Dimension and Design Best Practices

Version 1.0

Dimension and Design Best Practices

Create attribute relationships wherever possible
Attribute relationships are an important part of dimension design. They help the server optimize storage of data, define referential integrity rules within the dimension, control the presence of member properties, and determine how MDX restrictions on one hierarchy affect the values in another hierarchy. Spend time defining attribute relationships that accurately reflect relationships in the data.

Avoid creating unnecessary attributes
Attributes add to the complexity and storage requirements of a dimension, and the number of attributes in a dimension can significantly affect performance. This is especially of attributes which have **AttributeHierachyEnabled** set to **True**. Although SSAS can support many attributes in a dimension, having more attributes decreases performance unnecessarily.

Use numeric keys for attributes that contain many members (>1 million)
Using a numeric key column instead of a string key column / composite key will improve the performance of attributes that contain many members. This best practice is based on the same concept as using surrogate keys in relational tables for more efficient indexing. You can specify the numeric surrogate column as the key column and still use a string column as the name column so that the attribute members appear the same to end-users. As a guideline, if the attribute > 1 million members, consider using a numeric key.

Don't create hierarchies where an attribute of a lower level contains fewer members than an attribute of the level above
A hierarchy such as this is frequently an indication that your levels are in the incorrect order: for example, [State] above [Country]. It might also indicate that the key columns of the lower level are missing a column: for example, [Year] above [Quarter] instead of [Year] above [Year-Quarter]. Either of these situations will lead to confusion for end-users trying to use and understand the cube.

Don't include multiple non-aggregatable attributes per dimension
Because there is no All member, each non-aggregatable attribute will always have non-all member selected, even if not specified. If you include multiple non-aggregatable attributes in a dimension, the selected attributes will conflict and produce unexpected numbers.

Use key columns that completely define the uniqueness of the members in an attribute
Usually a single key column is sufficient, but sometimes multiple key columns are necessary to uniquely identify members of an attribute. For example, it is common in time dimensions to have a [Month] attribute include both [Year] and [Month] as key columns. This is known as a composite key and identifies January of 2009 as being a different member than January of 2010. When you use [Month] in a time hierarchy that also contains [Year] so this distinction between January of 2009 and January of 2010 is important.

Do perform Process Index after a Process Update if dimension contains flexible AttRelation or a parent-child hierarchy
An aggregation is considered flexible if any attribute included in the aggregation is related, either directly or indirectly, to the key of its dimension through an AttributeRelationship with RelationshipType set to Flexible. Aggregations that include parent-child hierarchies are also considered flexible.

When a dimension is processed by using the **Process Update** option, any flexible aggregations that the dimension participates in might be dropped, depending on the contents of the new dimension data. These aggregations are not rebuilt by default, so Process Index must then be explicitly performed to rebuild them.

Don't create redundant attribute relationships
Do not create attribute relationships that are transitively implied by other attribute relationships. The alternative paths created by these redundant attribute relationships can cause problems for the server and are of no benefit to the dimension. For example, if the relationships A->B, B->C, and A->C have been created, A->C is redundant and should be removed.

Include key columns of snowflake tables joined to nullable foreign keys as attributes with NullProcessing = UnknownMember
If tables that are used in a dimension are joined on a foreign key column that might contain nulls, it is important that you include in your design an attribute whose key column is the corresponding key in the lookup table. Without such an attribute, the OLAP server would have to issue a query to join the two tables during dimension processing. This makes processing slower; moreover, the default join that is created by the OLAP server would exclude any rows that contain nulls in the foreign key column. It is important to set the **NullProcessing** option on the key column of this attribute to **UnknownMember**. The reason is that, by default, nulls are converted to zeros or blanks when the engine processes attributes. This can be dangerous when you are processing a nullable foreign key. Conversion of a null to zero at best produces an error; in the worst case, the zero may be a legitimate value in the lookup table, thereby producing incorrect results.

To handle nullable foreign keys correctly, you must also set **UnknownMember** to **Visible** on the dimension. The Cube Wizard and Dimension Wizard currently set this property automatically; however, the Dimension Wizard lets you manually de-select the key attribute of snowflake tables. You must not deselect the key column if the corresponding foreign key is nullable.

If you do not want to browse the attribute that contains the lookup table key column, you can set **AttributeHierarchyVisible** to **False**. However, **AttributeHierarchyEnabled** must be set to **True** because it is necessary that all other attributes in the lookup table be directly or indirectly related to the lookup key attribute in order to avoid the automatic creation of new joins during dimension processing.

Note: You can also create a new record in look table (e.g. -1 as surrogate key) for NULL references.

Set RelationshipType property appropriately on AttributeRelationships based on whether the relationships between individual members change over time
The relationships between members of some attributes, such as dates in a given month or the gender of a customer, are not expected to change. Other relationships, such as SalesPeople in a given region or the Marital Status of a customer, are more prone to change over time. You should set **RelationshipType** to **Flexible** for those relationships that are expected to change and set **RelationshipType** to **Rigid** for relationships that are not expected to change. When you set RelationshipType appropriately, the server can optimize the processing of changes and re-building of aggregations.

Avoid using ErrorConfigurations with KeyDuplicate set to IgnoreError on dimensions
When KeyDuplicate is set to IgnoreError, it can be difficult to detect problems with incorrect key columns, incorrectly defined AttributeRelationships, and data consistency issues. Instead of using the IgnoreError option, in most cases it is better to correct your design and clean the data. The IgnoreError option may be useful in prototypes where correctness is less of a concern.

Consider creating user-defined hierarchies whenever you have a chain of related attributes in a dimension
Chains of related attributes usually represent an interesting navigation path for end-users, and defining hierarchies for these will also provide performance benefits.
Avoid creating user-defined hierarchies that do not have attribute relationships relating each level to the level above
Having attribute relationships between every level in a hierarchy makes the hierarchy strong and enables significant server optimizations.

Avoid creating diamond-shaped attribute relationships
A Diamond-shaped relationship refers to a chain of attribute relationships that splits and rejoins but contains no redundant relationships. For example, Day->Month->Year and Day->Quarter->Year have the same start and end points, but do not have any common relationships. The presence of multiple paths can create some ambiguity on the server. If preserving the multiple paths is important, it is strongly recommended that you resolve the ambiguity by creating user hierarchies that contain all the paths.

Consider setting AttributeHierarchyEnabled to False on attributes that have cardinality that closely matches the key attribute
When an attribute contains roughly one value for each distinct value of the key attribute, it usually means that the attribute contains only alternative identification information or secondary details. Such attributes are usually not interesting to pivot or group by. For example, the Social Security number or telephone number may be interesting properties to view, but there is very little value in being able to pivot and group based on SSN or telephone. Setting AttributeHierarchyEnabled to False on such attributes will reduce the complexity of the dimension for end-users and improve its performance.

Consider setting AttributeHierarchyVisible to False on the key attribute of parent-child dimensions
Because the members of the key attribute are also contained in the parent-child hierarchy in a more organized manner, it is usually unnecessary and confusing to the end-user to expose the flat list of members contained in the key attribute.

Avoid setting UnknownMember=Hidden
When you suppress unknown members, the effect is to hide relational integrity issues; moreover, because hidden members might contain data, results might appear not to add up. Therefore, we recommend that you avoid use of this setting except in prototype applications.

Use MOLAP storage mode for dimensions with outline calculations
Dimensions that contain custom rollups, semi-additive measures, and unary operators will perform significantly better using MOLAP storage. The following dimension types will also benefit from using MOLAP storage: an Account dimension in a measure group that contains measures aggregated using ByAccount; the first time dimension in a measure group that contains other semi-additive measures.

Use a 64 bit server if you have dimensions with more than 10 million members.
If a dimension contains more than 10 million members, using an x64 or an IA-64-based server is recommended for better performance.

Set the OrderBy property for time attributes and other attributes whose natural ordering is not alphabetical
By default, the server orders attribute members alphabetically, by name. This ordering is especially undesirable for time attributes. To obtain the desired ordering, use the **OrderBy** and **OrderByAttributes** properties and explicitly specify how you want the members ordered. For time-based attributes, there is frequently a date or numeric key column that can be used to obtain the correct chronological ordering.

Dimension and Design Best Practices

Expose a DateTime MemberValue for date attributes

Some clients, such as Excel, will take advantage of the MemberValue property of date members and use the DateTime value that is exposed. When Excel recognizes the value as DateTime, Excel can treat the value as a date type and apply date functions to the value, as well as provide better formatting and filtering. If the key column is a single DateTime column and the name column has not been set, this MemberValue is automatically derived from the key column and no action is necessary.

Avoid setting IsAggregatable to False on any attribute other than the parent attribute in a parent-child dimension

Non-aggregatable attributes have non-all default members. These default members affect the result of queries whenever the attributes are not explicitly included. Because parent-child hierarchies generally represent the most interesting exploration path in dimensions that contain them, it is best to avoid having non-aggregatable attributes other than the parent attribute.

Do not combine unrelated business entities into a single dimension

Combining attributes of independent business entities, such as customer and product or warehouse and time, into a single dimension will not only create a confusing model, but also reduce query performance because auto-exist will be applied across attributes within the dimension.

Another way to state this rule is that the values of the key attribute of a dimension should uniquely identify a single business entity and not a combination of entities. Generally this means having a single column key for the key attribute.

Consider setting AttributeHierarchyVisible to False for attributes included in user-defined hierarchies

It is usually not necessary to expose an attribute in its own single level hierarchy when that attribute is included in a user-defined hierarchy. This duplication only complicates the end-user experience without providing additional value.

One common case in which it is appropriate to present two views of an attribute is in time dimensions. The ability to browse by [Month] and the ability to browse by [Month-Quarter-Year] are both very valuable. However, these two month attributes are actually separate attributes. The first contains only the month value such as “January” while the second contains the month and the year such as “January 2010”.

Avoid making an attribute non-aggregatable unless it is at the end of the longest chain of attribute relationships in the dimension

Non-aggregatable attributes have non-all default members that affect the result of queries in which values for those attributes are not explicitly specified. Therefore, you should avoid making an attribute non-aggregatable unless that attribute is regularly used. Because the longest chain of attributes generally represents the most interesting exploration path for users, it is best to avoid having non-aggregatable attributes in other, less interesting chains.

Consider creating at least one user-defined hierarchy in each dimension that does not contain a parent-child hierarchy

Most (but not all) dimensions contain some hierarchical structure to the data which is worth exposing in the cube. Frequently the Cube Wizard or Dimension Wizard will not detect this hierarchy. In these cases, you should define a hierarchy manually.