cid:image001.jpg@01CA6144.728946B0

***OLAP Processing Standards***

***Microsoft Confidential***

***February 18, 2010***

Table of Contents

[Table of Contents 2](#_Toc254265180)

[Document History 2](#_Toc254265181)

[OLAP Process Conventions 2](#_Toc254265182)

[1. Why OLAP ? 2](#_Toc254265183)

[2. Do’s and Don’ts in OLAP design 2](#_Toc254265184)

|  |
| --- |
| Document History |

|  |  |  |
| --- | --- | --- |
| **Last Updated By** | **Last Updated Date** | **Comments** |
| Kevin Johnston | 2/18/2010 | Draft |
|  |  |  |

|  |
| --- |
| OLAP Process Conventions |

## Why OLAP ?

Fast query response times compared to relational store since data is cached and pre-aggregated, you can slice and dice data the way business requires and use hierarchies to efficiently organize and summarize data. Dimensional and Hierarchical data has faster responses in OLAP and pre-aggregation is something that comes for free with OLAP. Also SSAS handles complex design scenarios: Examples of complex design components include special aggregate functions, parent-child hierarchies, complex dimension relationships, Parallel periods, horizontal and vertical navigation of hierarchies, and “near real-time” data refreshes.

SSAS 2005 provides 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. 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.

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.

## Do’s and Don’ts in OLAP design

Do’s:

1. Use named queries for concatenation and column selection. Use SQL views for joins at ETL layer.
2. Use OLAP Partitions – with or without SQL partitioning – for performance and cube processing. This helps greatly reduce processing times from hours to minutes. Partition by date id or customer id or region id, etc. as data permits.
3. Show additional information within dimensions as attributes and member properties SSAS is great for handling that.
4. Use many-to-many relationships with intermediary fact tables –SSAS 2005 can achieve this easily.
5. CALCULATED MEMBERS should have minimal logic and IIF statements. Complex logic related to date dimensions should be pushed out of cube and into the Reporting end (like Pro-clarity or SSRS). Calculated members that change constantly should never be in the cube – that needs constant re-processing of the cube.
6. Logic like refresh status last processed or data pulled should be on relational end not inside the cube as a dimension.
7. Optimize MDX on Client-side to improve query performance, for example use appropriate where, sub clauses, named sets, cross-joins, non-empty keywords, etc along with named sets. Pro-clarity and SSRS can have named sets to handle date and complex logic and calculations.
8. Set Aggregations for Fact tables, along with attribute relationships to improve query performance -for pre-calculated summaries of data (ensure that you have an effective aggregation design that satisfies the needs of your specific workload).
9. Use Scope statements, “this” assignment for improved query performance. This reduces calculations space and is very efficient. Reducing calculation space, re-use across multiple users, and executing the most efficient execution path (run profiler to catch cached versus non-cached hits).
10. In some complex time calculations add another attribute to exiting date dim or create a new dummy dimension – then assign calculated members to that dimension, this is rarely needed but may help.
11. 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.
12. Referenced dimensions are also very helpful in2005 this can avoid creating some unnecessary views and dimensions.
13. Do set attribute relationships: these helps slicing and dicing performance, reporting performance. And finally, use natural hierarchies and set attribute relationships where ever possible (set many-to-many, flexible or rigid relationships).

Don’ts:

1. DO NOT combine dimensions like product and plant – try to have dimensions that are disparate business entities – try to separate them if they are combined, try to re-use common, static shared dimensions that are maintained by business. Otherwise, it is hard to set aggregations and improve performance.
2. Make sure the Data is at granular level – do not do a rollup or group by in named queries and ETL layers. The cube does this free for you.
3. DO NOT combine Fact tables. Have fact tables with IDs (not raw varchar fields) along with business values (sales, etc).
4. DO NOT Create Dimensions from fact tables – it is better to use and maintain Dimensions that are well defined (avoid fact dimensions unless absolutely necessary)
5. DO NOT create separate CUBES - Measure Groups are great in SSAS 2005. It is easy to add several fact tables to common dimensions for business. This helps at the reporting end, since merging datasets from different cubes is very cumbersome and impossible when it comes to reporting.
6. DO NOT use measure group expressions unless there are no left joins involved, otherwise there nulls will be ignored - the solution is to do this as views in the back-end ETL process.
7. DO NOT bloat the code inside Calculated Members – lot of code can be taken out of Calculated Members by setting the right aggregation types ( like lastchild(), lastnonempty(), etc which are new in SQL2005). Also, aggregations do not apply to Calculated Members so refer to based measures that are pre-aggregated.

DO NOT hard-code Date calculations inside Calculated Measures - instead separate the code into named Sets or use a shell dimensions or add another attribute to existing date Dim. This has created huge mess see Inventory Visibility Cube.

1. **OLAP Partitions for processing performance**

SQL 2005 SQL table partitioning versus OLAP cube partitioning concepts: These two are completely different – but note, however, SQL Partitioning can help speed up OLAP back-end queries. OLAP Partitions are different – they are in memory distinct Cache Segments, with their own aggregations and indexes but with different query bindings. Partitions allow the source data and aggregate data of a cube to be distributed across multiple hard drives and among multiple server computers. For a cube of moderate to large size, partitions can greatly improve query performance, load performance and ease of cube maintenance.

A partition is based on a table or view in a data source or a table or named query in a data source view. Typically, you can partition a measure group horizontally or vertically. A measure group and all of its partitions must have fact tables with the same structure, although the fact tables can have different names. In a vertically partitioned measure group, a measure group is based on a single table, so each partition is based on a SQL query that filters the data for the partition. For example, if a single table contains several months data, the measure group could still be partitioned by month by applying a WHERE clause that returns a separate month's data for each partition.

The location for partition data is defined by the data source binding. Different partitions in a measure group can reference tables in the same data source or in different data sources. The initial partition for a measure group is based on the same table in the data source view as that of the measure group. Partitions must be created and managed correctly to avoid inconsistent or inaccurate results. This requirement applies to multiple-partition cubes. It also applies when you incrementally update any cube, including a single-partition cube, because an incremental update creates a temporary partition and merges it into an existing partition.

The integrity of a cube's data relies on the data being distributed among the partitions of the cube such that no data is duplicated among the partitions. When data is summarized from the partitions, any data elements that are present in more than one partition will be summarized as if they were different data elements. This can result in incorrect summaries and erroneous data provided to the end user. For example, if a sales transaction for Product X is duplicated in the fact tables for two partitions, summaries of Product X sales can include a double accounting of the duplicated transaction.

Challenge is to add a query just to define the incremental slice of data and bind it to a new dynamically created partition segment in memory. We can point each partition to a different temp table but that will become too many over time. We can manually create each partition and bind it to a new query but then we need automation and a solution in Production Environment. New Solution (implemented in INV DECOMP) Define partitions with dynamic data binding queries, for example, In Inv Decomp - we know the data is weekly snapshots. So the trick is to point the partition to new data by restricting the query to latest week’s data. So we need some kind of Data base (olap metadata) to store last successfully partitioned week, status, etc. And some tool in c# that creates partition dynamically and binds the dynamic query based on last processed week from OLAP METADATA Database.

INVENTORY DECOMP PROJECT: You can use the Partition Wizard to create additional partitions in a cube. But this is manual process…and we need to automate it for each partition. We created a new “Olap-Metatdata” DB contains the information like number of measure groups, last start date, etc. I have created a new tool in c# to dynamically read from olap metadata database to get the last incremented date, to bind a dynamic query, to create a new olap partition and finally, process the partition - Code to read partitions, loop through measure groups, loop through dimensions, etc. This is custom written code and there is no tool in SSIS to do this as lot of stuff is application specific.

Multiple partitions can enhance processing performance by providing you with the ability to process smaller data components of a measure group in parallel.

With multiple partitions, you can isolate your data refresh operations to specific partitions.

**Distinct Count** is a non-additive aggregate function that counts the unique instances of an entity in a measure group. 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. And always this should be in a separate Measure Group.

1. **Using semi-additive measures**

Instead of writing complex MDX calculations to handle semi-additive measures, you can use the semi-additive aggregate functions like **FirstChild()**, **LastChild()**, etc. Note that semi-additive functions are a feature of SQL Server Enterprise Edition. In addition to using semi-additive 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.

The cube design should use the right semi-additive aggregates: semi-additive aggregate functions are useful and helps you avoid tons of code in MDX and Calculated measures (when to try to do the same thing instead of using Semi-Additive measures) these include **ByAccount()**, **AverageOfChildren**() **FirstChild()**, **LastChild()**, **FirstNonEmpty()**, and **LastNonEmpty()**. To return the correct data values, semi-additive functions must always retrieve data that includes the granularity attribute of the time dimension. Aggregations containing the granularity attribute of the time dimension are extremely beneficial for efficiently satisfying query requests for semi-additive measures. Aggregations containing other time attributes are never used to fulfill the request of a semi-additive measure.

1. **Run Profiler Traces**

Profiler traces can catch aggregation hits, cache hits, etc to help identify performance issues. Create partitions to spread load on different drives on the disks. Set duration and other events (The event that specifically pertains to aggregation hits is the Get Data from Aggregation event) to catch the bad performing MDX queries and tune them by re-writing those. Avoid using VB functions outside of MDX. 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.

Determine whether the performance bottleneck is within the formula engine or the storage engine. To accomplish this, determine the amount of time required by the formula engine and the storage engine to execute the poorly performing MDX query when the cache is cold. The time spent by the storage engine can be determined by adding up the elapsed time for each Query Subcube event in a SQL Server Profiler trace. The time spent by the formula engine can be determined by subtracting the time spent by the storage engine from the total execution time for the **Query End** event.

If a significant percentage (we use a guideline of approximately 30%) of the total time spent by Analysis Services resolving the poorly performing MDX query is spent in the storage engine, there is a storage engine bottleneck. Resolve storage engine bottlenecks by utilizing an appropriate partitioning strategy, defining a set of effective aggregations, eliminating excessive cache pre-fetching, and, in some cases, warming the cache. Examples of several queries that exhibit storage engine bottleneck behavior are illustrated in the examples later in this article, along with examples of each of the foregoing solutions.

If a significant percentage (we use a guideline of approximately 30%) of the total time spent by Analysis Services resolving the MDX query is spent by the formula engine, there is a formula engine bottleneck. Resolve formula engine bottlenecks by writing MDX queries and calculations that utilize MDX language constructs that support block-oriented evaluation, eliminating empty Tuples and cells before expensive calculations are evaluated.

If the storage engine is a bottleneck, determine if partitioning might help, if aggregations are being hit, and if the cache is being used efficiently. Where appropriate, partition each measure group to align to query patterns, define selected aggregations, and determine the best way to optimize the use of cache. Rewrite your MDX queries if that is an option or consider warming the cache.

If the formula engine is a bottleneck, use good troubleshooting techniques to reduce the complexity of a poorly performing query to isolate the portion at the heart of the performance problem. Read everything you can on how to write efficient MDX and enable Analysis Services to enable the formula engine to use block-oriented evaluation whenever possible. Also, you may need to learn how to use specialized tools other than SQL Server Profiler and System Monitor to try to isolate the source of certain types of problems.

1. **Appendix A – More Design /MDX Optimization**

Optimize MDX wherever possible – speed and response time depends on how efficiently MDX is written.

The server in SQL2005 uses the following path for MDX execution:

Session Management 🡪 (XML for Analysis (XMLA) over TCP IP or HTTP) 🡪 Analysis Services Session Manager 🡪 Security Manager 🡪 **Query ENGINE** 🡪 Data Retrieval 🡪 **Storage ENGINE** 🡪 Cache 🡪 Data from store files (for dimensions) and Data from Aggregations for fact data. **Run profiler to catch cached versus non-cached hits, same with aggregations!** The cube is built in passes, 0 for fact table, 1 for aggregations, etc.

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.
2. **Aggregations**—if relevant data is not in the cache, the Storage Engine checks for a pre-calculated data aggregation. In some scenarios, the aggregation may exactly fit the data request
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

The primary operation of the Query Execution Engine is to execute MDX queries. The Query Execution Engine decomposes each MDX query into data requests and communicates with the Storage Engine, the Query Execution Engine must translate the data requests into sub-cube requests that the Storage Engine can understand. A sub-cube represents a logical unit of querying, caching, and data retrieval. An MDX query may be The Query Execution Engine uses two kinds of execution plans to calculate results: it can **bulk calculate** an entire sub-cube, or it can calculate **individual cells**. In general, the sub-cube evaluation path is more efficient; however, the Query Execution Engine ultimately selects execution plans based on the complexities of each MDX query. Reducing calculation space, re-use across multiple users, and executing the most efficient execution path.

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.

Calculated members and Named sets: Calculated members are members of a dimension or of a measure group that are defined based on a combination of cube data, arithmetic operators, numbers, and functions. For example, you can create a calculated member that calculates the sum of two physical measures in the cube. Calculated member definitions are stored in cubes, but their values are calculated at query time. A calculated member in the Measures dimension is also frequently called a calculated measure. A named set is a Multidimensional Expressions (MDX) expression that returns a set of dimension members. You can define named sets and save them as part of the cube definition; you can also create named sets in client applications. You create named sets by combining cube data, arithmetic operators, numbers, and functions. Named sets can be used by users in MDX queries in client applications and can also be used to define sets in sub-cubes. A sub-cube is a collection of cross-joined sets that restricts the cube space to the defined subspace for subsequent statements. Defining a restricted cube space is a fundamental concept to MDX scripting.

**The default scope is the whole cube**, but as mentioned in the previous topic, you can use the **SCOPE** statement to define a more limited scope, known as a sub-cube, and then apply an MDX script to only that particular cube space. The SCOPE statement defines the scope of all subsequent MDX expressions and statements within the calculation script until the current scope ends or is re-scoped. The THIS statement is then used to apply an MDX expression to the current scope. You can use the BACK\_COLOR statement to specify a background cell color for the cells in the current scope, to help you during debugging.

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 just like SQL Server. 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.

The Dimension Attribute Store contains all of the information about dimension attributes (key stores, Property Store, and Relationship store).

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.

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.

* **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.
* **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.
* **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.

Move numeric attributes to measures

You canconvert 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.

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 less iteration 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.