

The premier AI platform for
**Integrated Planning and
Operations Management**



GraphCube-IBPL Server User's Guide

o9 Internal use Only 1

Table of Contents

[Introduction](#)

[IBPL Structure](#)

[Terminology](#)

where

- Dimension
- Member ()
- Level
- Hierarchy
- Attribute
- Property
- Member Set
- Plan scope: ([Time].[Fiscal Month] * &CWV * [Item].[SKU] * [Customer].[Customer] * [Personnel].[TM]);
- Measure.[Salesperson Budget Units (D02)] = Measure.[Baseline Budget Units (D01)] * Measure.[Customer x TM Association (A03)];
- end scope;
- Measure Group
- Measure
- Deletion of Measure Group Data
- Syntax
 - Square Brackets
 - Dot Notation

IBPL Rules

- Named Sets
- Active Rules
- Recurrence Rules
- Ordered Recurrence Rules
- Block Scope Rules
- Action Button Debug
- Save Performance Improvement

IBPL Procedures

- Parameterized IBPL procedures
- Parameterized IBPL Procedures Usage Examples
- Simple string input ...
 - ... to run stat forecast plugin
 - Defining procedure
 - Executing procedure
 - ... to run SCS
 - Defining procedure
 - Executing procedure
- Using #if / #else ..
 - ... on RHS of a regular scope statement
 - Defining procedure
 - Executing procedure
 - ... inside regular scope statement to run different assignments
 - Defining procedure
- o9 Internal use Only 2
 - Executing Procedure
 - ...torundifferentscopestatements
 - Defining procedure
 - Executingprocedure
 - Running insideanactionbutton

where

- Defining procedure
- Executing procedure manually
- Action button config
- Action button execution

IBPL Commands

- Read Queries
 - Member Query
 - Measure Query
- Write Queries
 - Single-Cell Update Query
 - Scope based (Multi-Cell) Update Query
 - Direct Inserts (DI)
- Member Management
 - Member Creation
 - Member Update
 - Member Delete (Trimming a Dataset)
- Version Management
- Release Memory IBPS command
- Set of Characters Allowed in the Graph Cube

Operators

- Unary Arithmetic Operators
- Binary Arithmetic Operators
 - Addition
 - Subtraction
 - Multiplication
 - Division
 - Power
- Binary String Operators
 - Concatenation
- Binary Comparison Operators
 - Equal
 - Not Equal
 - Less Than
 - Less Than Or Equal
 - Greater Than
 - Greater Than Or Equal
- Binary Logical Operators
 - Logical And
 - Logical Or
 - In
 - Not

o9 Internal use Only3

- Attribute Member Operators
 - At

Functions

where

String Functions

Upper

Lower

ToDateTime

Current User (Logged in User)

Numeric Functions

Abs

Coalesce Null

The Coalesce() function is used to replace a null value with another supplied value for a measure. The return value is a number. The function takes two numeric values. The first value is to check for the null, and the second is the number to replace the first with, if it is null.

ToString

Round

Ceiling

Floor

Safe Divide

Float

Integer

Sum

Avg

AvgWithNulls

Count

Min

Max

filterDateTime Functionscurrent timemem

ToString

DateADD Function

DateDiff Function

Logical Functions

If-Then-Else

IsNull

AttributeMember Functions

Ancestor

Children

LeadOffset

Between

AttributeMemberSet Functions

Find

Filter

Filter Predicate

Filter_By_Updates

Element

Count

AncestorsAtLevel

DescendantsAtLevel

o9 Internal use Only 4

QuerytoDisplayMemberswithoutDescendants

RelatedMembers

where

- UnionWith
 - IntersectWith
- RelationshipFunctions
 - Traverse
- DistinctCount
 - Members
 - UsingMembersfunction in other select queries
- RulesLanguage
- Command Language
 - SelectQueries
 - ExecutingProcedures
 - DataManagementCommands
 - DownloadDataFile
 - UploadDataFile
- SelectiveExportAll
 - ImportAll
- Member Management Commands
 - CreateMember
 - UpdateMember
 - CopyMember
 - DeleteMember
- MeasureManagementCommands
 - CopyMeasure
- VersionManagementCommands
 - CreateVersion
 - CreateVersionwithscope
 - UpdateVersionProperty
 - cre
 - UpdateScenario
 - UpdateScenariowithscope
 - DeleteScenario
- RelationshipCommands
 - updaterelationhiptype
- UpdatingMeasures
 - ScopeBasedUpdate
 - Single-Cell Update
- AccessControl (ACL)Commands
 - DataSecurity IBPLRules
 - MemberSecurity
 - DenyReadatDimension
 - DenyReadatLevel Attribute
 - DenyReadatMemberSet
 - GrantWriteatDimension

where

o9 Internal useOnly5

- GrantWriteatLevel Attribute
- GrantWriteatMemberSet
- Cell Security
- GrantReadAccessAtPlan
- GrantWriteAccessAtPlan
- GrantReadAccessAtModel (MeasureGroup)
- GrantWriteAccessAtModel (MeasureGroup)

Executing IBPL

- ModifyingtheRuleFile(s)
- Usingthe IBPLPlusCommand-LineClient
 - ActiveQueries
- Executingan IBPLScriptwith IBPLPlus
- UsingtheMeruClient
- Executing IBPL intheHTMLUI

Meta-Schema Dimensions

- Dimension: [_SchemaDimension]
- Hierarchiesofthe[_SchemaDimension]
- Dimension: [_SchemaPlan]
- Dimension: [_SchemaRelationship]

mPowerModeling: MemberRelationships

- BackgroundandMotivation
- Terminology
 - Relationship
 - RelationshipProperties
 - RelationshipType
 - Fromvs. To
 - Symmetry/Recursion/Cycles
 - Graphs/Edges/Nodes
- CreatingaRelationshipType intheDatabase
 - RelationshipType
 - NodeAttributeElements
 - NodeMeasureElements
 - RelationshipProperties
- Inspecting Records in the Database
- DeletionofGraphEdges
- ManipulatingRelationshipsUsing IBPL
 - AddRelationship
 - ModifyRelationship
 - DeleteRelationship
 - CombinationCommands
- Inspecting Relationships Using IBPL
- SimpleTraversal
- Starting MembersExpression
- RelationshipFilterExpression
- MoreComplexity
 - LocationBOMRelationshipType

where

DatabaseRecords

o9 Internal useOnly6

Add Relationship

Starting Members Expression

Level List Attribute

Traversal Direction

Traversal Distance

Interaction of Relationship Filter Expression with Recursion

Extraneous Information in Starting Members Expression

mPower Modeling: Aggregation

Background

Aggregation Policies

Sum

Min

Max Null

Median

Avg

AvgNonNull

AvgNonNullOrZero

FirstChild

LastChild

Computed

Null

Different Aggregation Across Dimensions for a Measure

mPower Modeling: Spreading

Background

Spreading Rules

CopyToLeaves

DistributeToLeaves

DistributeWeightedAverage

DistributeToLeavesNonSummable

Cartesian Scope

Spread Scope

Block Scope

Parallel Run

EvaluateMember(Member Property Based Evaluation)

Basis Measure Types

Distribute To Leaves

Basis Measure

Null Basis Measure

Respread Basis Measure

Assortment Basis Measure

Copy To Leaves

Basis Measure

Assortment Basis Measure

Transient Measures

Sample Graph Model

where

Querying a single graph

o9 Internal use Only 7

Leaf Level Graph Queries (TabularLeafGraph)

Leaf Level Query For just the from/to nodes

Leaf Level Qu

ery With Filters

Leaf Level Query With member filters, include member properties

Online Model Change

Querying external data source

Null Propagation Behaviour

Time roll forward

where

Introduction

IBPL Structure

IBPL stands for *Integrated Business Planning Language*. It is used to access and manipulate data and metadata in the GraphCube engine (also known as LiveServer).

The IBPL structure is broadly classified as

where

I. Rules

II. Commands

In the Rules language, IBPL is used to create named sets, set active rules for measures, define procedures and plug-ins, and create security rules. In the early versions of GraphCube, these actions were contained in a text file referred to as the rules file, which was read and executed during the initialization of the GraphCube engine as part of the startup process. Nowadays there are multiple rules files contained in the tenant/configuration database and managed through the configuration UI. They are still read and executed as part of the startup process. They are loaded in order according to the “position” value in the configuration UI.

In the Command language, IBPL is used to execute commands and queries on the GraphCube engine. The command language is used after the GraphCube engine has been started. There are three main ways it can be used:

- Commands can be executed in an interactive manner through one of the available clients (IBPLPlus, Meru, HTML UI)

•



Commands can also be contained in a script (text file) that is that is sent to GraphCube for execution by the IBPLPlus client

- Commands can be contained in action buttons that are executed by users in the HTML UI or Excel UI

The following figure outlines the IBPL structure:

where

Terminology

Dimension

A dimension is an entity such as a Product or *Sales Domain* that can be used to describe some facet of business planning data. A dimension entity is a collection of related objects called attributes. For

o9 Internal use Only 9

example, typical attributes in a product dimension are product name, category, sub-category and price.

| Product Name | Category Sub-Category | Price |
|--------------------------|-------------------------|-------|
| Cosmo Cola Diet 12 oz | Carbonated Cola Diet | \$5 |
| Cosmo Cola Regular 24 oz | Carbonated Cola Regular | \$3 |

There are two types of dimensions: Regular and Time . Time dimensions are treated in some special

ways in the UIs, such as in the aggregation policies FirstChild and LastChild, and in identifying which member of each level represents the “current” time (and therefore which members are in the past and which are in the future). A tenant is allowed to have more than one dimension of type Time.

There are several dimensions that come built into mPower:

- Version -- used to save snapshots of the business planning data and to manage scenarios. There is always one Version called CurrentWorkingView (CWV), which represents the plan of record. Saved versions are snapshots of the CWV taken at some time in the past and cannot be edited. Scenarios are dynamic versions that are used to evaluate some scenario or perform some what-if analysis. They can be snapshots of the CWV, or of a saved version, or even of another scenario. A scenario can be merged into the CWV in which case the data changes made in the scenario are applied to the CWV.
- DimPlugin -- <TBD>
- Algorithm -- <TBD>

Member

where

A member is a specific instance of an attribute in a dimension. For example, in the Product dimension, there may be members such as *Cosmo Cola Diet 12oz* and *Cosmo Cola Regular 24 oz*. A member is also referred to as an *AttributeMember*. Members have two intrinsic properties: Name and Key. Name is usually a string, and Key is usually an integer (for regular dimensions) or datetime (for time dimensions). However, other data types are possible. Note: there are performance implications if string is used as the data type for a Key.

Name is the name of the member (like *Cosmo Cola Diet 12oz* and *Carbonated Cola*). The Name of a member is unique among all members at the same level in the dimension (but may be shared by a member in another level).

Key is also unique among all members at the same level. While Name can be changed, the Key cannot change. For this reason, integers are usually used as keys, and the value has no intrinsic meaning. There are a few cases where the Key does have some meaning. In a dimension of type Time, the Key is a datetime value rather than an integer, and corresponds to the first day of the time period. Example, the first fiscal month of 2013 may start on Dec 30, 2012, in which case the Key for that member is 2012-12-30 00:00:00, and the Name is *M01-2013*.

It is a good modelling practice to maintain the Key of first Day as same as Key of Fiscal Week or Fiscal Month. So if Day 1 key is 2012-12-30 00:00:00, the same should be the key for fiscal week 1 and fiscal month 1. Follow the same construct for the other weeks and months.

In the Version dimension, the Key is an integer, but it has a meaning. The value of zero is reserved for the *CurrentWorkingView*. A negative value for Key indicates that the version is a *scenario* or *what-if* version. A positive value for Key indicates that the version is an archived snapshot of the plan.

¹ There's a third type "CurrentTime" that is no longer used.

o9 Internal use Only 10

Level

A level defines a group of members of a dimension that share the same granularity. Example, the member *Cosmo Cola Diet 12oz* is an actual product and is part of the level defined by the Product Name. There may also be a level called *Category* with members like *Carbonated Cola* and *Flavored*.

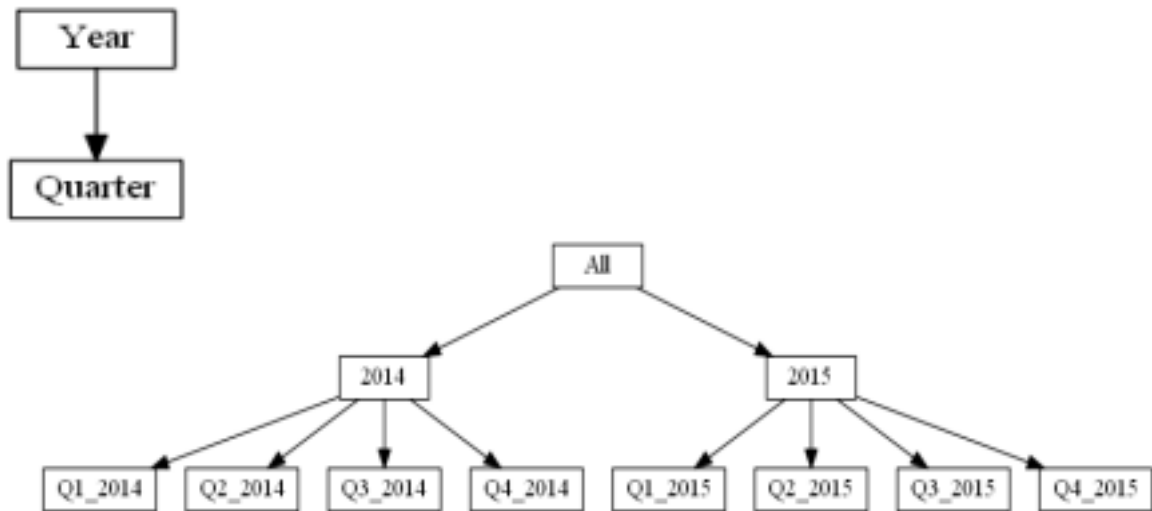
Hierarchy

A hierarchy is an ordered collection of levels, where the members of one level are *parents* to members in the next level. Example, the Product dimension can have a hierarchy with the levels of *Category*, *Sub-Category* and *Product Name*. Every member of the *Product Name* level has a parent member that is in the *Sub-Category* level, and every member of the *Sub-Category* level has a parent member that is in the *Category* level. Conversely, members of higher levels may have multiple children. So a member of the *Category* level may have one or more children in the *Sub-Category* level, and a member of the *Sub-Category* level may have one or more children in the *Product Name* level. It is not required that a higher level member have any children at all. A higher level member without children is referred to as a hanging member.

A dimension can have more than one hierarchy. For example, the Product dimension can have an alternate hierarchy with levels *Size* and *Product Name*. In this case, every member of the Product Name level will have two parent members: one in the *Size* level and one in the *Sub-Category* level.

where

Image: Sample Level Hierarchy Chart



Attribute

Attribute can either refer to an attribute of the dimension that defines a level in a hierarchy, or it can refer to a property of a member of some level. The former is also known as a LevelAttribute (and is synonymous with Level, discussed above), and the latter is also known as a Property.

o9 Internal use Only 11

Property

Properties are defined for members based on level. The names of properties (except for the intrinsic properties Name and Key) are unique within a dimension. So if, for example, there is a property called *Description* that is defined for members in the *Product Name* level, then there can't be a *Description* property for *Category* or *Subcategory* members. There can, however, be a *Category Description* property for the *Category* level and a *Sub-Category Description* for the *Subcategory* level.

There are two hidden properties that all members have, plus two additional hidden properties that all members of time dimensions have.

1. **The first hidden property that all members have is <level>\$DisplayName** (e.g., for the *Product Name* level, this property is called *Product Name\$DisplayName*). The display name is used in the UI instead of the Name property. By default it is the same as the Name, but it can be changed. Like Name, it must be unique among all members of the same level. If you change the display name and try to make it the same as the display name of another member, then GraphCube will automatically append " (x)" to the display name, where x is the smallest positive integer that makes the display name unique.
2. **The second hidden property that all members have is <level>\$Inactive** (e.g., for the *Product Name* level, this property is called *Product Name\$Inactive*). This is a boolean property that is true if a member has been marked for deletion, and false otherwise. If you execute a *DeleteMember* command, then this property is updated to true. The member is not actually deleted until you execute the *Purge Members* command (and even then, it won't be purged if there is fact data in a

saved version that is associated with this member). It is possible to update the in-active property directly, but it is not recommended.

where

3. **A hidden property that only time dimension members have is <level>\$IsCurrent** (e.g., for the *Fiscal Month* level, this property is called *Fiscal Month\$IsCurrent*). This is a boolean property that is true for the member that denotes the current time bucket, and false otherwise. Only one member in each level can be the current member. If you set this property to true for a different member, then the property for the former current member will be changed to false. If you set this property at the lowest level of the time dimension, then it will propagate up the hierarchy (i.e., if you set *Fiscal Month\$IsCurrent* to true for a certain month member, then the *Fiscal Quarter\$IsCurrent* property is set to true for the quarter member that is the parent for that month member, and the *Fiscal Year\$IsCurrent* property is set to true for the year member that is the parent of the quarter member). If you set this property at the higher level, it will not update the property at any lower levels, so it is recommended that you only set this property at the lowest level. Also is it recommended to only set this property to true. If you change the property to false for the current member, then there will not be any current member.
4. **Another hidden property that only time dimension members have is <level>\$InPast** (e.g., for the *Fiscal Month* level, this property is called *Fiscal Month\$InPast*). This is a boolean property that is true for members that occur chronologically in time before the current member, and false otherwise. This property is automatically updated when the current property is updated. This property should not be updated directly.

Member Set

A member set is a collection of members. It is also referred to as an AttributeMemberSet. All of the members in the member set are part of the same level of the same dimension. The set is ordered either by Name or Key, depending on how the attribute meta-data is configured.

o9 Internal use Only 12

Plan scope: ([Time].[Fiscal Month] * &CWV * [Item].[SKU] * [Customer].[Customer] * [Personnel].[TM]);

Measure.[Salesperson Budget Units (D02)] = Measure.[Baseline Budget Units (D01)] * Measure.[Customer x TM Association (A03)];

end scope;

A plan is a group of measure groups that are part of the same business process or functional area. It is a logical grouping only and has no bearing on the functionality of mPower.

Measure Group

A measure group is a group of measures that all share the same granularity (i.e., have the same dimensions and levels). It is sometimes also referred to as a Model.

Measure

A measure represents data (known as fact data, or simply facts) that exists at the intersection of one or more dimensions. For example, the measure *Net Revenue* is the net revenue from sales and exists at the intersection of Product, Sales Domain, Time and Version.

where

Each measure has a Data Type property. The most common property is number. Other options are integer, string, datetime, boolean, and picklist (which is a predefined list of allowable values of one of the other data types). There is also a Format property which is used to format the measure values when they appear in the HTML UI or Excel UI.

There are many ways that measure values can be set or modified. Measure data can be uploaded to GraphCube (such as through a file upload or data interface). Measures can also be populated sort through IBPL commands or through the execution of procedures or plug-ins. Measures can be manually edited through the HTML UI and/or Excel UI. Lastly, a measure can have an active rule defined for it that will set the value according to some formula.

For example, the measure *Net Revenue* is calculated as:

```
Measure.[Net Revenue] = Measure.[Gross Revenue] - Measure.[Trade Spend] -  
Measure.[Other Sales Reductions];
```

Another measure property is the Editable property. This is a boolean that controls whether the measure can be edited or not. In this example, *Gross Revenue*, *Trade Spend* and *Other Sales Reductions* are all editable. If a measure is editable, then it can be changed directly in the HTML UI and/or Excel UI. It is not necessary for a measure to be editable for the value to be changed through other ways such as file upload or IBPL commands. Measures with active rules are generally not marked as editable, though there is nothing to prevent it. In fact, a measure with an active rule can be changed through IBPL (though the value will likely change back the next time the active rule is executed).

There is a member property called Aggregation that controls how measure values at higher levels of the involved dimensions are calculated. Measure data is stored at the dimension level that is defined for the measure group (which is not necessarily the lowest level available in the dimension). This is referred to as the leaf level. For example, the measure group that *[Net Revenue]* belongs to has granularity defined as [Product].[Sub Category], [Sales Domain].[Region], [Time].[Fiscal Month] and [Version].[Version Name]. In that case, the measure data exists and is stored physically only at the intersection of these levels. The measure can be used at higher levels

o9 Internal use Only 13

in any or all of these dimensions, in which case it is calculated on the fly according to the Aggregation property defined for the measure. Here are the choices:

- Sum -- the value of the parent is the sum of the non-null child values. If all child values are null, then the sum is null.
- Min -- the value of the parent is the minimum of the non-null child values. If all child values are null, then the minimum is null.
- Max -- the value of the parent is the maximum of the non-null child values. If all child values are null, then the maximum is null.
- Median -- the value of the parent is the median of the non-null child values. If the number of values is odd, then this is the middle value in a sorted list of the values. If the number of values is even, then this is the average of the middle two values in a sorted list of the values. If all child values are null, then the median is null.
- Avg -- the value of the parent is the average of all child values, where null child values are treated as zero. If all child values are null, then the average is zero.
- AvgNonNull -- the value of the parent is the average of all non-null child values. If all child values are null, then the average is null.
- AvgNonNullOrZero -- the value of the parent is the average of all non-null and non-zero child values. If all child values are null or zero, then the average is null.
- FirstChild -- the value of the parent is the first child value in the time dimension, and is the sum of the non-null child values in the other dimensions. If the first child value is null, then the parent value is null.
- LastChild -- the value of the parent is the last child value in the time dimension, and is the sum of the non-null child values in the other dimensions. If the last child value is null, then the parent value is null.
- Computed -- the value of the parent is determined by a formula that references other measures. •
- Null -- the value of the parent is always null, regardless of the value of the children. It is possible to set a value for a measure at a higher level than the measure is defined at. In this case, the Spreading policy is used (along with any defined basis measures) to determine how to update

where

the values at the leaf level. If no spreading policy is defined, then an error is thrown and no updates are made. Here are the choices:

- CopyToLeaves -- the value at the higher level is copied to the leaf level. So if the parent value is set to 100, then all of the leaf values are set to 100. Obviously, this policy does not make sense if the Aggregation is set to Sum, since the value at the parent will not be 100 after the leaves are updated (assuming there is more than one child at the leaf level).
- CopyToLeaves with Assortment Basis -- <TBD>
- DistributeToLeaves with Basis -- <TBD>
- DistributeToLeaves with Null Basis -- <TBD>
- DistributeToLeaves with Assortment Basis -- <TBD>
- DistributeWeightedAverage -- <TBD>
- DistributeToLeavesNonSummable -- <TBD>

Deletion of Measuregroup Data

LS supports deleting fact data of measuregroups.

Examples:

- DELETE DATA FOR MODEL [Capacity Availability] WHERE {Version.[Version Name].[CurrentWorkingView]};

DELETE DATA FOR MODEL [Capacity Availability]

o9 Internal use Only 14

Syntax

Square Brackets

In IBPL, the names of objects such as dimensions, levels, members and so on are typically enclosed in square brackets. For example, the *Product* dimension is [Product], and the *Product Name* level is [Product].[Product Name].

The square brackets are optional if the name does not have any spaces or special characters. So, [Product].[Category] is the same as Product.Category. However, Product.Product Name is an error, because *Product Name* has a space in it.

Instead of square brackets, double quote marks can be used. So all of the following are all equivalent to one another:

- Product.[Product Name]
- Product."Product Name"
- "Product".[Product Name]
- "Product"."Product Name"
- [Product].[Product Name]
- [Product]."Product Name"

Dot Notation

In IBPL, a period is used to chain together objects and functions. One example is above, where the *Product Name* level is referred to by [Product].[Product Name]. In most contexts, when a dimension object is followed by a period, then the next object is a level. One counterexample is

where

found with the Children function:

```
[Time].[Fiscal Year].find([FY2013]).children([Time].[Fiscal YQM])
```

In this case, [Time].[Fiscal YQM] refers to a hierarchy.

Technically, the datatype of [Product].[Product Name] is a member set. So if you execute the following query in IBPL, the result will be the members of the Fiscal Year level:

```
select [Time].[Fiscal Year];
```

A member set followed by a period is then followed by a member object (or by a function that operates on a member set). So the following are all ways to refer to the member FY2013 which is a member of the Fiscal Year level:

- [Time].[Fiscal Year].[FY2013]
- [Time].[Fiscal Year].find([FY2013])
- [Time].[Fiscal Year].filter(#.Name==[FY2013]).element(0) A member object followed by a period is then followed by a level object (or by a function that operates on a member). So the following are all ways to get a member set with the Fiscal Quarter members that are children to FY2013:
- [Time].[Fiscal Year].[FY2013].[Fiscal Quarter]
- [Time].[Fiscal Year].[FY2013].children([Time].[Fiscal YQM])
- [Time].[Fiscal Year].[FY2013].relatedMembers([Fiscal Quarter])

o9 Internal use Only 15

IBPL Rules

Named Sets

A named set is like a variable whose type is a Member Set. Named sets can be used in the same places in IBPL as member sets are used.

Example:

```
create set CurrentWorkingView = [Version].[Version Name].filter(#.Key==0);
```

This example creates a set called “CurrentWorkingView” that includes only the member with Key equal to zero (which is the CurrentWorkingView member).

A named set is global and persists as long as GraphCube is running. Named sets are dynamic in the sense that the actual members in the list can change if members are created or deleted, or if attributes of the members change.

Example:

```
create set CurrentAndFutureFiscalMonths = [Time].[Fiscal Month].filter(~(#[Fiscal Month$InPast]));
```

This example defines a set of *Fiscal Month* members where the InPast property is false (notice the use of the Not operator (~)). In other words, this set contains the current and future fiscal month members. If the current month is changed, then this named set will automatically be updated.

where

Example:

```
create set PlanningMonths = &CurrentAndFutureFiscalMonths.filter(#.Key <=
    &CurrentAndFutureFiscalMonths.element(12));
```

This example defines a set of *Fiscal Month* members that are part of the *CurrentAndFutureFiscalMonths* named set and have a *Key* property that is less than the twelfth element of the same named set. I.e., It is the first twelve members of the named set. This example shows that a named set can be based off of another named set. The named set is identified by the ampersand (&) that precedes the name.

Note: you may see come across IBPL code where the keyword *dynamic* is specified. There is no functional difference to using the word *dynamic*. That is a vestigial feature from an earlier design of the IBPL code.

Example:

```
create dynamic set CwvAndScenarios = [Version].[Version Name].filter(#.Key
    <= 0); active rule
```

Active Rules

GraphCube is the (in-memory) storage & computational engine for o9 Business Planner. At startup, GraphCube reads a set of rules defined in one or more rule files. These rules define how any measure should be calculated by the GraphCube engine. At runtime, as users edit measures, GraphCube engine computes the entire chain of dependent computations. Hence, these are called “Active rules”, since these rules are evaluated immediately in response to edits.

o9 Internal use Only 16

Active rules are used to set the formulas for measures. The active rule declaration consists of a scope statement with the granularity of the active rules, followed by one or more measure formulas, and ending with and end scope statement.

Example:

```
scope: ([Version].[Version Name].[CurrentWorkingView] *
    [Product].[Sub-Category] * [Sales Domain].[Region] *
    Time.[Fiscal Month]);
Measure.[Net Revenue] = Measure.[Gross Revenue] - Measure.[Trade Spend] -
    Measure.[Other Sales Reductions];
end scope;
```

The scope statement contains member sets and/or single members, separated by asterisks (*). This example contains a single member for the [Version].[Version Name] level, and member sets for the other dimensions/levels. This active rule only applies to the *CurrentWorkingView* member.

Example:

```
scope: ([Version].[Version Name].filter(#[Key] == 0) *
    [Product].[Sub-Category] * [Sales Domain].[Region] *
    [Time].[Fiscal Month].filter(~(#[Fiscal Month$InPast])));
Measure.[Net Revenue] = Measure.[Gross Revenue] - Measure.[Trade Spend] -
    Measure.[Other Sales Reductions];
end scope;
```

This example uses the filter function to restrict the scope to the *Fiscal Month* members that are not in

where

the past (as well as to the *CurrentWorkingView* member).

Example:

```
scope: (&CurrentWorkingView * Product.[Product Name] * SalesDomain.[Sales
Domain Name] * &CurrentAndFutureFiscalMonths);
    Measure.[Net Revenue] = Measure.[Gross Revenue] - Measure.[Trade Spend] -
        Measure.[Other Sales Reductions];
    Measure.[Gross Profit] = Measure.[Net Revenue] - Measure.[COGS];
end scope;
```

This is an example with multiple rules defined with the same scope. Notice that the scope statement uses named sets in place of [Version].[Version Name] and [Time].[Fiscal Month].

Member properties can also be used in the active rules as shown below. [Organization] is a property of [Product Name].

```
scope:([Time].[Fiscal Month] * [Product].[Product Name] * [SalesDomain].[Sales Domain
Name] * [Version].[Version Name].[CurrentWorkingView]);

    Measure.[Cases] = if (Product.#.[Organization] == "Optimal Beverages") then
        Measure.[Cases] +10;

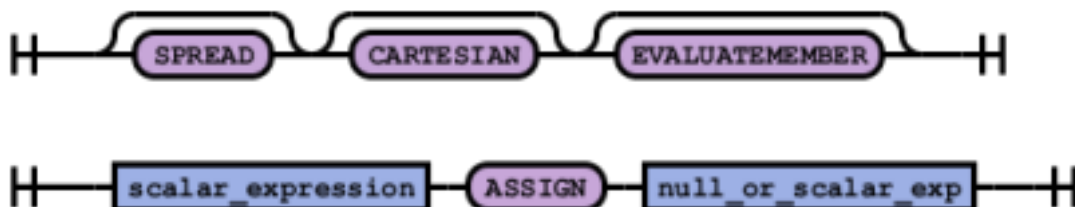
end scope;
```

The grammar productions for a scope statement are:

The scopePrefix can be one of the following keywords:

1. [Spread](#)
2. [Cartesian](#)
3. [evaluate member](#)

o9 Internal use Only 17



The scope consists of a sequence of AttributeMemberSets separated by asterisks (*). The dimensions and levels of the member lists must match the dimensionality defined for the measure group in the configuration database. However, a AttributeMemberSet can be a subset of the members (as is the case in this Example with the Version dimension and level "Version Name").

Named sets can also be used in the scope. Example, if the following named sets are defined:

```
create set CwvAndScenarios = [Version].[Version Name].filter(#.[Key] <= 0);
create set SCPPProductLevel = [Product].[Category];
create set SCPSalesDomainLevel = [SalesDomain].[P&L Region];
create set PlanningPeriods = [Time].[Fiscal Month];
```

Then the scope can be changed to the following:

```
scope:(&CwvAndScenarios * &SCPPProductLevel
    * &SCPSalesDomainLevel * &PlanningPeriods);
```

When setting a formula for a measure, the right-hand side (RHS) of the equation is allowed to have different dimensionality than the left-hand side (LHS).

where

```
scope: (&CurrentWorkingView * Product.[Product Name] * SalesDomain.[Sales Domain Name] *
Time.[Fiscal Months]);
  Measure.[Segment Forecast Detail Gross Revenue].Editable = 1;
end scope;
```

```
scope: (&CurrentWorkingView * Product.[Product Category] * SalesDomain.[Sales Region] *
Time.[Fiscal Months]);
    Measure.[Segment Forecast Gross Revenue] = Measure.[Segment Forecast Detail Gross
Revenue];
end scope;
```

In this Example, the active rule for Segment Forecast Gross Revenue is an aggregation rule. The RHS (e.g., Segment Forecast Detail Gross Revenue) is defined at a lower level in the Product and SalesDomain dimensions than the LHS (Segment Forecast Gross Revenue). The rule aggregates the RHS to a higher level in the Product and SalesDomain dimensions and sets it as the value for the LHS.

If using a named set in an Active rule and you are not specifying the exact set of version, the rule will be run only for CurrentWorkingView. So to avoid that you should define the exact set of versions you need to run the rule on.

Recurrence rules are used to model time-series based computations wherein the computation in a particular time bucket depends on the result of the computation of the previous time bucket, as in this example: `execut`

$$\begin{aligned} & \text{Diagram 1}(\text{Diagram 2}) = \\ & \text{Diagram 3} + \text{Diagram 4} - \text{Diagram 5} \end{aligned}$$
$$\begin{array}{c} \textcircled{1}\textcircled{2}\textcircled{3}\textcircled{4}\textcircled{5}\textcircled{6}\textcircled{7}\textcircled{8}\textcircled{9}\textcircled{10}\textcircled{11}\textcircled{12}\textcircled{13}\textcircled{14}\textcircled{15}\textcircled{16}\textcircled{17}\textcircled{18}\textcircled{19}\textcircled{20}\textcircled{21}\textcircled{22}\textcircled{23}\textcircled{24}\textcircled{25}\textcircled{26}\textcircled{27}\textcircled{28}\textcircled{29}\textcircled{30}\textcircled{31}\textcircled{32}(\textcircled{1}\textcircled{2}) = \\ \textcircled{1}\textcircled{2}\textcircled{3}\textcircled{4}\textcircled{5}\textcircled{6}\textcircled{7}\textcircled{8}\textcircled{9}\textcircled{10}\textcircled{11}\textcircled{12}\textcircled{13}\textcircled{14}\textcircled{15}\textcircled{16}\textcircled{17}\textcircled{18}\textcircled{19}\textcircled{20}\textcircled{21}\textcircled{22}\textcircled{23}\textcircled{24}\textcircled{25}\textcircled{26}\textcircled{27}\textcircled{28}\textcircled{29}\textcircled{30}\textcircled{31}\textcircled{32}(\textcircled{1}\textcircled{2} - 1) \end{array}$$

Recurrence rules are specified by using a scope statement with a recurrence prefix, as in this example: *recurrence scope: (Version.[Version Name].[CurrentWorkingView]):*

$$\text{Measure.[EOM AUC]} = (\text{Measure.[BOM\$]} + \text{Measure.[Receipts\$]}) / (\text{Measure.[BOM U]} + \text{Measure.[Receipts U]}):$$
$$\text{Measure.[BOM U]} = \text{Measure.[BOM\$]} / \text{Measure.[BOM AUC]}:$$

```
Measure.[BOM AUC] = Measure.[EOM AUC]@Time.#.LeadOffset(-1);
Measure.[BOM$] = if (Time.# == CurrentTimeBucket) then Measure.[Actual Inventory] else
    Measure.[EOM$]@Time.#.LeadOffset(-1);
```

$$Measure.[Receipts\ U] = Measure.[Receipts\$] / Measure.[AUC];$$
$$Measure.[COGS] = (Measure.[Sales] * Measure.[EOM AUC]) ;$$
$$\text{Measure.}[EOM\$] = \text{Measure.}[BOM\$] + \text{Measure.}[Receipts\$] - \text{Measure.}[COGS\$];$$
$$\text{Measure.}[EOM\ U] = \text{Measure.}[BOM\ U] + \text{Measure.}[Receipts\ U] - \text{Measure.}[Sales\ U]$$

where

U] end scope;



Constraints on recurrence scope equations:

1. Can only contain skew along time (Time.#.LeadOffset)
2. Cannot contain any skew in the LHS (any recurrence can be re-arranged to specify computation only in the current bucket)
3. There is no cycle in the recurrence. An example of cyclic recurrence would be the following two equations:
 - a. $\diamond\diamond(\diamond\diamond) = \diamond\diamond(\diamond\diamond - 1)$
 - b. $\diamond\diamond(\diamond\diamond) = \diamond\diamond(\diamond\diamond + 1)$
4. The recurrence works either move forward in time or backward in time; but not in both directions. In other words, all the leadoffset indexes used in the set of assignments can be either positive or negative; but not a combination of both.

Ordered Recurrence Rules

Execution of rules within an ordered recurrence is similar to rules within a recurrence rule, except that the rules are evaluated in the exact order specified in the rule file, whereas the rules in a recurrence scope are executed in dependency order.

Block Scope Rules

For better performance, the scope expression can be prefixed with a **block** keyword when the following conditions hold:

1. All LHS measures must belong to the same measure group
2. When the block scope appears in an active rule file,
 - a. None of the RHS measures can be of finer grain

o9 Internal use Only 19

- b. None of the RHS measures can have coordinate expressions, as in $\text{Measure.X} = \text{Measure.Y} @ (\text{Time.} @ . \text{LeadOffset}(-1))$, the measure Y on the RHS has a coordinate expression $@ (\text{Time.} @ . \text{LeadOffset}(-1))$
3. When the block scope is executed on-demand (when it is residing within a procedure, or when it is invoked from an action button), the restrictions 2.a and 2.b do not apply.
 4. A Spread scope cannot be used along with the block scope.

Action Button Debug Statements

In an action button one can execute many different IBPL statements. As part of debugging one would like to check logs to monitor the progress of each step. This requires to print some custom debug info. For eg.

```
print "my action button command is starting";
ibpl1;
print "first ibpl done";
ibpl2;
print "second ibpl done";
...
```

where

```
print "my action button is done".
```

This can be achieved as follows

Use a simple select statement with the information that you want to print. Syntax as follows **select** "my action button command is starting";

```
ibpl1;  
select "first ibpl done";  
ibpl2;  
select "second ibpl done";  
...  
select "my action button is done".
```

Save Performance Improvement

During save, the GraphCube service is not available for users. The save performance has been improved significantly.

Functionality Description

The GraphCube maintains a list of tables that were included in the previous backups. This list (BackupVersionInfo.json) has information about each MG-Version combination and in which backup they were included.

During save, when any table is saved, its information is updated in this file. All the tables are saved simultaneously, giving max parallelization and max performance, but the updates to this file were done serially or sequentially. This was a major performance drain and this bottleneck is removed. Almost every tenant will benefit from this change.

o9 Internal use Only 20

IBPL Procedures

An IBPL procedure contains commands that are executed only when the procedure is executed. In the example below: If a user edits Net Sales, Gross Profit does NOT change automatically. It changes only when the procedure is executed. Procedures are executed from a client, e.g. UI or Excel, by the "*exec Procedure <procedurename>*" command. Procedures are used to manipulate the values of editable measures.

Example:

```
Create Procedure UpdateGrossProfit  
Begin scope:  
    (Version.[Version Name].[CurrentWorkingView]);  
    Measure.[Gross Profit] = Measure.[Net Sales] – Measure.[COGS];  
end scope;  
end;
```

Example:

```
create procedure ForecastModel
```

where

```

begin scope:
    (&CwvAndScenarios * &Products * &SalesDomains * &PlanningPeriods);
    Measure.[FM Segment Forecast Detail Gross Revenue Three Month Moving
    Average] =
        (coalesce(Measure.[Segment Forecast Detail Gross
        Revenue]@(Time.#.LeadOffset(-3)), 0)
        + coalesce(Measure.[Segment Forecast Detail Gross
        Revenue]@(Time.#.LeadOffset(-2)), 0)
        + coalesce(Measure.[Segment Forecast Detail Gross
        Revenue]@(Time.#.LeadOffset(-1)), 0)) / 3;
end scope;
end;

```

The scope block in this procedure looks the same as in an active rule, however, it is not setting a formula for the measure “FM Segment Forecast Detail Gross Revenue Three Month Moving Average”. Instead, it is calculating and setting the value for the measure. When the procedure is executed, “FM Segment Forecast Detail Gross Revenue Three Month Moving Average” will be updated according to the current values of “Segment Forecast Detail Gross Revenue”. However, if the values of “Segment Forecast Detail Gross Revenue” change later, this will not have any impact on the value of “FM Segment Forecast Detail Gross Revenue Three Month Moving Average” (unless, of course, the procedure is executed again).

Note: incremental plan runs once per procedure after all statements in the procedure have been executed

Procedures can use Named Sets

o9 Internal use Only 21

Parameterized IBPL procedures

Starting from October 2017 release the parameterized IBPL procedures are supported in mPower. For example, one can define a parameterized procedure for a plugin command and link it to an action button on Web UI. During execution it will prompt for values for the specified parameters. User can input and complete the execution. For different runs, user can provide different values to the parameter and observe the plan results.

Syntax for Parameterized IBPL procedures

Create Procedure

- Use of “parameterized” keyword in create procedure syntax distinguishes the two types (Parameterized v/s Basic IBPL procedure)
- Parameterized procedures can optionally specify parameter schema json; but it is not allowed for basic procedures
- Body of the procedure must be valid ibpl for basic procedures. It can contain Mustache tags for parameterized procedures

where

- Body of the procedure of parameterized procedure must start with “begin procedure” and end with “end procedure;”. For basic procedures it can omit the procedure in the “begin procedure” and “end procedure” (for backward compatibility)
- Parameter schema for procedure creation and arguments for procedure execution are specified as Json. If the Json syntax conflicts with IBPL syntax (for example, square brackets for arrays; single quote for properties, etc), enclose the Json string within “begin jsonblock” and “end jsonblock” tags.

Execute Procedure

- Parameterized procedure execution must specify argument json; It's not relevant for basic procedures

Limitations

- Currently any default value for the parameters in the parameterized IBPL procedures cannot be specified
- This is yet to be supported with online model changes (which means Graphcube server save and restart are required)
- update members from parameterized procedure is not supported.

o9 Internal use Only 22

Parameterized IBPL Procedures Usage Examples

Simple string input ...

... to run stat forecast plugin

Defining procedure

```
create parameterized procedure p_RunStatForecast_1
begin procedure
    exec plugin instance [Stat Fcst] for Measures {[Cleansed Sales Actuals (L2)]}
using scope (&AllSalesParts_Item * Dealer.[Dealer Forecast Group].filter(#.Name ==
[{{DFG}}]) * &AllMonths * &CWV)
    using arguments {[Best Fit], true), ([Persist All], true)};
end procedure;
```

where

Executing procedure

```
exec procedure p_RunStatForecast_1 {"DFG" : "0/Southwest"};
```

IBPL executed for parameterized procedure P_RUNSTATFORECAST_1:

```
exec plugin instance [Stat Fcst] for Measures {[Cleansed Sales Actuals (L2)]}  
using scope (&AllSalesParts_Item * Dealer.[Dealer Forecast Group].filter(#.Name  
== [0/Southwest]) * &AllMonths * &CWV)  
using arguments {[Best Fit], true}, {[Persist All], true});
```

... to run SCS

Defining procedure

```
create parameterized procedure p_RunSCS
```

```
begin procedure
```

```
    exec plugin instance SCS_101 for Measures {[SCS Demand Quantity]} using scope (&CWV *  
Item.[Category] * &CurrentAndNext52Weeks)  
    arguments {[Material Constrained], "{{MC}}"}  
    , ([Capacity Constrained], "{{CC}}")  
    , ([Delete SCS Plan Data For Version], "True")  
    , ([End Time Bucket Name], &CurrentWeek.element(0).leadoffset(52).Name)  
    , ([Start Time Bucket Name], &CurrentWeek.element(0).Name)  
    , ([Inventory Plan Policy], "{{IPP}}")  
    , ([Inventory Plan Type], "QUANTITY")  
    , ([Respect Max Stock Constraint In JIT Planning], "False");  
end procedure;
```

Executing procedure

```
exec procedure p_RunSCS {"MC" : "True", "CC" : "False", "IPP" : "QUANTITY"}; o9 Internal use Only
```

23

Using #if / #else ..

... on RHS of a regular scope statement

Defining procedure

```
create parameterized procedure p_Test_If_1
```

```
begin procedure
```

```
    scope:(&CWVandS * &PastMonths);  
        Measure.[DBU Forecast (L10)] = {{#if Param_Check}} Measure.[Sales Actuals (L10)] *  
{{Param_1}}  
        {{#else}} Measure.[Sales Actuals (L10)]  
        {{/if}};
```

```
    end scope;
```

```
where
```

end procedure;

Executing procedure

exec procedure p_Test_If_1 {"Param_Check" : "True", "Param_1" : 5};

| |
|---|
| IBPL executed for parameterized procedure P_TEST_IF_1: |
| scope:(&CWVandS * &PastMonths); Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)] * 5 ; end scope; |

... inside regular scope statement to run different assignments

Defining procedure

create parameterized procedure p_Test_If_2

begin procedure

 scope:(&CWVandS * &PastMonths);

 {{#if Param_Check}}

 Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)] * {{Param_1}};

 {{#else}}

 Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)]; {{/if}}

 end scope;

end procedure;

Executing procedure

exec procedure p_Test_If_2 {"Param_Check" : "True", "Param_1" : 5};

IBPL executed for parameterized procedure P_TEST_IF_2:

o9 Internal use Only 24

| |
|---|
| IBPL executed for parameterized procedure P_TEST_IF_2: |
| scope:(&CWVandS * &PastMonths); Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)] * 5; end scope; |

... to run different scope statements

Defining procedure

create parameterized procedure p_Test_If_3

begin procedure

 {{#if Param_Check}}

 scope:(&CWVandS * &PastMonths);

 Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)] * {{Param_1}};

where

```

        end scope;
    {{#else}}
        scope:(&CWVandS * &PastMonths);
        Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)]; end scope;
    {{/if}}
end procedure;

```

Executing procedure

```
exec procedure p_Test_If_3 {"Param_Check" : "True", "Param_1" : 5};
```

IBPL executed for parameterized procedure P_TEST_IF_3:

```

scope:(&CWVandS * &PastMonths);
Measure.[DBU Forecast (L10)] = Measure.[Sales Actuals (L10)] * 5; end scope;

```

Running inside an action button

Defining procedure

```

create parameterized procedure p_RunStatForecast_2
begin procedure
    exec plugin instance [Stat Fcst] for Measures {[Cleansed Sales Actuals (L2)]}
    using scope (&AllSalesParts_Item * Dealer.[Dealer Forecast Group].filter(#.Name in
    {{#ibplList DFG}}) * &AllMonths * &CWV)
    using arguments {[Best Fit], true}, {[Persist All], true});
end procedure;

```

Executing procedure manually

```
exec procedure p_RunStatForecast_2 {"DFG" : ["0/Southwest", "0/Southeast"]}; o9 Internal
```

use Only 25

IBPL executed for parameterized procedure P_RUNSTATFORECAST_2:

```

exec plugin instance [Stat Fcst] for Measures {[Cleansed Sales Actuals (L2)]}
using scope (&AllSalesParts_Item * Dealer.[Dealer Forecast Group].filter(#.Name
in {"0/Southwest","0/Southeast"}) * &AllMonths * &CWV)
using arguments {[Best Fit], true}, {[Persist All], true});

```

Action button config

Fields -

```

Name - f_DFG
type - picklist
source - members -- Dealer.[Dealer Forecast Group]
selectionType - multiple

```

IBPL rules -

where

```
exec procedure p_RunStatForecast_2 {"DFG" : [{"#csv f_DFG"}]};
```

Action button execution

Selection -

```
f_DFG - "0/Northeast", "0/Others", "0/Southeast"
```

| IBPL executed for parameterized procedure P_RUNSTATFORECAST_2: |
|--|
| <pre>exec plugin instance [Stat Fcst] for Measures {[Cleansed Sales Actuals (L2)]} using scope (&AllSalesParts_Item * Dealer.[Dealer Forecast Group].filter(#.Name in {"0/Northeast", "0/Others", "0/Southeast"}) * &AllMonths * &CWV) using arguments {[Best Fit], true), ([Persist All], true)};</pre> |

Passing all members

syntax:

```
scope: (Product.[SKU]{#if IsInteractive}}.filter(#.Name in {{#ibplList SKUs}}){/if} * Time.[Month] *
...); ...
```

the syntax to send in an array parameter

Parameter setup:

Name : ItemList

Type : array

Item Type : string

In the scope, use:

```
[SCSItem].[Item].filter(#.Name in {{#ibplList ItemList}})
```

To run from meru:

```
exec procedure RunConstrainedPlanByItem {"Version":"{{Version}}"
```

```
, "ItemList":["Item1", "Item2"]}; o9 Internal use Only 26
```

Locked intersections and the volumes associated with it:



After you run a procedure, the measure is not updated if it is locked.

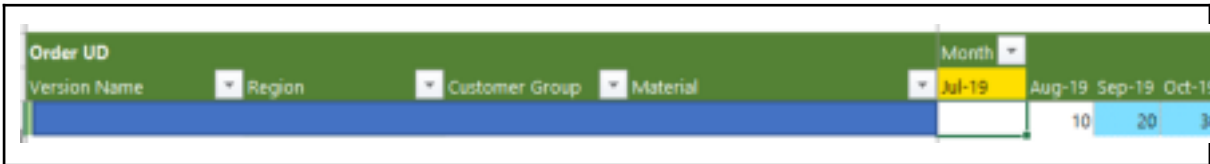
where

To identify the locked measure:

- 1. Run a select statement at the grain of the measure.
- 2. Include the lock command as one of the required output columns.

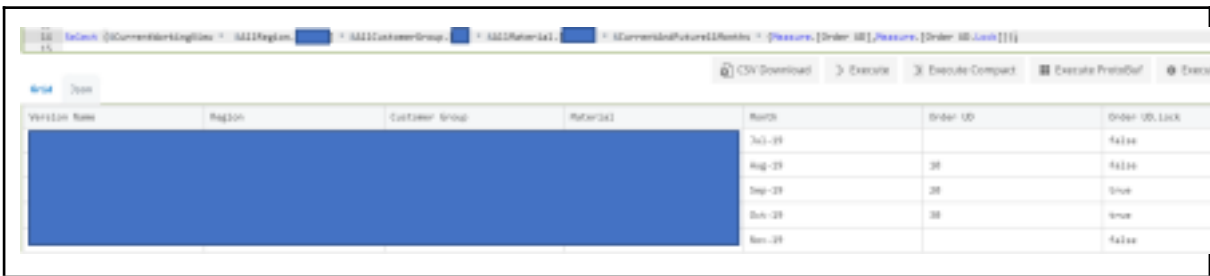
Example:

When the measure grain is Material-Region-Customer Group-Month, the data displayed in the excel UI is:



| Order UD | Version Name | Region | Customer Group | Material | Month | Aug-19 | Sep-19 | Oct-19 |
|----------|--------------|--------|----------------|----------|--------|--------|--------|--------|
| | | | | | Jul-19 | 10 | 20 | 3 |

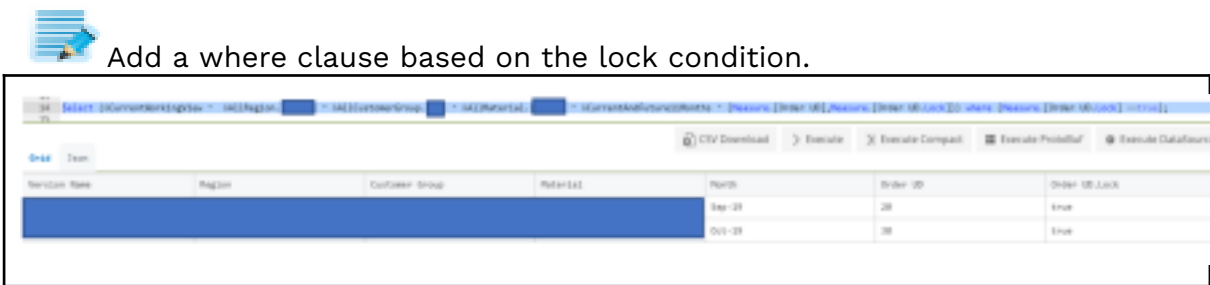
When you execute, the following query will output all data but also include the lock status.



```
SELECT [CurrentWorkingDir] + 'AllRegion' + 'AllCustomerGroup' + 'AllMaterial' + 'CurrentAndFutureMonths' + 'Measure [Order UD], Measure [Order UD Lock]'
```

| Version Name | Region | Customer Group | Material | Month | Order UD | Order UD Lock |
|--------------|--------|----------------|----------|--------|----------|---------------|
| | | | | Jul-19 | | False |
| | | | | Aug-19 | 30 | False |
| | | | | Sep-19 | 30 | True |
| | | | | Oct-19 | 30 | True |
| | | | | Nov-19 | | False |

When you execute, the following query will output only data where the intersection is locked.



```
SELECT [CurrentWorkingDir] + 'AllRegion' + 'AllCustomerGroup' + 'AllMaterial' + 'CurrentAndFutureMonths' + 'Measure [Order UD], Measure [Order UD Lock] WHERE [Measure [Order UD Lock]] = True'
```

| Version Name | Region | Customer Group | Material | Month | Order UD | Order UD Lock |
|--------------|--------|----------------|----------|--------|----------|---------------|
| | | | | Sep-19 | 30 | True |
| | | | | Oct-19 | 30 | True |

o9 Internal use Only 27
Based on the example displayed above, you can run the following query to get all locked intersections for a given measure:

where

*Select (&CurrentWorkingView * &AllRegion * &AllCustomerGroup * &AllMaterial*
&CurrentAndFutureMonths * {Measure.[Order UD],Measure.[Order UD.Lock]}) where
{Measure.[Order UD.Lock]}==true;*

IBPL Commands

Read Queries

Member Query

Which returns a list of all products in the model.

Select (Product.[Product Name]);

Query to know the current user (logged in users)

select CurrentUser();

Measure Query

Which returns the gross revenue for all product x sales domains x months x versions in the system

*Select (Version.[Version Name] * Product.[Product Name]*
SalesDomain.[Sales Domain Name] * Time.[Fiscal Months]*
{Measure.[Gross Revenue]});*

Below cmd returns the Beginning On Hand, Independent Demand and other measures specified in the query for all weeks x locations x SKUs x versions in the system

*Select ([Date].[Reporting Week] * [Location].[Location] * [Product].[BDC Name] * [Version].[Version Name]) on row,
({Measure.[Beginning On Hand], Measure.[Independent Demand], Measure.[Propagated Demand], Measure.[Suggested Replenishment], Measure.[Target Inventory], Measure.[Target WOS], Measure.[Total Demand]}) on column;*

Other Queries

Select "Rule Starts Here"; or select "Rule Ends Here";

Select (5+2); or select (5/2);

Attributes for a Measure, Measure Group

To get the list of Plan and Measure Group, below IBPL can be used

Select ([_SchemaPlan].[_Plan] * [_SchemaPlan].[_MeasureGroup]);

To get the list of Measure Group and Measure, below IBPL can be used

Select ([_SchemaPlan].[_MeasureGroup] * [_SchemaPlan].[_Measure]);

To get the leaf level attribute for a measure and measure group below IBPL can be used o9 Internal use

```

select (_SchemaPlan.[_Measure].[<Measure
Name>].relatedmembers([_MeasureGroup]).relatedmembers([_DimensionUsage]) ) on row, () on
column include memberproperties { _SchemaPlan.[_DimensionUsage],
[DimensionUsageDimensionName], [DimensionUsageDimensionProperty],
[DimensionUsageDimensionAttribute], [DimensionUsageAttributeProperty]};

select (_SchemaPlan.[_MeasureGroup].[<Measure Group
Name>].relatedmembers([_MeasureGroup]).relatedmembers([_DimensionUsage]) ) on row, () on
column include memberproperties { _SchemaPlan.[_DimensionUsage],
[DimensionUsageDimensionName], [DimensionUsageDimensionProperty],
[DimensionUsageDimensionAttribute], [DimensionUsageAttributeProperty]};

```

To get details of a measure

```
select _SchemaPlan.[_Measure].[measure name];
```

Write Queries

Single-Cell Update Query

Sets the value of this specific measure, Segment Target Gross Revenue, for Carbonated Cola, for Segment East, for M08-2013, to the value specified.

Scope based (Multi-Cell) Update Query

Sets the value for this measure for the specified scope, i.e. for CurrentWorkingView, for this product Cosmo Cola 12 oz., for sales domain Target, for all times, to be stated value. The right hand side of this expression could be a measure itself, not just a value.

Scope: (&CurrentWorkingView Product.[Product Name].Filter(#.Name = "Cosmo Cola 12 oz")* SalesDomain.[Sales Domain Name].filter(#.*

*Name == "Target") * Time.[Fiscal Months]);Measure.[Gross Revenue]=1000;end scope;*

Direct Inserts (DI)

Direct Inserts are an approach to adding data to a measure-group or graph-table where the entire contents of the table are being replaced by a fresh set of rows, i.e. the table is being truncated and repopulated. Plugins such as Supply-Chain-Solver perform such operations on their output tables, for example pegging export tables. Today such updates are made through a specialized version of the upsert operator that maintains the updated data as deltas to the base tables. Future queries on these tables require a merge between the base tables and the delta. The Direct-Insert operator performs the truncate-and-insert operation differently. It writes all the new data for a table to disk, and switches LiveServer to use the newly created "base" tables instead of the original tables. The advantage is that the new data in tables is immediately persisted on disk and future queries do not need to do a merge between the base table data and delta to retrieve the correct values for individual rows. Additionally, if there are no more changes to the table before the next save, then the on-disk tables can be directly used instead of doing a full save for the "direct-insert"ed tables, saving time.

o9 Internal use Only 29

where

Currently, the Direct-Insert capability has only been used in the pegging-table exports of the Supply-Chain Solver plugin. It is disabled by default. Direct-Insert can be enabled through a dynamic parameter **DirectInsertEnabled**.

Limitations

- Currently, on benchmark data-sets, Direct-Insert has been measured to be about 30% slower than the standard upserts for InMem btree mode, and approximately 50% faster than LSM btree mode.

Member Management

Member Creation

```
CREATEMEMBER(CurrentTimeModel.[Fiscal Quarter]=  
{ToDatetime("2013-04-21 00:00:00.000"),"Q2-2013"});
```

Member Update

Updates the "Version Creation Date" property of Version "V1" to specified date

```
UPDATEMEMBER(Version.[Version Name]= {"V1"},  
Version.[Version Creation Date]=  
{ToDatetime("2012-12-02 00:00:00.000").});
```

Updates the "Display Name" property of Product.[BDC Name].[SKU1] to "SKU One"

The below command searches for the member with key : mem_key, and **tries** to Update the "Name" of Product.[BDC Name].[SKU1] to "SKU_8", this is successful only if all conditions are met, along with Updating "Display Name" property. displayname

```
UPDATEMEMBER(Product.[BDC Name]= {mem_key, "SKU_8"},  
Product.[BDC Name$DisplayName]= {"SKU One",.});
```

Note : The order in which the values are given in { }.

Member Delete(Trimming a Dataset)

The below command is used to SOFT-Delete the member (i.e \$InActive flag is set to True), This command is used to delete just the member specified, **constraint** : Make sure there are no children (low level members related to this member in all Hierarchies). Deletes all the fact data against this member.

```
DELETEMEMBER Dim.[DimAtt] = {"AttMem"};
```

For Ex: DELETEMEMBER(Location.[City] = {"City1"});

The below command is for SOFT-Deleting the member (i.e \$InActive flag is set to True), and all the children (low level members related to this member in all Hierarchies) of this member. It does a cascaded delete to its descendants.

```
DELETEMEMBER(Dim.[DimAtt] = {"AttMem"})CASCADE;
```

For Ex: DELETEMEMBER(Location.[Region] = {"R1"}) CASCADE;

the model (Hard Delete). This command deletes all the members which are soft deleted. *PURGE MEMBERS(Customer.[Planning Customer]);*

Version Management

CreateVersion(0,1,"V2", false, false);

| | |
|------------|--|
| Parameter1 | Key of the source Version/Scenario |
| Parameter2 | Key of the plan to be version(future) |
| Parameter3 | Version Name of the plan to be version(future) |
| Parameter4 | IsOfficial? (Official, Published version from this planning cycle) |
| Parameter5 | isAOP? |

CreateScenario(0, 1, "Proposal Selection Scenario");

| | |
|------------|---------------------------------------|
| Parameter1 | Key of the source Version/Scenario |
| Parameter2 | Key of the plan to be version(future) |
| Parameter3 | Name of the Scenario to be created |

The below commands delete existing Version/Scenario V1.

DELETEVERSION(Version.[Version Name].[V1]);

DELETEVERSION("V1");

There are special properties to Version like [Access Role], [Category], [Is Official], [Is Target]. For updating these values use the UPDATEVERSIONPROPERTY cmd.

UPDATEVERSIONPROPERTY ("V1", Version.[Is Target], true);

Release Memory IBPL command

Users can release the memory consumed by measure-group and graph tables. When a tenant has loaded a lot of tables into memory for querying or computations and it is expected that these tables will not be used again soon, issuing this IBPL command will reclaim the memory while allowing the tables to be loaded into memory again when another query is issued against them at a later point. This is mostly helpful in the multi tenant world where dev-ops team can bring down the memory without running a save or shutting down the tenant service.

IBPL command that will allow admins to release the memory consumed by measure-group and graph tables. There is not expected to be changed in functionality with the exception that the first few queries following release-memory will be slower due to the tables having to be reloaded into memory.

Currently, this command should be used when user gets server (Graph Cube Service) alerts about low memory. Presently, the only solution to this problem is to run save or shut down the tenants. With this new IBPL command, user can reduce memory pressure in a more "tenant friendly" manner.

where

- Release memory for a single measure-group's table for CWV

```
RELEASE MEMORY INCLUDE MODEL [Account Plan] WHERE
{[Version].[Version Name].[CurrentWorkingView] };
```

o9 Internal use Only 31

- Release memory for a single graphs table for all versions (CWV and tuple-merged versions/scenarios)

```
RELEASE MEMORY INCLUDE GRAPH [CannibalizationFlow];
```

- Release memory for a subset of measure-groups and graphs (either a subset of versions as shown here or all versions if invoked without where clause)

```
RELEASE MEMORY INCLUDE MODELS [Account Plan] INCLUDE GRAPHS
[CannibalizationFlow] WHERE { [Version].[Version Name].
[CurrentWorkingView] };
```

- Release memory for all valid measure-groups (CWV and all tuple-merged scenarios will be included)

```
RELEASE MEMORY INCLUDE MODELS ALL;
```

- Release memory for all valid measure-groups and graphs - i.e. global release.
- ```
RELEASE MEMORY;
```

## Set of Characters Allowed in the GraphCube

- API-Names for Measure-Groups, Graphs and Dimensions must only contain [A-Z,a-z,0-9,\_,-]
- Excel sheets do not allow the following characters in name: / \ ? \* [ ]

## Round Function

With the round function in IBPL, a user can specify the decimals for which the value needs to be rounded off. This increases the accuracy in aggregations and conversions.

### Functionality Description

The various combinations of Round() function are:

1. Round(<value>)
 

For example: Round(13.5256) gives an output 14
2. Round(<value>, <precision>)
 

For example: Round(13.5256 , 1) gives an output 13.5
3. Round(<value>, <midpoint rounding>)
 

For example:

  - Round(13.5, ToEven) gives an output 14

where

- Round(13.5, AwayFromZero) gives an output 14
4. Round(<value>, <precision>, <midpoint rounding>)
- For example:
- Round(13.225, 2, ToEven) gives an output 13.22
  - Round(13.225, 2, AwayFromZero) gives an output 13.23



Parameters in round function are not case sensitive.

o9 Internal use Only 32  
o9 Internal use Only 33

# Operators

## Unary Arithmetic Operators

The following unary 0061 arithmetic operators are supported in IBPL. They accept a single scalar value expression as input and return a scalar value.

Unary Plus Example

+5 => 5

Unary Minus Example

-(5) => -5

## Binary Arithmetic Operators

The binary arithmetic operators supported in IBPL include:

addition (+), subtraction (-), multiplication (\*), division (/), power (^).

They each accept two numeric values as input and return a numeric value. (Please note the use of the Coalesce() function for handling null values correctly.)

## Addition

The Addition Operator is the plus sign (+). Here are some Examples,

**1. Example:**

5 + 3

result = 8

**2. Example:**

Measure.[Segment Forecast Gross Revenue] + 1000000

result = one million added to Segment Forecast Gross Revenue

**3. Example:**

Measure.[Gross Sales] + Measure.[Trade Spend]

result = sum of Gross Sales and Trade Spend

**Note**

where

When adding measures, if a measure has a null value, then the return value of the Addition Operator will be null. To avoid this, you should use the Coalesce() function.

o9 Internal use Only 34

## Subtraction

The Subtraction Operator is the minus sign (-). Here are some Examples:

**1. Example:**

5 - 3  
result = 2

**2. Example:**

Measure.[Segment Forecast Gross Revenue] - 1000000  
result = one million subtracted from Segment Forecast Gross Revenue

**3. Example:**

Measure.[Gross Sales] - Measure.[Trade Spend]  
result = difference between Gross Sales and Trade Spend

**Note**

When subtracting measures, if a measure has a null value, then the return value of the Subtraction Operator will be null. To avoid this, you should use the Coalesce() function.

## Multiplication

The Multiplication Operator is the asterisk (\*). Here are some Examples,

**1. Example:**

5 \* 3  
result = 15

**2. Example:**

Measure.[Segment Forecast Gross Revenue] \* 100  
result = one hundred times Segment Forecast Gross Revenue

**3. Example:**

Measure.[Gross Sales] \* Measure.[Trade Spend]  
result = product of Gross Sales and Trade Spend

**Note**

when multiplying measures, if a measure has a null value, then the return value of the Multiplication Operator will be null.

where

## Division

The Division Operator is the slash (/). Here are some Examples,

**1. Example:**

5 / 3  
result = 1.6666666667

**2. Example:**

Measure.[Segment Forecast Gross Revenue] / 100  
result = Segment Forecast Gross Revenue divided by one hundred

**3. Example:**

Measure.[Gross Sales] / Measure.[Trade Spend]  
result = ratio of Gross Sales to Trade Spend

**Note**

when dividing measures, if a measure has a null value, then the return value of the Division Operator will be null. Also, if the divisor is equal to zero, this will cause an error. To avoid this, you should use the if/then/else function to check whether the divisor is non-zero.

## Power

The Power Operator is the caret symbol (^). It raises the first input to the power of the second input and returns the result as a number. Here are some Examples,

**1. Example:**

2 ^ 2  
result = 4

**2. Example:**

4 ^ 0.5  
result = 2

**3. Example:**

Measure.[Segment Forecast Gross Revenue] ^ 2  
result = Segment Forecast Gross Revenue squared

**4. Example:**

Measure.[Gross Sales] ^ Measure.[Trade Sales]

where

result = Gross Sales raised to the power of Trade Sales

#### Note

when using measures with the Power Operator, if a measure has a null value, then the return value will be null.

## Binary String Operators

### Concatenation

The Concatenation Operator is the plus sign (+). It is used to concatenate two strings. o9 Internal use

Only 36

Example: "First" + "Last"

In this Example, the two strings are combined and the return value is the single string "FirstLast".

## Binary Comparison Operators

The binary comparison operators supported in IBPL include Equal (==), NotEqual (!=), LessThan (<), LessThanOrEqual (<=), GreaterThan (>), and GreaterThanOrEqual (>=). They each accept two value expressions as input and return a logical value. The inputs can be of type date, number or string.

Generally, the two inputs are of the same type, but that is not required. If one input is a string and the other is a date or a number, then the date/number is converted to a string before the comparison is made. In the case of a date, the conversion to string will look like "1/27/2013 12:00:00 AM". In the case of a number, the conversion to string may use scientific notation if the value is very large (e.g., 1,234,567,890,000,000 will become "1.23456789E+15").

However, if one input is a date and the other is a number, then you will get an error.

### Equal

The Equal Operator is two equal signs (==). It accepts two inputs and returns true if they are the same, and false otherwise. Here are some Examples,

#### 1. Example:

```
1==1
result = true
```

#### 2. Example:

```
1==2
result = false
```

#### 3. Example:

```
"Test"=="Test"
result = true
```

#### 4. Example:

```
"Test1"=="Test2"
result = false
```

where

### 5. Example:

```
1.0=="1"
result = true (because the number is converted to a string)
```

### 6. Example:

```
1.0=="1.0"
result = false (because conversion of number to string removes the extraneous zero and decimal point)
```

### 7. Example:

```
ToDateTime("2013-01-01 00:00:00.000")=="1/1/2013 12:00:00 AM"
result = true (because the date time is converted to a string with the same format as the string) o9
```

Internal use Only 37

### 8. Example:

```
ToDateTime("2013-01-01 00:00:00.000")==1
result = error
```

## NotEqual

The NotEqual Operator is an exclamation mark and an equal sign (!=). It accepts two inputs and returns false if they are the same, and true otherwise. The combination of greater than and less than signs (<>) can also be used as the NotEqual operator.

### 1. Example:

```
1!=1
result = false
```

### 2. Example:

```
1!=2
result = true
```

### 3. Example:

```
"Test1"!="Test"
result = true
```

### 4.Example:

```
"Test"!="Test2"
result = true
```

### 5. Example:

```
1.0!="1"
result = false (because the number is converted to a string)
```

### 6. Example:

```
1.0!="1.0"
result = true (because conversion of number to string removes the extraneous zero and decimal point)
```

### 7. Example:

where

```
DateTime("2013-01-01 00:00:00.000")!="1/1/2013 12:00:00 AM"
result = false (because the date time is converted to a string with the same format as the
string)
```

**8. Example:**

```
DateTime("2013-01-01 00:00:00.000")!=1
result = error
```

## LessThan

The LessThan Operator is a less than symbol (<). It accepts two inputs and returns true if the first is strictly less than the second, and false otherwise.

o9 Internal use Only 38

### 1. Example:

```
1<1
result = false
```

### 2. Example:

```
1<2
result = true
```

### 3. Example:

```
2<1
result = false
```

### 4. Example:

```
"Test"<"Test"
result = false
```

### 5. Example:

```
"Test1"<"Test2"
result = true
```

### 6. Example:

```
"Test2"<"Test1"
result = false
```

### 7. Example:

```
1.0<"1"
result = false (because the number is converted to the string "1" which is not less than "1")
```

### 8. Example:

```
11<"2"
result = true (because the number is converted to the string "11" which is less than "2")
```

### 9. Example:

```
2<"1"
result = false (because the number is converted to the string "2" which is less than "1")
```

where



## 10. Example:

```
ToDateTime("2013-01-01 00:00:00.000") < "1/1/2013 12:00:00 AM"
result = false (because the date is converted to the string "1/1/2013 12:00:00 AM" which is not
less than "1/1/2013 12:00:00 AM")
```

## 11. Example:

```
ToDateTime("2013-01-01 00:00:00.000") < "1"
result = false (because the date is converted to the string "1/1/2013 12:00:00 AM" which is greater
than "1")
```

## 12. Example:

```
ToDateTime("2013-01-01 00:00:00.000") < "11"
result = true (because the date is converted to the string "1/1/2013 12:00:00 AM" which is less than
"11")
```

## 13. Example:

o9 Internal use Only 39

```
ToDateTime("2013-01-01 00:00:00.000") < 1
result = error
```

# LessThanOrEqual

The LessThanOrEqual Operator is a less than symbol and an equal sign (<=). It accepts two inputs and returns true if the first is less than or equal to the second, and false otherwise.

## 1. Example:

```
1 <= 1
result = true
```

## 2. Example:

```
1 <= 2
result = true
```

## 3. Example:

```
2 <= 1
result = false
```

## 4. Example:

```
"Test" <= "Test"
result = true
```

## 5. Example:

```
"Test1" <= "Test2"
result = true
```

## 6. Example:

```
"Test2" <= "Test1"
result = false
```

## 7. Example:

where

```
1.0<="1"
```

result = true (because the number is converted to the string "1" which is equal to

"1") **8. Example:**

```
11<="2"
```

result = true (because the number is converted to the string "11" which is less than "2")

**9. Example:**

```
2<="1"
```

result = false (because the number is converted to the string "2" which is greater than "1")

**10. Example:**

```
DateTime("2013-01-01 00:00:00.000")<="1/1/2013 12:00:00 AM"
```

result = true (because the number is converted to the string "1/1/2013 12:00:00 AM" which is equal to "1/1/2013 12:00:00 AM")

**11. Example:**

```
DateTime("2013-01-01 00:00:00.000")<="1"
```

o9 Internal use Only 40

result = false (because the date is converted to the string "1/1/2013 12:00:00 AM" which is greater than "1")

**12. Example:**

```
DateTime("2013-01-01 00:00:00.000")<="11"
```

result = true (because the date is converted to the string "1/1/2013 12:00:00 AM" which is less than "11")

## GreatherThan

The GreaterThan Operator is a greater than symbol (>). It accepts two inputs and returns true if the first is strictly greater than the second, and false otherwise.

**1. Example:**

```
1>1
```

result = false

**2. Example:**

```
1>2
```

result = false

**3. Example:**

```
2>1
```

result = true

**4. Example:**

```
"Test">"Test"
```

result = false

where

### 5. Example:

```
"Test1">"Test2"
result = false
```

### 6. Example:

```
"Test2">"Test1"
result = true
```

### 7. Example:

```
1.0>"1"
result = false (because the number is converted to the string "1" which is not greater than "1")
```

### 8. Example:

```
2>"11"
result = true (because the number is converted to the string "2" which is greater than "11")
```

### 9. Example:

```
1>"2"
result = false (because the number is converted to the string "1" which is greater than "2")
```

use Only 41

### 10. Example:

```
DateTime("2013-01-01 00:00:00.000")>"1/1/2013 12:00:00 AM"
result = false (because the date is converted to the string "1/1/2013 12:00:00 AM" which is not greater than "1/1/2013 12:00:00 AM")
```

### 11. Example:

```
DateTime("2013-01-01 00:00:00.000")>"11"
result = false (because the date is converted to the string "1/1/2013 12:00:00 AM" which is less than "11")
```

### 12. Example:

```
DateTime("2013-01-01 00:00:00.000")>"1"
result = true (because the date is converted to the string "1/1/2013 12:00:00 AM" which is greater than "1")
```

### 13. Example:

```
DateTime("2013-01-01 00:00:00.000")>1
result = error
```

## GreaterThanOrEqual

The GreaterThanOrEqual Operator is a greater than symbol and an equal sign (>=). It accepts two inputs and returns true if the first is greater than or equal to the second, and false otherwise.

### 1. Example:

```
1>=1
result = true
```

where

## 2. Example:

```
1>=2
result = false
```

## 3. Example:

```
2>=1
result = true
```

## 4. Example:

```
"Test">="Test"
result = true
```

## 5. Example:

```
"Test1">="Test2"
result = false
```

## 6. Example:

```
"Test2">="Test1"
result = true
```

## 7. Example:

```
1.0>="1"
```

o9 Internal use Only 42

```
result = true (because the number is converted to the string "1" which is equal to "1")
```

## 8. Example:

```
2>="11"
result = true (because the number is converted to the string "2" which is greater than "11")
```

## 9. Example:

```
1>="2"
result = false (because the number is converted to the string "1" which is greater than "2")
```

## 10. Example:

```
DateTime("2013-01-01 00:00:00.000")>="1/1/2013 12:00:00 AM"
result = true (because the date is converted to the string "1/1/2013 12:00:00 AM" which is not greater than "1/1/2013 12:00:00 AM")
```

## 11. Example:

```
DateTime("2013-01-01 00:00:00.000")>="11"
result = false (because the date is converted to the string "1/1/2013 12:00:00 AM" which is less than "11")
```

## 12. Example:

```
DateTime("2013-01-01 00:00:00.000")>="1"
result = true (because the date is converted to the string "1/1/2013 12:00:00 AM" which is greater than "1")
```

## 13. Example:

```
DateTime("2013-01-01 00:00:00.000")>=1
result = error
```

where

# Binary Logical Operators

The binary logical operators supported in IBPL include AND (&&) and OR (||). They accept two logical value expressions as input and return a logical value. For examples && and || operators can be used as below. This statement will compute Base Sales-Out for the time buckets between start date and end of life dates or for new products.

```
scope:(&AllProducts * &AllAccounts * &PlanningWeeks * &CwvAndScenarios); Measure.[Base
Sales-Out] = if(Time.#.Key >= Measure.[Sales-Out Start Date] && Time.#.Key <= Measure.[EOL Date]
|| Product.#.[IsNewProduct]) then Measure.[Sales-Out Rate] else 0.0;
```

## Logical And

The Logical And Operator is two ampersands (&&). It returns a true value if both of the input expressions evaluate to true, and returns false if either expression evaluates to false. Here are some Examples,

### 1. Example:

```
(1>0) && (1<2)
```

```
result = true
```

### 2. Example:

```
(1<0) && (1<2)
```

```
result = false
```

o9 Internal use Only 43

### 3. Example:

```
(1<0) && (1>2)
```

```
result = false
```

If either expression evaluates to null, then the Logical And Operator returns null.

## Logical Or

The Logical Or Operator is two vertical bars (||). It returns a true value if either of the input expressions evaluate to true, and returns false only if both expressions evaluate to false.

### 1. Example:

```
(1>0) || (1<2)
```

```
result = true
```

### 2. Example:

```
(1<0) || (1<2)
```

```
result = true
```

### 3. Example:

```
(1<0) || (1>2)
```

```
result = false
```

If one of the expressions evaluates to null, then the Logical Or Operator returns the value of the

where

other non-null expression. If both expressions evaluate to null, then the Logical Or Operator returns null.

**In**

**Not**

## AttributeMember Operators

**At**

The At Operator uses the at symbol (@). It is used in a measure formula to refer to the value of a measure for a different member of one of the dimensions. The return value is an AttributeMember.

**Example:**

```
Measure.[Segment Forecast Detail Gross Revenue LY] = Measure.[Segment Forecast Detail Gross Revenue]@([Time].#.LeadOffset(-12))
```

o9 Internal use Only 44

```
Measure.[Segment Forecast Detail Gross Revenue] = Measure.[Segment Forecast Detail Gross Revenue Actual]@(Time.Week.filter(#.Key == &CurrentWeek.element(0).Key).element(0)) * Measure.[Growth %];
```

This Example finds the value of “Segment Forecast Detail Gross Revenue” for the member twelve months earlier, and uses that as the value for “Segment Forecast Detail Gross Revenue LY”.

where

where

# **Functions**

## **String Functions**

where



## Upper

The Upper() function converts one string to another string where all the lowercase characters are replaced with uppercase characters. Non-alphabetic characters remain unchanged. The function takes one string parameter. The return value is of string type.

Example: Upper("Test 123")

The result is "TEST 123".

## Lower todatetime

The Lower() function converts one string to another string where all the uppercase characters are replaced with lowercase characters. Non-alphabetic characters remain unchanged.

Example: Lower("Test 123")

The result is "test 123".

## ToDateTime

The ToDateTime() function converts a string representing a date into a date value. The return value is a date. The function takes one parameter which is a string. Here are some Examples,

Examples:

- ToDateTime("1/27/2013")
- ToDateTime("01/27/2013")
- ToDateTime("1/27/2013 12:00 AM")
- ToDateTime("1/27/2013 12:00:00 AM")
- ToDateTime("2013-01-27")
- ToDateTime("2013-1-27")
- ToDateTime("2013-01-27 00:00")
- ToDateTime("2013-01-27 00:00:00")
- ToDateTime("2013-01-27 00:00:00.000")

All of these Examples evaluate to the same date value.

## Current User (Logged in User)

This ibpl will return the logged in user.

```
select CurrentUser();
```

o9 Internal use Only 46

| Key | Name | DisplayName Email\$InAc | Image Is | User Id | User Name |
|-----|------|-------------------------|----------|---------|-----------|
|     |      | tive                    | MPo      |         | e         |

where

|   |                             |                          |                                    |            |         |
|---|-----------------------------|--------------------------|------------------------------------|------------|---------|
|   |                             |                          | wer<br>User                        |            |         |
| 6 | "admin@relea<br>se scs.com" | "admin@releasc<br>s.com" | FALSE "asset/Unknown.<br>png"<br>E | 2099<br>25 | "admin" |

o9 Internal use Only 47

## Numeric Functions

### Abs

The Abs() function takes a numeric value and returns the corresponding absolute value. In other words, if the number is less than zero, it returns the number multiplied by -1, otherwise it returns the number itself. The return value is a number. The function takes a single parameter which is a number. Here are few examples:

Examples

- abs(1)  
result = 1
- abs(-1)  
result = 1
- abs(Measure.[Segment Forecast Gross Revenue])  
result = absolute value of Segment Forecast Gross Revenue (where it is non-null)

### Coalescenull

The Coalesce() function is used to replace a null value with another supplied value for a measure. The return value is a number. The function takes two numeric values. The first value is to check for the null, and the second is the number to replace the first with, if it is null.

In the below example, if Segment Target Gross Profit is null, then the Coalesce() function returns zero, else it returns the value of Segment Target Gross Profit.

coalesce(Measure.[Segment Target Gross Profit], 0)

**Note:** It is extremely important to always use coalesce in all arithmetic rules involving addition/subtractions.

**Example:**

Measure.[Net Sales] = Measure.[Gross Sales] – Measure[Trade Spend]

This is not a correct rule, since if there is no trade spend for a row (i.e., the value is null), then the Net Sales will be null, even when there is non-zero Gross Sales. This is not the expected business result. The correct rule is:

Measure.[Net Sales] = coalesce(measure.[Gross Sales],0) – coalesce(Measure[Trade Spend],0)

### ToString

where

The ToString() function converts a numeric value to a string. The return value is a string. The function takes one parameter which is a number.

**Example:**

```
ToString(1234567890000000)
```

In this Example, the ToString() function returns the value "1.23456789E+15".

o9 Internal use Only 48

## Round

The Round() function takes a numeric value and returns the same rounded value, if the decimal value is lesser than 0.5. Otherwise, the value will be rounded to the next integer value when decimal value is greater than or equal to 0.5.

Here are some Examples,

**1. Example:**

```
round(3.3)
result = 3
```

**2. Example:**

```
round(3.5)
result = 4
```

## Ceiling

The Ceiling() function takes a numeric value and returns the next integer rounded value, when decimal value exists.

Here are some Examples,

**1. Example:**

```
ceiling(3.3)
result = 4
```

**2. Example:**

```
ceiling(3.5)
result = 4
```

## Floor

The Floor() function takes a numeric value and returns the same integer rounded value, even though decimal value exists.

Here are some Examples,

**1. Example:**

```
floor(3.3)
```

where

```
result = 3
```

## 2. Example:

```
floor(3.5)
result = 3
```

## Safe Divide

Consider the ibpl below

```
Select SafeDivide(5.0,2.0,10.0); // let's say a=5.0, b=2.0, c=10.0
```

o9 Internal use Only 49

```
Select SafeDivide(5.0, 0.0, 10.0); // let's say a=5.0, b=0.0, c=10.0
```

Output of the Safedivide functions will be -

a/b when  $b \neq 0$ .

And c when  $b = 0$ .

So, output of first ibpl will be  $5.0/2.0 = 2.5$

And that of second ibpl will be 10.0

Safedivide function can also accept measure expressions as inputs as shown in the ibpl below.

```
Select([Time].[Fiscal Month].[M12-2014] *[Product].[Product Name].[Cosmo Cola
Regular 24oz] * SalesDomain.[Sales Domain Name].[BJ's Wholesale Club]*
[Version].[Version Name].[CurrentWorkingView] *{Measure.[Cases],
Measure.[Sales],
SafeDivide(Measure.[Cases],Coalesce(Measure.[Sales],0),200.0) as
Transient.[First]
});
```

Output of above ibpl will be equal to  $(\text{Measure}[\text{Cases}]/\text{Coalesce}(\text{Measure}[\text{Sales}],0))$  when  $\text{Coalesce}(\text{Measure}[\text{Sales}],0) \neq 0$  and equal to 200.0 when  $\text{Coalesce}(\text{Measure}[\text{Sales}],0) = 0$

## Float

## Integer

## Sum

## Avg

## AvgWithNulls

where

## Count

## Min

o9 Internal use Only 50

## Max

# filterDateTime Functionscurrent timemem

## ToString

The ToString() function converts a date value to a string. The return value is a string. The function takes one parameter which is a date.

### Example:

```
ToString(ToDateTime("2013-01-27"))
```

In this Example, the ToString() function returns the value "1/27/2013 12:00:00 AM".

## DateADD Function

DATEADD(<date>,<number\_of\_intervals>,<interval>)

Returns a date , shifted either forward or backward in time by the specified number of intervals.

<date>: an expression that evaluates to a date

<number\_of\_intervals>: An integer that specifies the number of intervals to add to or subtract from the dates.

<interval>: The interval by which to shift the dates. The value for interval can be one of the following:  
year,month,week,day

0. example: Assign the ship week by subtracting lead time

```
scope(...);
```

```
Measure.[Ship Week] = DATEADD(Time.#.Key , -1 * Measure.[LeadTime]);
```

```
end scope;
```

0.1 Similar example:

```
scope:[Customer].[Channel Name] * [Date].[Reporting Month] * [Product].[Brand Name] *
[Version].[Version Name].[CurrentWorkingView];
```

```
Measure.[Dummy Datetime] = DATEADD(Date.#.Key,coalesce(Measure.[Dummy
Integer],0),"week"); end scope;
```

where

Other examples:

```
1. select NOW() // select current time
"08/13/2015 16:46:41"
```

```
2.select DATEADD(NOW(),1,"day"); // Add one day to current time
"Result"
"8/14/2015 5:31:53 PM"
```

```
3. select DATEADD(NOW(),1,"week"); //Add one week to current time
"Result"
"8/20/2015 5:32:14 PM"
```

```
4.select DATEADD(NOW(),1,"month"); // Add one month to current time
"Result"
"9/13/2015 5:32:51 PM"
```

```
5. select DATEADD(NOW(),1,"year"); // Add one year to current time
"Result"
"8/13/2016 5:33:11 PM"
```

o9 Internal use Only 51

```
6.select DATEADD(NOW(), INTEGER(RANDOM(1,10)), "week"); // Add a random number of weeks
(between 1 and 10, to the current time)
"Result"
"9/17/2015 5:33:35 PM"
```

```
7. select DATEADD(Date.[Reporting Week].element(0).Key, 1, "week"); // Same as Date.[Reporting
Week].element(0).LeadOffset(1)
```

```
8. select DATEADD(Date.[Reporting Week].element(0).Key, -11, "week"); // Same as Date.[Reporting
Week].element(0).LeadOffset(-11)
```

## DateDiff Function

Planners use date measures to plan dates such as Event Start Date, Event End Date etc. mPower now supports DateDiff function to calculate number of time buckets between two dates. Planner can know number of Days, Weeks, Months or Year between two date measures as shown below. A string parameter "DAY" or "WEEK" is required to know the granularity of the time bucket. DATEDIFF function can also be used inside an update statement or active rule.

```
1. select DATEDIFF(ToDateTime("2012-01-01 00:00:00.000"), ToDateTime("2012-01-01
00:00:00.000"), "DAY");
```

Ans: 0

```
2. select DATEDIFF(ToDateTime("2012-01-01 00:00:00.000"), ToDateTime("2012-01-02
00:00:00.000"), "DAY");
```

Ans: 1

```
3. select DATEDIFF(ToDateTime("2012-01-02 00:00:00.000"), ToDateTime("2012-01-01
00:00:00.000"), "DAY");
```

Ans: -1

```
4. select DATEDIFF(ToDateTime("2012-01-01 00:00:00.000"), ToDateTime("2012-02-01
00:00:00.000"), "week");
```

Ans: 4

```
5. select DATEDIFF(ToDateTime("2012-01-01 00:00:00.000"), ToDateTime("2012-02-01
```

where

```
00:00:00.000"), "MONTH");
```

Ans: 1

```
6. select DATEDIFF(ToDateTime("2012-01-01 00:00:00.000"), ToDateTime("2013-02-01
00:00:00.000"), "YEAR");
```

Ans: 1

```
7. scope:(BusinessUnit.[Vertical] * [OrganizationStructure].[Role] * Practice.[Sub Practice] *
[Suite].[Suite] * [Location].[City] * [Version].[Version Name].Filter(#.Key <= 0));
```

```
Measure.[True Indents] = DATEDIFF(Measure.[Planned Start Date], Measure.[Start Date], "day");
end scope;
```

o9 Internal use Only 52

## Logical Functions

### If-Then-Else

The if-then-else function is used to specify conditional logic in calculating the value for a measure.

#### Example:

```
if(Measure.[Segment Target Net Revenue] > 0)
then Measure.[Segment Target Gross Profit] / Measure.[Segment Target Net Revenue]
else 0
```

The “else” is optional and if missing the value will be NULL in case the condition in the “if” statement is not true.

### IsNull

IsNull() function verifies if the measure is null or not. If the value of measure is null then the function returns 1. so if user needs to execute a rule only for those intersection where Assortment measure is not null then a scope can be written as

```
scope: (Product.[Product] *);
```

```
Measure.Revenue = if(~IsNull(Measure.Assortment)) then Measure.Units *
```

```
Measure.ASP$; end scope;
```

## AttributeMember Functions

AttributeMember functions act on a single member.

### Ancestor

where

The Ancestor() function is used to traverse the hierarchy from a member at one level to a parent at a higher level. The return value is an AttributeMember. The function takes two parameters: the first is the hierarchy to use, and the second is how many levels up the hierarchy to traverse.

**Example:**

```
[Time].[Fiscal Month].[M01-2013].ancestor([Time].[Fiscal YQM], 2)
```

In this Example, there is a hierarchy called "Fiscal YQM" that has levels "Fiscal Year", "Fiscal Quarter" and "Fiscal Month". The Ancestor() function traverses from "Fiscal Month" up two levels to "Fiscal Year" and returns the "Fiscal Year" member "FY2013".

o9 Internal use Only 53

## Children

The Children() function is used to traverse the hierarchy from a member at one level to the children at the next lower level. The return value is an AttributeMemberSet. The function takes one parameter which is the hierarchy to use.

**Example:**

```
[Time].[Fiscal Year].[FY2013].children([Time].[Fiscal YQM])
```

In this Example, the Children() function finds the level that is one below "Fiscal Year" in the "Fiscal YQM" hierarchy (which is "Fiscal Quarter") and returns the descendants of "FY2013" at that level, which is "Q1-2013", "Q2-2013" and "Q3-2013".

## LeadOffset

The LeadOffset() function is used to traverse from one member to another member at the same level. The return value is an AttributeMember. The function takes one parameter, which is an integer that can be either positive or negative.

**Example:**

```
[Time].[Fiscal Month].[M01-2013].leadOffset(12)
```

In this Example, the LeadOffset() function will return the "Fiscal Month" that is twelve positions after "M01-2013", which is "M01-2014". Another Example:

```
[Time].[Fiscal Month].[M01-2013].leadOffset(-12)
```

In this Example, the result is twelve positions before "M01-2013", which is "M01-2012".

## Between

The Between() function is used to find all of the members with keys greater than or equal to the first member and less than or equal to the second member. The return value is an AttributeMemberSet. The function takes two attribute members as inputs.

where



**Example:**

```
between([Time].[Fiscal Month].[M12-2013], [Time].[Fiscal Month].[M02-2014])
```

In this Example, the Between() function returns all "Fiscal Month" members with Key greater than or equal to "2013-12-29 00:00:00.000" and less than or equal to "2013-02-23 00:00:00.000", which is "M12-2013", "M01-2014" and "M02-2014".

Note: the above Example is equivalent to:

```
Time.[Fiscal Month].filter(#.Key>=[Time].[Fiscal Month].[M12-2013].Key && #.Key<=[Time].[Fiscal Month].[M02-2014].Key)
```

**Here are some more Examples,****1. Example:**

```
between(Time.[Fiscal Month].[M12-2013], Time.[Fiscal Month].[M12-2013])
result = M12-2013
```

**2. Example:**

o9 Internal use Only 54

```
between(Time.[Fiscal Month].[M02-2014], Time.[Fiscal Month].[M12-2013])
result = M12-2013, M01-2014, M02-2014
```

**3. Example:**

```
between(Product.[Brand].element(1), Product.[Brand].element(3))
result = the 2nd, 3rd and 4th members of Product.Brand, which are Cosmo Flavored, Garden Fresh, Hydro Cola
```

**4. Example:**

```
between(Product.[Brand].filter(#.Name=="Cosmo Flavored").element(0),
Product.[Brand].filter(#.Name=="Hydro Cola").element(0))
result = Cosmo Flavored, Garden Fresh, Hydro Cola
```

## AttributeMemberSet Functions

AttributeMemberSet functions act on a list of members. The functions take the form of:  
&SomeAttributeMemberSet.function(parameter1, parameter2)

### Find

The Find() function is used to find a specific member in a list. The return value is an AttributeMember. The function takes one parameter which is the name of the member to find.

**Example:**

```
[Time].[Fiscal Year].find("FY2013")
```

In this Example, the Find() function finds the "Fiscal Year" member named "FY2013".

Note: the above Example is equivalent to the following:

```
[Time].[Fiscal Year].[FY2013]
```

where

## Filter

The `Filter()` function is used to filter a list of members according to some provided condition(s). The return value is an `AttributeMemberSet`. The function takes one parameter, which is an expression that is evaluated for each member of the list, and returns either true or false. In the expression, `#` stands in for a member in the list.

### 1.Example:

```
[Product].[Product Name].filter(#[Organization]==“Optimal Beverages”)
```

In this Example, the members of the Product Name level in the Product dimension are filtered based on the property Organization. If Organization equals “Optimal Beverages”, then the Product Name member is included in the result, otherwise it is excluded.

### 2. Example:

```
[Product].[Product Name].filter(#[Name]==“Cosmo Cola Diet 12oz”)
```

### o9 Internal use Only 55

The `Filter()` function can use any property that is defined for members of the level in question (and as with dimension and levels, the square brackets are optional if the property does not have spaces in it).

### 3. Example:

```
[Version].[Version Name].filter(#[Is Latest]==true)
```

The “logical and” (`&&`) and “logical or” (`||`) operators can be used to specify multiple conditions.

### 4. Example:

```
[Product].[Product Name].filter(#[Organization]==“Optimal Beverages” && #[Type]==“Retail”)
```

In this Example, the resulting `AttributeMemberSet` contains all of the members of the Product Name level for which Organization equals “Optimal Beverages” and Type equals “Retail”. If only one of these conditions holds true, then the member is not included in the result.

### 5. Example:

```
Product.[Brand].filter(Upper(#[Name])==“COSMO COLA”)
```

### 6. Example:

```
Time.[Fiscal Month].filter(#[Key]==ToDateTime(“2012-01-01 00:00:00.000”))
```

### 7. Example:

```
Time.[Fiscal Month].filter(#[Key]!=ToDateTime(“2012-01-01 00:00:00.000”))
```

### 8. Example:

```
Time.[Fiscal Month].filter(#[Key]>ToDateTime(“2012-01-01 00:00:00.000”))
```

### 9. Example:

```
Time.[Fiscal Month].filter(#[Key]<ToDateTime(“2012-01-01 00:00:00.000”))
```

### 10. Example:

```
Time.[Fiscal Month].filter(#[Name]==“P1-2012” || #[Name]==“P2-2012”)
```

### 11. Example:

```
Time.[Fiscal Month].filter(#[Key]==[Current Time].[Fiscal Month].element(0).Key).element(0).leadOffset(1)
```

where

**12. Example:**

```
Time.[Fiscal Month].filter(#.Key==[Current Time].[Fiscal
Month].element(0).Key).element(0).leadOffset(2).LeadOffset(-2)
```

**13. Example:**

```
Time.[Fiscal Month].filter(#.Key==[Current Time].[Fiscal Month].element(0).Key).element(0).leadOffset(-1)
```

**14. Example:**

```
Time.[Fiscal Month].filter(#.Key==[Current Time].[Fiscal
Month].element(0).Key).element(0).leadOffset(-100)
```

**15. Example:**

```
Time.[Fiscal Month].filter(#.Key==[Current Time].[Fiscal
Month].element(0).Key).element(0).leadOffset(-1).leadOffset(1)
```

**16. Example:**

```
Product.[Brand].filter(#.Name==Product.[Brand].find("Cosmo
```

```
Cola").leadOffset(-1).Name) 17. Example:
```

o9 Internal use Only 56

```
Time.[Fiscal Month].filter(#.Key==[Current Time].[Fiscal Month].element(0).Key).element(0).leadOffset(1+1)
```

**18. Example:**

```
Time.[Fiscal Month].filter(#.Key==[Current Time].[Fiscal
Month].element(0).Key).element(0).leadOffset(Product.[Brand].Element(0).Key - 150)
```

**19. Example:**

```
[Version].[Version Name].filter(#.Key<(-1))
```

**20. Example:**

```
Between(Time.[Fiscal Month].filter(#.Key==ToDateTime("2012-01-01 00:00:00.000")).element(0),
Time.[Fiscal Month].filter(#.Key==ToDateTime("2010-04-18 00:00:00.000")).element(0))
```

## Filter Predicate

Filter Predicate option filters the members of Level Attribute based on the given expression.

Some of the sample keywords are

- Startswith
- Endswith
- Contains
- Like
- containsregexp etc.

**1. Example:** Returns members which have names starts with 'AB'

```
#.Name startswith [AB]
```

**2. Example:** Returns the members which are 'Marketing' type

```
#[Type]=="Marketing"
```

**3. Example:** Returns the members which contains name 'Estimates'

```
#.Name contains "Estimates"
```

**4. Example:** Returns the members whose category is 'CurrentWorkingView' or 'What-if'

```
#.Category == "CurrentWorkingView" || #.Category == "What-if"
```

**5. Example:** Returns members whose names doesn't start with 'AB'

```
~(#.Name startswith [AB])
```

where

Filter Predicate option filters the members of String Measures based on the given expression.

Some of the sample keywords are

- Startswith
- Endswith
- Contains
- Like
- containsregexp etc.

1. **Example:** Returns members which have names starts with 'AB'  
Measure.[Y] startswith "AB"
2. **Example:** Returns members which have names ends with 'AB'  
Measure.[Y] endswith "AB"
3. **Example:** Returns the members which contains name 'AB'  
Measure.[Y] contains "AB"
4. **Example:** Returns members whose names doesn't start with 'AB'  
~(Measure.[Y] startswith "AB")

## Filter\_By\_Updates

Filter By Updates function returns only updated cells in the same transaction. if no cell has been updated, then none of the rows are returned. .

```
SELECT ([Order].[Order Name].Filter(#.Name in { "Dell101" })) * [SalesAccount].[Account] *
[Product].[Product] * {Measure.[Demand Layer] , Measure.[Demand Node Demand]}) on row, ([Time].[Fiscal
Week].Filter(#.Name in { "W01-2015" , "W02-2015" , "W03-2015" }))) on column include Icid 1033,
cellproperties {LOCK, BG_COLOR, FG_COLOR} FILTER_BY_UPDATES where {[Version].[Version
Name].Filter(#.Name in { "CurrentWorkingView" })
};
```

where

## Element

The `Element()` function is used to select a single member out of a list of members. The return value is an `AttributeMember`. In the `Filter()` function, even if the result is a single member, syntactically it is still an `AttributeMemberSet`. To convert it from an `AttributeMemberSet` to an `AttributeMember`, you can use the `Element()` function.

### Example:

```
[Product].[Product Name].filter(#[Name]=="Cosmo Cola Diet 12oz").element(0)
```

Lists are indexed starting from zero, so the first element of the list is 0, the second is 1, the third is 2, and so on.

Note: the above Example is equivalent to the following:

```
[Product].[Product Name].[Cosmo Cola Diet 12oz]
```

## Count

The `Count()` function counts the number of members in an `AttributeMemberSet`. The return value is an integer. The function does not take any parameters.

### Example:

```
[Time].[Fiscal Year].count
```

o9 Internal use Only 58

In this Example, the return value is the number of Fiscal Year members. Another

Example: 

```
[Time].[Fiscal Year].[FY2013].[Fiscal Month].count
```

In this Example, the return value is the number of Fiscal Month members that are descendants of Fiscal Year "FY2013".

## AncestorsAtLevel

The `AncestorsAtLevel()` function is used to traverse up the hierarchy using the name of a higher level. The return value is an `AttributeMemberSet`. The function takes one parameter, which is the name of a level. Note: the level must be in the same dimension as the members of the `AttributeMemberSet`, and higher in the hierarchy.

### Example:

```
[Time].[Fiscal Month].filter(#[Name]=="M01-2013").ancestorsAtLevel([Fiscal Year])
```

In this Example, the `AncestorsAtLevel()` function finds the ancestor members in the Fiscal Year level, which is "FY2013".

Note: the above Example is equivalent to the following:

```
[Time].[Fiscal Month].[M01-2013].[Fiscal Year]
```

## DescendantsAtLevel

The `DescendantsAtLevel()` function is used to traverse down

where

the hierarchy using the name of a lower level. The return value is an AttributeMemberSet. The function takes one parameter, which is the name of a level. Note: the level must be in the same dimension as the members of the AttributeMemberSet, and lower in the hierarchy.

#### Example:

```
[Time].[Fiscal Year].filter(#.Name=="FY2013").descendantsAtLevel([Fiscal Quarter])
```

In this Example, the function finds the descendants of Fiscal Year "FY2013" at the Fiscal Month level, which are "Q1-2013", "Q2-2013", "Q3-2013" and "Q4-2013".

Note: the above Example is equivalent to the following:

```
[Time].[Fiscal Year].[FY2013].[Fiscal Quarter]
```

## Query to Display Members without Descendants

With a query to display the hanging members, a user can fetch all the attributes without any parents. This makes it easier to evaluate the accuracy of the data, and delete the ones which are no longer needed as part of the data management.

### Functionality Description

The following are the different ways to display the hanging members:

1. Select (DimName.DimAttribute.filter(#.HasNoDescendants) on row, () on column); For example Select (Time.Year.filter(#.HasNoDescendants) on row, () on column); returns all the years which has no quarters.

o9 Internal use Only 59

2. Select (DimName.DimAttribute.filter(#.Name in {"MemberName"}).filter(#.HasNoDescendants));

For example Select (Time.Year.filter(#.Name in {"2019","2018"}).filter(#.HasNoDescendants)); returns an output 2018 if 2018 has no child

3. Select (DimName.DimAttribute.filter(#.Key > 10).filter(#.HasNoDescendants)); For example Select (Item.Product.filter(#.Key > 10).filter(#.HasNoDescendants)); checks for all products with key greater than 10 and returns an output. #HasNoDescendants can also be used in NamedSets and scope

1. CustomerWithNoDescendants = Customer.[Customer Name].Filter(#.HasNoDescendants)

Select &CustomerWithNoDescendants; - returns all Customers without any children

2. scope: ([Customer].[Customer Name].[Costco].Filter(#.HasNoDescendants) \* [Product].[Brand Name]);

Measure.[ACP Bill Back Plan] = 1;

end scope;

Updates value of Measure [ACP Bill Back Plan] =1 if Costco has no descendants

- Query restricted to Dimension Attribute i.e, Querying over the dimension for hanging members is not implemented.
- Not an implied property and does not have any effect on the query with



where

- member properties.
- Not Supported for ACLs and RHS of a rule.

## RelatedMembers

The RelatedMembers() function finds related members at either a higher or lower level of the hierarchy. The return value is an AttributeMemberSet. The function takes one parameter, which is the name of a level that is in the same dimension as the members of the AttributeMemberSet. If the level is higher in the hierarchy, then this function is the same as the AncestorsAtLevel() function. If the level is lower in the hierarchy, then this function is the same as the DescendantsAtLevel() function.

### Example:

```
[Time].[Fiscal Month].filter(#[Name]=="M01-2013").relatedMembers([Fiscal Year])
```

In this Example, the level is higher in the hierarchy, so this is equivalent to using the AncestorsAtLevel() function. Another Example:

```
[Time].[Fiscal Year].filter(#[Name]=="FY2013").relatedMembers([Fiscal Quarter])
```

In this Example, the level is lower in the hierarchy, so this is equivalent to using the DescendantsAtLevel() function.

## UnionWith

The UnionWith() function joins two lists of members together into one unique list of members. The return value is an AttributeMemberSet. The function takes one parameter, which is the second AttributeMemberSet to combine with the first. Note: the members of both AttributeMemberSets must share the same dimension and level.

### Example:

o9 Internal use Only 60

```
[Time].[Fiscal Year].filter(#[Name]=="FY2013").unionWith([Time].[Fiscal Year].filter(#[Name]=="FY2014"))
```

In this Example, the first AttributeMemberSet contains "FY2013", and it is unioned with a second AttributeMemberSet that contains "FY2014". The result contains both "FY2013" and "FY2014".

**Note:** the above Example is equivalent to the following:

```
[Time].[Fiscal Year].filter(#[Name]=="FY2013" || #[Name]=="FY2014")
```

If the two AttributeMemberSets have a member in common, the result will only have the member once.

### 1. Example:

```
[Time].[Fiscal Year].filter(#[Name]=="FY2013" || #[Name]=="FY2014").unionWith([Time].[Fiscal Year].filter(#[Name]=="FY2014"))
```

In this Example, the result will contain two members: "FY2013" and "FY2014". In other words, even though both AttributeMemberSets contain "FY2014", the result will contain this member once, not twice.

### 2. Example:

```
Product.[Brand].filter(#[Name]=="Cosmo Cola").unionWith(Product.[Brand].filter(#[Name]=="Cosmo Flavored"))
result = Cosmo Cola, Cosmo Flavored
```

### 3. Example:

where

```
Product.[Brand].filter(#.Name=="Cosmo Cola").unionWith(Product.[Brand].filter(#.Name=="NON EXISTENT"))
result = Cosmo Cola
```

#### 4. Example:

```
Product.[Brand].filter(#.Name=="Cosmo Cola").unionWith(Product.[Brand].filter(#.Name=="Cosmo Cola")) result = Cosmo Cola
```

#### 5. Example:

```
Product.[Brand].filter(#.Name=="Cosmo Cola").unionWith(Product.[Brand].filter(#.Name=="Cosmo Flavored"))).unionWith(Product.[Brand].filter(#.Name=="Garden Fresh"))
result = Cosmo Cola, Cosmo Flavored, Garden Fresh
```

#### 6. Example:

```
Product.[Brand].unionWith(Product.[Brand].filter(#.Name=="Cosmo Cola"))
result = all brands
```

#### 7. Example:

```
Product.[Brand].unionWith(Product.[Brand])
result = all brands
```

## IntersectWith

The `IntersectWith()` functions finds the members in one list that are also contained in another list. The return value is an `AttributeMemberSet`. The function takes one parameter, which is the second `AttributeMemberSet` to intersect with the first.

Note: the members of both `AttributeMemberSets` must share the same dimension and level.

#### 1. Example:

```
[Time].[Fiscal Year].intersectWith([Time].[Fiscal Year].filter(#.Name=="FY2014"))
```

o9 Internal use Only 61

In this Example, the return value is "FY2014" since that is the only member that is common to both `AttributeMemberSets`.

#### 2. Example:

```
Product.[Brand].intersectWith(Product.[Brand].filter(#.Name=="Cosmo Cola"))
result = Cosmo Cola
```

#### 3. Example:

```
Product.[Brand].filter(#.Name=="Cosmo Cola").intersectWith(Product.[Brand].filter(#.Name=="NON EXISTENT"))
result = empty set
```

#### 4. Example:

```
Product.[Brand].filter(#.Name=="Cosmo cola").intersectWith(Product.[Brand].filter(#.Name=="Cosmo Cola"))
result = Cosmo Cola
```

#### 5. Example:

```
Product.[Brand].filter(#.Name=="Cosmo Cola").unionWith(Product.[Brand].filter(#.Name=="Cosmo Flavored")).intersectWith(Product.[Brand].filter(#.Name=="Cosmo Cola"))
```

where



result = Cosmo Cola

**6. Example:**

Product.[Brand].intersectWith(Product.[Brand])

result = all brands

**7. Example:**

**select &CurrentFiscalMonth.element(0).ancestor(Time.[Fiscal  
YQMF],2).relatedMembers([Fiscal Month]).intersectwith(Time.[Fiscal  
Month].Filter(#.Key <= &CurrentFiscalMonth.element(0).Key))**

**8. Example:**

SCSItem.[Brand].Filter(#.Name == "Raw Material").relatedmembers([Item])

**.INTERSECTWITH(SCSItem.[ProductClass].Filter(#.Name == "PACK").relatedmembers([Item]))** o9 Internal

where

# Relationship Functions

## Traverse

The `Traverse()` function is used to traverse a relationship type and generate a set of unique relationships that are connected to the starting member tuple set. In general, traverse uses a member tuple set (or part of a member tuple set) and collects all the relationships that meet the specified criteria along the path of traversal. Currently the traverse command can be invoked only as a component of the `Members()` function. The return value is a set of relationships. The function takes up to five parameters (the first two are required, the other three are optional and any combination of them may be used).

### Syntax:

```
<starting_member_exp>.TRAVERSE(<relationship_type_name>, <version_exp>, <traverse_direction>?,
<distance>?, <relationship_filter_exp>?);
```

`starting_member_exp` - This is an expression that generates a subset of member tuple sets for a relationship type. The expression consists of one or more `AttributeMembers` or `AttributeMemberSets`, separated by asterisks and enclosed in parenthesis.

### 1. Example:

```
(Product.[Brand].filter(#.Name!=Cosmo Cola))
```

### 2. Example:

```
(Product.[Brand].Filter(#.Name!=Cosmo Cola) * SalesDomain.[Sales Region].[North America])
```

### 3. Example:

```
(Product.[Brand].Filter(#.Name!=Cosmo Cola) * [_SchemaPlan].[_Measure].Filter(#.Name in {[Regional Sales], [Gross Sales]}))
```

### 4. Example:

```
(Product.[Brand].[Cosmo Cola] * Measure.[Regional Sales])
```

### 5. Example:

where

(Product.[Brand].Filter(#.Name!=Cosmo Cola)) \* Measure in {[Regional Sales], [Gross Sales]})

Note the use of measures in the member tuple set. Since a measure can be part of a member tuple set, it can also be included in the expression of the *starting\_member\_exp*. The third Example shows how to use a measure as a member using an arbitrary member set expression. The last two Examples show a simpler syntax for the more common use cases of checking for measure names.

relationship\_type\_name - The name of the relationship type.

version\_exp - An IBPL expression for a version/scenario.

traverse\_direction - An optional parameter that specifies the direction for the traversal. Values can be one of UPSTREAM, DOWNSTREAM, or ALL. The default value is ALL. If the traversal direction is UPSTREAM, the *starting\_member\_exp* will be matched against the TO member tuple sets to select the first set of relationships in traversal. If the traversal direction is DOWNSTREAM, the *starting\_member\_exp* will be matched against the FROM member tuple sets. If the choice is ALL, an relationship is selected if either the TO or the FROM member tuple set matches the *starting\_member\_exp*.

distance - An optional parameter that specifies the maximum number of hops for the traversal (assuming that recursion is possible). The value can be either an integer or the string ALL. The default value is 1; it returns only the relationships that match the *starting\_member\_exp* and other

#### o9 Internal use Only 63

conditions. For integers greater than 1, the function collects all relationships that match the filter conditions, and then continues traversing upstream or downstream (or both) recursively until the specified distance is reached. The value of ALL indicates that the traversal should continue as much as possible. The traversal ends when no additional connected relationships that match the criteria can be found. Note: a value greater than 1 can only be used if the relationship type is symmetrical, otherwise an error will result.

If there are cycles in the relationships, the Traverse() function will detect them and stop traversing along the cycle.

relationship\_filter\_exp - An optional parameter that is a logical IBPL expression which specifies a filter to be applied to the relationships encountered by traversal. The expression can be an arbitrary logical expression and check against the properties of attribute members (including measures) as well as relationship property values. Additional syntactic elements include: Prefixes of "To." or "From." (to denote TO or FROM nodes), "Measure.Name" to handle measure members, "Relationship." prefix (to denote properties of the relationship).

#### Examples for relationship\_filter\_exp:

##### 1. Example:

From.Date.[Month].#.Name!=P3-2010] && Relationship.Size < 6 && To.Measure.Name==Sales]

##### 2. Example:

To.Date.[Month].#.Name!=P3-2010] && To.Measure.Name IN {[Sales], [Revenue]}

Note that Measure.[Sales] should not be used since it refers to the measure value at a cell in arithmetic/logical expression. Here, we are checking for the measure itself. Therefore the correct, long version of the usage is:

[\_SchemaPlan].[\_Measure].Filter(#.Name == Sales])

The above is a simplified syntax.

#### Examples for traverse():

##### 1. Example:

traverse([ProductAffinity], Version.[Version Name].filter(#.Key==0).element(0))

##### 2. Example:

where

```
(Product.[Product].filter(#.Name==[Cosmo Cola Diet 12oz]))
.traverse([ProductAffinity], Version.[Version Name].filter(#.Key==0).element(0),UPSTREAM, 3)
```

### 3. Example:

```
(Product.[Brand].filter(#.Name==[Cosmo Cola]) * Measure.[Sales])
.traverse([Brand_Sales_Month_Profit], Version.[VersionName].filter(#.Key==0).element(0),
DOWNSTREAM, ALL)
```

### 4. Example:

```
(Product.[Product].Filter(#.Name==[Cosmo Cola Diet 12oz])).traverse([ProductAffinity], Version.[Version
Name].filter(#.Key==0).element(0),DOWNSTREAM, 2, From.Product.[Product].#.Name!= [Friendly Cola])
```

### 5. Example:

```
(Product.[Brand].filter(#.Name==[Cosmo Cola]) * Measure.[Revenue])
.traverse([Brand_Sales_Month_Profit], Version.[Version Name]
.filter(#.Key==0).element(0), DOWNSTREAM, 6,
From.Date.[Month].#.Name!= [P3-2010] && Edge.Size < 6 && To.Measure.Name==[Sales])
```

An additional note on how `starting_members_exp` get used as a filter for identifying the first set of relationships in the graph. If no `starting_members_exp` is specified, then every relationship of the specified relationship type passes the filter. However, when `starting_members_exp` is specified,

#### o9 Internal use Only 64

structurally it may or may not match either the TO or FROM member tuple set of the relationship type. If so, how does the filtering work? We address by checking for partial matches in some cases. Assume F and H are sets of level attributes that define the members in the filter expression and the TO node of the relationship type respectively. If H is a subset of F, then each TO node attribute can be fully tested and filtered against the set of filter attributes. If H and F are overlapping sets, then we filter by checking against members in the filter set for common level attributes. If the intersection of H and F is a null set, then all TO nodes automatically fail the filter test. The same approach is independently used for applying the filter against the results. In summary, if there is a starting member expression, we ignore the non-applicable attribute members for filtering purposes.

On a more general note, subsequent hops in the traversal use the TOs/FROMs from the relationships of the previous step to identify the connected relationships. If the TO and FROM of a relationship have different defining level attributes, we use the same partial match technique to identify the relationships that are connected.

## Distinct Count

### Distinct Count on MeasureGroups

#### ● Distinct Count on Measure Groups.

- DistinctCount query gives the number of records in the table based on dimension attributes /measures selected, filters and measure condition given in the using clause .
- All selected dimensionAttributes and measures should belong to same measure group.
- Can get count with filters on dimensions and measures.
- Possible to group-by primary keys (ie:attribute members).
- Output of distinctCount is given as a transient measure.
- Can specify a measure condition with using clause.

where

- It can be combined with other measures, measure cell properties, NULLS\_FOR\_FINER\_GRAIN\_SELECT etc.
- Can give multiple distinctCount in same query.
- DistinctCount using Table Function. transi
  - It acts as a placeholder for distinctcount query that generates a table with output as transient measure along with grouped attribute members if specified.
  - Implementation Approach
 

This table function- given a specified set of attributes, measures, filters, measure condition from using clause and a querycontext, generates a table of rows. (For table function of distinctCount, we have single transient measure as output or grouped attribute members along with count as transient measure).
  - Table function aspects:
    - i. An implementation of VirtualTableFunctionMeasure => DistinctCountTableFunctionMeasure. It should use the

o9 Internal use Only 65

TupleAggregateMeasureValueExpression info for generating a query plan

- ii. An implementation of BaseTableFunctionQueryPlan => DistinctCountTableFunctionQueryPlan that uses constructor to build the operator for distinct count and override ExecuteQuery() to generate the query results for the single DistinctCountTableFunctionMeasure.
- iii. An implementation of AbstractTableFunctionExpression => DistinctCountTableFunctionExpression. Creates the above virtual measure at appropriate granularity for execution
- iv. An implementation of MeasureValueExpression => TupleAggregateMeasureValueExpression to represent the value of a single distinct count in a query

- Let us consider an example

```
select (Product.[Brand Name] * [Date].[Reporting Month] * {Measure.[ACP Bill Back Plan] , DistinctCount(Customer.[Customer Name] * measure.[ACP Ticket Sales Plan]) as Transient.ProdCustCount using measure.[ACP Ticket Sales Plan] > 0)) on row, () on column where {Version.[Version Name].Filter(#.Key == 0)};
```

Above query can be rewritten into different subqueries:

```
subquery1 => Select ([Product].[Brand Name] * [Date].[Reporting Month] * {Measure.[ACP Bill Back Plan]}) where {Version.[Version Name].Filter(#.Key == 0)};
```

where

```
subquery2 => Select ([Date].[Reporting Month] * [Product].[Brand Name] *
{DistinctCount (Customer.[Customer Name] * measure.[ACP Ticket Sales
Plan]) as ProdCustCount}) from (Select ([Date].[Reporting Month] *
[Product].[Brand Name] * Customer.[Customer Name] * measure.[ACP
Ticket Sales Plan]) where {Version.[Version Name].Filter(#.Key == 0),
measure.[ACP Ticket Sales Plan]> 0})
```

```
subquery3=>subquery1 FullOuterJoin subquery2
```

Subquery2 is made through DistinctCount TableFunction. In DistinctCountTableFunctionQueryPlan, we do data source operator over the selected attributes([Date].[Reporting Month] \* [Product].[Brand Name] \* Customer.[Customer Name]),measure(measure.[ACP Ticket Sales Plan]) based on filters and queryContext. Over this we make an aggregation operator that count distinct values of measure.[ACP Ticket Sales Plan] or Customer.[Customer Name]grouped over [Date].[Reporting Month] \* [Product].[Brand Name].

o9 Internal use Only 66

Examples:

1. To count of customers with [ACP Ticket Sales Plan]  
select ({DistinctCount(Customer.[Customer Name]) as  
Transient.CustCount using measure.[ACP Ticket Sales Plan] >= 0}) on  
row, () on column where {Version.[Version Name].Filter(#.Key == 0)};
2. To count of customer \* [ACP Ticket Sales Plan] with [ACP Ticket Sales  
Plan]  
select ({DistinctCount(Customer.[Customer Name] \* measure.[ACP  
Ticket Sales Plan]) as Transient.ProdCustCount using measure.[ACP  
Ticket Sales Plan] > 0}) on row, () on column where {Version.[Version  
Name].Filter(#.Key == 0)};
3. To count of Month \* customer \* [ACP Ticket Sales Plan] with [ACP Ticket  
Sales Plan] group by Product  
select (Product.[Brand Name] \* {DistinctCount([Date].[Reporting  
Month] \* Customer.[Customer Name] \* measure.[ACP Ticket Sales Plan])  
as Transient.ProdCustCount using measure.[ACP Ticket Sales Plan] > 0})  
on row, () on column where {Version.[Version Name].Filter(#.Key == 0)};
4. To count of Month \* customer \* [ACP Ticket Sales Plan] with [ACP Ticket  
Sales Plan] group by higher level product Category and get the count of  
Month \* [ACP Ticket Sales Plan] with [ACP Ticket Sales Plan] for the  
same grouping.(Mutiple distinctCount in same query)  
select (Product.[Category Name] \* {DistinctCount([Date].[Reporting

where

Month] \* Customer.[Customer Name] \* measure.[ACP Ticket Sales Plan]) as Transient.MonthCustCount using measure.[ACP Ticket Sales Plan] > 0, DistinctCount([Date].[Reporting Month] \* measure.[ACP Ticket Sales Plan]) as Transient.MonthCount using measure.[ACP Ticket Sales Plan] > 0)) on row, () on column where {Version.[Version Name].Filter(#.Key == 0)});

5. Distinct Count with other measures, cellproperty and NULLS\_FOR\_FINER\_GRAIN\_SELECT  
 select (Product.[BDC Name] \* {measure.[ACP Ticket Sales Plan], DistinctCount(Date.[Reporting Month]) as Transient.MonthCount using Measure.[APD Base Ticket Sales Forecast] >= 0, Measure.[APD Base Ticket Sales Forecast], Measure.[APD Base Ticket Sales Forecast]/Transient.[MonthCount] as Transient.[Monthly Avg Sales] }) on row, () on column include cellproperties {LOCK, BG\_COLOR} NULLS\_FOR\_FINER\_GRAIN\_SELECT where {&CurrentWorkingView};

### More Examples for MeasureGroups (based on unittest Tenant)

o9 Internal use Only 67

1. **Select ([Customer].[Customer Name] \* [Date].[Reporting Month] \* [Product].[Brand Name]) on row, ({Measure.[ACP Ticket Sales Plan]}) on column where {Version.[Version Name].Filter(#.Key == 0), Measure.[ACP Ticket Sales Plan]>=0};**

Data at leaf level (910 rows affected)

2. **select ({DistinctCount(Product.[Brand Name] \* Customer.[Customer Name]) as Transient.ProdCustCount using measure.[ACP Ticket Sales Plan] > 0}) on row, () on column where {Version.[Version Name].Filter(#.Key == 0)});**

"ProdCustCount"

33

3. **select ({DistinctCount(Customer.[Customer Name] \* measure.[ACP Ticket Sales Plan]) as Transient.CustCount using measure.[ACP Ticket Sales Plan] > 0}) on row, () on column where {Version.[Version Name].Filter(#.Key == 0)});**

"CustCount"

21

4. **select (Product.[Category Name] \* {DistinctCount([Date].[Reporting Month] \* Customer.[Customer Name] \* measure.[ACP Ticket Sales**

where

Plan]) as Transient.MonthCustCount using measure.[ACP Ticket Sales Plan] > 0, DistinctCount([Date].[Reporting Month] \* measure.[ACP Ticket Sales Plan]) as Transient.MonthCount using measure.[ACP Ticket Sales Plan] > 0)) on row, () on column where {Version.[Version Name].Filter(#.Key == 0)};

"Category Name" "MonthCustCount" "MonthCount"  
 "Deli Fresh" 182 13  
 "Fuels" 194 13

5. select (Product.[Category Name] \* Product.[Brand Name] \* {DistinctCount([Date].[Reporting Month] \* Customer.[Customer Name] \* measure.[ACP Ticket Sales Plan]) as Transient.[Count] using measure.[ACP Ticket Sales Plan] > 0)) on row, () on column where {Version.[Version Name].Filter(#.Key == 0)};

"Category Name" "Brand Name" "Count"  
 "Deli Fresh" "Stacy's" 182  
 "Fuels" "Stax" 142  
 "Fuels" "Spitz" 78

o9 Internal use Only 68

6. Select ([Product].[Brand Name]) on row, ({DistinctCount([Product].[Brand Name]) as Transient.ExpCount using measure.[ACP Ticket Sales Plan] > 0)) on column where {Version.[Version Name].Filter(#.Key == 0)};

"Brand Name" "ExpCount"  
 "Stacy's" 1  
 "Stax" 1  
 "Spitz" 1

7. Select ([Product].[Brand Name]) on row, ({DistinctCount([Product].[Brand Name]\*[Date].[Reporting Month]) as Transient.ExpCount using measure.[ACP Ticket Sales Plan] > 0)) on column where {Version.[Version Name].Filter(#.Key == 0)};

"Brand Name" "ExpCount"  
 "Stacy's" 13  
 "Stax" 13  
 "Spitz" 13

8. Select ([Product].[Category Name]) on row, ({DistinctCount([Product].[Brand Name]\*[Date].[Reporting Month]) as Transient.ExpCount using measure.[ACP Ticket Sales Plan] > 0)) on column where {Version.[Version Name].Filter(#.Key == 0)};

where



"Category Name" "ExpCount"  
"Deli Fresh" 13  
"Fuels" 26

9. **Select** ([Product].[Category Name]\*[Product].[Brand Name]\*[Date].[Reporting Month]) **on row**,  
(**{DistinctCount**([Product].[Brand Name]) **as** Transient.ExpCount **using**  
**measure**.[ACP Ticket Sales Plan] > 0) **on column where**  
**{Version.[Version Name].Filter(#.Key == 0)}**;

"Category Name" "Brand Name" "Reporting Month"  
"ExpCount"  
"Deli Fresh" "Stacy's" "P1-2012" 1  
"Fuels" "Stax" "P1-2012" 1  
"Fuels" "Spitz" "P1-2012" 1  
"Deli Fresh" "Stacy's" "P2-2012" 1  
"Fuels" "Stax" "P2-2012" 1  
"Fuels" "Spitz" "P2-2012" 1  
"Deli Fresh" "Stacy's" "P3-2012" 1  
"Fuels" "Stax" "P3-2012" 1  
"Fuels" "Spitz" "P3-2012" 1

o9 Internal use Only 69

"Deli Fresh" "Stacy's" "P4-2012" 1  
"Fuels" "Stax" "P4-2012" 1  
"Fuels" "Spitz" "P4-2012" 1  
"Deli Fresh" "Stacy's" "P5-2012" 1  
"Fuels" "Stax" "P5-2012" 1  
"Fuels" "Spitz" "P5-2012" 1  
"Deli Fresh" "Stacy's" "P6-2012" 1  
"Fuels" "Stax" "P6-2012" 1  
"Fuels" "Spitz" "P6-2012" 1  
"Deli Fresh" "Stacy's" "P7-2012" 1  
"Fuels" "Stax" "P7-2012" 1  
"Fuels" "Spitz" "P7-2012" 1  
"Deli Fresh" "Stacy's" "P8-2012" 1  
"Fuels" "Stax" "P8-2012" 1  
"Fuels" "Spitz" "P8-2012" 1  
"Deli Fresh" "Stacy's" "P9-2012" 1  
"Fuels" "Stax" "P9-2012" 1  
"Fuels" "Spitz" "P9-2012" 1  
"Deli Fresh" "Stacy's" "P10-2012" 1  
"Fuels" "Stax" "P10-2012" 1  
"Fuels" "Spitz" "P10-2012" 1  
"Deli Fresh" "Stacy's" "P11-2012" 1  
"Fuels" "Stax" "P11-2012" 1

where

```

"Fuels" "Spitz" "P11-2012" 1
"Deli Fresh" "Stacy's" "P12-2012" 1
"Fuels" "Stax" "P12-2012" 1
"Fuels" "Spitz" "P12-2012" 1
"Deli Fresh" "Stacy's" "P13-2012" 1
"Fuels" "Stax" "P13-2012" 1
"Fuels" "Spitz" "P13-2012" 1

```

```

10. Select ([Product].[Category Name]*{DistinctCount([Product].[Brand
Name]) using measure.[ACP Ticket Sales Plan] > 0 as
Transient.ExpCount}) on row, ()
on column where {Version.[Version Name].Filter(#.Key ==
0),[Product].[Brand Name].filter(#.Name in {"Stacy's","Spitz"})};

```

```

"Category Name" "ExpCount"
"Deli Fresh" 1
"Fuels" 1

```

o9 Internal use Only 70

## Members

The Members() function is used for fetching the attribute members from a set of relationships (generated by Traverse() or any other function with similar output). The return value is one or more Member Tuple Sets. The function takes two optional parameters.

### Syntax:

```
SELECT DISTINCT <traverse_exp>.MEMBERS(<level_attr_list>?, <node_type>?);
```

The DISTINCT keyword is optional. However, in most graphs a given node can be a TO or FROM node of multiple edges. Therefore, to avoid duplicates, it is better to use the DISTINCT keyword in the select statement.

**level\_attr\_list** - An optional parameter that specifies a cross-join (i.e., an asterisk-separated list) of one or more attribute names. The Members() function uses this list as a template for projecting the subset of members that make up a given member tuple set. For each member tuple set, we extract only the members matching specified level attributes and include them in the output. This is very useful when the set of member tuple sets from the traversed relationships may not be homogenous with respect to the defining level attributes. If level\_attr\_list is not provided, all members of each member tuple set are included in the output. The syntax allows simple way of specifying a level attribute for measure member, by keyword Measure.

**node\_type** - An optional parameter that specifies whether the output should be restricted to only the TO or FROM of the relationships. Possible values are TO and FROM. If omitted, the output will include both TO and FROM member tuple sets.

### 1. Example:

```
select traverse([ProductAffinity], Version.[Version Name].filter(#.Key==0).element(0)) .members();
```

where

## 2. Example:

```
select distinct traverse([ProductAffinity],
 Version.[VersionName].filter(#.Key==0).element(0)) .members(FROM);
```

## 3. Example:

```
select distinct traverse([ProductAffinity], Version.[VersionName].filter(#.Key==0).element(0))
 .members(Product.[Product], TO);
```

## 4. Example:

```
select distinct traverse([ProductAffinity], Version.[VersionName].filter(#.Key==0).element(0), UPSTREAM, 2,
 From.Product.[Product].#.Name==[Cosmo Cola Diet 12oz] &&
 Relationship.Coefficient>=1.75).members(FROM);
```

## 5. Example:

```
select distinct (Product.[Brand].filter(#.Name==[Cosmo Cola]) *
 Measure.[Sales]).traverse([Brand_Sales_Month_Profit], Version.[Version
 name].filter(#.Key==0).element(0), DOWNSTREAM, 3)
 .members(Product.[Brand] * Measure, FROM);
```

## 6. Example:

```
select distinct (Product.[Brand].filter(#.Name==[Cosmo Cola]))
 .traverse([Brand_Sales_Month_Profit], Version.[Version Name].filter(#.Key==0).element(0),
 DOWNSTREAM, 3, To.Measure.Name!= [Revenue].members(Product.[Brand] *
 [Date].[Month]));
```

## 7. Example:

```
select distinct (Product.[Brand].filter(#.Name==[Cosmo Cola]) * Measure.[Revenue]) o9 Internal use Only
```

71

```
.traverse([Brand_Sales_Month_Profit], Version.[Version Name].filter(#.Key==0).element(0),
 DOWNSTREAM, 6, From.Date.[Month].#.Name!= [P3-2010] && Relationship.Size < 6 &&
 To.Measure.Name==[Sales]).members(Date.[Month] * Measure, TO);
```

## Using Members function in other select queries

Members() function produces a set of member intersections as specified by the level attributes in the <level\_attr\_list> clause. If the clause contains a single attribute, the output is reduced to a set of attribute members. In either case, the Members() function can be used in a select query in combination with other attribute membersets. Such use results in the query results being filtered not only by the attribute members in the relationship, but also by the specific intersections generated by the Members() function. Additionally, Members() functions based on a single attribute can also be used in WHERE clause of the queries.

```
SELECT ((<attr_member_set> <traverse_exp>.MEMBERS(<level_attr_list>?,<node_type>?))+
{Measure.[measure_name]});
```

## 1. Example:

```
select (&CwvAndScenarios * &SalesDomains * &PlanningPeriods *
 traverse([ProductAffinity], &Cwv.element(0)).members(Product.[Product], TO)
 {Measure.[Segment Forecast Detail Gross Revenue], Measure.[Segment Forecast Detail Trade Spend]});
```

## 2. Example:

where

```
select (&CwvAndScenarios * &PlanningPeriods * &PlanningQuarters * Product.[Brand] *
 traverse([ProductAffinity], &Cwv.element(0)).members(Product.[Product], TO)
 {Measure.[Segment Forecast Detail Gross Revenue], Measure.[Segment Forecast Detail Trade Spend]});
```

### 3. Example:

```
select (&CwvAndScenarios * traverse([Brand_Sales_Month_Profit],
 Cwv.element(0)).members(Product.[Brand] * Date.[Month], FROM) *
 &SalesDomains * {Measure.[Segment Forecast Detail Gross Revenue], Measure.[Segment Forecast Detail
 Trade Spend]});
```

### 4. Example:

```
select (&CwvAndScenarios * &PlanningPeriods * &PlanningQuarters *
 {Measure.[Segment Forecast Detail Gross Revenue], Measure.[Segment Forecast Detail Trade Spend]) on
 row, () on column WHERE {traverse([ProductAffinity], {Measure.[measure_name]+});
```

Current Limitations:

1. If the members function produces members for more than one attribute, each of those attributes must be the lowest level attribute among the selected attributes for that dimension.
2. if the members function produces members for more than one attribute, it cannot be used in the WHERE clause.
3. The INCLUDE\_SUBTOTALS option will not work correctly in combination with Members() function. [o9](#)

Internal use Only 72

## Rules Language

The “rules language” refers to the statements that are used during the startup of the GraphCube Widget engine. It is used to create named sets, set active rules for measures, and define procedures that can be executed later. There are statements that are permitted during startup that are not permitted (or do not have the same syntax) as when using IBPL in a script or command-line (the “command language”). And vice versa.

Named Sets can be specified in both startup and in the command language.

Active Rules can be specified in both startup and in the command language.

where

Procedures cannot be created in the command language.

# Command Language

The “command language” refers to the statements that are executed on the GraphCube Widget engine orderbyafter startup using the command-line client, or through a script.

## Select Queries

A select query in IBPL consists of the command “select”, followed by a list of AttributeMemberSets (or named sets) separated by asterisks, followed by a list of measures enclosed in curly braces, with the whole thing enclosed in parenthesis. Order by and Limit keywords can be used to sort the results and limit the number of intersections displayed. Keywords asc or desc can be used to display the results in ascending or descending orders.

### Example:

```
select (&CwvAndScenarios * &Products * &SalesDomains * &PlanningPeriods
* {Measure.[Segment Forecast Detail Gross Revenue],
Measure.[Segment Forecast Detail Trade Spend],
Measure.[Segment Forecast Detail Other Sales Reductions],
Measure.[Segment Forecast Detail Net Revenue]});
```

```
Select ([Time].[Fiscal Month] * [Product].[Product Name] * [SalesDomain].[Sales
Domain Name] * [Version].[Version Name]) on row,
({ Measure.[Cases], Measure.[Sales Dependent]}) on column orderby Measure.[Cases]
desc limit 10;
```

This will return a list of every combination of Version, Product, Sales Domain and Fiscal Month for which these measures have values. If you want to sum up the measures across all products and all sales domains, you can leave those dimensions out of the select query:

```
select (&CwvAndScenarios * &PlanningPeriods * {Measure.[Segment Forecast Detail
Gross Revenue],
Measure.[Segment Forecast Detail Trade Spend], Measure.[Segment Forecast Detail
Other Sales Reductions], Measure.[Segment Forecast Detail Net Revenue]});
```

o9 Internal use Only 73

Syntax for ascending sorting query

```
Select ([Time].[Fiscal Month] * [Product].[Product Name] * [SalesDomain].[Sales Domain Name] *
[Version].[Version Name]) on row,
({ Measure.[Cases], Measure.[Sales Dependent]}) on column orderby Measure.[Cases] asc limit
```

## 10; Executing Procedures

The command to execute a procedure is “exec procedure” and the name of the procedure. Example, to where

execute the procedure "ForecastModel", use the following command:  
exec procedure ForecastModel;

## Data Management Commands

### DownloadDataFile

The DownloadDataFile command is used to create Excel or text files from an IBPL select query.

```
DOWNLOADDATAFILE(<selectquery>,<path>,<includeData>,<fileType>,<delimiter>
,<useTranslationNames>);
```

#### Parameters:

- <selectQuery> – a select query whose result is output to an Excel or text file.
- <path> – either a local path in GraphCube Widget where the new Excel file will be created, or a cloud file Id.
- <includeData> – a boolean indicating if data is to be included. If false, then the output will contain the dimension members, but the measure values will all be missing. This is a way to create a data template for manually creating data in Excel and then loading it into GraphCube Widget (see the ImportDataFile command).
- <fileType> – a string which can take to values "EXCEL" or "DSV". If "EXCEL", then the output will be written into an Excel file, and if "CSV", then the output will be written into a text file.
- <delimiter> – an optional parameter indicating the delimiter to be used for text files. If no delimiter is specified the character '#' is used as default. For Excel files this value is ignored.

Download datafile

```
(select (Product.[Sub Category] * Product.[Group] * Product.[Category]
* Product.[Sub Group] * Product.[Product Name]) on row, () on column,
select (Proposal.[Group] * Proposal.[Proposal Name] * Proposal.[Sub Group])
on row, () on column,
select(Product.[Product Name] * Version.[Version Name] *
SalesDomain.[Sales Domain Name] * Proposal.[Proposal Name] *
{Measure.[Proposal Detail Incremental Gross Revenue]})
on row, (Time.[Fiscal Month]) on column,
"D:\scratchpad\MemberDataUpload\AE\RefModel.xlsx", false, true, "EXCEL");
```

download members of dimensions with its key and properties. This might be needed to create new members with reference to key,

```
DOWNLOADDATAFILE(Select ([Version].[Version Name]) on row, () on column include
memberproperties {Version.[Version Name], key}, "filelocation":"D:\ValidatedRDs.CSV",
"withdata":true, "usetranslationnames":false, "filetype":"CSV");
```

o9 Internal use Only 74

### UploadDataFile

```
UPLOADDATAFILE(<path>,

```

```
<isCloudFile>,<fileType>,<overrideWithNulls>,<delimiter>); Parameters:
```

- <path> : Path of the file to be uploaded. Or a cloudFile Id.
- <isCloudFile> : A boolean indicating if the file is a cloud file.
- <fileType> : A string which can take to values, "Excel" or "DSV", indicating the type of file.
- <overrideWithNulls> : Boolean value, Used when particular intersection should be given NULL value.
- <delimiter> : Incase the file is a dsv file this parameter indicates the delimiter. If no delimiter is

where

specified then a default delimiter is used as "#".

**Example:**

```
UPLOADDATAFILE("D:\\scratchpad\\MemberDataUpload\\AE\\RefModel.xlsx", false, "Excel");
```

```
UPLOADDATAFILE("s3://guVKBOyaDVgpmU-JuiguRKoo8ZOUPBKF-23pslqHD4K04AnpaeL7
bZkwhEZVFI3I1Oxuq42v_P-A_uSQyUyC1qNBCdpVu_ceUp_FBr1b3Ljr5qh41oXTnllNjXbr_
4oxswLUvhNgkNXAr3x8VlrAA2", True, "CSV", False, ",");
```

Uploading multiple datafiles:

Here's the syntax - `UPLOADDATAFILE(file1, file2, file3, false, "excel");`

Example:

```
UPLOADDATAFILE(
"D:\\Arun\\DemandPlanning_20181004\\Dimension.Customer.dsv",
"D:\\Arun\\DemandPlanning_20181004\\Dimension.Product.dsv",
"D:\\Arun\\DemandPlanning_20181004\\Fact.ProductRegionAssociation.dsv",
"D:\\Arun\\DemandPlanning_20181004\\Fact.Shipments.dsv",
"D:\\Arun\\DemandPlanning_20181004\\Fact.UnitOfMeasure.dsv", false, "DSV", false, "|");
```

## ExportAll

This command dumps the entire Live Server data in form of dsv files.

`ExportAll(<path>, <delimiter>, <includeKeys>);`

**Parameters:**

- `<path>` : An optional parameter indicating the local path of the folder where export files will be created. If no path is provided then a default path ([DataFolder]/export) is used.
- `<delimiter>` : An optional parameter indicating the delimiter of the dsv files created during export.
- `<includeKeys>` : Optional boolean flag indicating if Keys are to be include in downloaded files

## Selective ExportAll

Support present for selective Export All, i.e all combinations of Dimensions, Models(Fact) & Graphs(RelationshipTypes).

ex of query structure:

`ExportAll();` : exports all data in GraphCube

`ExportAll() include Dimensions All; // Exports only the master data, All dim tables`

o9 Internal use Only 75

`ExportAll() include Models All; // Exports all fact data`

`ExportAll() include Graphs All; //Exports all graph data`

`ExportAll() include Dimensions All except Product, Customer; : Exports all dimensions except Product and Customer dims.(similarly for fact and graphs.. measureGroups which have space as separator should be given in " [ ] " ).`

`ExportAll() include dimension Customer include Models [Account Plan], [Demand Plan] include Graphs All except [CannibalizationFlow];`

etc

Changes in the result message :

Need to zip the files created by `ExportAll()`, and upload it to cloud, send the zipped cloud Id as a part of

where

Result message, So that this link can be displayed in UI for download.

new format of result message : {"Localpath" : "D:/vmbaba" , "CloudId" : "Cloud Path"}

## ImportAll

This command imports all data from export folder created by ExportAll command into GraphCube Widget. It can only be run once, that too when the GraphCube Widget is totally empty.

ImportAll(<path>, <delimiter>);

### Parameters:

- <path> : An optional parameter indicating the local path of the Export files. If no path is provided then a default path ([DataFolder]/export) is used.
- <delimiter> : An optional parameter indicating the delimiter of the dsv files created during export.

# Member Management Commands

## CreateMember

CREATEMEMBER(CurrentTimeModel.[Fiscal Quarter] = {ToDatetime("2013-04-21 00:00:00.000"), "Q2-2013"});

### Explanation:

Creates a new Member in "CurrentTimeModel" dimension, with name "Q2-2013" and key as the specified date. The key is optional for creating members in dimensions where the keys are integers, (Example: Product, Sales Domain, Proposals etc), they will be auto generated as shown below.

Createmember(Product.[Product Name] = {, "Cosmo Cola Diet 12oz"}, Product.[Weight] = {"12 oz", }, Product.[Brand] = {, "Cosmo Cola"});

Above ibpl command will create a product called 'Cosmo Cola Diet 12oz' under the brand 'Cosmo Cola'. Weight is the property of the product. The Key Generation method for Product has to be Auto for above command to work (see the picture below).





## UpdateMember

### Example:

```
UPDATEMEMBER(Version.[Version Name]={,"V1"}, Version.[Version Creation Date]={ToDatetime("2012-12-02 00:00:00.000"),});
```

### Explanation:

Updates the “Version Creation Date” property of Version “V1” to specified date.

### Realignment Example:

```
hierarchy:Product.[Name] -> Product.[Sub Group]->Product.[Group]
UPDATEMEMBER(Product.[Name]={,"P1"}, Product.[Sub Group]={,"New Sub Group"}); Explanation:
```

Product.[Name] member named P1 is realigned from “Old Sub Group” to “New Sub Group”.

### Example:

```
UPDATEMEMBER(Product.[Name]={,"P1"}, Product.[Group]={,"New Group"});
```

### Explanation:

This update command will fail. P1 is being realigned to move to a “New Group” but its parent(Sub Group) is not specified in the command.

## CopyMember

### Example:

```
COPYMEMBER(Product.[BDC Name].Filter(#.Name == "SY PITA SUPER REG" || #.Name == "SY PITA SUPER GARLIC"));
```

Or

```
COPYMEMBER(Product.[Flavor Name].[Vanilla].Children(Product.BrandFlavorHierarchy)); o9 Internal
```

where

## Delete Member

For example there is this following condition:

When there is fact data against the dimension-member in the version then the member is not deleted from CWV, Scenario and Version but fact data is deleted from CWV and Scenario after a purge command.

Now the user also does not want to delete fact data from CWV.

This is not yet supported because the delete member works in the following way:

- An attribute member is not a version-specific entity. If it exists, it will exist for all versions/scenarios.
- When a member is deleted, all the fact data in CWV and all scenarios gets wiped out. Only the fact data in read-only versions is retained.

- This fact data deletion happens right away as a part of member deletion. It is unrelated to purge command.
- After a member is deleted, you cannot add any fact data for that member in CWV or any scenario (that includes existing and new scenarios).

## Measure Management Commands

### CopyMeasure

Syntax:

```
COPYMEASURE([Dim].[Dim attr]={{"{src member}"},{tgt member}}},{measures},{bool}
RollBackOnConflict) WHERE {other filters,Version.[Version Name].filter(#.Name ==
"CurrentWorkingView")};,
```

COPYMEASURE

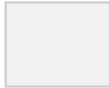
```
(Product.[BDC Name]={{"STACY PIT SNK MIX","STACY' PALLET LEM/HE"}},
{Measure.[OPP Override Forecast Cases], Measure.[OPP Target Inventory Cases]}, false)
WHERE {Version.[Version Name].Filter(#.Name == "CurrentWorkingView"),
Date.[Reporting Week].Filter(#.Name == "13x4-2013")};
```

In this command we are trying to Copy Measure.[OPP Override Forecast Cases] and Measure.[OPP Target Inventory Cases].

All intersections in the Measure Group where BDC Name = "STACY PIT SNK MIX" will be copied. In the copied intersections "STACY PIT SNK MIX" will be replaced by "STACY' PALLET LEM/HE".

Boolean argument after the measure names specify if measure value has to be overwritten if the destination intersection already exists. Version Filter is mandatory in the where clause.

where



NULLIFY is an IBPL command. It is used to nullify one or more measures in a large measure group. This is faster and shows memory efficiency than writing a scope statement which nullifies across the full scope.

#### o9 Internal use Only 78

However, do not use NULLIFY in an action button as it acquires a global lock (similar to save).

The following command deletes the extra intersections that are created in the system because of explosion of one Boolean measure over the time:

```
NULLIFY {Measure.[M1]} for Version.[Version Name].[CurrentWorkingView];
```

Another example which can be used in action button

```
COPYMEASURE([Version].[Version
Name]={{{"{{SelectSourceScenario}}"},"{{SelectTargetScenario}}"}},{Measure.[RM Transfer to FOB
vendor],Measure.[RM Consumption],Measure.[RM Expected Inwards],Measure.[RM Sales or
Liquidation],Measure.[Trims Consumption],Measure.[Trims Expected Inwards],Measure.[WIP
Closing Inventory]},true) WHERE {[Time].[Month].filter(#.Key >= [Time].[Month]{{#filter
StartMonth}}.element(0).leadoffset(0).Key && #.Key <=
[Time].[Month].{{EndMonth}}.Key),Version.[Version Name].filter(#.Name ==
"CurrentWorkingView")};
```

## Version Management Commands

### CreateVersion

```
CreateVersion(0, 1, "V2", false, false);
```

#### Explanation:

- Parameter1: Key of the source Version/Scenario
- Parameter2: Key of the plan to be version(future)
- Parameter3: Name of the new Version
- Parameter4: IsOfficial (Official, Published version from this planning cycle)
- Parameter5: isAOP (Whether this version is the AOP/target defining version)

### Create Version with scope

```
CreateVersion(0, 0, "Scoped Version", false, false,
for Model [Target WOS] using scope(Date.[Reporting Month].Filter(#.Name == "P1-2010")), for Model
[Seasonal Profiles] using scope (Product.[Category Name].Filter(#.Name == "Fuels"));
```

#### Explanation:

- Parameter 1: source Version/Scenario Key
- Parameter 2: Plan Key
- Parameter 3: Version Name
- Parameter 4: IsOfficial

where

- Parameter 5: IsAOP
- Parameter 6: Measure Group specific scope

Unlike a non scoped version a scoped version will only have data populated at the scopes specified in the argument. User can provide specify any number of measure groups in the argument. The scope can be at higher or leaf level.

o9 Internal use Only 79

## DeleteVersion

```
deleteversion("Version_name");
```

## Update Version Property

## CreateScenario

```
CreateScenario(0, 1, "Proposal Selection Scenario");
```

### Explanation:

- Parameter1 : Key of the source Version/Scenario
- Parameter2: Key of the plan to be version(future)
- Parameter3: Name of the Scenario to be created

## UpdateScenario

```
UpdateScenario(<toScenarioName>, <fromScenarioName>, bool);
```

### Explanation:

- Parameter1: Destination Scenario(or CWV) for the commit
- Parameter2:Source Scenario for the Commit
- Parameter3: ToScenarioWins: If the same intersection has 2 different values, in the source scenario vs the destination scenario, then which value should prevail after the scenario commit action.

## Update Scenario with scope

```
UpdateScenario("Test", "Source", true, for model [Account Plan] using scope ([Customer].[Channel Name].Filter(#.Name == "Club")[Date].[Reporting Month].Filter(#.Name == "P1-2011") [Product].[Brand Name].Filter(#.Name == "Spitz")));
```

### Explanation:

- Parameter1: Destination Scenario(or CWV) for the commit
- Parameter2: Source Scenario for the Commitcreate scenario
- Parameter3: ToScenarioWins: If the same intersection has 2

Fourth argument is similar to the scope argument in Create Scenario with scope. Only the specified scope will be committed from FromScenario to ToScenario. Multiple Measure Groups can be specified in the scope, as shown below.:

```
UpdateScenario([CurrentWorkingView], [MyScenario], false,
 for model [A] using scope (Time.[Year].[FY2018]),
 for model [B] using scope (Time.[Year].[FY2018] * Product.[Brand].[Some]),
 for graph [C] using scope (FROM Time.[Year].[FY2018]);Grouping
```

where

syntax for private scenario:

```
createScenario("MasterVersion",1,"ScenarioName") FOR "email-id of user"
```

## **DeleteScenario**

o9 Internal use Only 80

where