation to append data to the existing partition files. If you frequently perform **ProcessAdd**, it is advised 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.

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.

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.

6.4.2 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*)*

6.5 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 subsection below, we will 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 will use 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.

6.5.1 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 this query. 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.

For background on relational star schemas and how to design and denormalize for optimal performance, refer to:

- Ralph Kimball, *The Data Warehouse Toolkit*

6.5.2 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 Figure 22.

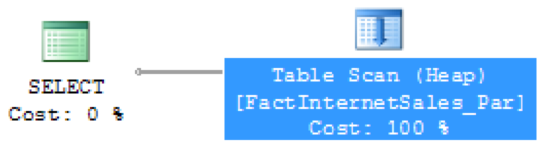


Figure 22 An optimal partition processing query

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

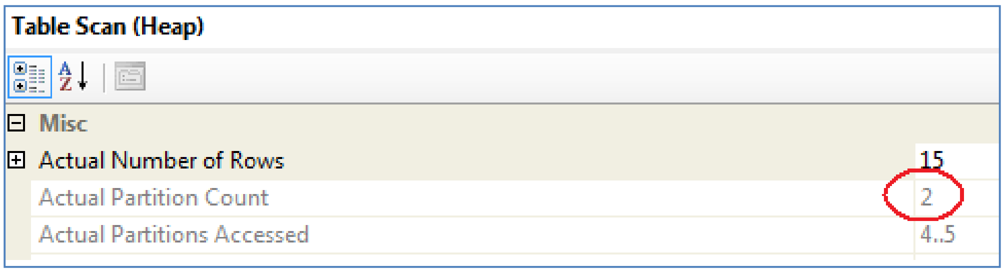


Table Scan (Heap)

Misc	
Actual Number of Rows	15
Actual Partition Count	2
Actual Partitions Accessed	4..5

Figure 23 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 above), you should consider repartitioning the relational source data so that each cube partition touches at most one relational partition.

6.5.2.1 Special Case: Distinct Count

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

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.

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

From Analysis Services 2005 and forward it is possible to use noninteger columns for distinct count measure groups. However, for performance reasons you should avoid this. The white paper below 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.

- Analysis Services Distinct Count Optimization

6.5.3 Getting Relational Indexing Right

While you generally want each cube partition to touch at most one relational partition, the opposite is not true. It is perfectly viable to have to have more than one cube partition accessing the same relational partition. As an example, a relational source that is partitioned by year with a cube that is partitioned by month can still provide optimal processing performance.

If you do not have a 1-1 relationship between relational and cube partitions, you generally want an index to support the fact processing query. The best choice of index for this purpose is a clustered index; if your load strategy allows you to maintain such an index, this is what you should aim for.

When a processing query is supported by an index the plan should look like this.

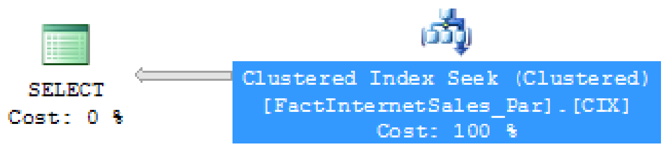

SELECT
Cost: 0 %
Clustered Index Seek (Clustered)
[FactInternetSales_Par].[CIX]
Cost: 100 %

Figure 24 Index correctly supporting processing

For more information about optimizing queries, see:

- Top 10 SQL Server 2005 Performance Issues for Data Warehouse and Reporting Applications
- Itzik Ben-Gan and Lubor Kollar, *Inside Microsoft SQL Server 2005: T-SQL Querying*

6.5.3.1 Special Case: Distinct Count

As with partitioning, distinct count is again a special case for indexing.

The distinct count processing queries will have an ORDER BY clause added to them by Analysis Services. For example, if you create a distinct count on **CustomerPONumber** in **FactInternetSales,** you would get this query while processing:

```
SELECT … FROM FactInternetSales
ORDER BY [CustomerPONumber]
```

If your partition contains a large amount of rows, ordering the data can take a long time. Without supporting indexes, the query plan will look something like this.

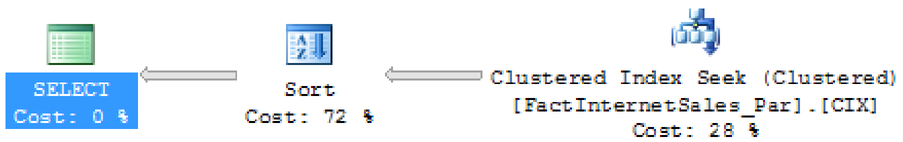

SELECT
Cost: 0 %
Sort
Cost: 72 %
Clustered Index Seek (Clustered)
[FactInternetSales_Par].[CIX]
Cost: 28 %

Figure 25 Relational sorting caused by distinct count

Notice the long time spent on the Sort operation? By creating a clustered index sorted on the distinct count column (in this case **CustomerPONumber**), you can eliminate this sort operation and get a query plan that looks like this.

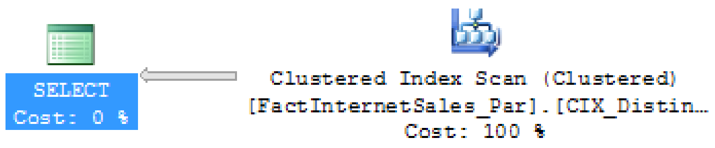

SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[FactInternetSales_Par].[CIX_Distin…
Cost: 100 %

Figure 26 Distinct count query supported by a good index

Of course, this index will have to be maintained. But having it in place will speed up the processing queries.

6.5.4 Using Index FILLFACTOR = 100 and Data Compression

If page splitting occurs in an index, the pages of the index may end up less than 100% full. The effect is that SQL Server will be reading more database pages than necessary when scanning the index.

You can check for index pages are not full by querying the SQL Server DMV **sys.dm_db_index_physical_stats**. If the column **avg_page_space_used_in_percent** is significantly lower than 100%, a FILLFACTOR 100 rebuild of the index may be in order. It is not always possible to rebuild the index like this, but this trick has the ability to reduce I/O. For stale data, rebuilding the indexes on the table is often a good idea before you mark the data as read-only.

In SQL Server 2008 you have the option of using either Row or Page compression to further reduce the amount of I/O required by the relational database to serve the fact process query. Compression has a CPU overhead, but reduction in I/O operations is often worth it.

6.6 Eliminating Database Locking Overhead

When SQL Server scans an index or table, page locks are acquired as the rows are being read. This ensures that many users can access the table concurrently. However, for data warehouse workloads, this page level locking is not always the optimal strategy – especially when large data retrieval queries like fact processing access the data.

By measuring the **Perfmon** counter **MSSQL:Locks – Lock Requests / Sec** and looking for **LCK** events in **sys.dm_os_wait_stats**, you can see how much locking overhead you are incurring during processing.

To eliminate this locking overhead, you have three options:

- Option 1: Set the relational database in Read Only mode before processing.
- Option 2: Build the fact indexes with `ALLOW_PAGE_LOCKS = OFF` and `ALLOW_ROW_LOCKS = OFF`.
- Option 3: Process through a view, specifying the `WITH (NOLOCK)` or `WITH (TABLOCK)` query hint.

Option 1 may not always fit your scenario, since setting the database to read-only mode requires exclusive access to the database. However, it is a quick and easy way to completely remove any lock waits you may have.

Option 2 is often a good strategy for data warehouses. Because SQL Server Read locks (S-locks) are compatible with other S-locks, two readers can access the same table twice, without requiring the fine granularity of page and rows locking. If insert operations are only done during batch time, relying solely on table locks may be a viable option. To disable row/page locking on a table and index rebuild ALL like this.

```
ALTER INDEX ALL ON FactInternetSales REBUILD
WITH (ALLOW_PAGE_LOCKS = OFF, ALLOW_ROW_LOCKS = OFF)
```

Option 3 is a very useful technique. Processing through a view provides you with an extra layer of abstraction on top of the database –a good design strategy. In the view definition you can add a NOLOCK or TABLOCK hint to remove database locking overhead during processing. This has the advantage of making your locking elimination independent of how indexes are built and managed.

```
CREATE VIEW vFactInternetSales
AS
SELECT [ProductKey], [OrderDateKey], [DueDateKey]
      ,[ShipDateKey], [CustomerKey], [PromotionKey]
      ,[CurrencyKey], [SalesTerritoryKey], [SalesOrderNumber]
      ,[SalesOrderLineNumber], [RevisionNumber], [OrderQuantity]
      ,[UnitPrice], [ExtendedAmount], [UnitPriceDiscountPct]
      ,[DiscountAmount], [ProductStandardCost], [TotalProductCost]
```

```
            ,[SalesAmount], [TaxAmt], [Freight]
            ,[CarrierTrackingNumber] ,[CustomerPONumber]
      FROM [dbo].[FactInternetSales] WITH (NOLOCK)
```

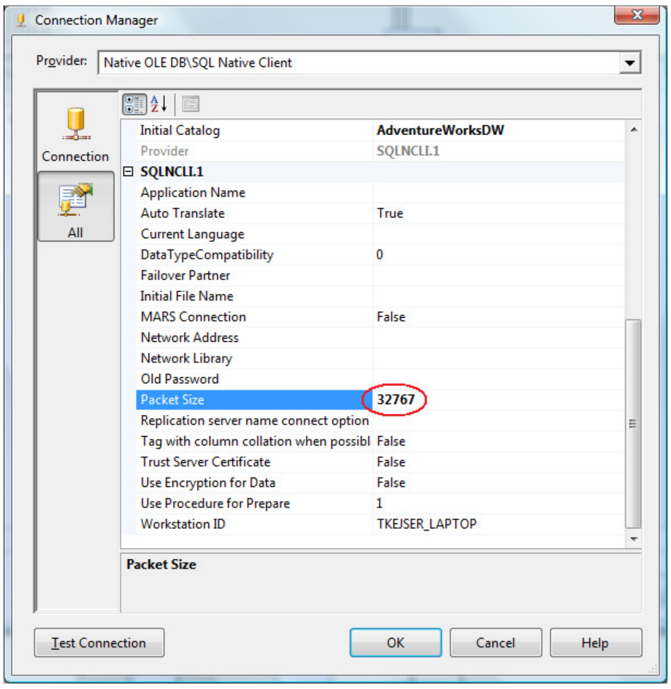
If you use the **NOLOCK** hint, beware of the dirty reads that can occur. For more information about locking behaviors, see SET TRANSACTION ISOLATION LEVEL in SQL Server Books Online.

6.7 Optimizing Network Throughput

During **ProcessData**, rows must be transferred between SQL Server and Analysis Services. If these two services are installed on different computers, TCP/IP network traffic occurs. You should make sure all your network components are configured to support the throughput you need. If your Ethernet throughput is consistently close to 80% of your maximum capacity, adding more network capacity will typically speed up **ProcessData**. Also, if your network is becoming a bottleneck, you will see waits for **ASYNC_NETWORK_IO** in SQL Server.

In addition to creating a high-speed network, there are some additional configurations you can change to further speed up network traffic.

Under the properties of your data source, increasing the network packet size for SQL Server will minimize the protocol overhead require to build many, small packages. The default value for SQL Server 2008 is 4096. With a data warehouse load, a packet size of 32K (in SQL Server, this means assigning the value 32767) can benefit processing. Instead of changing the value of the SQL Server, override it in your data source.



Provider: Native OLE DB\SQL Native Client

Initial Catalog	AdventureWorksDW
Provider	SQLNCLI.1
⊟ SQLNCLI.1	
Application Name	
Auto Translate	True
Current Language	
DataTypeCompatibility	0
Failover Partner	
Initial File Name	
MARS Connection	False
Network Address	
Network Library	
Old Password	
Packet Size	32767
Replication server name connect option	
Tag with column collation when possibl	False
Trust Server Certificate	False
Use Encryption for Data	False
Use Procedure for Prepare	1
Workstation ID	TKEJSER_LAPTOP

Packet Size

Figure 14 Tuning network packet size

You should be aware that the overhead of transporting data over the TCP/IP network is significant. Recall that Analysis Services, if it is installed on the same machine as SQL Server, is capable of using Shared Memory connections. Shared Memory connections incur minimum overhead during data exchange between SQL Server and Analysis Services. Depending on your processing workload, you may therefore be able to speed up processing by consolidating SQL server and Analysis Services to the same computer.

You can check if your connection is running shared memory by executing the following SELECT statement.

```
SELECT session_id, net_transport, net_packet_size
FROM sys.dm_exec_connections
```

The **net_transport** for the Analysis Services SPID should show: **Shared memory**.

For more information about shared memory connections, see:

- Creating a Valid Connection String Using Shared Memory Protocol

6.8 Improving the I/O Subsystem

If you have fully tuned the relational source system and eliminated network bottlenecks, it is time to look at the I/O subsystem.

From SQL Server's perspective, you can measure the I/O latency from **sys.dm_os_wait_stats**. If you consistently see high waiting for **PAGELATCH_IO**, you can benefit from a faster I/O subsystem for SQL Server.

If you have placed your Analysis Services files on a separate drive letter or mount point (which we recommend), you can use the **Logical Disk perfmon** counter to measure I/O wait times. If your wait times are consistently over 0.015 seconds, you can also benefit from faster I/O on the Analysis Services drives.

Techniques for tuning I/O subsystems are beyond the scope of this document. For more information, see the following technical notes and white papers:

- Storage Top 10 Best Practices
- SQL Server 2000 I/O Basics
- Predeployment I/O Best Practices

6.9 Increasing Concurrency by Adding More Partitions

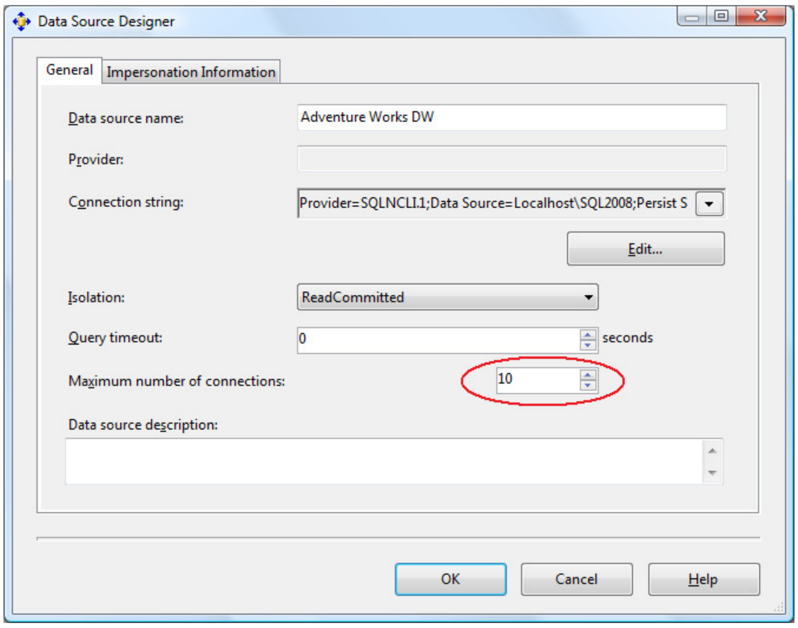
At this point of the tuning you are now bound only by the amount of CPU power you have and the ability to issue high-concurrency operations. It is time to have a look at the **Processor:Total** counter from the baseline trace. If this counter is not 100%, you are not taking full advantage of your CPU 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. Since 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.

6.10 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.



Data source name: Adventure Works DW

Provider:

Connection string: Provider=SQLNCLI1.1;Data Source=Localhost\SQL2008;Persist S

Isolation: ReadCommitted

Query timeout: 0 seconds

Maximum number of connections: 10

Data source description:

Figure 28 Adding more database connections

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

6.11 Adjusting ThreadPool and CoordinatorExecutionMode

These server-wide properties increase the number of threads that can be used to support parallel processing operations.

ThreadPool\Process\MaxThreads determines the maximum number of available threads to Analysis Services during processing. On large, multiple-CPU machines, the default value of this setting may be too low to take advantage of all CPU cores. However, as you increase this counter, bear in mind that increased parallelism of processing also has an effect on queries running at the system. As you dedicate more CPU power and threads to processing, less CPU will be use for query responses. Of course, if you are processing the cubes during a batch window, this may not be an issue.

To optimize these settings for the **ProcessData** phase, check your **perfmon** counter on the object **MSOLAP: Threads** and use the table below for guidance.

Situation	Action
Processing pool job queue length > 0 and **Processing pool idle threads** = 0 for longer periods during processing.	Increase **Threadpool\Process\MaxThreads** and retest.
Both **Processing pool job queue length** > 0 and **Processing pool idle threads** > 0 at same time during processing.	Decrease **CoordinatorExecutionMode** and retest.

You can use the **Processor –% Processor Time – Total** counter as a rough indicator of how much you should change these settings. You are aiming to get as close to 100% CPU utilization as possible. For example, if your CPU load is 50% you can double **Threadpool\Process\MaxThreads** to see if this also doubles your CPU usage.

For more information about adjusting thread pools, see the following white paper:

- SQL Server 2005 Analysis Services (SSAS) Server Properties

6.12 Adjusting BufferMemoryLimit

OLAP\Process\BufferMemoryLimit determines the size of the fact data buffers used during partition processing. While the default value of the **OLAP\Process\BufferMemoryLimit** is sufficient for most deployments, you may find it useful to alter the property in the following scenario.

If the granularity of your measure group is more summarized than the relational source fact table, you may want to consider increasing the size of the buffers to facilitate data grouping. For example, if the source data has a granularity of day and the measure group has a granularity of month; Analysis Services must group the daily data by month before writing to disk. This grouping occurs within a single buffer and it is flushed to disk after it is full. By increasing the size of the buffer, you decrease the number of times that the buffers are swapped to disk. Because this allows higher compression ratio, the size of the fact data on disk is decreased, which provides higher performance. However, be aware that high values for the **BufferMemoryLimit** will use more memory. If memory runs out, parallelism is decreased.

6.13 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 SQL Server 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.

6.13.1 Avoid Spilling Temporary Data to Disk

During processing, the aggregation buffer determines the amount of memory that is available to build aggregations for a given partition. If the aggregation buffer is too small, Analysis Services supplements the aggregation buffer with temporary files. Temporary files are created in the **TempDir** folder when memory is filled and data is sorted and written to disk. When all necessary files are created, they are merged together to the final destination. Using temporary files can potentially result in some performance degradation during processing. To monitor any temporary files used during processing, review **MSOLAP:Proc Aggregations\Temp file bytes written/sec**.

In addition, when processing multiple partitions in parallel or processing an entire cube in a single transaction, you must ensure that the total memory required does not exceed the value of the **Memory\TotalMemoryLimit** setting. If Analysis Services reaches the **Memory\TotalMemoryLimit** during processing, it does not allow the aggregation buffer to grow and may cause temporary files to be used during aggregation processing. Furthermore, if you have insufficient virtual address space for these simultaneous operations, you may receive out-of-memory errors. If you have insufficient physical memory, memory paging will occur. If processing in parallel and you have limited resources, consider doing less in parallel.

Under the default configuration, Analysis Services will throw an out-of-memory exception if you try to request too much memory during processing. It is possible to disable this error by setting the **MemoryLimitErrorEnabled** to **false** in the server properties. However, this may cause disk spill and slow down the processing operation.

If there is no way you can avoid spilling data to disk, you should at least make sure the **TempDir** folder and Page file is a fast I/O system.

6.13.2 Eliminate I/O Bottlenecks

During **ProcessIndex** the disk activity is generally lower than during **ProcessData**. If you have enough I/O to not bottleneck on **ProcessData**, chances are that the I/O speed will be sufficient for **ProcessIndex** too. However, you should still monitor the I/O using the guidelines from Improving the I/O Subsystem.

6.13.3 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.

6.13.4 Tune Threads and AggregationMemorySettings

During **ProcessIndex**, Analysis Services will scan and aggregate the partitions created during **ProcessData**. There are two ways to perform this operation in parallel:

- Use several threads to concurrently scan and aggregate the segments of one partition at a time.
- Scan and aggregate several partitions at the same time with a lower number of threads.

Both techniques can be used at the same time, to a lesser or greater degree. Using server properties, you can control how this is executed. Also, both settings are limited by the thread settings on the server, as described in Adjusting ThreadPool and CoordinatorExecutionMode.

If you are adding more partitions to increase parallelism you may need to change the **AggregationMemory** settings. If your design does not allow you to add more partitions, you have the option of changing **CoordinatorBuildMaxThreads** to further increase parallelism.

When you measure CPU utilization with the counter **Processor –% Processor Time – Total** that is less than 100%, and assuming you have no I/O bottlenecks, there is a good chance that you can increase the speed of **ProcessIndex** phase further by using the techniques in this section.

6.13.4.1 Adjusting Thread Settings

Just as was the case under **ProcessData** you may have to adjust your thread pool setting to achieve optimal performance. Use the guidelines from Adjusting ThreadPool and CoordinatorExecutionMode.

6.13.4.2 Adjusting the AggregationMemoryMin Setting

Under the server properties you will find the settings:

- **OLAP\Process\AggregationMemoryLimitMin**
- **OLAP\Process\AggregationMemoryLimitMax**

These settings, expressed as a percentage of the Analysis Services memory, determine how much memory is allocated for the creation of aggregations in each partition. When Analysis Services starts partition processing, parallelism is throttled based on the **AggregationMemoryMin** setting. For example, if you start five concurrent partition processing jobs with **AggregationMemoryMin** = 10, an estimated 50% of memory will be allocated for the processing. If memory runs out, new partition processing jobs will block while they way for memory to become available. If you process many partitions in parallel, lowering the value of **AggregationMemoryLimitMin** can increase **ProcessIndex** speed. By limiting the minimum amount of memory allocated per partition, you can drive a higher degree of parallelism in the process index phase.

Like the other Analysis Services counters, if this setting has a value greater than 100 it is interpreted as a fixed amount of kilobytes. For machines with large amounts of memory, using an absolute kilobyte value may provide a better control of memory than using a percentage value.

6.13.4.3 Adjusting CoordinatorBuildMaxThreads

When scanning a single partition, the amount of threads used to scan extents is limited by the **CoordinatorBuildMaxThreads** setting. The setting determines the maximum number of threads allocated per partition processing job. It behaves in the same way as **CoordinatorExecutionMode** (refer to the section on Job Architecture.) If this setting has a negative value, its absolute value is multiplied by the number of cores in the machine to determine the maximum number of threads that can run. If the setting has a positive value, it is the absolute number of threads that can be run. Keep in mind that you will still be limited by the number of threads in the process threadpool when processing, so increasing this value may mean increasing the process threadpool too.

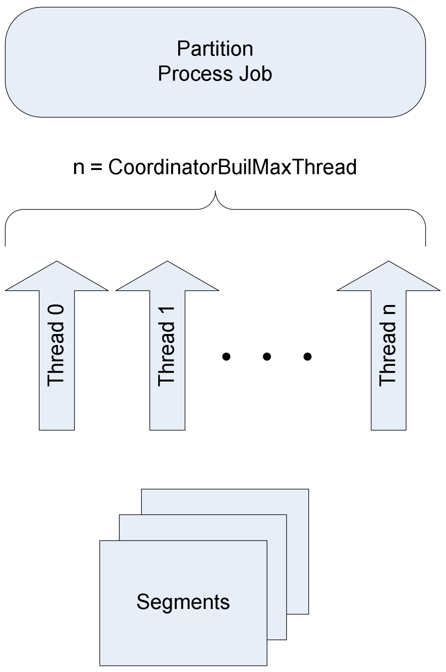


Figure 29 CoordinatorBuildMaxThreads

If you are not able to drive high parallelism by using more partitions, you can change the **CoordinatorBuildMaxThreads** value. Increasing this allows you to use more threads per partition.

Note that you may also need to adjust the **AggregationMemoryMin** settings too, to get optimal results.

7 Tuning Server Resources

While you can tune your application, there are times when you simply need to add more hardware of tune the server itself. This section describes server wide settings you can apply to increase performance.

7.1 Using PreAllocate

The **PreAllocate** setting found in msmdsrv.ini can be used to reserve physical memory for Analysis Services. For installations where Analysis Services coexists with other services on the same machine, setting **PreAllocate** can provide a more stable memory configuration.

Note that if the service account used to run Analysis Services also has the **Lock pages in Memory** privilege, **PreAllocate** will cause Analysis Services to use large memory pages. **Lock pages in Memory** is set using gpedit.msc. Bear in mind that large memory pages cannot be swapped out to the page file. While this can be an advantage from a performance perspective, a high number of allocated large pages can cause the system to become unresponsive.

You should generally leave around 20% of total system memory for the operation system when using **PreAllocate** with large pages.

Important: PreAllocate has the largest impact on the Windows Server® 2003 operating system. With the introduction of Windows Server 2008, memory management has been much improved. We have been testing this setting on Windows 2008 Server, but have not measured any benefits of using **PreAllocate** on this platform. Considering the drawbacks of **PreAllocate**, there is probably very little benefit of this setting under Windows Server 2008.

To learn more about the effects of **PreAllocate**, see the following technical note:

- Running Microsoft SQL Server 2008 Analysis Services on Windows Server 2008 vs. Windows Server 2003 and Memory Preallocation: Lessons Learned

7.2 Disable Flight Recorder

Flight Recorder provides a mechanism to record Analysis Services server activity into a short-term log. Flight Recorder provides a great deal of benefit when you are trying to troubleshoot specific querying and processing problems; however, it introduces a certain amount of I/O overheard. If you are in a production environment and you do not require Flight Recorder capabilities, you can disable its logging and remove the I/O overhead. The server property that controls whether Flight Recorder is enabled is the **Log\Flight Recorder\Enabled** property. By default, this property is set to **true**.

7.3 Monitoring and Adjusting Server Memory

Generally, the more memory you have the better. If the data files can reside in the operating system cache, storage engine performance is very forgiving. If the formula engine can cache its results, cell values are reused rather than recomputed. During processing, not spilling results to disk also improves performance. However, be aware that Analysis Services will not use AWE memory on 32-bit systems. If your cube requires a high amount of memory, we highly recommend 64-bit hardware.

Key memory settings are **Memory\TotalMemoryLimit** and **Memory\LowMemoryLimit** and are expressed as a percentage of available memory. You can monitor memory from Task Manager or from the following the performance counters:

- MSAS2008:Memory\Memory Usage Kb
- MSAS2008:Memory\Memory Limit Low Kb
- MSAS2008:Memory\Memory Limit High Kb

Unless **PreAllocate** is used, Analysis Services gives up memory when not under load – other applications (such as the SQL Server engine) may consume freed memory and not give it up. So, it is important to configure not only Analysis Services properly but also other applications on the machine.

Before deciding that more memory is required, take the steps outlined in the querying and processing sections to optimize cube performance.

8 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 Richard Tkachuk at richtk@Microsoft.com or Thomas Kejser at tjkejser@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.

Microsoft

Microsoft SQL Server 2008 R2

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

6

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.

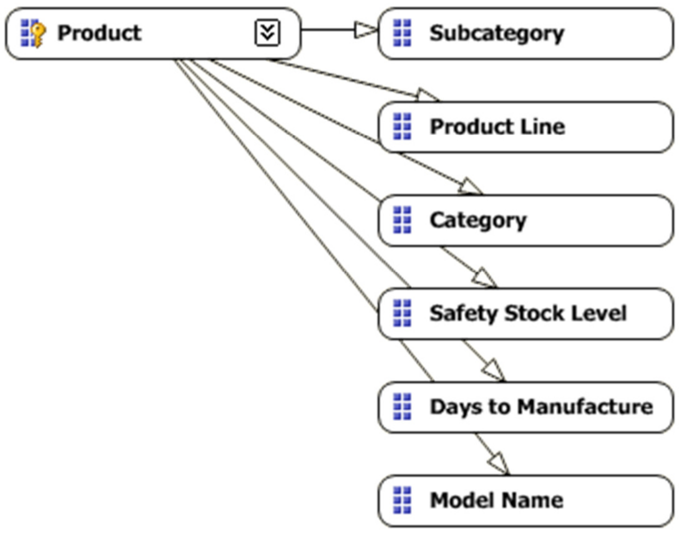


Figure 1: Bushy attribute relationships

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.

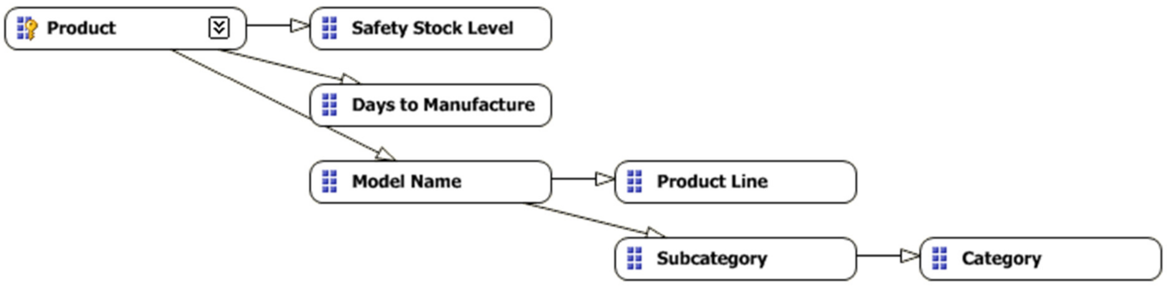


Figure 2: Redefined attribute relationships

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.

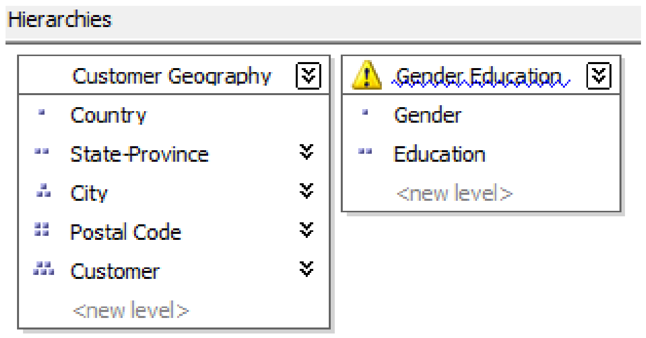


Figure 3: Natural and unnatural hierarchies

12

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.

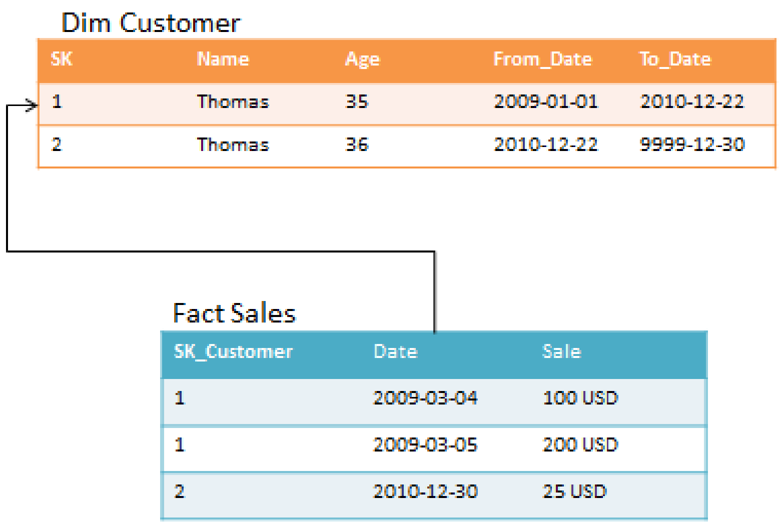
Type2 Fast-Changing Attributes - If you track every change to a fast-changing attribute, this may cause the dimension containing the attribute to grow very large. Type 2 attributes are typically added to a dimension with a **Process Add** command. At some point, running **Process Add** on a large dimension 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

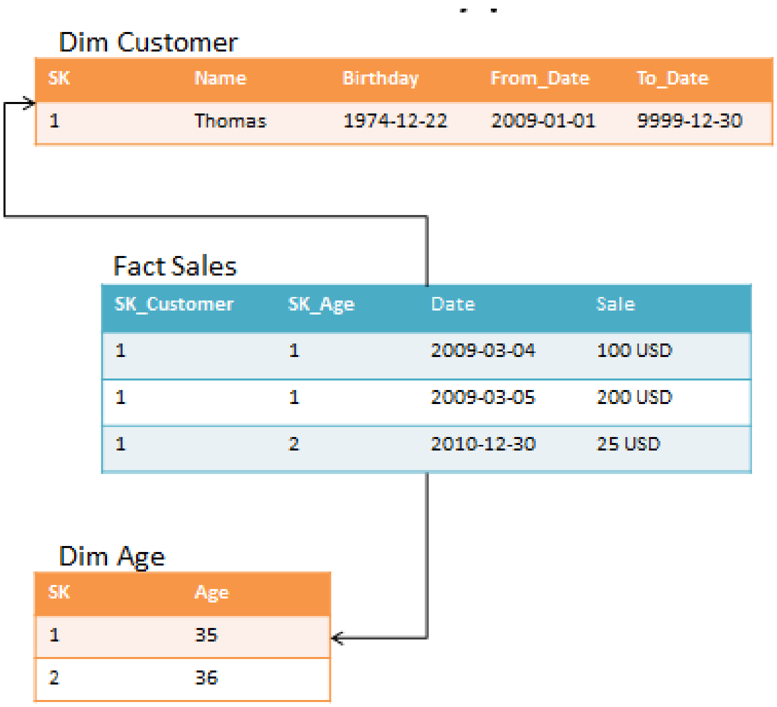
SK	Name	Age	From_Date	To_Date
1	Thomas	35	2009-01-01	2010-12-22
2	Thomas	36	2010-12-22	9999-12-30

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 4: 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

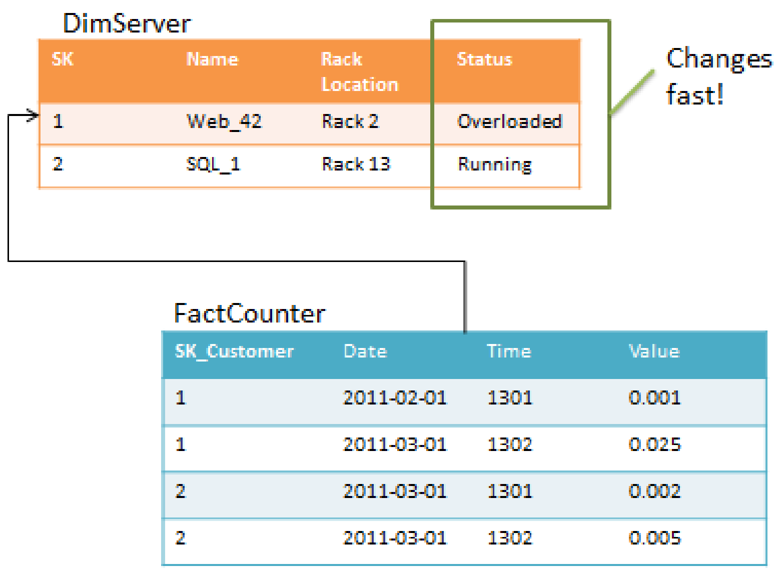
Figure 5: Age in its own dimension

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.



DimServer

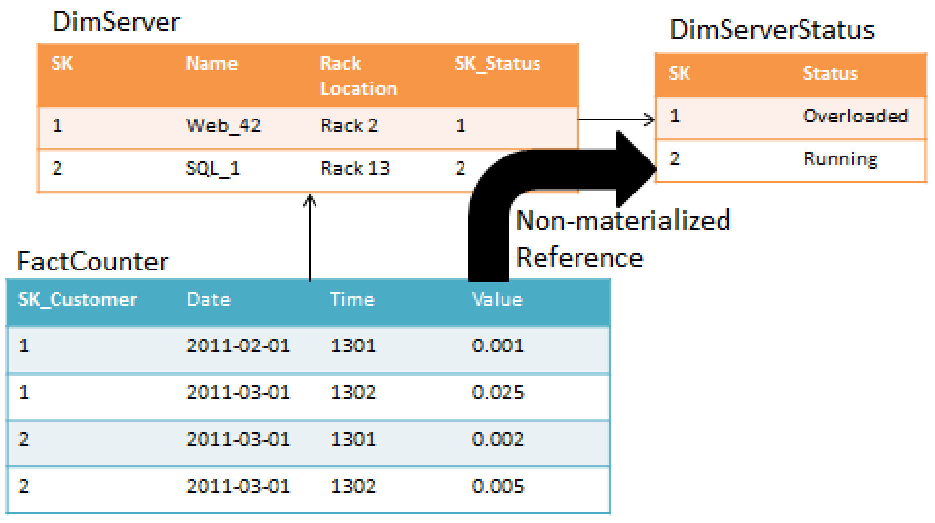
SK	Name	Rack Location	Status
1	Web_42	Rack 2	Overloaded
2	SQL_1	Rack 13	Running

FactCounter

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 6: 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.



DimServer

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

DimServerStatus

SK	Status
1	Overloaded
2	Running

FactCounter

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 7: 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.

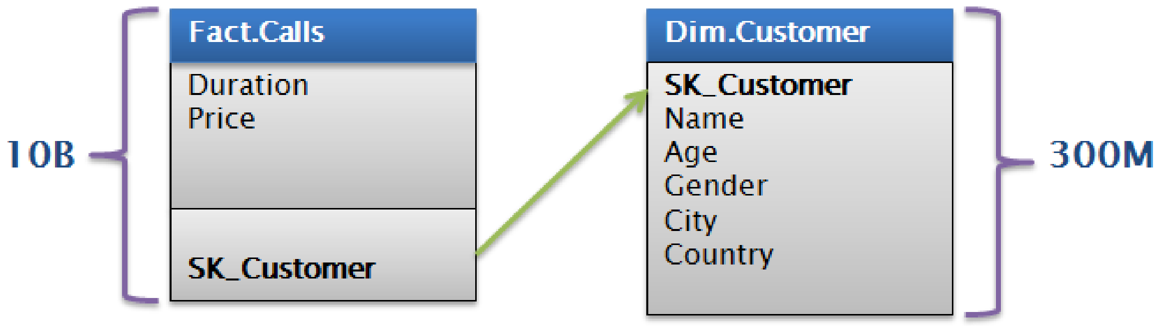
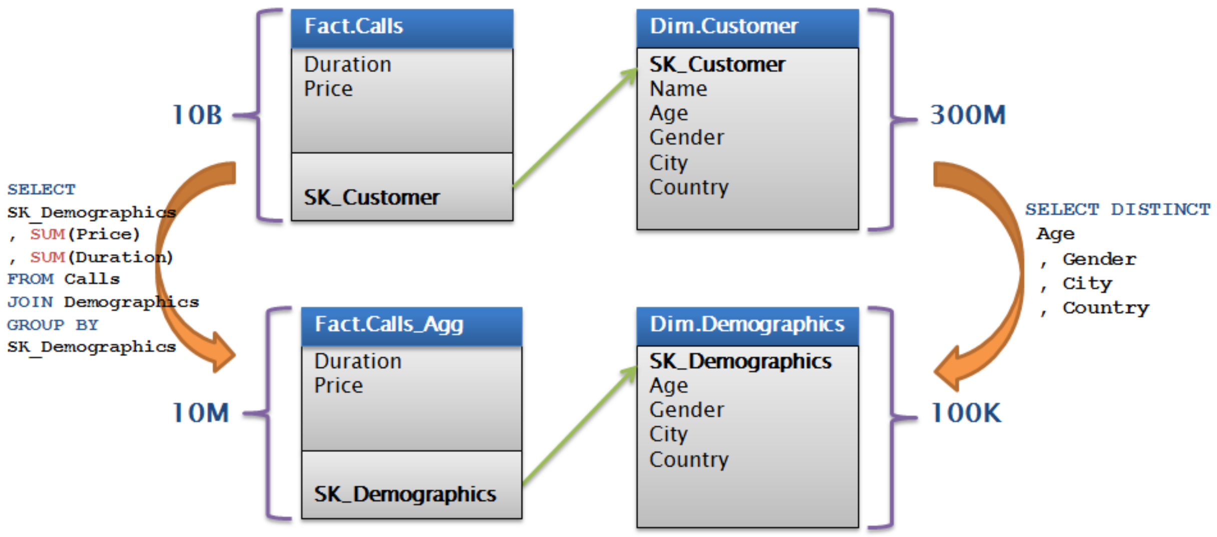


Figure 8: Call detail records (CDRs)

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.



```
SELECT
SK_Demographics
, SUM(Price)
, SUM(Duration)
FROM Calls
JOIN Demographics
GROUP BY
SK_Demographics
```

```
SELECT DISTINCT
 Age
 , Gender
 , City
 , Country
```

Figure 9: 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.

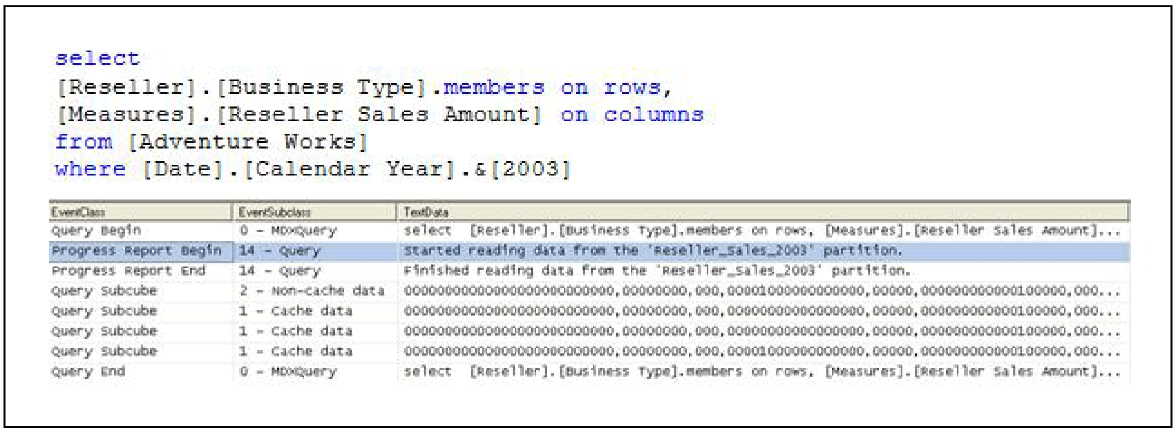
2.2 Partitioning a Cube

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	0000000000000000000000000,00000000,000,00001000000000000,00000,00000000000100000,000...
Query Subcube	1 - Cache data	0000000000000000000000000,00000000,000,00000000000000000,00000,00000000000100000,000...
Query Subcube	1 - Cache data	0000000000000000000000000,00000000,000,00000000000000000,00000,00000000000100000,000...
Query Subcube	1 - Cache data	0000000000000000000000000,00000000,000,00001000000000000,00000,00000000000100000,000...
Query End	0 - MDXQuery	select [Reseller].[Business Type].members on rows, [Measures].[Reseller Sales Amount]...

Figure 10: 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.

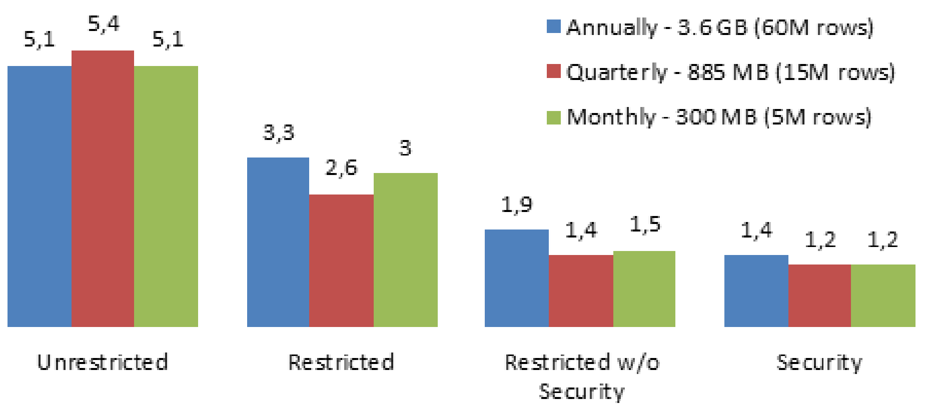


Figure 11: Throughput by partition size (higher is better)

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.

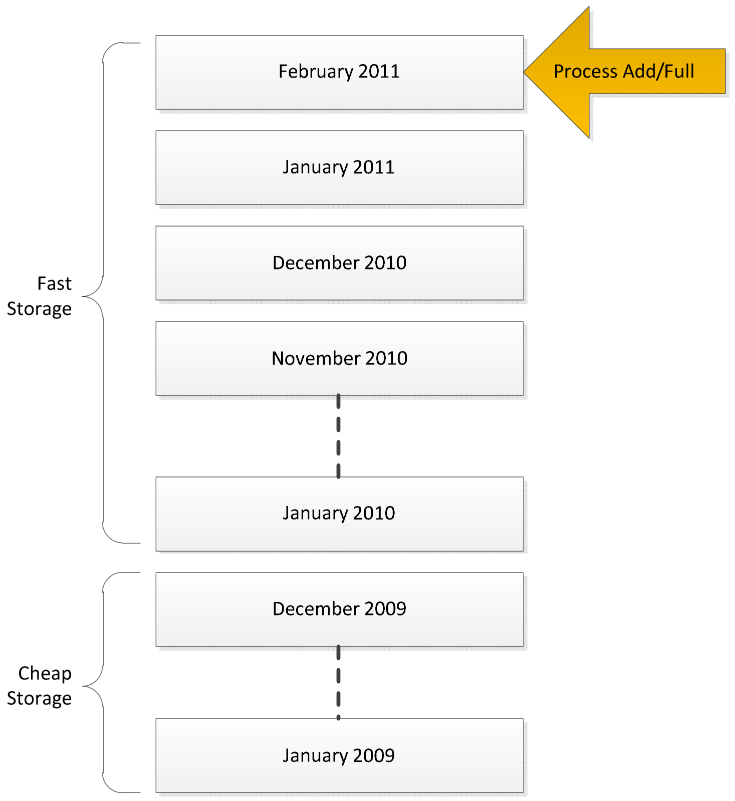


Figure 12: Partitioning by Date

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.

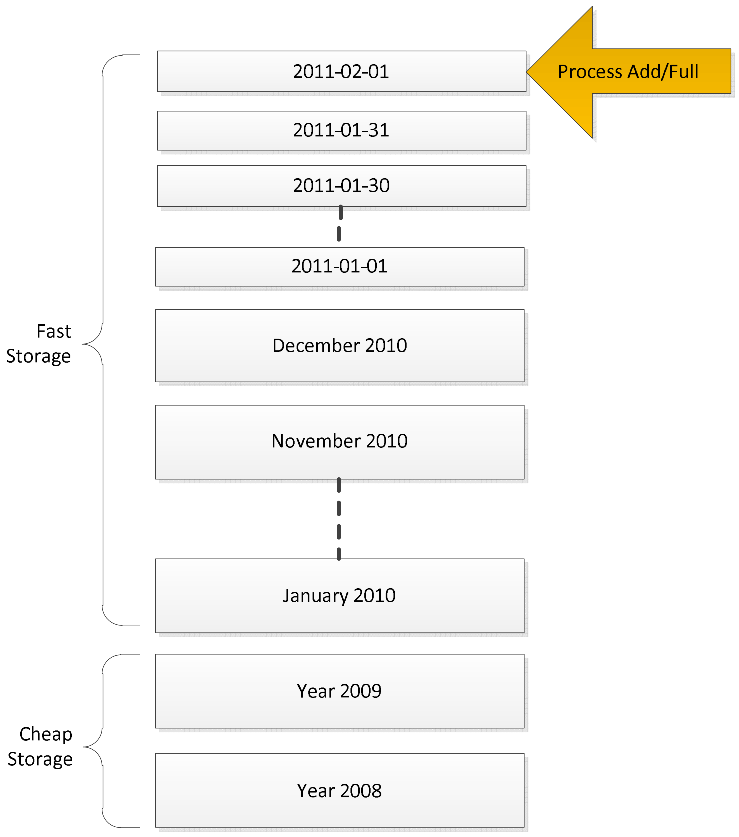


Figure 13: Modified Date Partitioning

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.

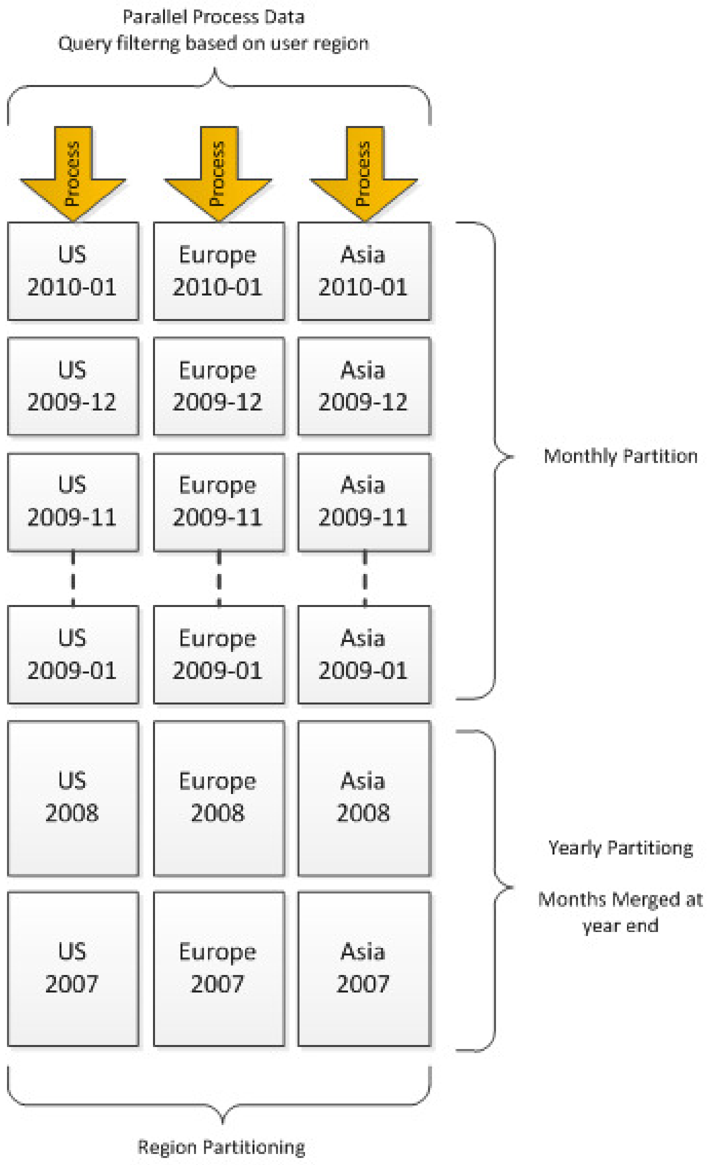


Figure 14: Example of matrix partitioning

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

25

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.

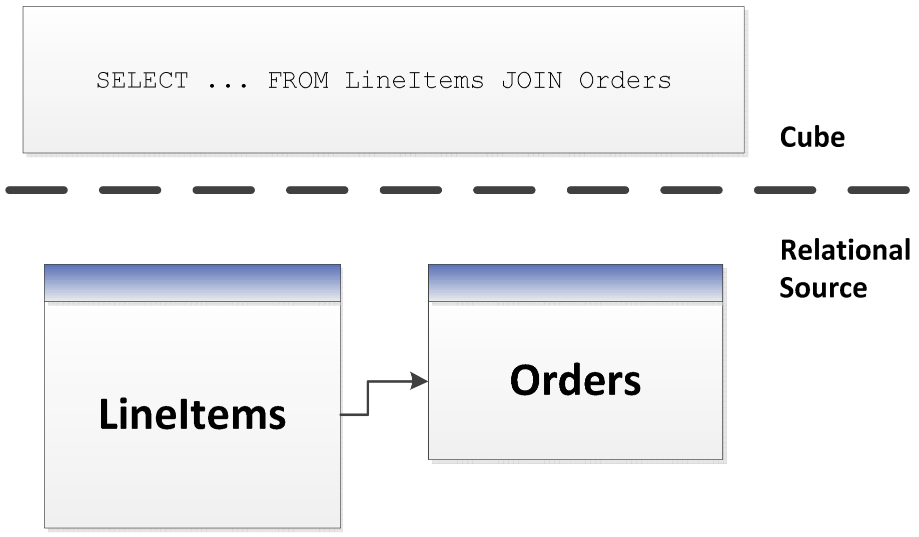


Figure 15: Using named queries in UDM

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

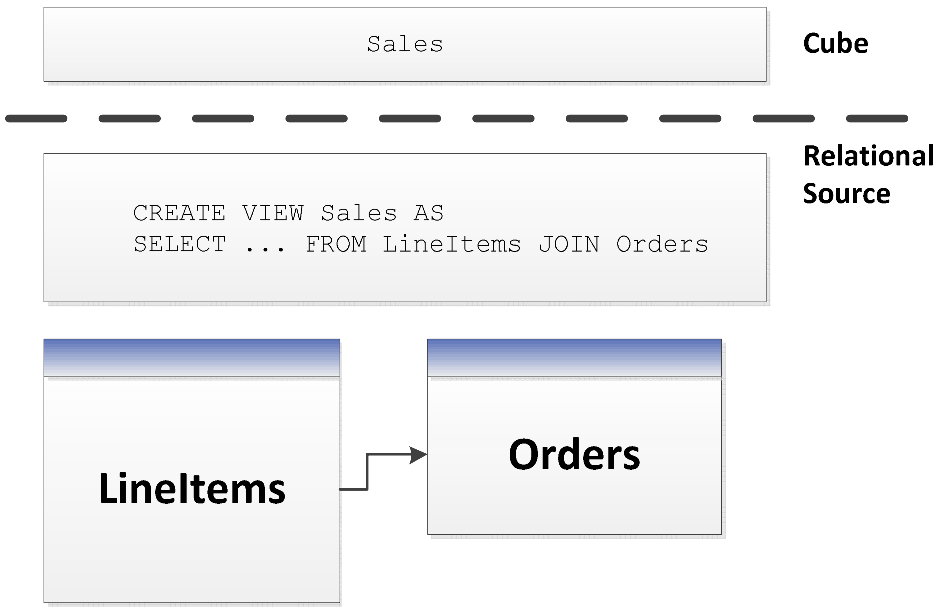


Figure 16: Implementing UDM on top of views

This 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.

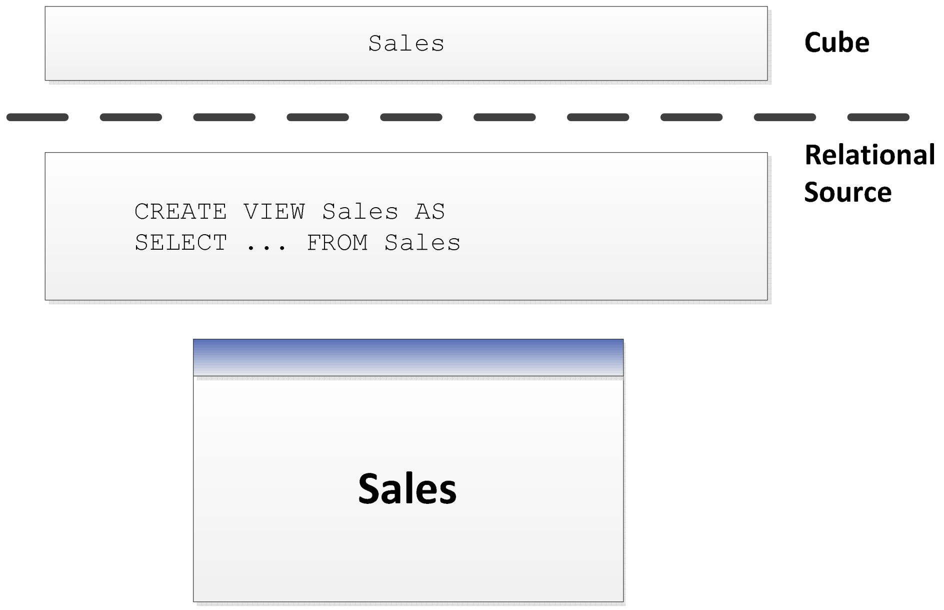


Figure 17: Implementing UDM on top of pre-joined tables

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.

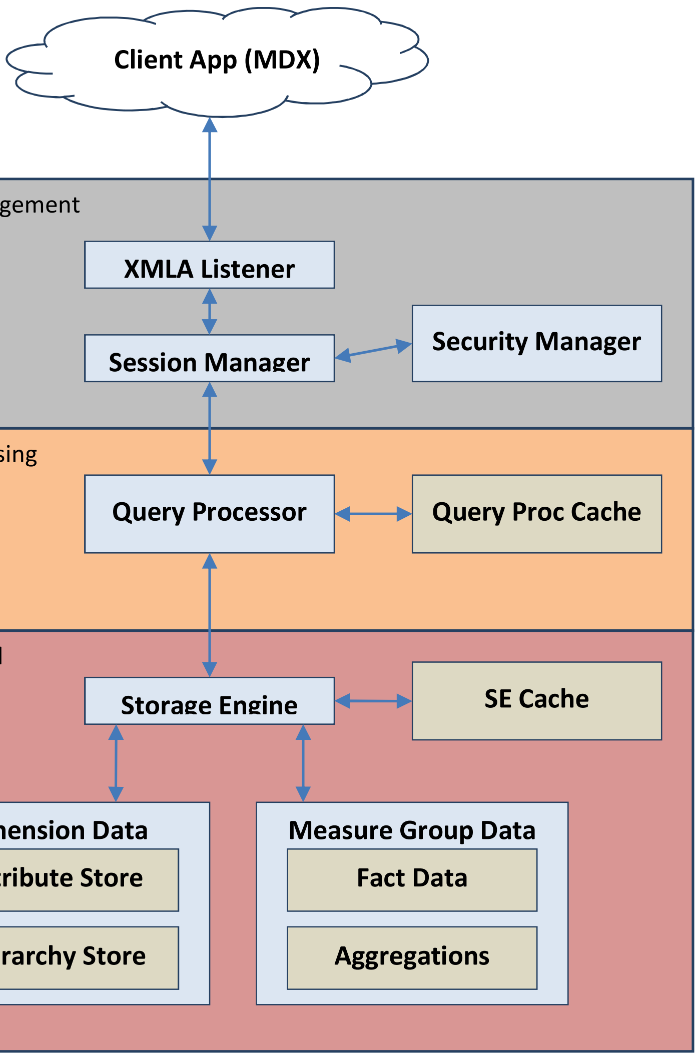


Figure 18: Analysis Services query processor architecture

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.

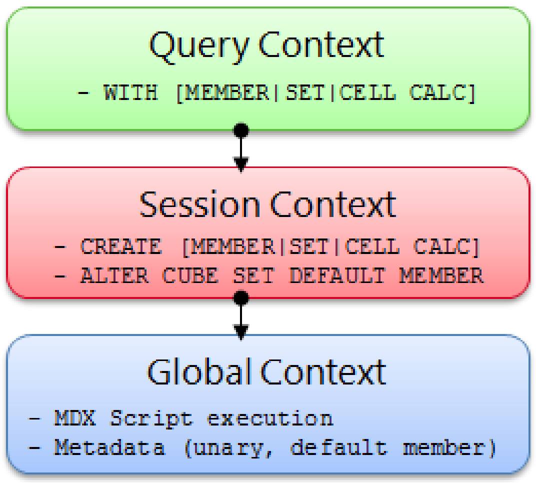

Query Context
- WITH [MEMBER|SET|CELL CALC]
Session Context
- CREATE [MEMBER|SET|CELL CALC]
- ALTER CUBE SET DEFAULT MEMBER
Global Context
- MDX Script execution
- Metadata (unary, default member)

Figure 19: 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.

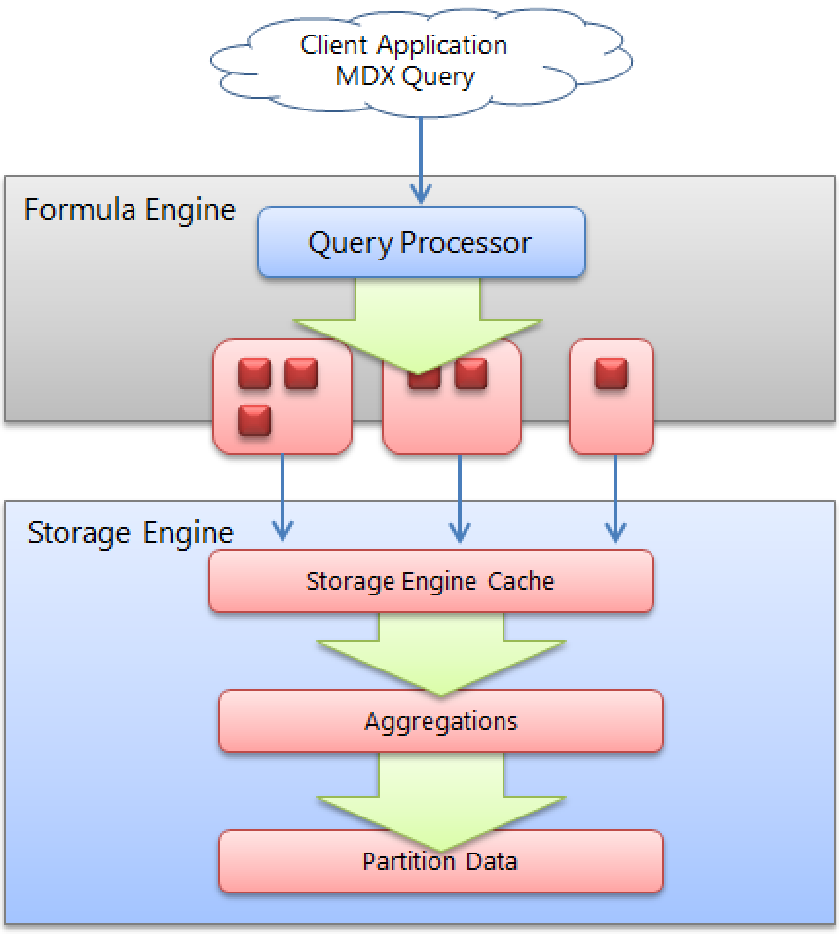


Figure 20: High-level overview of the data retrieval process

38

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:

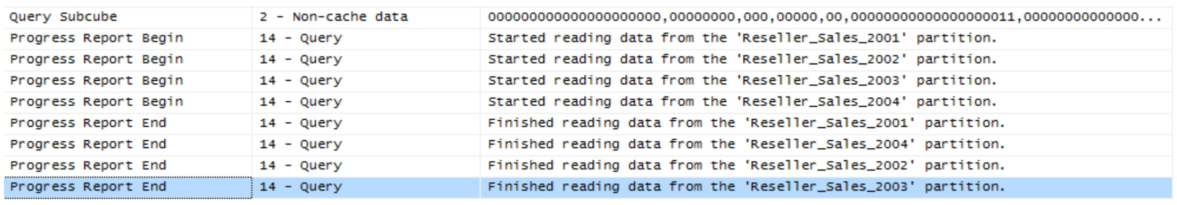
- If there is a matching subcube, the corresponding data cache is returned.
- If a subcube superset is found, a new data cache is generated and the results are filtered to fit the subcube request.
- If lower-grain data exists, the data cache registry can aggregate this data and make it available as well – 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	000000000000000000000,00000000,000,00000,00,00000000000000000011,0000000000000...
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 21: 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.

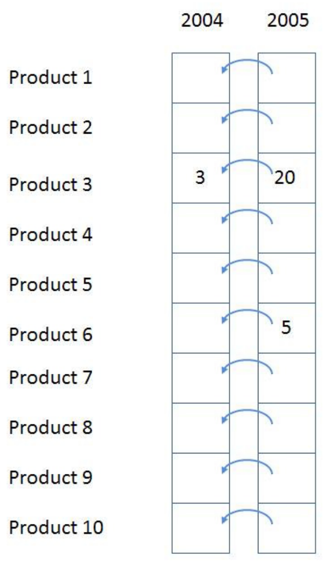


Figure 22: Cell-by-cell evaluation

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

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.

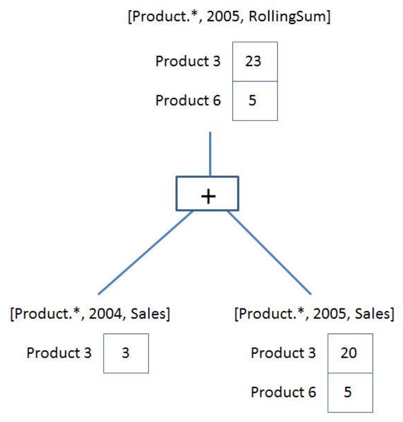


Figure 23: Execution plan

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

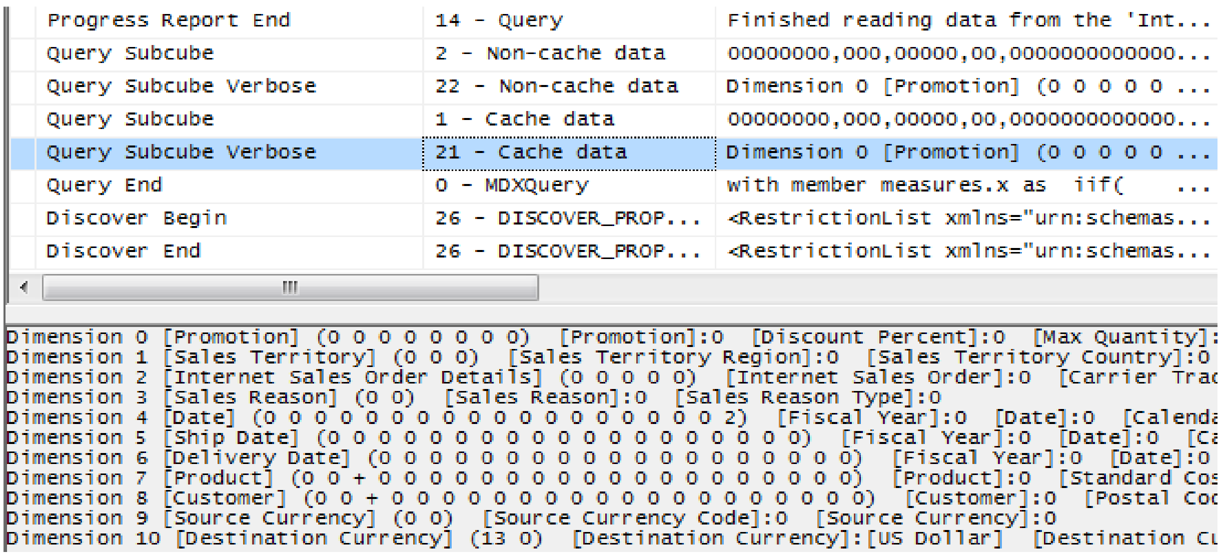
When all caches are clear, you should initialize the calculation script by executing a query that returns and caches nothing. Here is an example.

```
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 24: 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:

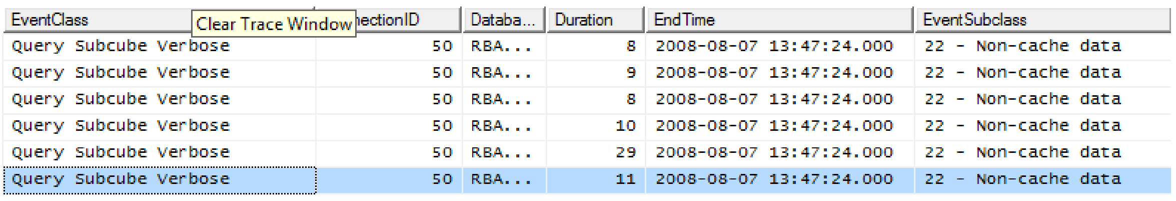
- Identifying and Resolving MDX Query Performance Bottlenecks in SQL Server 2005 Analysis Services (http://sqlcat.com/sqlcat/b/whitepapers/archive/2007/12/16/identifying-and-resolving-mdx-query-performance-bottlenecks-in-sql-server-2005-analysis-services.aspx)
- Configuring the Analysis Services Query 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	ectionID	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 25: 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])
```

53

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:

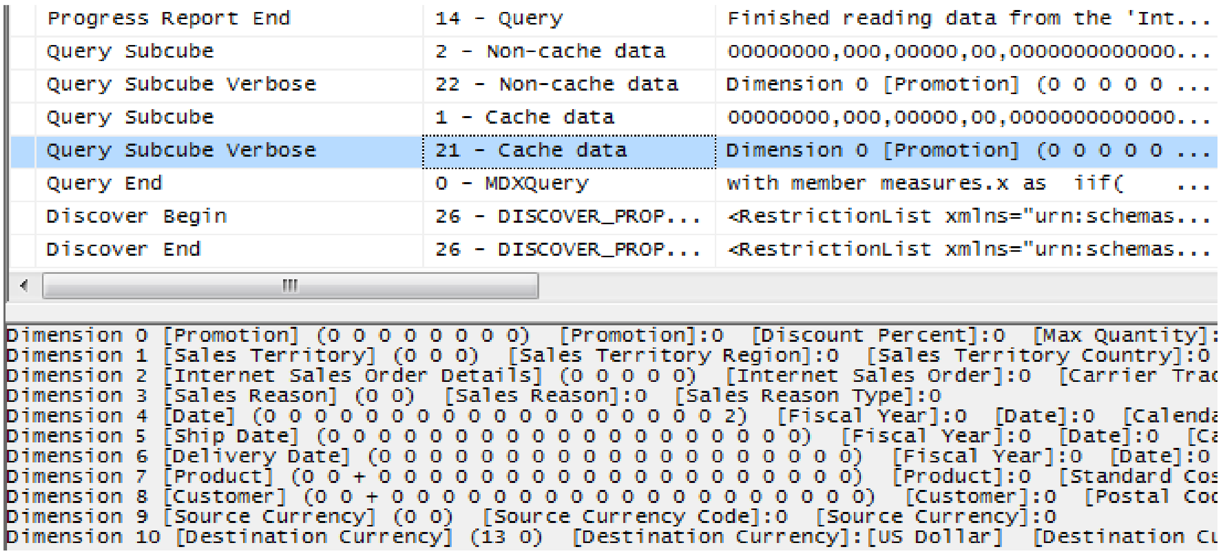
- The engine determines the query plan for the condition is expensive and evaluates each branch in strict mode.
- The condition is evaluated in cell-by-cell mode, and each branch is evaluated in eager mode.
- The branch expression is dense but easily evaluated.

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     00000000,000,00000,00,0000000000000...
Query Subcube Verbose    22 - Non-cache data    Dimension 0 [Promotion] (0 0 0 0 0 ...
Query Subcube            1 - Cache data         00000000,000,00000,00,0000000000000...
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 26: Default IIf query trace

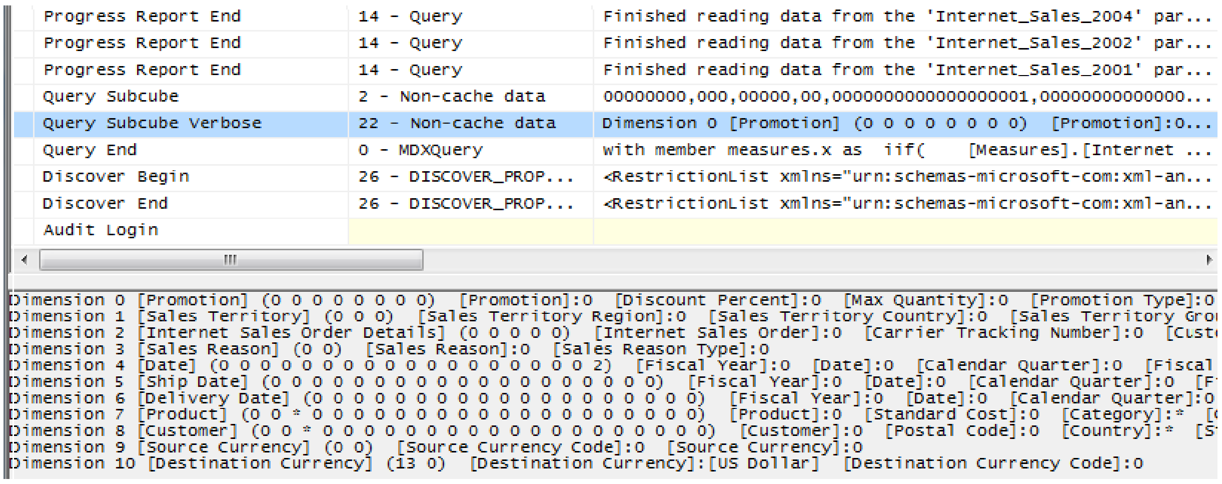
Note the subcube definition for the Product and Customer dimensions (dimensions 7 and 8 respectively) with the '+' indicator on the Country and Category attributes. This means that more than one but not all members are included – the query processor has determined which tuples meet the condition and 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
```

55

```
   , (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 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 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 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 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 27: 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

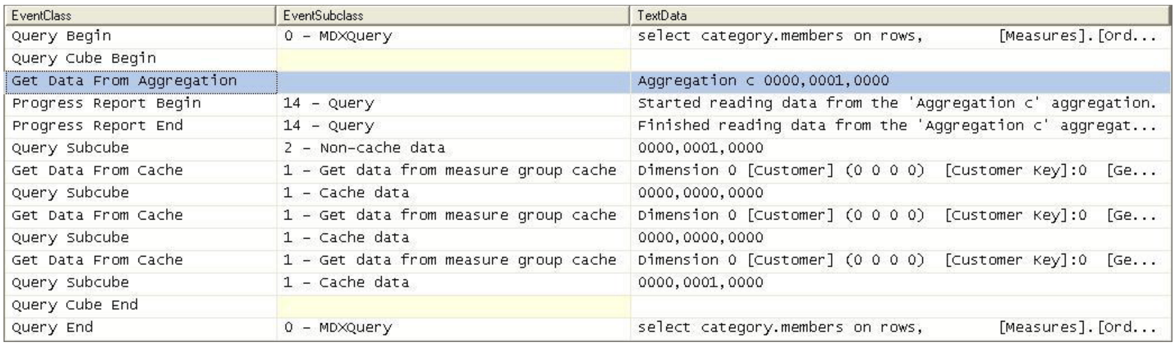
An *aggregation* is a data structure that stores precalculated data that Analysis Services uses to enhance query performance. You can define the aggregation design for each partition independently. Each partition can be thought of as being an aggregation at the lowest granularity of the measure group.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



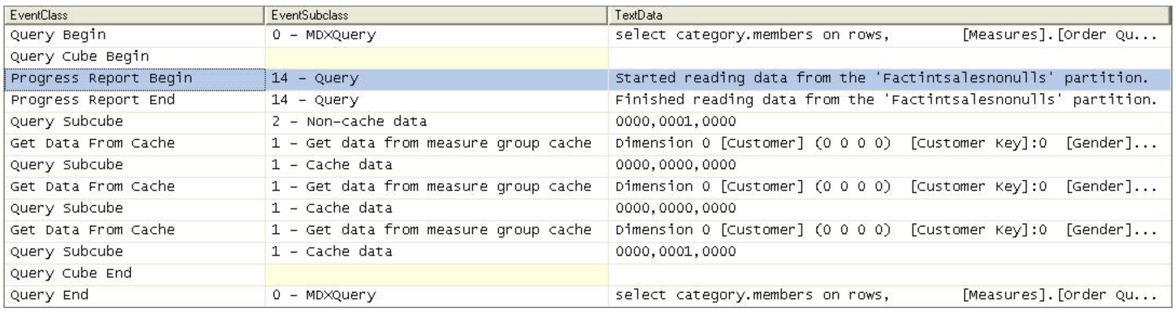
EventClass	EventSubclass	TextData
Query Begin	0 - MDXQuery	select category.members on rows, [Measures].[Ord...
Query Cube Begin		
Get Data From Aggregation		Aggregation c 0000,0001,0000
Progress Report Begin	14 - Query	Started reading data from the 'Aggregation c' aggregation.
Progress Report End	14 - 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 28: 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 29: 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.

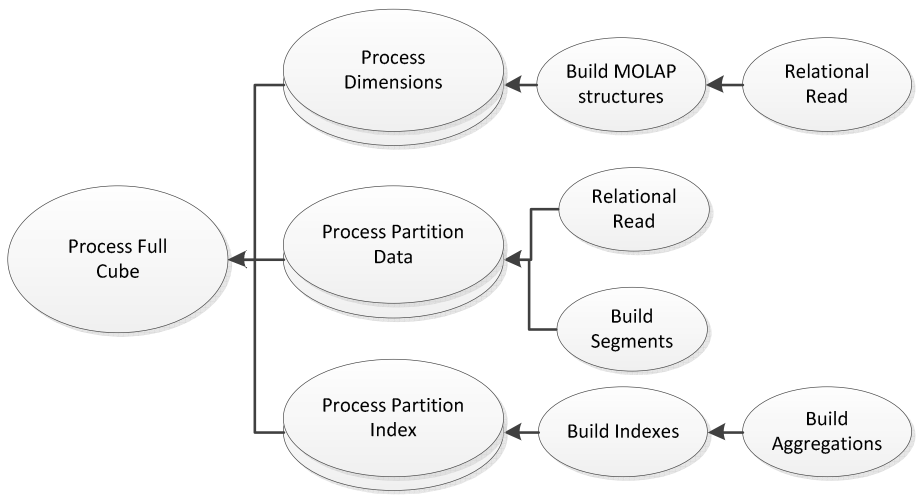


Figure 30: Full cube processing overview

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.

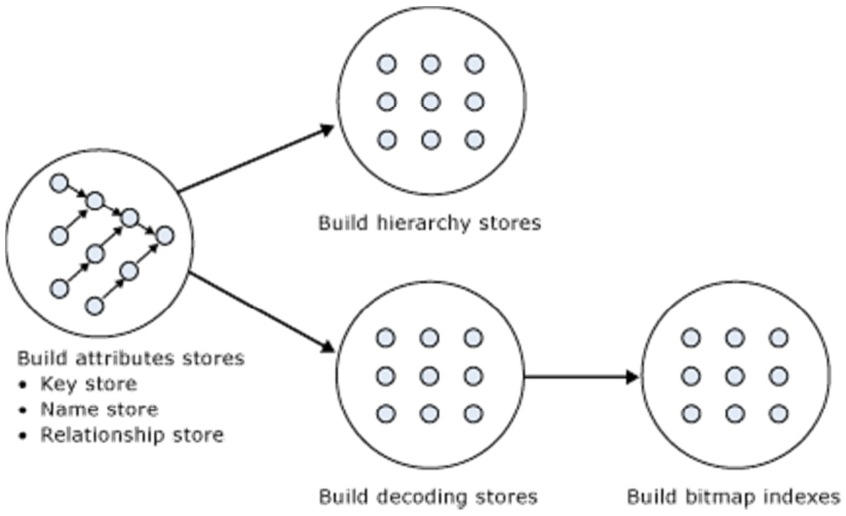


Figure 31: Dimension processing jobs

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 32: 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.

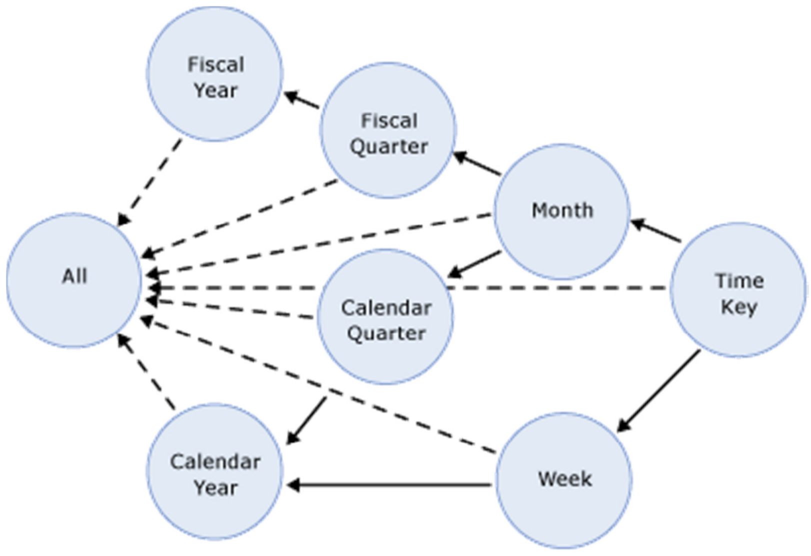


Figure 32: Execution tree example

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.

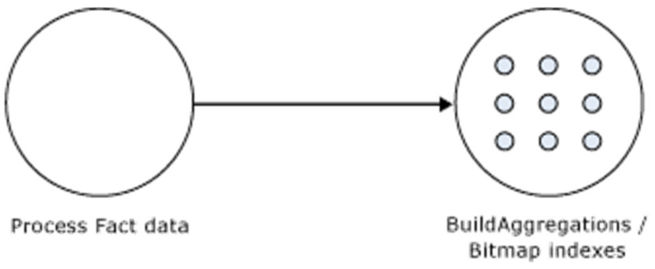


Figure 33: Partition processing jobs

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.

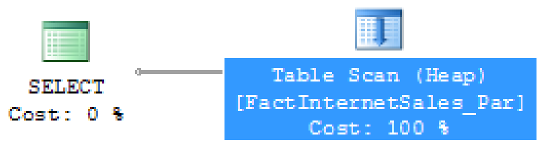


Figure 34: An optimal partition processing query

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

83

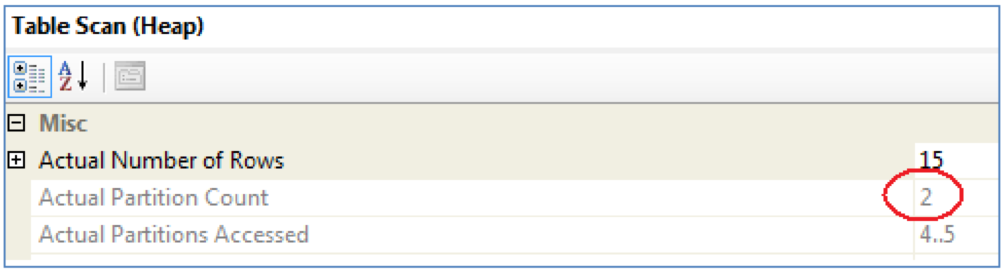


Table Scan (Heap)

⊟ Misc	
⊞ Actual Number of Rows	15
Actual Partition Count	2
Actual Partitions Accessed	4..5

Figure 35: 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

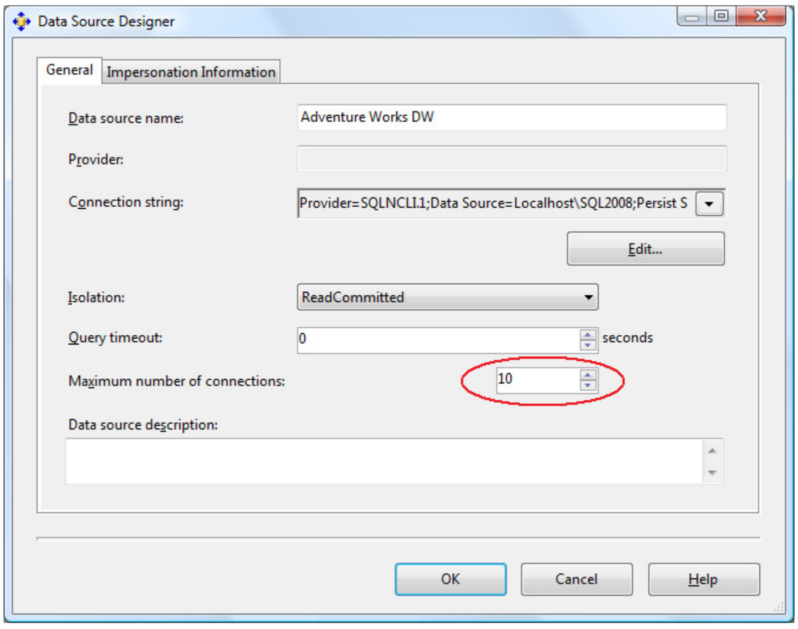
If your tuning is bound only by the amount of CPU power you have (as opposed to I/O, for example), you should optimize to make the best use of the CPU cores available to you. It is time to have a look at the **Processor:Total** counter from the baseline trace. If this counter is not 100%, you are not 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 36: 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.

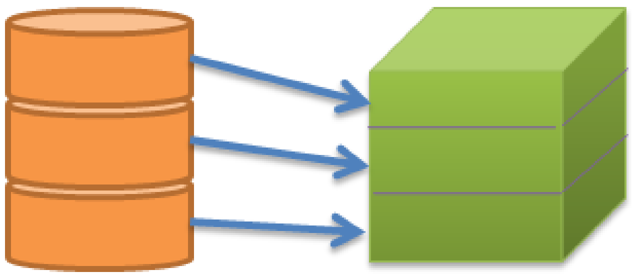


Figure 37: Matching partition strategies

86

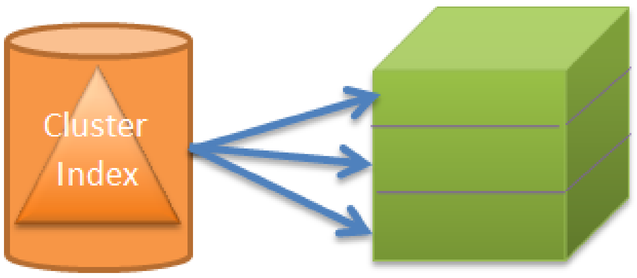


Figure 38: Clustering the relational table

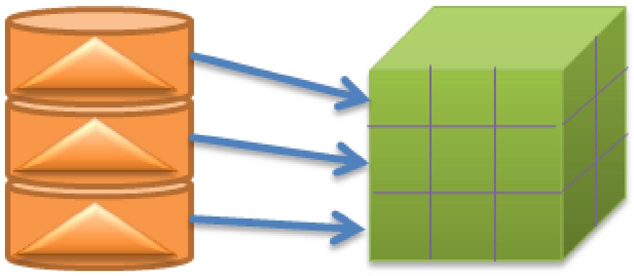


Figure 39: Supporting matrix partitioning with a combination of relational layouts

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)

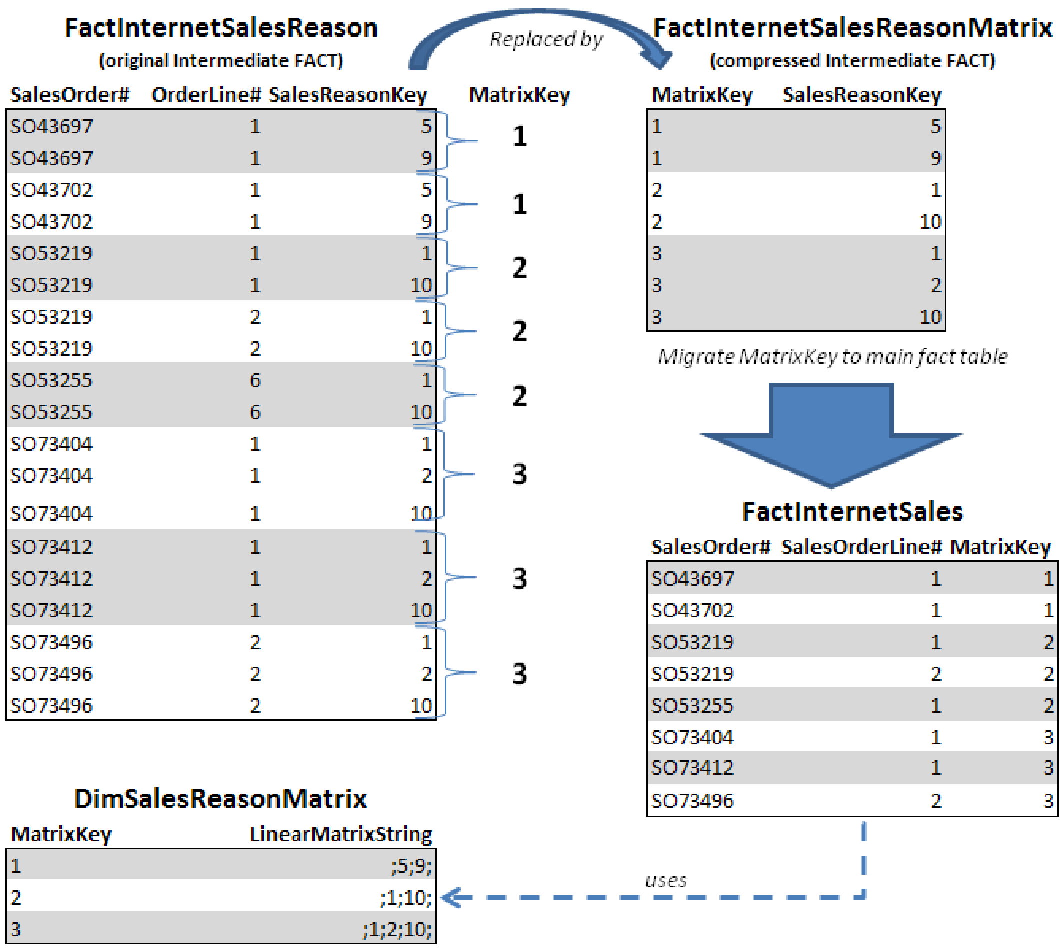
5.2 Large Many-to-Many Dimensions

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	
SO73404	1	2	3
SO73404	1	10	
SO73412	1	1	
SO73412	1	2	3
SO73412	1	10	
SO73496	2	1	
SO73496	2	2	3
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 40: 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.

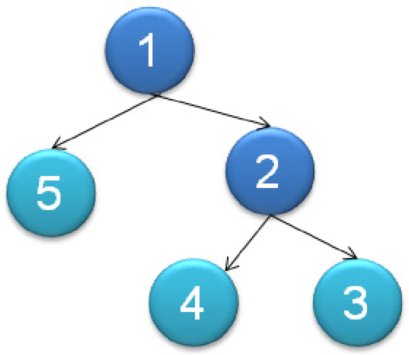


Figure 41: Sample parent-child hierarchy

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).

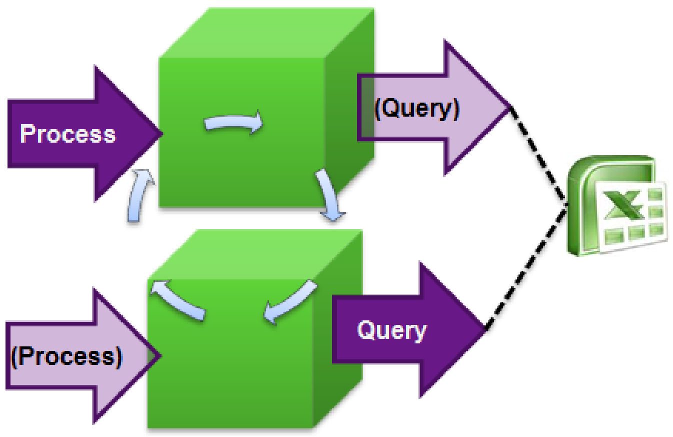


Figure 42: Cube-flipping concept

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).

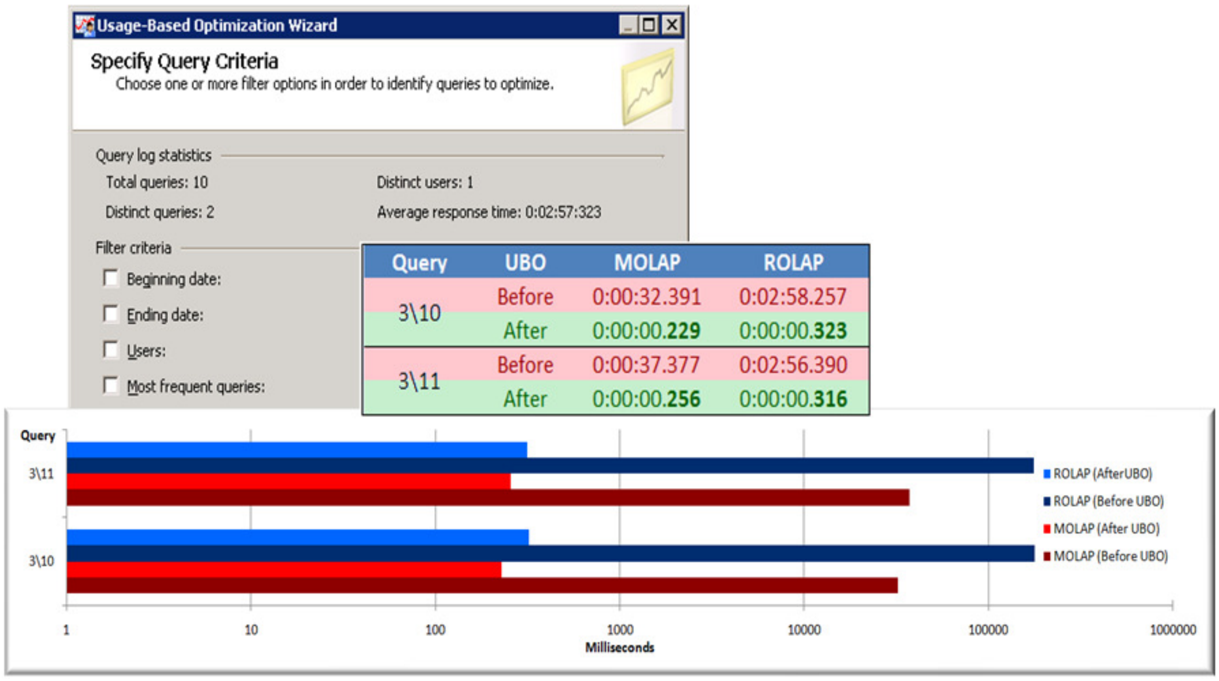
5.4.3 Comparing MOLAP Switching and ROLAP + MOLAP

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 43: 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.

Scale-Out Querying for Analysis Services with Read-Only Databases

SQL Server Technical White Paper

Writer: Denny Lee, Kay Unkroth

Contributors: James Podgorski

Technical Reviewer: Akshai Mirchandani, Murshed Zaman, T.K. Anand

Published: June 2010

Applies to: SQL Server 2008 and SQL Server 2008 R2

Summary: This white paper describes recommended design techniques and methodologies to maximize the performance and scalability of SQL Server 2008 Analysis Services deployments by using read-only query servers.

Copyright

Contents

Executive Summary

Some of the world's largest enterprise servers run Microsoft® SQL Server™ 2008 Analysis Services. Top-end systems, such as HP Integrity Superdome, Unisys ES7000, and IBM x3950 M2 deliver massive processor power and memory capacity for even the most complex, multi-terabyte data warehouses to handle their business-critical workloads with record-setting performance, as well as high reliability, availability, and serviceability.

It makes sense to use top-end server models in a large Analysis Services environment. With all cube cells loaded into memory Online Analytical Processing (OLAP) queries perform most efficiently, query optimizers tend to generate more efficient query plans, communication between worker threads is extremely fast, and latencies are minimal. Yet, Analysis Services cannot use more than 64 logical processors. Only the relational engine of SQL Server 2008 R2 can fully exploit an Integrity Superdome with 64 Intel Itanium CPUs, 256 logical processors, and 2 terabytes of memory running Microsoft Windows Server® 2008 R2.

With a hard limit of 64 logical processors, an Analysis Services server cannot support an unlimited number of concurrent users or queries. Although an optimized cube design and efficient multidimensional expressions (MDX) can help to maximize performance, concurrent queries depleting server resources eventually require a scale-out approach. A common technique is to distribute the workload across multiple dedicated query servers, which has many advantages, yet the downside is an inefficient use of expensive storage resources given that each query server requires a separate database copy.

In a Storage Area Network (SAN) environment, it is possible to mitigate redundancies to some degree. Virtual copy snapshots provide an option to present the same database to each query server by means of a separate logical unit number (LUN). In this way, multiple query servers can read data from a single underlying database folder on a SAN array, yet the query servers still require read/write access to their database snapshots, so each query server's LUN must still be writeable. Read/write snapshots complicate the SAN design and not all SAN systems support this feature, but the read-only database feature of SQL Server 2008 Analysis Services eliminates this requirement. It enables multiple query servers to access the same database concurrently, which can help to save terabytes of storage space without complicated configurations. The shared database LUN only has to be able to sustain the combined input/output (I/O) workloads that the parallel query servers might generate.

SQL Server Customer Advisory Team (SQLCAT) performance tests prove that read-only databases can be a viable option for scale-out querying, both, for formula-engine-heavy queries that primarily process data already in memory as well as storage-engine-heavy queries that involve a scan of the files on disk to perform a calculation. The key in both cases is to optimize the SAN environment for the anticipated levels of random I/O operations.

This white paper contains information for data warehouse architects, database administrators, and storage engineers who are planning to deploy read-only query servers in a SAN-based environment for Analysis Services scalability. This paper assumes the audience is already familiar with the concepts of Storage Area Networks, Windows Server, SQL Server, and SQL

Server Analysis Services. A high-level understanding of Analysis Services optimization techniques for cube processing and query performance is also helpful. Detailed information is available in the SQL Server 2008 Analysis Services Performance Guide at http://sqlcat.com/whitepapers/archive/2009/02/15/the-analysis-services-2008-performance-guide.aspx.

Introduction

Performance and scalability are closely related, but they are not the same. Performance deals with the maximum workload and capacity possible in a system with existing resources. Scalability, on the other hand, describes a system's ability to utilize additional resources for an increase in system capacity to perform additional work. It is important to differentiate between these aspects in Analysis Services environments with multiple query servers.

SQL Server Analysis Services provides an abundance of internal optimizations and performance tuning options to maximize the utilization of available system resources. You can enhance query performance by optimizing the design of dimension attributes, aggregations, cubes, and MDX expressions. You can improve cube processing performance by implementing an economical processing strategy. You can increase server performance by fine-tuning worker threads, pre-allocating memory, and controlling resource allocation. And all of these measures contribute to Analysis Services scalability, yet their primary goal is to increase the performance of the system for a maximum number of users, a maximum number of concurrent queries, a maximum amount of data, and the fastest possible cube loads within a given system capacity. Going beyond maximum achievable performance levels requires additional resources.

Ideally, system capacity increases with each additional resource in a linear or perhaps even better than linear fashion, but this is rarely the case in reality. It is not easy to balance processors, memory, and I/O on a scale-up server. While most server applications today can scale fairly well up to four or eight logical processors, it becomes difficult to obtain linear scalability beyond this point. While it is possible to ramp up Analysis Services on a high-end system with 16 or 32 logical processors by adjusting the size of the thread pools for the formula engine and the storage engine, Analysis Services scalability increasingly levels off. There is bus contention whenever I/O operations occur, Interrupt Service Routines (ISR) can lock a processor, the data has to be in the local CPU cache before a logical processor can use it, and so forth. The system capacity gains on a server with 64 logical processors might still warrant the hardware investment, but this is the absolute limit. Analysis Services cannot utilize more than 64 logical processors.

Note: The relational database engine of SQL Server 2008 R2 can use more than 64 logical processors. In close collaboration with hardware partners and Windows® developers, the SQL Server team worked hard to achieve an almost linear scalability up to 256 logical processors by removing core locks and using finer-grain locks in the system.

Achieving High Scalability with Analysis Services

If Analysis Services scalability increasingly levels off and reaches an asymptotic point at 64 logical processors per server, then this raises the question at what point does a scale-out system that uses many moderate-capacity servers perform better than a scale-up system that uses fewer but more powerful high-capacity servers. Given that scale-out system capacity increases in a more linear fashion than scale-up system capacity and assuming the same total workload, the scale-up system performs better initially, but as the total number of logical processors increases, it is a mathematical certainty that the scale-up and scale-out lines cross, and at this point, the scale-out system begins to outpace the scale-up system. Figure 1 illustrates this aspect conceptually.

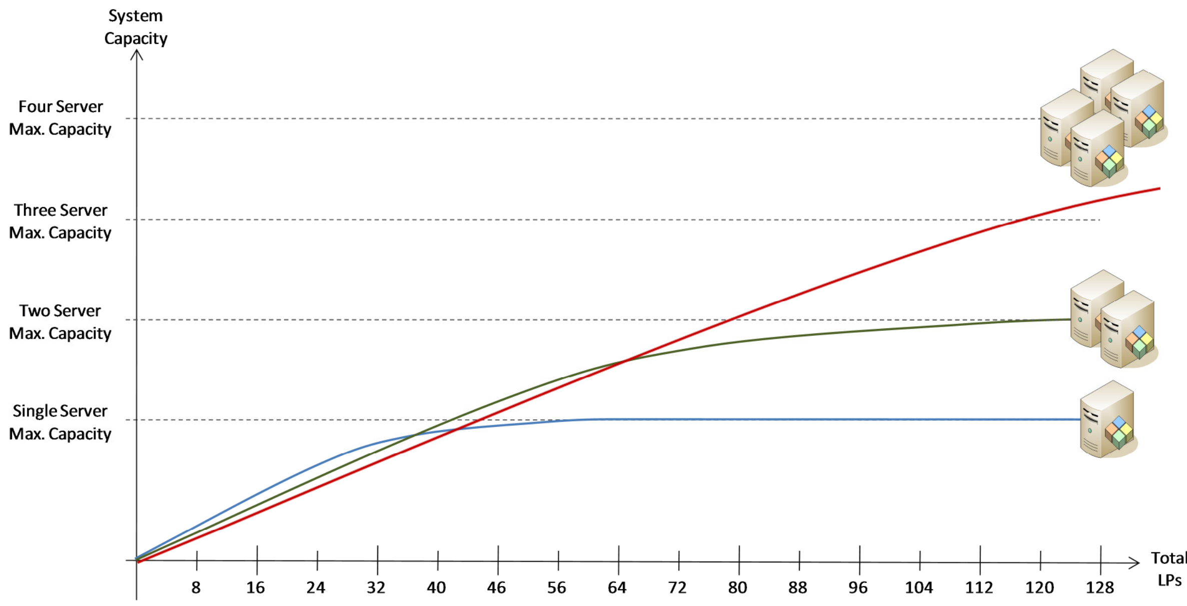


Figure 1: SQL Server Analysis Services Scale-Up and Scale-Out Behavior

Although the exact scalability behavior depends on individual hardware configurations and workload characteristics, Figure 1 shows that at a total of 128 logical processors, four Analysis Services servers with 32 logical processors perform much better than two servers each with 64 logical processors, simply because of the compounded differences in system capacity between less-linear scale-up and more-linear scale-out.

There are two important points to take away from Figure 1:

1. **Performance optimization is important** To realize high scalability with Analysis Services, it is vital to optimize the system configuration for highest possible performance. This includes server tuning as well as SAN storage optimization. The objective is to achieve the most linear scale-up behavior possible. The scale-out then multiplies the scale-up gains.

2. **Balanced scale-out is better than aggressive scale-up** Achieving high scalability requires a balanced approach of scale-up and scale-out. Read-only query servers eliminate the need for an aggressive scale-up strategy to avoid storage redundancies. At the same time, read-only query servers do not eliminate the need for scale-up designs. According to Figure 1, a single system with eight logical processors still performs better than four query servers each with two logical processors.

Note: The high-scalability strategy discussed in this white paper assumes a scale-out with identical query servers. It is difficult to achieve predictable performance in an environment with varying server designs. Microsoft recommends using identical hardware, software versions, and configurations settings.

Analysis Services High-Scalability Architecture

Figure 2 shows an Analysis Services scale-out architecture based on a read-only database for querying. The overall architecture deviates only slightly from the scale-out design discussed in the SQL Server Best Practices Article "*Scale-Out Querying with Analysis Services*," available from Microsoft TechNet at http://technet.microsoft.com/en-us/library/cc966449.aspx.

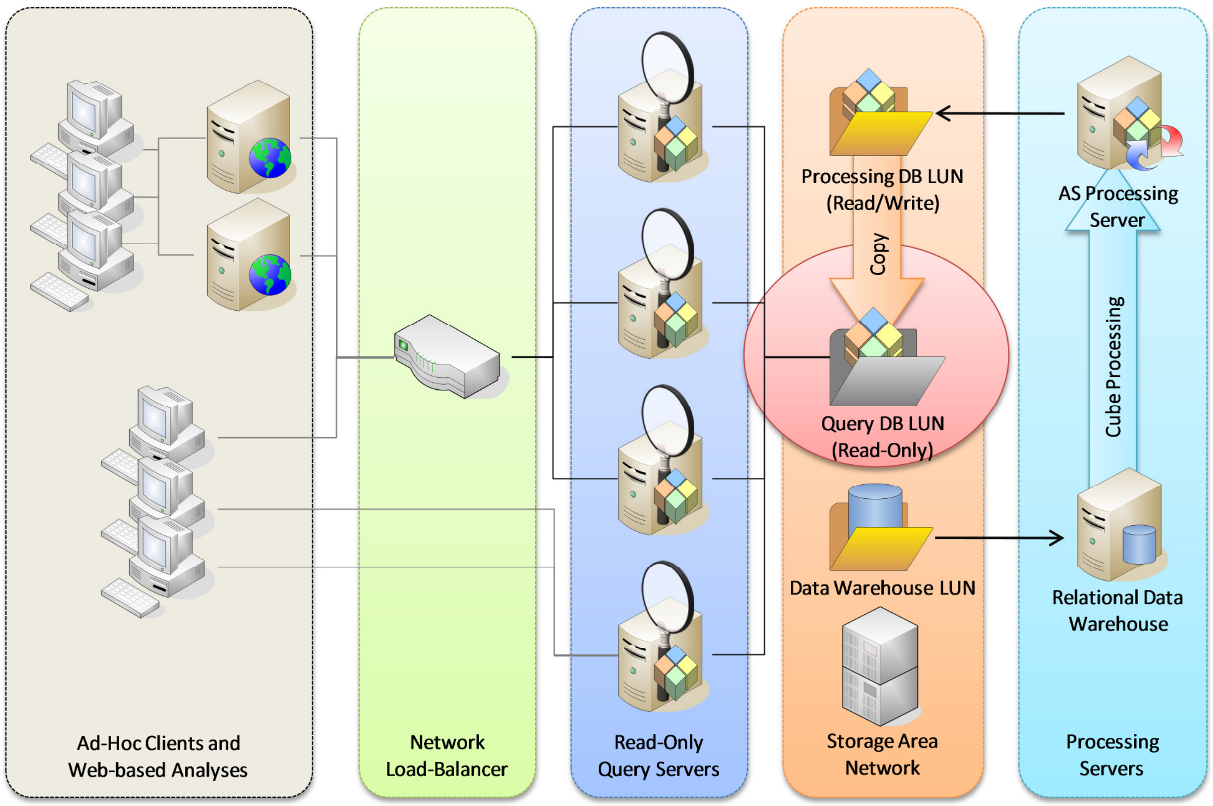


Figure 2: Analysis Services Read-Only Querying Architecture

7

The important innovation in Figure 2 is that all query servers use a single database copy. SQL Server 2005 Analysis Services required a separate database for each query server and elaborate server synchronization techniques. With SQL Server 2008 Analysis Services, it is possible to attach the same database in read-only mode to all query servers.

High-scalability designs based on a read-only database provide the following advantages:

- **Decreased scalability complexity** Because the load-balancer dynamically distributes concurrent users across all available query servers, it is possible to respond to resource depletion due to increased workloads simply by adding further query servers. It is generally easier to scale-out than to scale-up, as long as the SAN environment is able to sustain the total I/O load.
- **Increased cube processing flexibility** Given that there are separate servers for cube querying and cube processing, the processing server does not need to wait for long-running queries to complete before it can obtain an exclusive lock on the database. Consequentially, the processing server has a much larger processing window and can update cubes more frequently, and any processing errors do not affect cube availability on the query servers.
- **Server maintenance without availability impact** Network load-balancers direct clients to an available query server automatically, so individual server downtime does not affect overall Analysis Services availability. Provided that the cube uses Multidimensional OLAP (MOLAP) storage mode, it is possible to shut down the relational data warehouse, the processing server, and individual query servers for maintenance while the remaining query servers continue to service clients and applications.
- **High cube availability** Load-balanced query servers make it possible to update cubes with minimal interruption of business processes. The key is to exclude a subset of query servers from network load balancing so that these servers do not receive new query requests, attach the updated database to these servers, and then add these servers back into the load-balancing cluster, and after that exclude the remaining query servers to repeat this process.
- **Separation of clients** Ad-hoc clients and Web-based applications can access the same Analysis Services database through different sets of query servers. In this way, users generating long-running queries in Microsoft Office Excel® or other ad-hoc tools do not interfere with Web-based analysis and reporting solutions.

Note: The actual number and ratio of servers in a high-scalability design depends on specific query and processing requirements. For example, Microsoft does not necessarily recommend or not recommend using separate data warehouse and processing servers. It is important to perform requirements analysis, planning, and testing in each environment individually.

Distributing Workloads across Query Servers

It is relatively uncomplicated to distribute Analysis Services clients across multiple query servers. One option is to distribute clients statically by pointing the *Source* property in each connection string to a different Analysis Services instance. A disadvantage of this approach is that there is no load balancing for connections within a single application and no automatic redirection to an available query server if the specified server is offline.

The alternative to static load balancing is network load balancing, which distributes the load dynamically by performing network address translation (NAT) between a virtual server IP address and the IP addresses of the actual query servers. The server name that clients use in their connection strings must resolve to the virtual IP address, which points to the load-balancer to intercept the network traffic. However, the load-balancer cannot simply distribute the client requests across all query servers in a round-robin fashion. It must route all requests that belong to the same client session to the same query server.

XMLA Sessions and Network Load Balancing

Figure 3 illustrates the Analysis Services communication architecture. The most important point is that the clients always communicate through XML for Analysis (XMLA), which is a transport-neutral communication method that can use Simple Object Access Protocol (SOAP) over Direct Internet Message Encapsulation (DIME) or over Hypertext Transfer Protocol (HTTP). While DIME enables clients to communicate with an Analysis Services instance over long-lived TCP/IP connections, HTTP connections are always short-lived. The client connects, sends the request, receives the response, and disconnects. As a consequence, clients cannot rely on the underlying transport to maintain session state. If an operation requires session context, XMLA clients must maintain their own sessions on top of the transport.

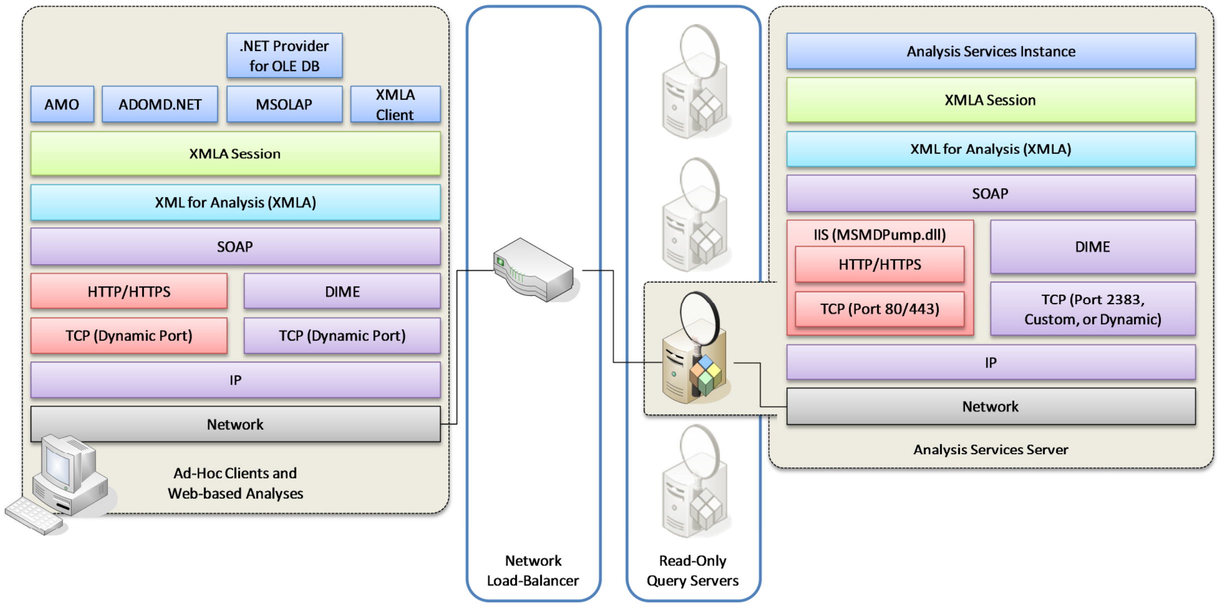


Figure 3: Client/Server Transports and XMLA Sessions

9

The XMLA session behavior is significant for network load balancing. To give an example, the CREATE SUBCUBE statement requires an XMLA session so that the Analysis Services instance can apply the same filter to all subsequent MDX queries, which can span multiple network connections. The client sends a *BeginSession* element in the SOAP header of the CREATE SUBCUBE command to the Analysis Services instance, which returns a *Session* element that identifies the new session, and then the client includes this *Session* element in the SOAP header of all subsequent commands so that the Analysis Services instance can apply the correct filter, until the client finally ends the session by sending the DROP SUBCUBE command to the server, which contains the *EndSession* element in the SOAP header. In short, for this session handling to work, the load-balancer must direct all client requests to the same query server because only this one query server owns the session. Query servers do not share XMLA session information. Directing client connections within a session to a different query server inevitably leads to communication problems.

In order to ensure XMLA session persistence over short-lived connections, the network load-balancer must support cookie-based affinity or source-IP-address affinity. Cookie-based affinity is superior to IP-based affinity because reverse proxies and other NAT devices can hide the actual client IP addresses behind a single proxy IP address, which prevents the load-balancer from distinguishing the client connections, so all clients would end up at the same query server. Cookie-based affinity avoids this issue by enabling the load-balancer to identify each client based on a cookie. Initially, the load-balancer uses round robin to select a query server for the first HTTP-based request and then inserts a cookie into the HTTP response from this server. The client receives the cookie along with the HTTP response and then sends the cookie back in all further HTTP requests. In this way, the load-balancer can recognize the client and direct the client to the correct query server. However, cookie-based affinity only works if the client uses HTTP and supports cookie management. Thin XMLA clients that make SOAP calls directly might require IP-based affinity.

Note: By default, SQL Server Analysis Services compresses and encrypts the XMLA communication, which prevents network devices from analyzing XMLA session details to make sophisticated load-balancing decisions. Cookie-based affinity and source-IP-address affinity work because these methods do not require an inspection of the Analysis Services payload.

Load Balancing Web-Based Analytics Applications

Affinity-based load balancing can help to distribute connections from multiple clients across multiple query servers, but it is difficult to distribute multiple requests from a single client, especially if the load-balancer uses IP-based affinity. This can be an issue for large Web-based applications because the Web server only uses a single source IP address associated with the network adapter for outbound communication. Accordingly, the load-balancer selects a query server based on round robin for the first request, but then continues to direct all traffic from this Web server to the same query server regardless of the availability of other query servers in the load-balancing cluster.

The best way to ensure scale-out efficiency for large Web-based Analysis Services applications is to deploy multiple, load-balanced Web servers and a proportionate number of query servers, as illustrated in Figure 4. The load-balancer in front of the Web servers distributes the browser workload. The load-balancer that sits between the Web servers and the query servers primarily ensures high availability by maintaining a list of online and offline query servers and directing the requests to an available server.

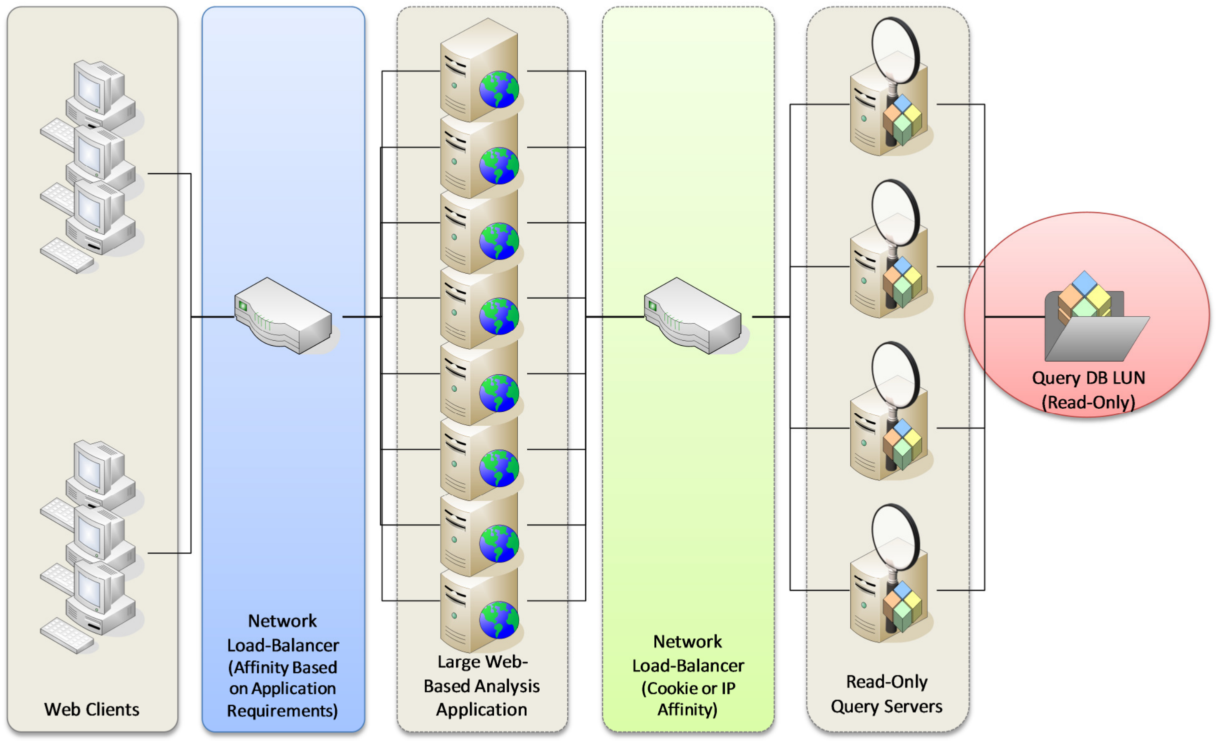

Web Clients
Network Load-Balancer (Affinity Based on Application Requirements)
Large Web-Based Analysis Application
Network Load-Balancer (Cookie or IP Affinity)
Read-Only Query Servers
Query DB LUN (Read-Only)

Figure 4: Load Balancing Web Servers and Query Servers

Sharing and Updating an Analysis Services Database

With multiple query servers in a load-balancing cluster, the next important consideration is how to share a single Analysis Services database. This includes two requirements. Most obviously, all query servers require access to the same database, but it also requires updating the query database after each processing cycle. A subtle issue is that the shared database volume must be write-protected for multiple query servers to access the volume without causing data corruption, yet it is not possible to update a database on a read-only volume. A SAN configuration with multiple LUNs provides a solution.

Note: A processing server cannot directly share a database with read-only query servers even if the SAN has its own distributed file system and lock manager to support concurrent write access because the processing server locks the database in read/write mode.

Database LUN Arrangement

Figure 5 shows a sample SAN configuration to share and update an Analysis Services database. This configuration includes three database LUNs hosted in one or multiple storage enclosures that connect to a common fabric. All query servers and the processing servers can theoretically access all database LUNs. Typically, each server has multiple host bus adapters (HBA) with preferred and alternate storage paths for fault tolerance. For a real-world, enterprise-scale SAN configuration example, refer to the technical white paper "*Accelerating Microsoft adCenter with Microsoft SQL Server 2008 Analysis Services*" available at http://sqlcat.com/whitepapers/archive/2009/09/19/accelerating-microsoft-adcenter-with-microsoft-sql-server-2008-analysis-services.aspx.

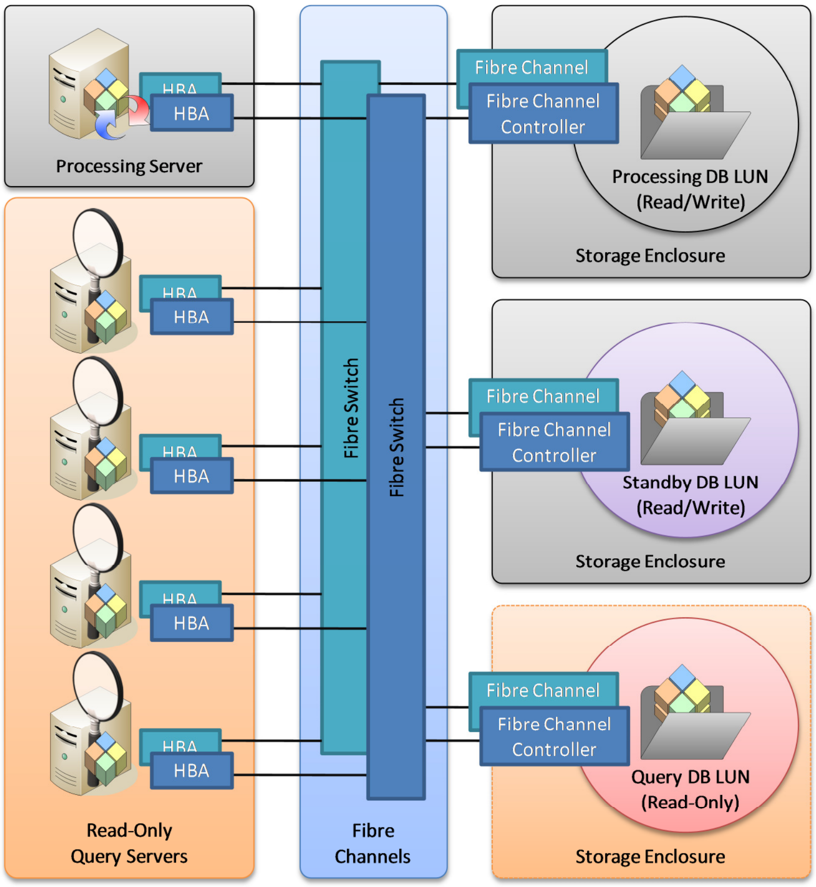


Figure 5: Sharing an Analysis Services Database in a SAN Environment

Figure 5 illustrates a SAN configuration that connects processing server and query servers to three database LUNs as follows:

- **Processing server** The processing server exclusively owns the Processing DB LUN. This LUN hosts the read/write copy of the database that Analysis Services refreshes during each processing cycle. The processing server also has read/write access to an optional Standby DB LUN, which is the target of the database copy after each processing cycle.

- **Query servers** The LUN masks on the storage controllers exclude the Processing DB and Standby DB LUNs as targets for query servers, but include the Query DB LUN, which hosts the read-only version of the Analysis Services database. Because multiple query servers cannot have write access to the same LUN, it is important to configure the Query DB LUN as a read-only volume at the file-system level.

Note: The Standby DB and Query DB LUNs do not host the DataDir folder. Each query server must have its own DataDir folder on a different drive. Given that read-only query servers with sufficient memory capacities do not maintain databases in the DataDir folder, the drive that contains the DataDir folder has no special capacity or performance requirements.

Database Refresh Cycle

The Standby DB LUN in Figure 5 is optional, but highly recommended. Without this LUN, it is necessary to shut down or disconnect all query servers prior to updating the read-only database directly on the Query DB LUN. This implies that the Analysis Services query environment is entirely unavailable for the duration of the database refresh process. Another important advantage of the Standby DB LUN is flexibility to accommodate an increase in I/O demand over time. In an expanding Analysis Services environment, it is possible to optimize the design of the Standby DB LUN without affecting business processes. The new design takes effect when swapping the Standby DB LUN and the Query DB LUN as part of the regular database refresh cycle, as illustrated in Figure 6.

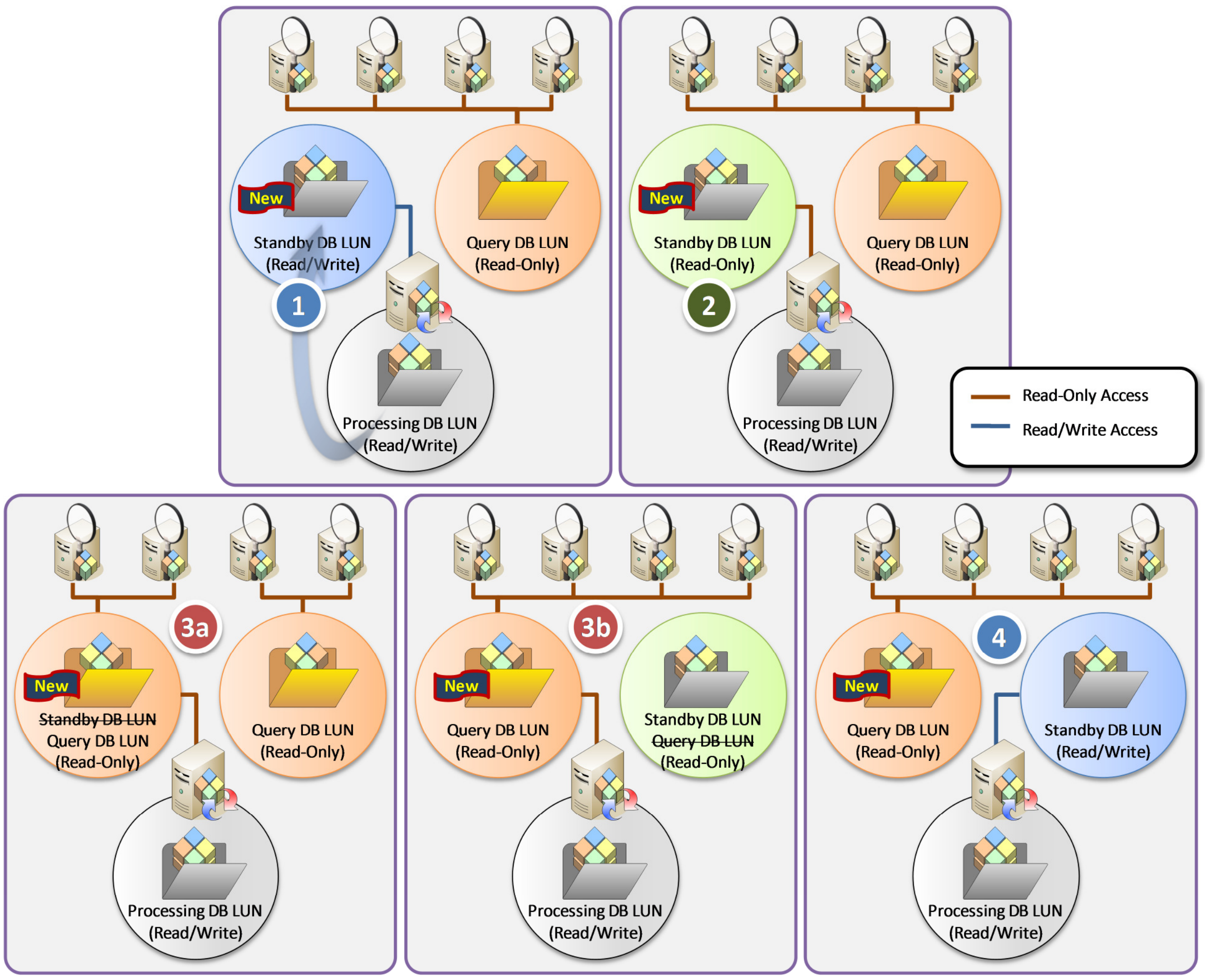


Figure 6: Database Update by using a Standby DB LUN

The SAN configuration with a Standby DB LUN supports the following procedure to update a read-only database shared between multiple query servers:

1. **Copying the updated database** The processing server updates the cube's dimensions and facts as usual. After all cube processing finishes, it is possible to detach the database from the processing server, copy it to the Standby DB LUN, and then reattach the database on the processing server.

2. **Write-protecting the standby volume** Before swapping Standby DB and Query DB LUNs on query servers, it is necessary to configure the standby NTFS partition as a read-only volume by using the Diskpart command-line tool that ships with Windows Server 2008. Run **diskpart.exe**, type **list volume** to determine the correct volume number, then type **select volume *<volume number>*** to select the standby volume, and then use **attributes volume set readonly** to write-protect the standby volume.

3. **Swapping databases on query servers** Now that the standby volume is write-protected, you can swap the standby and query databases. Exclude the desired query

server from load-balancing and, after draining all users, detach the current read-only database, dismount the Query DB LUN, mount the Standby DB LUN instead, and then attach the new database copy in read-only mode. If you manually attach the database in SQL Server Management Studio, make sure you select the **Read-only** check box in the **Attach Database** dialog box. For automated methods, refer to the SQL Server 2008 Books Online topic "*Switching an Analysis Services database between ReadOnly and ReadWrite modes*" at http://msdn.microsoft.com/en-us/library/cc280661.aspx.

4. **Swapping LUNs on the processing server** Upon completion of the LUN swap on all query servers, dismount the Standby DB LUN on the processing server, and then mount the previous Query DB LUN and enable write access so that the previous Query DB LUN can now act as the Standby DB LUN. Run **diskpart.exe**, type **list volume** to determine the volume number, type **select volume *<volume number>*** to select the volume, and then type **attribute clear readonly** to remove the read-only flag.

Note: The Standby DB LUN facilitates database updates, offers design flexibility, and helps to increase cube availability, yet it also imposes additional storage costs. It is a beneficial component, but not a strict requirement for a scale-out design.

Database Copying

The SQL Server Best Practices Article "*Scale-Out Querying with Analysis Services*" describes various techniques to copy an offline Analysis Services database. In a scale-out environment based on a read-only database, the best choices are as follows:

- **SAN storage cloning** Enterprise-scale SAN solutions support various storage replication capabilities, which can be used to create a copy of the Processing DB volume. At a high level, the storage replicas fall into categories of snapshots or clones. A snapshot defines an original dataset and then tracks any changes in a separate storage location so that the changes can be rolled back if necessary. Snapshots are not a good choice for database copying because of noticeable performance penalties and the dependencies on the original dataset. A clone, on the other hand, is an exact duplicate of the original LUN, offering an efficient method to copy an offline Analysis Services database. Similar to disk mirroring, SAN-based cloning is almost instantaneous. Note, however, that clones might inherit file system issues that can cause performance degradation over time, specifically disk fragmentation. This is particularly an issue with older SAN systems. If your SAN does not support volume defragmentation, you must manually defragment the Processing DB volume after cube processing for best performance. The actual cloning mechanism depends on the specific SAN system. Front-line operators require special training to perform LUN cloning correctly.

- **Robocopy** A viable alternative to LUN cloning is the straightforward copying of database files at the file system level. This approach requires no SAN administration. It is easy to reformat the target volume prior to file copying, a simple script can automate all relevant tasks, and front-line operators do not require any special skills. The technical article "*Sample Robocopy Script to Synchronize Analysis Services Databases*" describes an efficient method. The article is available at http://sqlcat.com/technicalnotes/archive/2008/01/17/sample-robocopy-script-to-customer-synchronize-analysis-services-databases.aspx.

Optimizing an Analysis Services High-Scalability Design

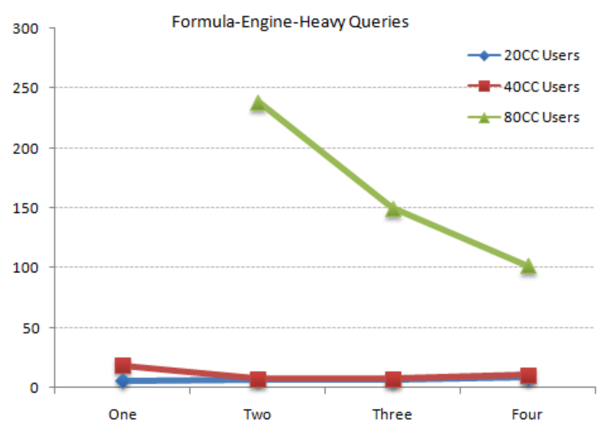
As mentioned in the beginning of this white paper, it is vital to optimize the Analysis Services design to realize high scalability with cost efficiency. This includes query optimization, server tuning, and storage optimization. A suboptimal query performance requires more query servers to support the desired number of concurrent users and places higher demand on the storage subsystem than necessary. A suboptimal storage design can prevent the Analysis Services environment from reaching the desired query performance regardless of the number of query servers.

In order to achieve high scalability with cost efficiency, perform the following optimization tasks:

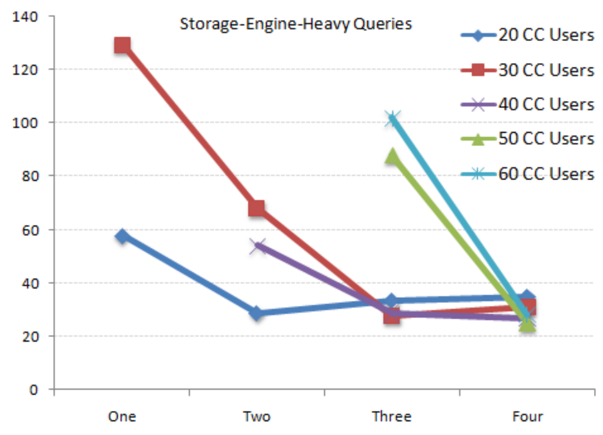
1. **Query Performance Optimization** The primary purpose of this step is to process individual queries as efficiently as possible. Optimizations can include redesigning the cube's dimensions, aggregations, and partitions as well as rewriting MDX expressions. An important outcome of this step is a reproducible performance baseline, which helps to evaluate the processing efficiency of each individual query server.
2. **Server Tuning** The purpose of this step is to tune CPU, memory, and I/O capacity on a single server to support as many concurrent users and queries as possible without degrading query performance in comparison to the performance baseline established previously. An important outcome of this step is a reproducible I/O workload baseline, which helps to estimate the overall I/O performance requirements to support the total number of concurrent users and queries across all query servers.
3. **I/O Performance Optimization** The purpose of this step is to design the SAN storage subsystem according to the combined I/O throughput and performance requirements of all query servers combined. An important outcome of this step is a Redundant Array of Independent Disks (RAID) configuration for the database LUNs and a correctly configured SAN environment.

Establishing Query Performance Baselines

Figure 7 illustrates the scale-out behavior of SQL Server 2008 Analysis Services for two test cases based on actual customer data. The first test case relies on formula-engine-heavy queries and inexpensive query plans. On average, query processing completes in less than ten seconds because the formula engine can reuse data that is already in memory. The second test case uses storage-engine-heavy queries and expensive query plans. This cube includes a large percentage of many-to-many dimensions and roughly 30 distinct count measures that require a scan of the files on disk each time to calculate the results. Accordingly, queries take approximately 30 seconds to complete under normal circumstances. In both cases, the test results show that the scale-out design can accommodate increased workloads, but it does not help to improve the individual performance baselines. The storage-engine-heavy test case is a good candidate for MDX query optimizations. For most detailed recommendations, refer to the SQL Server 2008 Analysis Services Performance Guide at http://sqlcat.com/whitepapers/archive/2009/02/15/the-analysis-services-2008-performance-guide.aspx



	Query Servers			
CC Users	One	Two	Three	Four
20	5.938218	6.70625	6.62531	8.699035
40	18.41792	7.569751	7.674358	10.78396
80		238.5635	150.3148	101.917



	Query Servers			
CC Users	One	Two	Three	Four
20	57.88929	28.80727	33.26458	34.70754
30	129.578	68.01255	27.71317	30.93606
40		54.24718	28.50949	26.79812
50			88.178	25.12548
60			102.0223	28.35425

Figure 7: Analysis Services Scalability Results

Note: Empty table cells in Figure 7 signify unobtainable test results, meaning the concurrent workload overtaxed the capabilities of the corresponding test environment. All displayed results refer to average query times.

As part of query optimization, it is important to establish performance baselines by using representative user queries extracted from SQL Server Profiler traces. These baselines build the foundation to measure scale-out efficiency. In the example of Figure 7, it is possible to recognize that the formula-engine-heavy environment does not support 80 concurrent users very well, while the storage-engine-heavy environment has reached scale-out efficiency. Although the storage-engine-heavy environment has a much higher I/O load than the formula-engine-heavy environment, its storage subsystem is able to support the load of 60 concurrent users. The formula-engine-heavy environment, on the other hand, shows a bottleneck. Even though the I/O load is generally low, 80 concurrent users seem to overwhelm the current storage subsystem and so query performance suffers. If two query servers can support 40 users with query times of eight seconds, then four servers should be able to support 80 users with about the same performance. Instead, query times exceed 100 seconds. It is predictable that adding further query servers will not bring the query time down to 10 seconds. It is necessary to review the scale-out design and locate the actual bottleneck.

To understand why four servers for 80 users are not equivalent to two servers with 40 users in the formula-engine-heavy environment, it is important to keep in mind that even formula-engine-heavy queries must first retrieve the data via the storage engine. As Figure 8 reveals, the initial set of queries flooded the storage subsystem with read requests. Subsequent queries did not have to read data from disk and finished much faster.

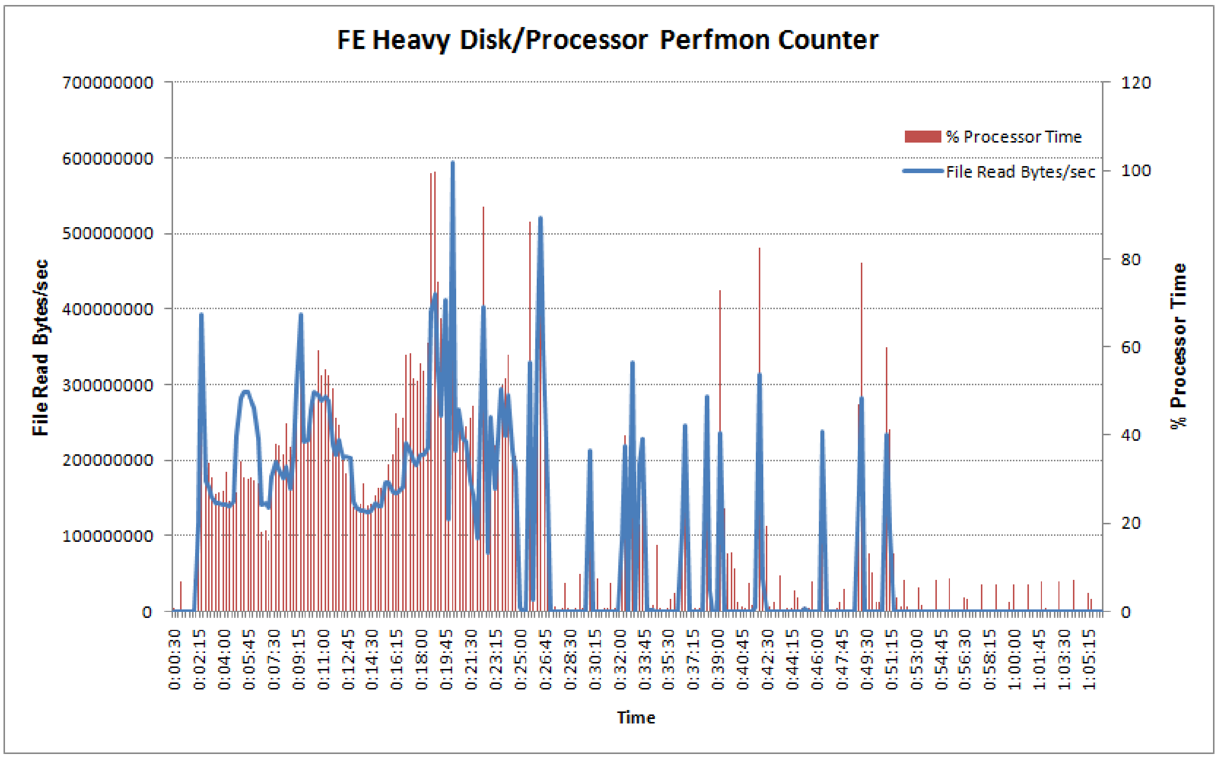


Figure 8: I/O Load Pattern in a Formula-Engine-Heavy Environment

Server Tuning and I/O Workload

Server tuning is an important aspect of query optimization for multiple-user processing performance. The goal is to exploit all available processor and memory resources to scale up to the maximum number of users before scaling out. Both, scale-up and scale-out help to increase overall query parallelism, but the scale-up offers a higher I/O efficiency due to better memory utilization. In the context of high scalability, the primary goal of server tuning is to minimize the I/O workload while maximizing parallelism on each individual query server.

Figure 9 illustrates the Analysis Services system architecture, which shows that a number of hardware, operating system, and Analysis Services components must work together on a query server to handle client connections, manage XMLA sessions, process MDX queries, and return meaningful results. The point is that all these components consume processor cycles and memory, but not all generate I/O activity, and those that do generate I/O include optimizations to reduce the footprint. The TCP/IP stack can automatically tune its window sizes to receive more data per network I/O and a variety of other components, including the formula engine, storage engine, file system, and storage hardware, maintain cache memory to service frequently accessed data without having to repeat the corresponding I/O requests.

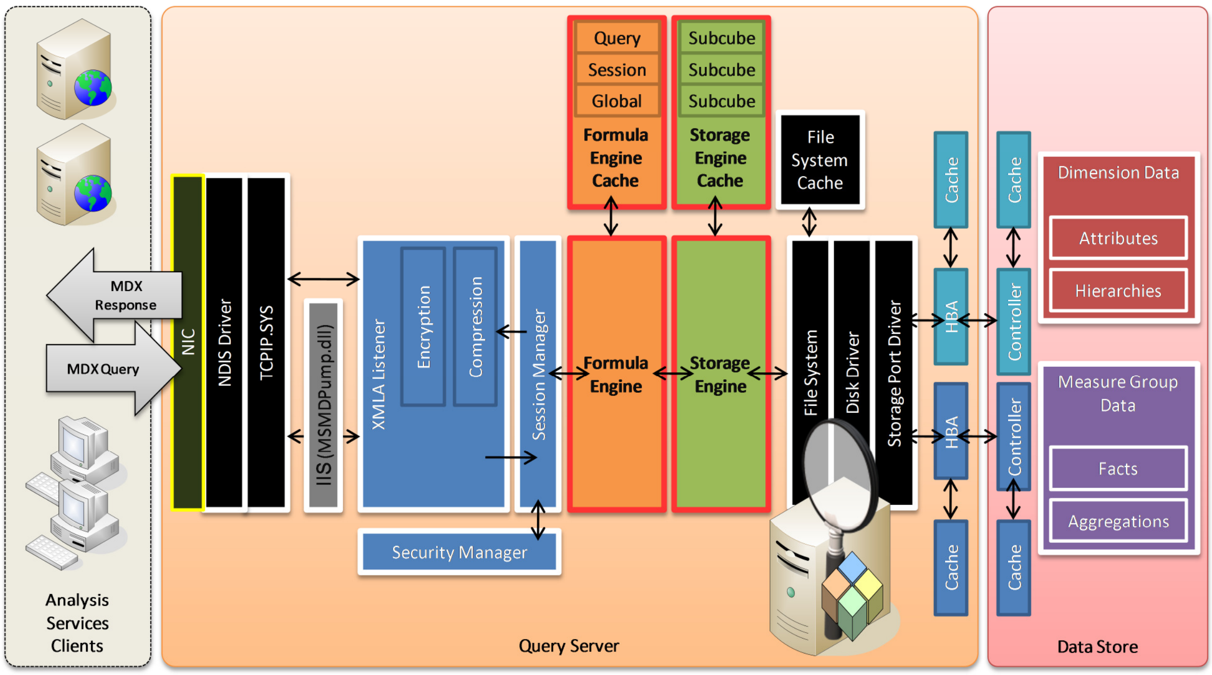


Figure 9: Analysis Services System Architecture

Caching is important for scale-out scenarios. Following a system restart or an update of the read-only database, the server is cold and I/O load is high because the storage engine must translate the formula engine's subcube requests into I/O requests to retrieve the cube data from disk, yet as the server warms up, I/O load goes down due to caching and settles at a lower level as the server reaches the normal operational state. The SQL Server 2008 Analysis Services Performance Guide describes various techniques to warm up a query server quickly. Cache warming methods, such as pre-executing queries or running dedicated cache-warming queries, combine small but time-consuming I/O requests into one larger request. The storage engine retrieves the data once and then returns all future results from cache.

On a warm query server, CPU utilization continues to stay at 80-100 percent because of query processing, but the performance counters *% Disk Read Time* and *Avg. Disk Read Queue Length* go down and settle at a low level as the server reaches the normal operational state. This is the ideal scale-out behavior because low disk utilization per query server during normal operation implies that a given storage subsystem can support more query servers. As long as the storage subsystem has excess performance capacity, additional query servers can help to deliver an increased processing power to support the desired total number of concurrent users and queries.

Storage and I/O Requirements

The server tuning exercises provide a clear understanding of the maximum number of concurrent queries that an individual query server can handle and the corresponding I/O workload in terms of *Disk Transfers/sec*, *Disk Read Bytes/sec*, and *Disk Queue Length*. These characteristics of a single server then help to determine the overall I/O throughput and performance requirements for the desired total number of users.

For example, if a single server generates 500 I/O requests per second (IOPS) to support 20 concurrent users during normal operation, then it is reasonable to assume that four servers generate 2000 IOPS to support 80 concurrent users. The storage subsystem must deliver the required I/O bandwidth and disk performance for all servers and processes combined. Figure 10 illustrates the relationship between processing capacity, I/O throughput, and disk performance in a SAN-based Analysis Services environment.

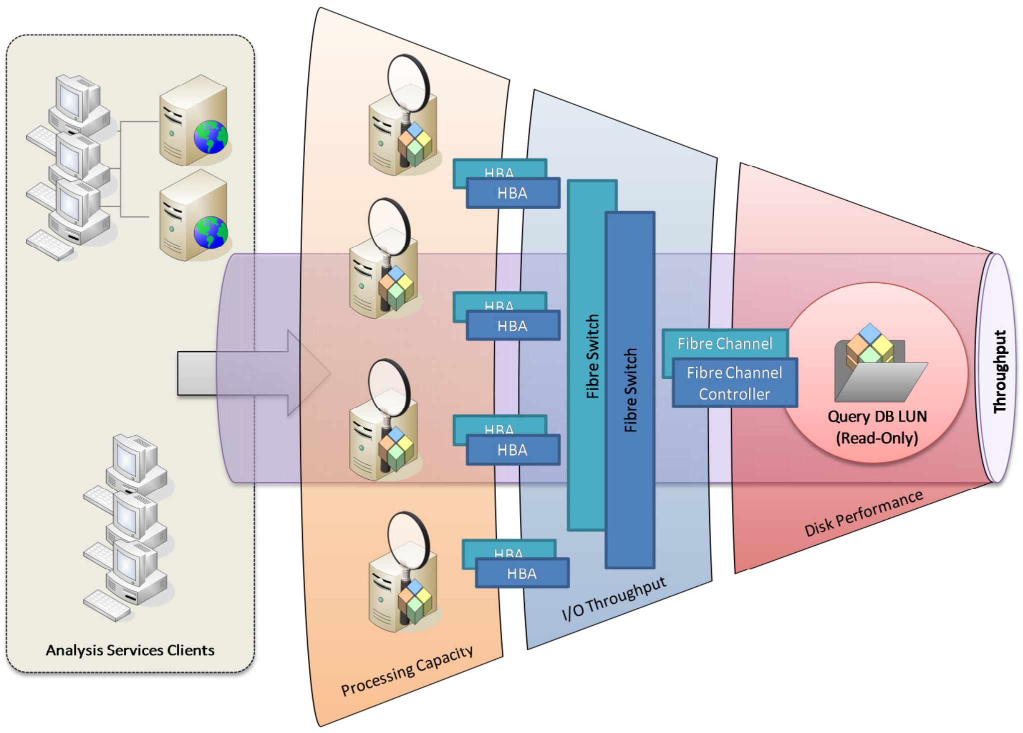


Figure 10: Scalability Constraints in an Analysis Services Scale-Out Design

Read-Only Database Storage Design

For storage engineers, an important advantage of read-only databases is simplicity in the disk layout. There are no write activities to consider in I/O calculations. As a consequence, cost-efficient RAID configurations with a high write penalty, such as RAID 5, remain a viable option. In fact, SQLCAT observed a slightly better read-only performance with RAID 5 in comparison to RAID 10. Ignoring the benefits of the controller cache, the calculation is very straightforward and independent of RAID-type factors: *Number of Disks = Required Total IOPS / IOPS per Disk*.

For example, assuming a total I/O load of 2000 IOPS, twelve disks (15000 rpm, 180 IOPS) can deliver the required performance in a RAID 5 configuration. Assuming 450 gigabyte disks, this RAID 5 array provides roughly 4.8 terabytes of usable capacity (450*[12-1]/1024=4.83), which corresponds to a performance-bound storage design. If the database requires more capacity, further disks must be added according to the capacity-bound design requirements. On the other hand, if the cube size is relatively small and the size of each individual disk large, then it can be an option to short-stroke the disks, as illustrated in Figure 11. Short-stroking is a common technique to reduce disk seek time by limiting the physical range used on the disk platters for data storage. By leaving a sizeable portion of the platters unused, the disk heads do not need to move to distant regions, which benefits I/O performance at the expense of sacrificed storage capacity. In the example of Figure 11, six short-stroked 450GB disks can deliver the required 2000 IOPS with 1 terabyte of useable capacity.

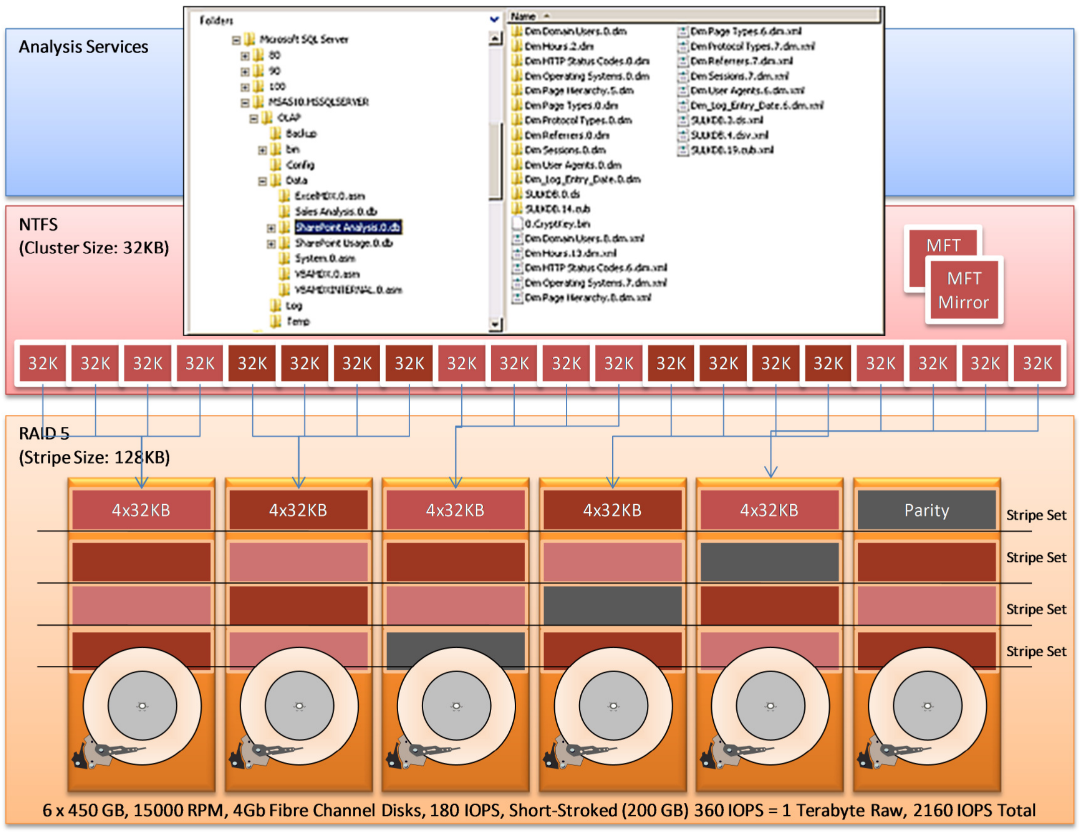
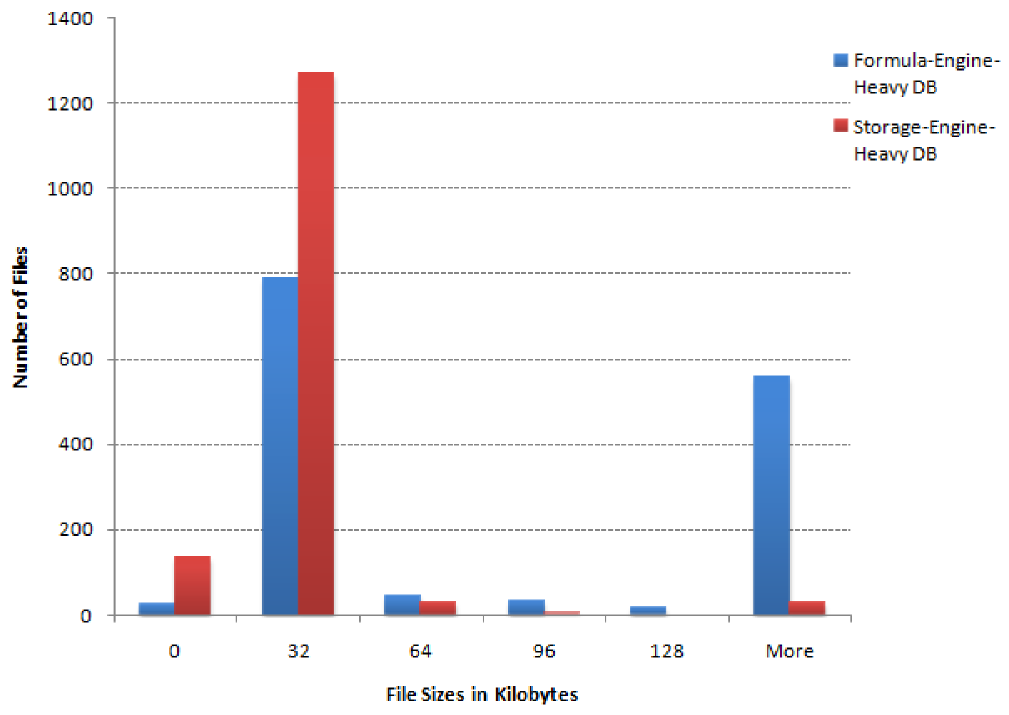

Analysis Services
NTFS
(Cluster Size: 32KB)
MFT
MFT Mirror
32K
RAID 5
(Stripe Size: 128KB)
4x32KB
Parity
Stripe Set
6 x 450 GB, 15000 RPM, 4Gb Fibre Channel Disks, 180 IOPS, Short-Stroked (200 GB) 360 IOPS = 1 Terabyte Raw, 2160 IOPS Total

Figure 11: A Read-Only Database Storage Design with Short-Stroked Disks

Note: Do not repurpose unused disk capacity to provision LUNs for other applications. Microsoft does not recommend sharing physical disks between server applications. If multiple applications with different I/O patterns share the same physical disks, unpredictable performance behavior occurs. Note also, however, that this design recommendation only applies to disk drives. Solid state drives (SSD) and enterprise-class flash drives (EFD) eliminate this design constraint. SQLCAT plans to cover SSD-based storage designs in a future white paper.

Block Sizes and Stripe Sizes

The allocation unit refers to the smallest storage block or cluster of disk space that NTFS can use to store file data. For relational SQL Server databases, it is a standard recommendation to use a 64KB Allocation Unit size because the relational engine reads data in 64KB extents. Yet, this recommendation does not apply to Analysis Services. It is important to note that the storage engine of Analysis Services does not read data in extents of eight 8KB data pages. Instead, an Analysis Services database corresponds to a file structure with a very large number of relatively small files for dimension keys, hash tables, and hierarchical information as well as larger files for facts and aggregations. The majority of the files are less than 32KB in size, many are even smaller than 1024 bytes, and yet others can exceed several gigabytes. Figure 12 shows a detailed breakdown for two large production Analysis Services databases.



Stats	Formula-Engine-Heavy DB	Storage-Engine-Heavy DB
Average	54 MB	1283 KB
Median	15 KB	945 Bytes
Max	10 GB	1214 MB

Figure 12: Analysis Services Database File Size Histogram

Given that the majority of Analysis Services files are less than 32KB, an Allocation Unit size of 64KB is not optimal. Storing a 32KB file in a 64KB cluster not only wastes 32KB of storage capacity but also lowers performance because the disk controller must read more raw data than necessary. For this reason, SQLCAT recommends an Allocation Unit size of 32KB. Lower Allocation Unit sizes are not a good choice because they only increase the I/O load when scanning large files for facts and aggregations. Figure 13 shows performance test results for an 8KB Allocation Unit size in comparison to 32KB for various RAID configurations. Note that the Allocation Unit size does not apply to files that are less than 1KB in size, which NTFS stores directly in the MFT File Record Segment.

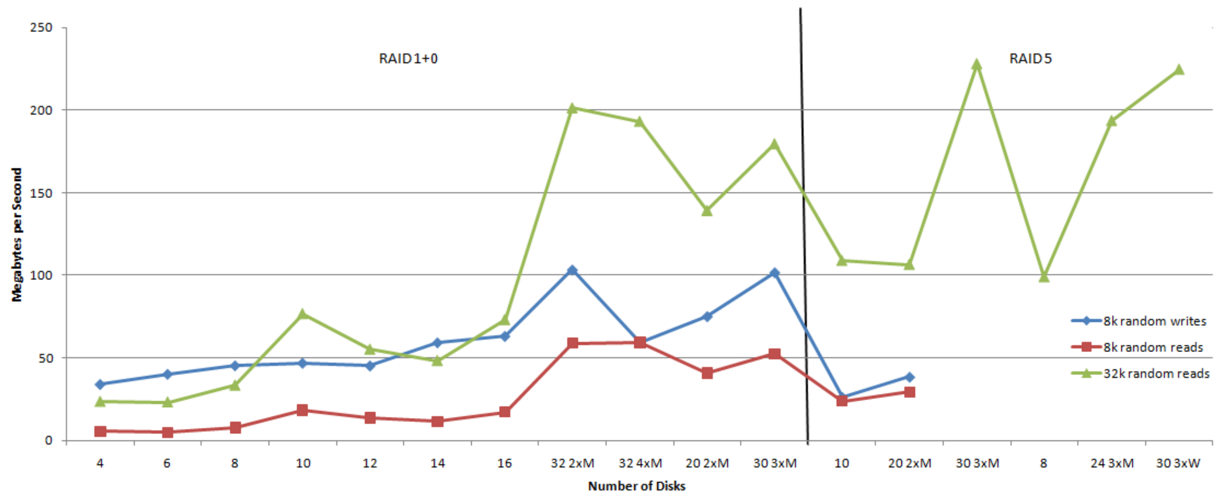
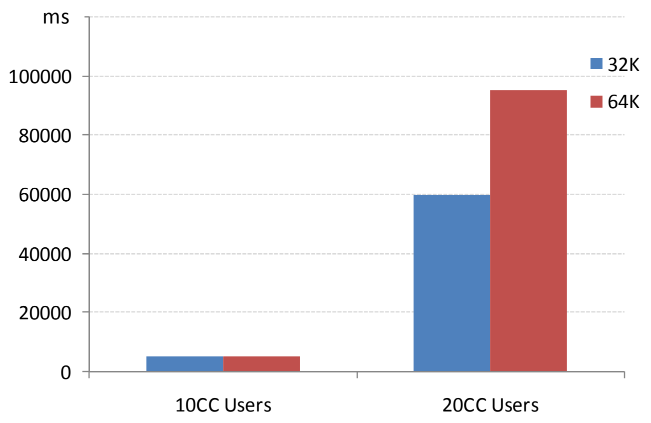


Figure 13: I/O Throughput Comparison for 8KB and 32KB Allocation Unit Sizes

Figure 14 confirms that the 32KB Allocation Unit size achieves best performance even in comparison to a 64KB size. It illustrates the I/O characteristics of a storage-engine-heavy Analysis Services environment. With ten concurrent users, performance is almost equal between 32KB and 64KB clusters because the storage subsystem can sustain the load without bottlenecks. However, the picture changes with 20 concurrent users overwhelming the storage subsystem. Under heavy load, the 32KB Allocation Unit size performs significantly better than the 64KB Allocation Unit size. The performance decline is less abrupt because of higher data density on the disks enabling the storage controllers to use cache more efficiently.



Block size	10CC Users	20CC Users
32K	5382.369	59999.809
64K	5409.986	95259.629

Figure 14: Impact of Block Size on Analysis Services Performance

Note that there is no relationship between the NTFS Allocation Unit size and the stripe size of the RAID array. NTFS is unaware of striping and cannot employ sophisticated methods to align clusters with stripes. To avoid spillover to a second disk, choose a large stripe size, such as 128KB, 256KB, or 512KB, and align the NTFS volumes with the stripe unit boundaries. Failure to align the volume can lead to significant performance degradation. The DiskPart.exe tool that ships with Windows Server 2008 uses a starting offset of 1024 KB, which works well for almost any array configuration and eliminates the need to establish the correct alignment manually.

Note: For details regarding the configuration and testing of an I/O subsystem for SQL Server, refer to the SQL Server Best Practices Article "*Predeployment I/O Best Practices*" available from the Microsoft Web site at http://technet.microsoft.com/en-us/library/cc966412.aspx. Keep in mind, however, that this article targets the relational engine with its recommendation of a 64KB Allocation Unit size. Use a 32KB Allocation Unit size for Analysis Services instead.

I/O Throughput

The I/O channel is a building block frequently overlooked in the storage design. As shown in Figure 15, this includes the host bus adapters (HBA), fibre channel switches, and storage controllers, and it can be the source of excessively unpredictable performance behavior, particularly in cluttered environments with numerous different types of servers accessing storage devices without proper isolation. With overlapping zones, storage controllers might service multiple systems with different I/O patterns, causing sporadic throughput breakdowns.

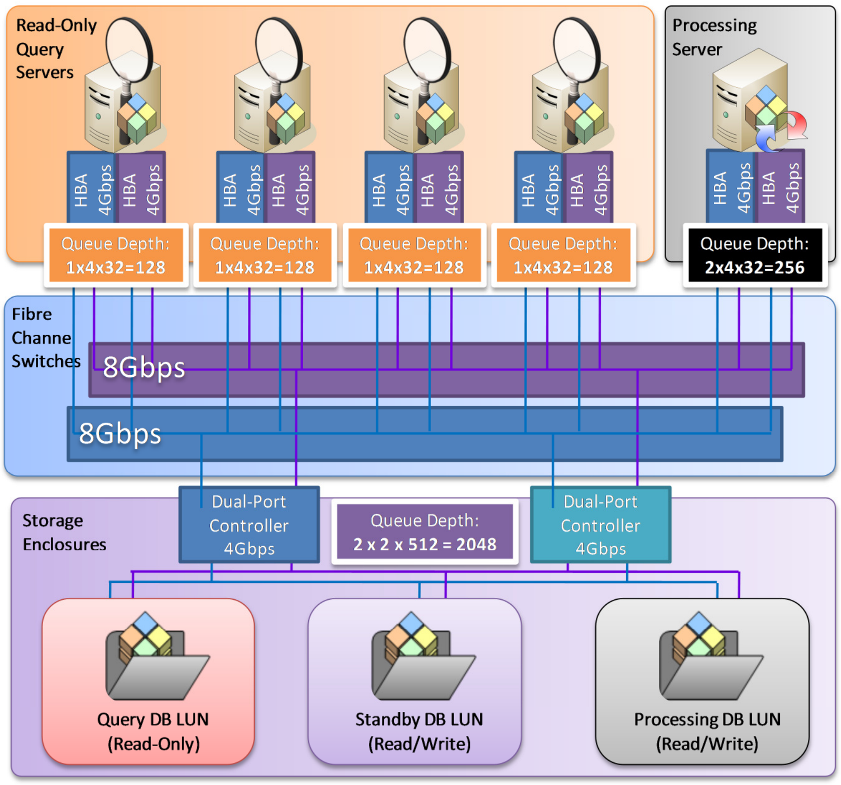


Figure 15: Dedicated I/O Channel for Analysis Services

24

SQLCAT strongly recommends using an isolated SAN environment to achieve high scalability with predictable query performance. In addition, consider the following best practices to optimize the I/O throughput:

- **Enable Multipath I/O (MPIO)** By using MPIO you can assign multiple fibre channel ports to the same LUN. Multiple data paths to the same LUN help to increase fault tolerance and performance by means of load balancing.
- **Use a dedicated fibre channel switch or zone** A fibre channel switch is required to aggregate the bandwidth. If it is not possible to dedicate a switch to the query servers and storage devices, define a fibre channel zone for the ports in use in the Analysis Services environment. The port-based zoning scheme isolates the query servers from other systems on the SAN so that they have exclusive access to their storage arrays.
- **Optimize the HBA queue depth** In general, Microsoft recommends using the default configuration settings for the HBAs. Although it can benefit performance to increase the queue depth per LUN (QueueDepth parameter) to support an increased number of concurrent I/O requests through each HBA, it is important to note that the setting must correlate with the queue depth of the target storage ports within the SAN array. For example, a storage port with a queue depth of 256 I/O requests can support four dual-connected query servers using the default HBA queue depth of 32 I/Os (256 / [32*2] = 4) or two dual-connected query servers with a queue depth of 64 I/Os (256 / [64*2] = 2), but not four dual-connected query servers using a queue depth of 64 I/Os. If the workload overwhelms the storage port, I/O performance drops sharply.
- **Update the HBA drivers** When choosing drivers for the HBAs that are installed on a host, ensure that you are using the recommended drivers for your particular storage array. Be aware that there might be significant differences between different drivers. For example, Storport drivers support a maximum queue depth of 255 I/Os per LUN as opposed to SCSIport drivers, which only support a queue depth of 255 I/Os per adapter.

Note: Use the SQLIO Disk Subsystem Benchmark Tool (SQLIO) and Windows Server Performance and Reliability Monitor (Perfmon) to benchmark available I/O bandwidth and synthetic performance of the storage subsystem. For details, refer to the SQLIO *readme.txt* file or run **SQLIO.exe -?** from the command prompt.

SQLCAT Lab Design

To help data warehouse architects and storage engineers determine appropriate storage configurations for their scale-out environments, the following section outlines the SQLCAT lab configuration for performance tests. SQLCAT used this configuration to clarify the best Allocation Unit size for Analysis Services, as discussed earlier under "*Block Sizes and Stripe Sizes*".

Figure 16 shows the lab configuration based on RAID 5 groups, which offered the best 32K random read performance per disk count. This particular test lab relied on a stripe set configured at the operating system level, which combined up to 12 RAID groups. In each RAID group, SQLCAT created one 50GB RAID 5 LUN, which overall amounted to a 600GB volume in Windows. Each RAID group consisted of eight disks, heavily short-stroked to achieve a very high I/O performance. The even numbered RAID groups SQLCAT assigned to storage processor A and the odd numbered RAID groups were assigned to storage processor B. In addition, SQLCAT created a RAID group consisting of 8 disks for a RAID 1+0 LUN. This LUN was presented to the computer running Analysis Services as the temp and log drive.

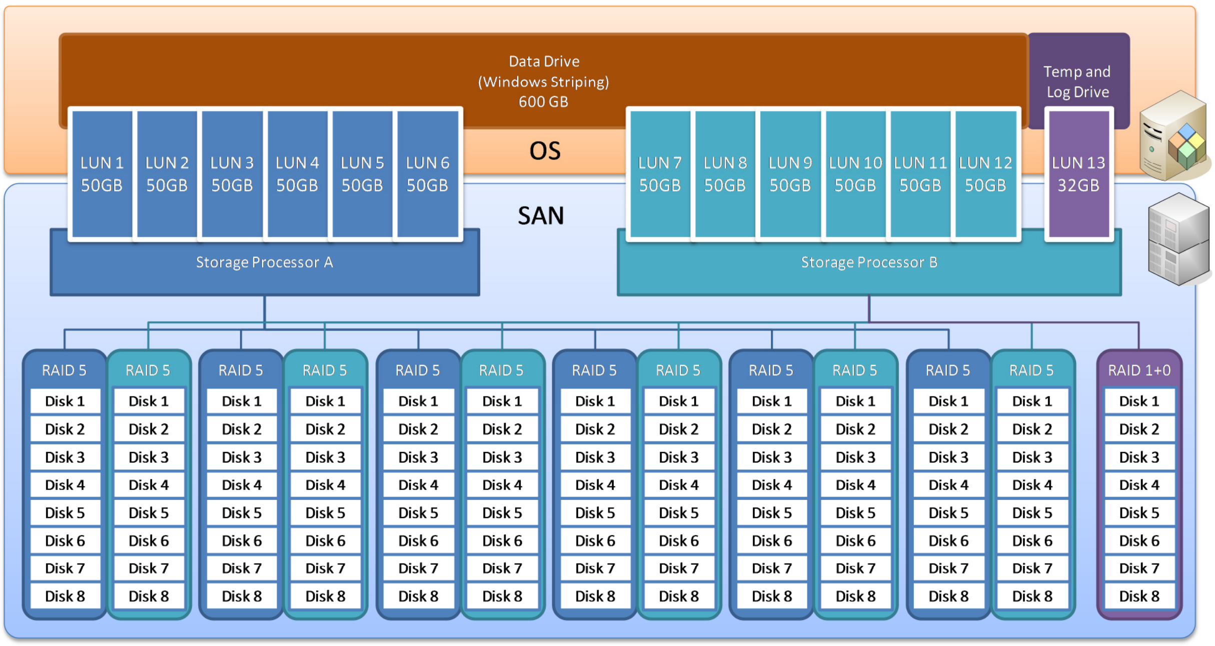


Figure 16: SQLCAT Storage Design for Allocation Unit Size Performance Tests

Note: The architecture depicted in Figure 16 is a lab configuration. It is for reference purposes only and not suitable for a production environment.

Conclusion

The read-only database feature of SQL Server 2008 Analysis Services enables high scalability with storage efficiency in a SAN-based environment. It decreases the complexity of the scale-out design, eliminates the need to provision write-enabled snapshot LUNs, increases cube processing flexibility, helps to ensure high cube availability during server maintenance and database updates, and provides flexible options to separate ad-hoc and Web-based analysis and reporting clients. In many enterprise-scale environments, it is the only cost-efficient option to support the required workload.

Achieving high scalability with Analysis Services requires a balanced approach of scale-up and scale-out. It is important to optimize query processing, tune the servers, and optimize the storage subsystem for highest possible performance. The scale-out then multiplies the scale-up gains. Hosted on a properly designed storage subsystem, read-only databases can be a viable scale-out option for both, formula-engine-heavy queries that primarily process data already in memory as well as storage-engine-heavy queries that involve a scan of the files on disk to perform a calculation.

On the front end, the scale-out architecture relies on round-robin network load balancing between multiple query servers with cookie-based or IP-based affinity. On the back end, the query servers access the same database LUN. It is strongly recommended to use an isolated SAN environment for Analysis Services. It is also recommended to provision a separate standby LUN for database updates. The standby LUN helps to increase cube availability during database updates and offers design flexibility to accommodate future growth more easily.

The individual database LUN design also benefits greatly from the read-only database feature. Because there are no write activities to consider, the RAID configuration is very straightforward. You can use cost-efficient RAID configurations, such as RAID 5, and ignore the write penalty. The read performance does not depend on RAID factors and the write penalty only applies during database updates on the processing server. It does not affect query performance.

SQL Server 2008 Analysis Services will continue to run on some of the world's largest enterprise servers, while the read-only database feature breaks down scalability barriers. It is the key to maintain even the world's largest analytics environments with cost efficiency and manageable complexity.

For more information:

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

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.