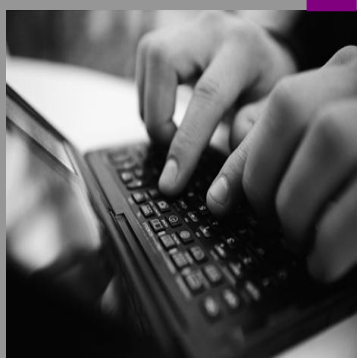


How-to Guide
SAP CPM

How To Use BPC Scripting Logic

Version 1.00 – April 21, 2006

Applicable Releases:
SAP BPC 4.2 SP3

Purpose of this document

This document describes the design of the Everest Modeling Logic module, and how it works. It is intended for administrators that understand the functionality and architecture of Everest. A basic understanding of the MDX syntax will also be helpful for writing modeling logics in MDX format. On the other hand the so-called SQL-based logics do not need practically any knowledge of SQL language, as most of the instructions are strictly Everest-specific jargon, which is fully documented here below.

Major differences from version 4.2 SP2

The most significant enhancement is the support for a new syntax for handling allocations. This feature is described in a new section of this document.

Introduction

The Everest Modeling Logic module enhances the capability of Everest in executing user-defined calculations and in deriving new data to store in the cube.

This module enables Everest to perform what can be referred to as "Base Level Logic" calculations, i.e. calculations that only need to be performed on base level cells (cells where all members have no children in any dimension). The results of such calculations need to be simply aggregated up the dimensional hierarchy as they are, without being re-calculated at upper levels. For this reason, Modeling Logics are also called "base level logics" or "base level formulas". In some case these logics have been referred to as "calc and store formula" because the results are stored in the database. Finally (in the most misleading of the definitions) you may also find them named "advanced formula".

With such functionality the user can perform:

- Calculations that we usually call "units-times-price calculations". Such calculations were hard to define in early versions of the product because, as already said, they must only be performed on base level cells.

- Currency conversions, i.e. calculations that derive translated numbers from the amounts in local currency and store them in the appropriate reporting currency members of the currency dimension

- Inter-company eliminations, i.e. eliminations of inter-company relationships between entities within a common parent in a given hierarchy of entities

Technical note: the logic module engine K2Logic.DLL is fully backwards-compatible with any version of Everest and EAP, in the sense that later builds can also be installed and used on prior versions of the product.

When to use modeling logic

The administrators can choose among different possible techniques for calculating data in Everest, and modeling logic must only be used when needed. To perform calculations, the administrator can:

Use Excel formulas stored in the Excel sheets used for data entry
Use dimension formulas
Use hierarchical relationships to define aggregations
Use modeling logic

Each technique has advantages and limitations and the correct one should be carefully selected. As a simple rule of thumb:

- Excel formulas are flexible and fast but are hard to control and do not work with data loads.

- Dimension formulas are easy to use but not very flexible, as they apply to all levels and to all cubes that use a given dimension. They are also fast enough if used sparingly, but do not scale well on larger applications.

- Hierarchical relationships can only be used to define some aggregations and nothing else. This is where relying onto an OLAP cube pays off. On the other hand, OLAP does not handle very efficiently multiple hierarchies on the same dimension.

- Modeling Logics are powerful, flexible and efficient, but must be used in the appropriate way. They also increase the amount of stored data and must be re-executed whenever data change. They should only be used when all other options do not apply.

The approach

Here is how modeling logic work: The Logic Module loads in memory a specific selection of data from an Everest cube, applies to it a set of user-defined formulas and derives an appropriate set of calculated amounts. These amounts are then written back into the fact table as if they were additional input data.

The Logic Module is invoked automatically every time data are sent to the cube, but the user can also decide to invoke this module in a batch mode (using some appropriate DTS package), to perform specific calculations against a desired selection of data of the cube at any point in time.

For example, some initial calculations could be performed every time the user enters data via WebExcel, i.e. using Excel spreadsheets containing the usual EVSND( ) or EVDRE( ) functions. Some other calculations, like currency conversions, could be deferred to a later phase: after all data in local currency have been entered and/or calculated, the administrator may decide that it’s time to generate the translated amounts in the desired reporting currencies. He can do this using the Logic Module in a batch mode, by running an appropriate DTS package from Data Manager.

Note: the Logic Module can be invoked from a DTS package using Data Manager’s custom task EvDTSRunLogic. More on this later.

Logic files

The formulas or instructions used by the Logic Module are NOT stored in the dimension members sheet, as done with dimension formulas, but are written in text files called “logic files” having an extension LGF (for example AnyLogic.LGF), that are stored in an application folder below the AdminApp folder of a given AppSet. The full path to access them is:

{webfolders}\{AppSet}\AdminApp\{App}\

For example, a logic file for the AppSet NEWSHELL2, application MAIN, could be stored in the directory:

C:\Everest\WebFoldes\NEWSHELL2\AdminApp\MAIN

An additional interface layer has been added in the first release of Everest 4.0 represented by excel workbooks where the logic instructions are initially written. When the workbook is validated, the LGF file is anyway created behind the scenes. The users who prefer to write their logics with their text editor of choice can still do it editing directly a text file that can then simply included in the Excel workbook with a simple INCLUDE instruction (more on this syntax later).

Validated logic files

When a logic is executed, the Logic Module does not actually process the instructions contained in the original LGF file, but those found in an executable version of such file, called like the original LGF file, but with the extension LGX, and stored in the same folder. Such file is generated by the Logic Module whenever a logic is ‘validated’.

This means that:

- For each source logic file edited by the administrator (the LGF files), there will be a corresponding LGX file that will contain the executable MDX instructions.

- All logics must be validated and saved in their LGX format, in order to be ready for execution. This can be done with a specific task provided by the admin module.

The validation process serves several purposes:

- It checks the validity of the syntax used in the original logic file
- It executes special instructions that can be entered in the logic file to generate additional statements in the executable file.
- It provides an intermediate step that we can use to hide to the user some of the intricacies of the syntax.

Technical considerations: the LGX files are text files stored in the same folder of the LGF files. They could as well be loaded in any text editor like Notepad to verify the results of the validation process.

Remark: if the user tries to run a logic file with an extension different from LGX, the Logic Module will not return an error, but it will be smart enough to automatically validate this file in memory. If the validation is passed, the logic is executed. In this case, however, a corresponding LGX file will NOT be generated, nor saved. It will have been created in memory, executed, and then forgotten.

Which logic is executed

When the Logic Module is invoked, it must be instructed what logic to execute.

When data are sent to the cube via EVSND( ) and EVINP( ) functions, WebExcel calls the logic Module without passing to it any logic name. In this case, the Logic Module assumes that a default logic should be used. (This logic is indeed called DEFAULT.LGX).

The same principle applies when the Logic Module is invoked from Data Manager for batch executions. In this case, however, it is possible to pass to the Logic Module a selected logic name, if desired. Such logic name could be hard-coded in the package or prompted to the user with a PROMPT( ) instruction added to an EAPMODIFY task included in the package.

Remark on the default logic

The default logic DEFAULT.LGF can be edited and validated like any other logic. However, whenever a dimension is re-built, the default logic is always automatically validated in all Apps using such dimension. This means that in most cases the validation of the default logic is taken care of automatically.

When data are entered in the cube, the logic DEFAULT.LGX is always executed. However, if the default logic is not found, no error message will be generated. This allows the definition of applications not using any default logic.

The calculation query and the scope of the logic execution

When the Logic Module is invoked, it needs to know two fundamental pieces of information to run correctly:

- The name of the logic file to execute
- The data region against which the formulas must be executed (the “scope” of the execution)

The way the logic module gets such information varies, depending on the way it is invoked.

Running from DTS:

If the Logic Module is invoked as a task inside a **Data Manager package**, the package itself needs to provide such information, and it could do it through a *prompt to the user*. For example, with a prompt, the package could ask the user “what logic shall I run?” and “What is the data region you want me to process?” And the user could answer: ”Run the logic called Translation for January 2002, category ACTUAL, entity SalesItaly”. The logic module will then run the selected logic (Translation) only for the selected members of the selected dimensions. The members of all other non-specified dimensions will be assumed as “all non-calculated” (if the app in the example included a product dimension, and the user did not specify a product, this would imply that the logic will run for all non-calculated products). The only exception to the rule is the currency dimension (if existing in the cube). If no currency is selected, and a currency dimension is used in the cube, the logic will be executed just for currency “LC”.

In an execution triggered by a Data Manager package, the definition of the data region could also be left to the logic module itself, and not be asked to the user through a prompt. The package could tell the logic module to just go and figure out by itself the scope of the execution *by reading the content of a given data file*. The scope of the resulting query will be restricted by the logic module to only the members found in the data file belonging to the dimensions of type CATEGORY, TIME and ENTITY. This behavior can be adjusted using some appropriate logic instructions.

***Running from Excel**:*

In case the Logic Module has been invoked after some data have been entered from an **Excel** sheet, the logic module will follow these rules:

(1) it will run the DEFAULT logic, and …
(2) it will run against a data region that is derived by the scope of the data that have just been entered into the cube.

In order to build the correct data selection, the logic module will rely on all occurrences of ALL different combinations of dimension members that are found in the posted data, including the currency dimension, but not including the ACCOUNT dimension.

For example, let's assume that the user has entered data for six cells in the cube. The corresponding records could be the following:

CATEGORY	TIME	ENTITY	CURRENCY	ACCOUNT	PRODUCT	PERIODIC
ACTUAL	2000.JAN	SALESITALY	LC	UNITS	PRODUCTA1	12345
ACTUAL	2000.JAN	SALESITALY	LC	INPUTPRICE	PRODUCTA1	100
ACTUAL	2000.FEB	SALESITALY	LC	UNITS	PRODUCTA1	54321
ACTUAL	2000.FEB	SALESITALY	LC	INPUTPRICE	PRODUCTA1	200
ACTUAL	2000.FEB	SALESITALY	LC	UNITS	PRODUCTA2	6789
ACTUAL	2000.FEB	SALESITALY	LC	INPUTPRICE	PRODUCTA2	300

In this case, WebExcel will pass a selection containing the category ACTUAL, the time periods 2000.JAN and 2000.FEB, the products PRODUCTA1 and PRODUCTA2, the currency LC, and the entity SALESITALY.

With the information specifying the scope of the data to process and the formulas found in the default logic, the Logic Module will be able to return the appropriate set of calculated cells.

Remarks in logic

To improve readability, logic files support blank lines and remarks. Remarks must be preceded by a double slash (//). Anything to the right of the double slash will be ignored during the validation process. Example:

```
//This is a remark
#COST = -[#GROSSSALES]*80/100          //this is also a valid remark
```

Special logic Instructions

Over and above regular MDX expressions or remarks, the logic files can contain special instructions, which are interpreted by the Logic Module during the validation process or when the logic is executed. Such instructions are all expressions beginning with an asterisk (*) character.

The original list of special instructions has grown tremendously since the introduction of modeling logic, and now its not uncommon to find logics that are made entirely of these special instructions, especially since MDX-based logics have lost most of their appeal. The full list of the instructions currently supported is shown here below (in alphabetical order):

*ADD / *ENDADD structure
*ADD_DIM {dim name}={member name}
*ADD_TABLE={tablename}
*BEGIN / *END structure
*BLOCK_SECURITY
*CALC_DUMMY_ORG {dim name}={property name}
*CALC_EACH_PERIOD
*CALC_ORG {dim name}={property name}[,DUMMY]
*CALCULATE_DIFFERENCE = 0 | 1
*COMMIT
*COMMIT_EACH_LEVEL = {Dimension name}
*COMMIT_EACH_MEMBER = {Dimension name}
*COMMIT_MAXMEMBERS
*CLEAR_DESTINATION
*DESTINATION
*DESTINATION_APP = {app name}
*FOR / *NEXT structure
*FUNCTION {functionname}({Param1}[,{Param2}…]) = {Function Text}
*FUNCTION / *ENDFUNCTION structure
*GO
*IGNORE_SECURITY
*IGNORE_STATUS
*INCLUDE {includedfile}(Param1,Param2,….)
*JOIN({tablename} , {dimension.property} [, {tablefield}] [, {selected fields}])
*LAST_MEMBER {Dimension name} = {Member }
*LOCAL_CURRENCY={Member}
*LOGIC_BY = {dimensions list}
*LOGIC_MODE = 0 | 1 | 2
*LOGIC_PROPERTY = {property name}
*LOOKUP / ENDLOOKUP structure
*MEASURES = {dimension}
*MEMBERSET({variable}, {member set in MDX format})
*NO_PARALLEL_QUERY
*PROCESS_EACH_MEMBER = {dimensions list}
*PROCESS_FAC2
*PUT( )
*PUTSCOPE_BY
*OLAPLOOKUP [{Application name}]
*QUERY_FILTER = {member set}
*QUERY_SIZE = 0 | 1 | 2
*QUERY_TYPE = 0 | 1 | 2
*REC( )
*RENAME_DIM {dim name}={new dim name}
*RUNLOGIC / ENDRUNLOGIC
*RUN_STORED_PROCEDURE={stored procedure name}([{params}])
*SCOPE_BY = {dimensions list}
*SKIP_DIM = {dimension name}
*SELECT ({variable}, {What}, {From}, {Where})
*SELECTCASE / *ENDSELECT structure
*STORE_ORG {dimname}={propertyname}
*SUB (Param1,Param2,….) / ENDSUB

*SYSLIB {Includedfile} (Param1,Param2,….)
*TEST_WHEN
*USE {filename}
*USE_UNSIGNEDDATA
*WHEN / ENDWHEN
*WRITE_TO_FILE = {filename}
*WRITE_TO_FAC2
*XDIM_ADDMEMBERSET {Dimensio name} = {Member Set}
*XDIM_DEFAULT {Dimension name} = {Members Set}
*XDIM_FILTER {Dimension name} = {Members Set}
*XDIM_GETINPUTSET / *ENDXDIM structure
*XDIM_GETMEMBERSET / *ENDXDIM structure
*XDIM_MAXMEMBERS {Dimension name} = {Number of members}
*XDIM_MEMBER {Dimension name} = {Member1} [TO {Member2}]
*XDIM_MEMBERSET {Dimension name} = {Members Set}
*XDIM_NOSCAN {Dimension name} = {Members Set}
*XDIM_REQUIRED = {dimensions list}

Here below you will find an explanation of each one of them.

Included Files

Let's assume that our application contains two logics: the default logic and one translation logic. The default logic is always automatically executed when data are entered from WebExcel, but the translation logic must be executed manually after the initial data have been entered.

This may be acceptable for some cases, but it may happen that the administrator prefers the translation logic to be executed immediately every time some data have been entered. The benefit would be that, even if the data entry will become slower, there would be no need to translate the numbers at a later stage.

One simple way to obtain this result will be to copy-paste the entire translation logic at the bottom of the default logic file (preceded by a *COMMIT instruction). By so doing, the default logic will perform the entire process in one pass, whenever data are entered (execute the default formulas, save the results then execute the translation formulas).

Instead of duplicating the entire translation logic at the bottom of the default logic, a more elegant way to accomplish the same result is to write, at the bottom of the default logic, the instruction:

*INCLUDE TranslationLogic

…where TranslationLogic is the name of the file containing the translation logic instructions.

This will accomplish the same result of copy-pasting the entire translation logic in the default logic, with the benefit of keeping open the possibility to execute each logic independently while avoiding the maintenance of the translation logic in two different files.

Remarks:

- 'Include' instructions can be entered anywhere in a file
- Multiple 'include' instructions can be entered in the same file
- 'included' files can be nested for as many levels as desired, i.e. an 'included' file can in turn include other files, etc.

- circular references will generate an error message during logic validation.
- Missing included files will generate an error during logic validation
- If the included file has no path specified, the default path is: {WebFolders}\{AppSet}\AdminApp\{App}
- If the included file has no extension specified, the extension defaults to LGF.

Passing parameters to included files

When a logic or library file is included with the instruction *INCLUDE, any number or parameter can be passed to it. The syntax is:

*INCLUDE {includedfile}(Param1,Param2,….)

The parameters, inside the included file, MUST be referenced to with the reserved names %Pn%, where n is the position of the parameter in the passed list.

Example:

*INCLUDE MyModule.LGF (REVENUE, COST)

The content of the file MyModule.LGF could be:

```
#GROSSPROFIT= [ACCOUNT].[%P1%] – [ACCOUNT].[%P2%]
#GROSSMARGIN= (([ACCOUNT].[%P1%] – [ACCOUNT].[%P2%])/[ACCOUNT].[%P1%])*100
```

The benefit of using included files instead of multi-line functions is that this method permits to include entire logic sections, containing any type of instructions like *COMMIT, *XDIM_MEMBER, etc., while logic functions can only perform string substitutions on single line statements. On the other hand, functions do not need to be written in separate files.

Remark: this type of functionality can now be better obtained using *SUB procedures, as later described. SUB procedures can have user-named parameters, and (while they still can) they do not need to be stored in individual external files.

Reserved folder for System Logic Libraries

A special instruction can be used to include logic library files in a modeling logic. This instruction is:

*SYSLIB {Includedfile} (Param1,Param2,….)

This instruction works exactly like the *INCLUDE instruction, however the file to be included will be searched for in a reserved folder, that is only intended to contain logic libraries provided by Outlooksoft, and should not be modified by the user. This folder will exist below the Appset folder with the name “SystemLibrary\Logic Library”. Version Mont Blanc and subsequent will come with a certain number of library files provided by Outlooksoft.

The COMMIT instruction

It may happen that a logic file contains formulas that depend on the result of calculations performed by the cube, and that these calculations in turn depend on the results of some other formulas in the same logic.

Take this example:

[Account].[#1] = {expression}
[Account].[#2] = ([ENTITY].currentmember.parent,[Account].[#1])

In this example account 2 depends on the calculation of the parent entity values performed by the cube, and this calculation in turn depends on the calculation of account 1.

This logic, if written in the above format, will not work correctly, because account 1 cannot be retrieved from the parent of current entity until its result has not been posted to the cube. To get the right results, account 1 must be calculated AND stored in the cube. THEN, the 'calculated' result can be retrieved from the parent and be used to calculate account 2.

In order to force a write back of the result of the calculation of account 1 into the cube, the instruction "*COMMIT" can be inserted between the two calculations, to enforce a write back of account 1 before the calculation of account 2. The logic will then work if written as follows:

[Account].[#1] = {expression}
*COMMIT
[Account].[#2] = ([ENTITY].currentmember.parent,[Account].[1]

Note that in this case account 1 in the second formula does not have the pound sign (#), because it is a 'stored' amount read from the cube.

Remark: Any number of commit instructions can be entered in a logic file. However, the number of commit instructions should be kept to the minimum, because they have a negative impact on the overall performance of the logic execution.

*COMMIT_EACH_MEMBER={dimname}

This instruction will enforce a commit for each member of the selected dimension. If the dimension is of type TIME, the members will also be sorted in ascending sequence, so that older periods will be processed first. In addition, the logic will be executed for all periods between the oldest and the youngest, filling any gaps existing in the range of selected periods. This can be useful for formulas performing a carry-forward of prior periods values like in this example.

#ClosingBalance= (ClosingBalance, lag([TIME].currentmember,1)) + Changes

*COMMIT_EACH_LEVEL={dimname}

This instruction will sort and group all members of the selected dimension and enforce a logic execution from the bottom to the top of the structure, with a commit between each level. This feature can be used to

ensure that the eliminations performed on each level take into account the results of the eliminations performed on all lower levels.

***LAST_MEMBER {Dimension name} = {Member}**

With this instruction, the logic will be run for the indicated dimension from the first passed member to the member specified in the instruction itself. For example, if the instruction says:

*LAST_MEMBER TIME=2002.DEC

…the logic will run from the first modified period up to period 2002.DEC.

The year can be made dynamic using the keyword %PREFIX% as follows:

*LAST_MEMBER TIME=%PREFIX%.DEC

In this case, the year will be the year of the last modified period.

The instruction can also interpret some MDX functions in the passed parameters. These can be used, for example, to define an offset from a given period. The following syntax:

*LAST_MEMBER TIME=[%PREFIX%.DEC].LEAD(6)

…will enable the logic to run from the first selected period until June of the following year.

Hints: the member name must be enclosed in [brackets] to be recognized as an MDX call. The dimension name must be however omitted from the expression.

The instruction *XDIM_REQUIRED

Sometimes it may happen that a user, when running a logic from Data Manager, forgets to select a member for one of the proposed dimensions and, for example, executes the logic for all entities instead of just one or two. This situation can be controlled inside the logic, using the instruction:

*XDIM_REQUIRED={dimname}[,{dimname}]

Example:

*XDIM_REQUIRED=CATEGORY, TIME

Multiple dimensions can be passed in the same instruction, separated by commas.

If no selection is passed for any of the required dimensions, the logic will abort and an appropriate error message will be returned.

The instruction is valid for the entire logic and not just for one COMMIT section.

The instruction *PROCESS_EACH_MEMBER

The instruction *PROCESS_EACH_MEMBER has been implemented, in addition to the already described COMMIT_EACH_MEMBER, to serve a more specific purpose. This instruction supports the exact same syntax of COMMIT_EACH_MEMBER, i.e.:

***PROCESS_EACH_MEMBER={dimname1}[, {dimname2}, …]**

However, it works in a significantly different way.

- **It applies to the entire logic and not to a specific COMMIT section**
- It is processed separately, before the logic is interpreted. This makes the logic behave exactly as if it had been called multiple times (once for each member in the dimensions listed in the instruction).
- The member set to process for the named dimensions must be explicitly passed to the logic call, as if such dimension sets were required (see instruction XDIM_REQUIRED).
- It applies to both MDX logics as well as SQL logics (COMMIT_EACH_MEMBER, on the other hand, only works on MDX logic)

Similarly to COMMIT_EACH_MEMBER, this instruction processes the TIME dimension in a special way, i.e., it sorts the time members from the oldest to the newest and fills all gaps in the range. For example, if the members passed are:

2001.Mar, 2001.Jan

…the resulting set whose members will be processed individually will be:

2001.Jan, 2001.Feb, 2001.Mar

***FOR… *NEXT structures**

The Logic Module supports any number of nesting of FOR…NEXT loops in the body of the logic files.

The syntax is:

```
*FOR  {variable1} = {set1} [ AND {variable2={set2}]
        {text}
        {text}
        …
*NEXT
```

For example, in the default translation logic we may need to repeat some calculation for each reporting currency. This works automatically if we write:

```
*FOR %CURR%=USD,EURO
        //Average Rate for currency %CURR%
        [measures].[!Avg_%CURR%] = {expression}
*NEXT
```

This could correspond to writing in the logic:

```
//Average Rate for currency USD
[measures].[!Avg_USD] = {expression}
//Average Rate for currency EURO
[measures].[!Avg_EURO] = {expression}
```

*FOR/NEXT loops where the set of variables is empty will be skipped without generating any error message.

*FOR/NEXT loops support up to two variables iterating on two independent sets of members.

Example:

```
*FOR %ACC1%=ThisA,ThisB,ThisC AND %ACC2%= ThatA, ThatB, ThatC
        [ACCOUNT].[#%ACC1%] = ([ACCOUNT].[ %ACC2%],[TIME].currentmember.lag(1))
*NEXT
```

The correct number of members is driven by the set of the first variable. If the first variable has less values than the second variable, the extra values of the second variable will be ignored. If the first variable has more values than the second variable, the missing values of the second variable will be assumed null.

Remark: this is not a nested loop. It's just one loop on two sets of variables.

Note that FOR/NEXT loops executed on empty lists of elements will not cause the logic execution to fail.

Nested FOR / NEXT loops

The for/next structure supports any level of nesting. Following is an example of a valid syntax:

```
*WHEN TIME
*IS <>TOT.INP
   *WHEN ACCOUNT
      *IS PERCENT.ALLOC
         *FOR %YEAR%=2003,2004,2005
            *FOR %MONTH%=JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
               *REC(FACTOR=GET(ACCOUNT="TOT.OVRHEAD",TIME="TOT.INP")/100,TIME="%YEAR%.%MONTH%")
            *NEXT
      *NEXT
   *ENDWHEN
*ENDWHEN
```

Note that, in case of a single-level loop, if the set of elements is empty the logic will still validate and execute correctly. However, in case of nested for / next loops, none of the loops can contain an empty set of elements, otherwise the logic will not validate.

Nested loops, similarly to single-level loops, can handle up to two sets of "parallel" variables, with the syntax:

```
*FOR %VariableOne%=FIRSTSET AND %VariableTwo%=SECONDSET
```

(Note that this is NOT a nested loop; it's just one loop handling two variables)

For example, these are two nested loops both using "parallel" variables:

```
*FOR %X%=1,2,3 AND %Y%=A,B,C
  *FOR %M%=4,5 AND %N%=E,F
    //…
  *NEXT
*NEXT
```

Special use of run-time FOR/NEXT structures

A logic validated at run time could contain a *FOR / NEXT loop that is based on the set of members passed for a given dimension. For example the user could write:

*FOR %MYTIME% = %TIME_SET%

    // logic content

*NEXT

If the dimension is the TIME dimension, the user might require the passed periods to be ordered in the correct sequence (from the earliest to the latest). This can be obtained inserting the keyword **ORDER_TIME** after the *FOR instruction, as follows:

*FOR ORDER_TIME %MYTIME% = %TIME_SET%

Remarks: It is imperative that the set of items of the FOR loop represent members of the TIME dimension, otherwise an error message will be returned.

The ordered set of members will also automatically fill the gaps in the time sequence. For example, if the members to order are:

2005.JUN, 2005.FEB

…the ordered set will be:

2005.FEB, 2005.MAR, 2005.APR, 2005.MAY, 2005.JUN

Forcing the selection to contain a specific range of members

While the Logic Module automatically builds, for each dimension, the set of members that must be included in the logic query, this set can be also controlled by the logic itself, with the command:

*XDIM_MEMBERSET {Dimension name} = {Members Set}

For example, in the exception translation logic we can enforce the query to generate results for all reporting currencies with the instruction:

*XDIM_MEMBERSET CURRENCY=USD,EURO

The instruction * XDIM_MEMBERSET supports also the “not equal to” operator with the syntax:

    * XDIM_MEMBERSET {Dimension}<>{MemberSet}

This operator is only supported for SQL logic (in MDX logic this feature is not quite needed, as the MDX syntax allows to build any sort of sets), and can be handy to pass to the SQL query smaller lists of valid members, that will be more efficiently parsed by Microsoft SQL engine.

Example:

*XDIM_MEMBERSET INTCO<>NonInterco

This corresponds to passing the list of all intercompany members, excluding the NonInterco member.

Forcing a dimension to read all members

When the user wants to make sure that all members of a given dimension would be loaded, irrespective of what specified in the passed region, he usually writes something as follows:

```
// a workaround
*XDIM_MEMBERSET INTCO<>INVALID
```

Today the logic engine supports a cleaner definition, through the keyword <ALL>. The above example can be written as follows:

```
// a better syntax
*XDIM_MEMBERSET INTCO = <ALL>
```

This will improve the readability of the logic and will also generate faster SQL queries.

Redefining the default member set of a dimension

When no member set is passed to a dimension, the logic module automatically queries all non-calculated members of such dimension (with the exception of the currency dimension, if existing, which defaults to the LC member). The default filtering criteria for a dimension can be redefined using the command:

*XDIM_DEFAULT {Dimension name} = {Members Set}

The difference with XDIM_MEMBERSET is that XDIM_DEFAULT will only be used if no selection is passed for the specified dimension, while XDIM_MEMBERSET will be used in any case. In practical terms this only applies to logics called by a TDS package, because WebExcel always passes the selection for all dimensions (except the accounts).

Filtering the passed member set of a dimension

The member set used for a given dimension can be filtered using some user-defined criteria with the instruction:

*XDIM_FILTER {Dimension name} = {Members Set}

This instruction does not replace the passed set with a hard-coded set, but filters the passed set using some predefined criteria.

For example, if the members passed for the ACCOUNT dimension are Cash, Receivables and Payables, this instruction…:

*XDIM_FILTER ACCOUNT = [account].properties(“ACCTYPE”)=”AST”

…will accept only asset accounts, so that the resulting account set will be limited to Cash and Receivables

Remark 1: if no member is passed, the filter criteria will not apply to only the non-calculated members (the default members set for all dimensions except currency), but to all members in the dimension. This could also be used as a way to modify the default filtering criteria.

Remark 2: the instruction automatically removes duplicates from the filtered set. This could be helpful when a returned member set contains duplicates, a situation that can easily be encountered in the entity dimension where the double hierarchy returns twice the same entity.

Remark 3: a typical use of this feature is to filter a list of members against one or more properties. When the members must be filtered against some values in the cube (like: take only the entities that have a value<:> 0 in a given account), the instruction to use is *XDIM_GETMEMBERSET.

Adding members to the passed member set

With the keyword XDIM_ADDMEMBERSET the logic can merge a specific set of members with the members passed in the region for which the logic should be executed.

This instruction is similar to the instruction *XDIM_MEMBERSET. The difference is that, while XDIM_MEMBERSET redefines the region passed by the user, XDIM_ADDMEMBERSET adds the defined set to the passed region.

The syntax is:

*XDIM_ADDMEMBERSET {dimension} = {members set}

For example, if a user enters a value in entity SalesItaly, and the default logic says:

*XDIM_ADDMEMBERSET ENTITY=SalesFrance

…the logic will be executed for SalesItaly AND SalesFrance.

On the other hand, if the logic simply said:

*XDIM_MEMBERSET ENTITY=SalesFrance

…the logic would have been executed for SalesFrance only, irrespective of the entity where the data had been entered.

Limiting the maximum number of members per query

In most cases, the fastest results are obtained running the logic as one single query. However, if the number of members in a dimension is too big, the performance can deteriorate significantly. In this case it may be preferable to break the execution in multiple queries. This can be accomplished using the following instruction in the logic:

*XDIM_MAXMEMBERS {dimension}= {max number of members}

Example: *XDIM_MAXMEMBERS Entity = 50

This instruction, in case the entities to process exceed the limit of 50 members, will break the query into multiple queries of no more than 50 entities each.

The selected dimension can be any one, including a dimension explicitly mentioned in the passed region.

The default for this option is, in MDX logic, one entity per query. In other words, if the user does not specify any value for this option, the logic module will automatically generate one separate MDX query for each entity being processed.

To reset the default value of maxmembers to ANY number of entities in MDX logics, the user can set the instruction to zero, as follows:

```
*XDIM_MAXMEMBERS Entity = 0          // unlimited number of entities
```

Important: the maximum number of members can be specified for up to TWO dimensions, like in these examples:

```
*XDIM_MAXMEMBERS Entity = 50
*XDIM_MAXMEMBERS Time = 1
```

Or..

```
*XDIM_MAXMEMBERS Entity = 50, PRODUCT=100
```

In SQL-type logics, the default value for the MAXMEMBERS option is not set. In other words the SQL query engine by default will try do read in one lump all the selected members of all dimensions. This default has been chosen because in the majority of cases there is no need to restrict the size of the query and the engine will run pretty fast with just one big query. If this is not the case, the option can be set similarly to what above described for the MDX logic.

Another difference in the behavior of the XDIM_MAXMEMBERS option is SQL logic is that the process will commit the generated result as they are generated by each single query, while in MDX logic all results are posted at the end of the entire execution.

Limiting the maximum number of committed members

When the instruction XDIM_MAXMEMBERS is used in logics of type MDX, the logic query is broken in as many queries as required to accommodate all members to process. However, all resulting records are committed to the database in one lump at the end of the loop of queries, and not after of each query. There may be cases where this is not desired (for example for memory limitations), and it may be preferable to perform a commit to the database after each individual query.

This can be accomplished inserting the instruction:

```
*COMMIT_MAXMEMBERS
```

…anywhere inside the commit section using the XDIM_MAXMEMBERS instruction.

Dynamic set of members

The special instructions:

*SELECT ({variable}, {What}, {From}, {Where})

…and…

*MEMBERSET({variable}, {member set in MDX format})

..allow the user to retrieve a list of elements from a dimension and save it in a user-defined variable for later use anywhere else in the logic.

For example, with this instruction:

```
*SELECT(%REPORTING_CURRENCIES%, "ID", "CURRENCY", "[GROUP] = 'REP'")
```

…the user can retrieve the ID of all members in the CURRENCY dimension where the property GROUP has the value REP.

By so doing, the user will fill the variable %REPORTING_CURRENCIES% with the list of reporting currencies defined in the current application.

The same selection could be obtained using an MDX statement with the MEMBERSET instruction as follows:

```
*MEMBERSET((%REPORTING_CURRENCIES%, "filter{[CURRENCY].members,
        [currency].properties("GROUP")="REP"")
```

The content of the resulting variable can then be used anywhere in the logic, like in this example:

```
*XDIM_MEMBER_SET CURRENCY=%REPORTING_CURRENCIES%
```

Important remarks:

1) The SELECT statement generates a SQL query, and NOT an MDX query. This implies that it can be executed against any SQL table existing in the appset database, and not just against the properties of a dimension in the cube. If the table name is the name of a valid dimension in the current app, the prefix "mbr" will automatically be added, otherwise, the name of the table will be taken as is.

2) The MEMBERSET statement generates an MDX query, and can be preferred to the SELECT statement to perform complex hierarchical selections or in general whenever an MDX query is more appropriate than a SQL one.

3) In case any parameter contains embedded commas (like in the above MEMBERSET example), the entire parameter must be enclosed in an extra set of double quotes.

4) These statements are executed at the time the logic is validated, and the expanded result is written in the LGX file. This means that if the related dimension is modified, it may be necessary to re-validate the logic, or (better) make sure the logic is re-validated at run time.

5) Statements returning no members will not cause the validation of the logic to fail. A warning will anyway be entered in the validation log.

6) These instructions are not specific to a given logic section, but they can be written once anywhere in the logic and used across multiple commit sections. The following example will work correctly.

```
//example ---------------------------------------------------------------------------
*SELECT(%INCACC%, "[ID]", "ACCOUNT", "ACCTYPE='INC'")

*XDIM_MEMBERSET ACCOUNT=%INCACC%
[category].[#realistic]=[category].[actual] *1. 2

*COMMIT

*XDIM_MEMBERSET ACCOUNT=%INCACC%
[category].[#optimistic]=[category].[actual] *1. 3
//end of example ---------------------------------------------------------------------
```

Note also that, since the select and memberset statements are expanded first, they can be placed anywhere in the logic. The above example would work even if the select statement was the LAST line in the logic, like this:

```
//example ---------------------------------------------------------------------------
*XDIM_MEMBERSET ACCOUNT=%EXPACC%
[category].[#realistic]=[category].[actual] * 1.2

*COMMIT

*XDIM_MEMBERSET ACCOUNT=%EXPACC%
[category].[#optimistic]=[category].[actual] *1. 3

*SELECT(%EXPACC%, "[ID]", "ACCOUNT", "ACCTYPE='INC'")
//end of example ---------------------------------------------------------------------
```

In addition the following example of logic will validate correctly, even if the same keyword %MYSET% is used in two different SELECT statements (in this case the substitution will be position-sensitive):

```
*SELECT(%MYSET%,ID,ENTITY,[CURRENCY]='GBP')
*XDIM_MEMBERSET ENTITY=%MYSET%
*SELECT(%MYSET%,ID,CATEGORY,[COMPARISON]='BUDGET')
*XDIM_MEMBERSET CATEGORY=%MYSET%
```

The validated logic might become:

```
*XDIM_MEMBERSET ENTITY=SalesUK,MfgUK,ResDevUK
*XDIM_MEMBERSET CATEGORY=ACTUAL,ACTBUD
```

Note also that cascading substitutions of variables will work correctly, as per the following example:

```
*SELECT(%VAR1%,.....)
*SELECT(%VAR2%,...., "[SOMEFIELD]='%VAR1%'")
```

Execution on empty lists of members

SELECT( ) instructions and XDIM_GETMEMBERSET( ) instructions returning an empty list of members will not cause the logic execution to fail. The log file will contain a warning that no members where found, but the logic will complete the execution successfully.

The special format **SELECT( )

The instruction SELECT can also be invoked with a double leading asterisk. This will trigger the execution of the instruction before any expansion is performed on the logic file (like function substitutions or file inclusions). This feature can be useful in case the result of the SELECT instruction is needed to control other tasks like INCLUDE or RUNLOGIC, which could be dependant on the result of the SELECT itself.

Example:

```
//-------------------------------------------------------------------------------------------
**SELECT(%MYFILE%, [filename], MyTable, "[CATEGORY]='%CATEGORY_SET%' AND [ENTITY]=%ENTITY_SET%'")
*INCLUDE %MYFILE%.LGF
//-------------------------------------------------------------------------------------------
```

In the above example an included logic file is derived from an entry in a special table using the passed category and entity as key. Note that in this case the keywords %CATEGORY_SET% or %ENTITY_SET% must contain only one member each.

Remarks: a logic containing a **SELECT instruction can only be executed in LGF format. Such logic might also refuse to validate, if the result of the instruction is a function of the region passed at runtime, like in the above example.

***XDIM_GETMEMBERSET….*ENDXDIM structure**

This instruction filters the members of the selected region for a given dimension according to their compliance with some user-defined criteria that must be met by the values in the cube.

The syntax is:

```
*XDIM_GETMEMBERSET {dimension} [={member set}]
        [*APP={application}]                                    //optional
        [*XDIM_MEMBERSET {dimension} [={member set}]            //as many of these as needed
        [*QUERY_TYPE= 0 | 1 | 2]                                //optional
        *CRITERIA {expression}                                  //required
*ENDXDIM
```

The portions between [square brackets] are optional.

Example:

```
*XDIM_GETMEMBERSET ENTITY=[ENTITY].[PARENT1].CHILDREN
        *APP=OWNERSHIP
        *XDIM_MEMBERSET INTCO=I_NONE
        *CRITERIA [ACCOUNTOWN].[METHOD]>1
*ENDXDIM
```

This will filter all children of entity PARENT1 to only those that in application OWNERSHIP have a value greater than 1 in account METHOD, intercompany I_NONE.

For all the dimensions not specified in the instruction, the search will be performed in the corresponding members of the selected region. For example, the category and period will be those for which the logic was being executed.

Hints:

This instruction can be very useful to restrict the scope of a logic execution where the user has selected a large region of data to process. For example, a currency conversion selected for all entities could be automatically limited to only those entities that have a value in a specified account.

Another use of this feature could be to break a complex modeling logic into independent sections separated by multiple COMMIT instructions, and to only execute those for which some specific accounts have changed.

Another possibility is to trigger a different type of logic based on the value of an account. For example a FORECAST logic could only be executed for just those months where [account].[One_If_Forecast]=1.

Special case: Change of dimension name across applications

The instruction *XDIM_GETMEMBERSET can redefine the name of the dimension being filtered, if, when querying a different cube, such dimension has a different name than in the source application.

The syntax supports an optional "AS" statement as follows:

*XDIM_ GETMEMBERSET {ThatDimension} [AS {ThisDimension}] [={member set}]

Example:

```
*XDIM_GETMEMBERSET SOMEENTITY as ENTITY=[SOMEENTITY].members
        *APP SOMEOWN
        *XDIM_MEMBERSET RPTCURRENCY=GROUP1
        *XDIM_MEMBERSET INTCO=Non_Interco
        *CRITERIA [ACCOUNTOWN].[METHOD]<>0
*ENDXDIM
```

In the above example the members of the ENTITY dimension are extracted from the members of the SOMEENTITY dimension existing in the SOMEOWN cube.

***XDIM_MEMBER**

This instruction is similar to the *XDIM_MEMBERSET instruction, but, while it only supports ONE member to be passed for the specified dimension, it permits to specify a different destination member into which the results of the logic execution must be written.

The syntax is:

*XDIM_MEMBER {dimension}={member} [TO {member}]

Example:

*XDIM_MEMBER CATEGORY=ACTUAL TO BUDGET

The above statement will force the logic to be executed reading the desired values from the ACTUAL category, but will write the results into the BUDGET category.

Multiple DIM_MEMBER instructions can be entered in the same logic, like in this example:

```
*DIM_MEMBER DATASRC=INPUT TO ELIM
*DIM_MEMBER PARENTDIM=NONE TO GROUP1
```

Hints:

This feature can be used to simplify some logic expressions. For example this logic….

```
#ACC1 =  ([ACCOUNT].[X],[CATEGORY].[BUDGET])
#ACC2 =  ([ACCOUNT].[Y],[CATEGORY].[BUDGET])
#ACC3 =  ([ACCOUNT].[Z],[CATEGORY].[BUDGET])
```

…could become:

```
*DIM_MEMBER CATEGORY=ACTUAL TO BUDGET
#ACC1 =  [ACCOUNT].[X]
#ACC2 =  [ACCOUNT].[Y]
#ACC3 =  [ACCOUNT].[Z]
```

This instruction can also be useful when the QUERY_TYPE=2 is used. In such case, if multiple dimensions must be nested on rows, a NonEmptyCrossJoin query could result, and it may be important to run the query directly from the data region containing the source values, in order not to miss some of them in the calculation (see the MDX language documentation for an explanation of the NonEmptyCrossJoin function).

The keyword %SET%

The keyword %SET% can be used in these instructions:

*XDIM_MEMBER
*XDIM_MEMBERSET
*DIM of a *LOOKUP instruction

The keyword %SET% can be used to indicate the set of members passed by the user for the specified dimension. This feature is helpful to redefine the set of members to process on the basis of the originally selected members.

For example the instruction:

*XDIM_MEMBERSET ENTITY = %SET% , SPECIAL_ENTITY

..could be used to add the entity SPECIAL_ENTITY to the list of passed entities.

When the passed set of members contains only ONE element, the %SET% keyword can be used in a *XDIM_MEMBER instruction to redirect either the source or the destination region for a given dimension starting from the passed member. Here are some examples:

```
// read from current period and write in prior period:

*XDIM_MEMBER TIME=%SET% to [TIME].[%SET%].lag(1)

// read from December of last year and write in current period:

*XDIM_MEMBER TIME=[time].[%SET%].parent.parent.lag(1).lastchild.lastchild TO %SET%

// read from the category specified in the COMPARISON property of curent category and write in current category:

*XDIM_MEMBER CATEGORY=strtomember([category].[%SET%].properties("COMP")) to %SET%
```

Finally the %SET% instruction can be used to redirect the region of a LOOKUP instruction like in the following example (see SQL logic below):

```
//-------------------------------------------------------------------------------------------------------------------
// this sample reads the values from prior year closing period,
// but the LOOKUP values it uses are read from the
// passed period (remark: the passed region must only contain ONE period)

*XDIM_MEMBER TIME=[time].[%SET%].parent.parent.lag(1).lastchild.lastchild TO %SET%

*LOOKUP RATE
        *DIM ENTITY2="DEFAULT"
        *DIM TIME="%SET%"
        *DIM INPUTCURRENCY=ENTITY.CURR
        *DIM RATE=ACCOUNT.RATETYPE
*ENDLOOKUP

*WHEN *
*IS *
        *REC(EXPRESSION=LOOKUP)
*ENDWHEN
//-------------------------------------------------------------------------------------------------------------------
```

The Keyword %{DimName}_SET%

An implied keyword is available for each dimension. This keyword will hold the set of members passed to the logic engine for a give dimension. This keyword can be used as a replacement string to use anywhere in the logic. The syntax is:

%{DimName}_SET%

…Where {DimName} is the name of a valid dimension in the application. For example the keyword %INTCO_SET% will contain the set of members passed to the logic for the dimension INTCO.

This keyword is very similar to the %SET% keyword, but has the advantage that it can be used anywhere in the logic and not just within some specific statement like XDIM_MEMBERSET.

This keyword is not modified by the XDIM_MEMBERSET instruction, as it always returns the original set passed to the logic.

Also, this keyword will not return a default set, if no set is passed. Its default is an empty set.

The Keyword %{DimName}_SQLSET%

The keyword %{dimension}_SET% (like %ENTITY_SET% or %ACCOUNT_SET%) has a “quoted” counterpart that can be used in SQL statements requiring the use of a list of members enclosed between single quotes. Its syntax is:

%{dimension}_SQLSET%

Example:

%ENTITY_SQLSET%

For example, if the selected Entities are SalesItaly, SalesFrance and SalesUK, the two keywords will contain the following strings:

%ENTITY_SET%: SalesItaly, SalesFrance, SalesUK
%ENTITY_SQLSET%: 'SalesItaly', 'SalesFrance', 'SalesUK'

Here is an example of how to use it:

*SELECT(%MYLIST%,[ID], INTCO, "[ENTITY] in (%ENTITY_SQLSET%)")

User-defined functions

A user-defined function, in this context, is the name of a placeholder that the user can decide to insert in his formulas in place of a corresponding MDX statements (or part of it). This can greatly improve the readability of a logic statement.

The definitions of the logic functions can be inserted anywhere in a logic file or in an included file. Their syntax is:

(For single line functions)

*FUNCTION {functionname}({Param1}[,{Param2}…]) = {Function Text}

(For multi-line functions)

*FUNCTION {functionname}({Param1}[,{Param2}…])
    {Function text}
    {Function text}
*ENDFUNCTION

An unlimited number of functions can be defined.

An unlimited number of parameters can be passed to a function in order dynamically modify the corresponding MDX string.

Functions can be nested (a function can invoke another function), and any level of nesting is supported.

The position of the functions in the logic file is irrelevant.

If multiple instances of the same function are entered in the logic file, the FIRST occurrence will prevail on the subsequent ones. This behavior can be used to redefine a function using the "add formula" text box in the TDSRunLogic task.

The values of the passed parameters are replaced in the function text without any validation, even if they are embedded in longer words, like in this example:

```
*FUNCTION TEST(Param1,Param2)
        AParam1Param2D
*ENDFUNCTION
```

The following logic line, calling the above function as follows…

[#123]=[TEST(B,C)]

…will expand into:

[#123]=[ABCD]

For the above reason some caution should be used in defining the names of the parameters, to avoid the risk of conflicts with MDX reserved words and in general with the text surrounding them in logic. One good practice could be to always surround the name of the parameters with some delimiter, like in this example:

*FUNCTION GET_PROPERTY(%DIMNAME% , %PROPERTYNAME%) = ….

Some characters are invalid in logic functions names. These characters are all those listed here below, plus the blank character**:**

+ - / * ^ % > < = ( ) [ ] { } , . ; ' : & \ | # ~ "

Invalid characters are trapped during logic validation.

The Instruction *SUB( ) / *ENDSUB

A SUB structure allows the user to define reusable logic sections that can be invoked anywhere in the body of the logic to make the logic easier to read and maintain.

A *SUB( ) structure is declared like a multi-line *FUNCTION( ) structure, with the following syntax:

```
*SUB {SubName}({Param1}[,{Param2}…])
        {body text}
        {body text}
        [….]
*ENDSUB
```

When a SUB is then invoked somewhere else in the logic, its body lines will be inserted in the logic with all the values passed to its parameters appropriately replaced.

SUBs behave similarly to included files to which any number of parameters can be passed. When the logic is validated, the invoked subs are inserted in the body of the logic as if they were included files invoked with an *INCLUDE instruction. However, to invoke a SUB structure, no special keyword is required. A SUB is simply called inserting a line with the name of the SUB, followed by the values assigned to its parameter enclosed in brackets. The other important difference from included files is that a SUB does not need to be written in a file of its own, but can be written in any part of the logic, more similarly to a FUNCTION.

Example:

```
// Here the sub is defined
//---------------------------------
*SUB MYSUB(Param1,Param2,Param3,Param4)
        [AccountDim].[#Param1]= [AccountDim].[Param2]+ [AccountDim].[Param3]
```

```
        [AccountDim].[#Param4]= [AccountDim].[#Param1]* [AccountDim].[Factor_Param4]
*ENDSUB

// Here the sub is used
//---------------------------------
MySub(A1,B1,C1,D1)
MySub(A2,B2,C2,D2)
MySub(A3,B3,C3,D3)
```

Remarks:

Similarly to FUNCTIONs, SUBs are not position sensitive, and can be defined anywhere in a logic, as well as, if so desired, stored in separate library files that must then be merged with the logic using an INCLUDE instruction. Also, SUBs can be invoked in any commit section of the logic without the need to be "re-defined" in each section.

SUBs without parameters are supported, but they must always be followed by brackets when invoked, like in this example:

```
// Here the sub is defined
//---------------------------------
*SUB SetTheAppropriateRegionToClear
        *CLEAR_DESTINATION
        *XDIM_MEMBERSET INTCO=NonIntco
        *XDIM_MEMBERSET PRODUCT=NoProduct
        *XDIM_CURRENCY=%REPORTING_CURRENCIES%
*ENDSUB

// Here the sub is used
//---------------------------------
SetTheAppropriateRegionToClear( )
```

Controlling the scope of the query

As above said, when the region of data is built reading an existing data file, or derived by the values entered from an Excel sheet, the logic module by default restricts its size to only the members found in such data for all dimensions except the account dimension.

The default set of dimensions to use for building the final region can be modified with the logic instruction:

*SCOPE_BY = {dimensions list)

…where the dimension list is any combination of dimensions.

Using this instruction can be extremely useful to expand the scope of a logic execution, when data are entered via Excel or via an import file. If, for example, the formulas to execute should span across all products, irrespective of what products have been modified, it is possible to expand the scope of the execution to all products by redefining the "scope_by" with a list of dimensions that does not include the product. In this case the instruction could be:

*SCOPE_BY = CATEGORY, TIME, ENTITY

(To be precise, the logic will be run not exactly for all products but for all *non-calculated* products).

Remark: in many cases it may be more practical to modify the scope of the logic execution using the instruction *XDIM_MEMBERSET. This instruction was introduced for different purposes but it allows the administrator to exactly specify what members to process for a given dimension, in effect overriding the default scope with greater precision.

Using one or multiple logics

The logic module can select the logic to execute according to three different “logic modes”:

Mode 0: run the default logic DEFAULT.LGX

Mode 1: override the default logic, and use a different one.

Mode 2: override the default logic, and use one or more logics, identified using some special criteria.

By default, the logic module runs in mode 0 (use the default logic), but the mode could be overridden with the logic instruction:

*LOGIC_MODE = 0 | 1 | 2

In reality this instruction makes only sense in the form: *LOGIC_MODE = 2, as the former two settings would be useless inside a logic file, because the values 0 or 1 can nly be controlled by the UI of the EvDTSRunLogic task.

The instruction:

*LOGIC_MODE = 2

… will tell the program NOT to run the default logic or any other given logic, but to run one or more logics, to be identified according to some specific criteria, as below described.

Remark: LOGIC_MODE is automatically set to 2 in case the instruction *LOGIC_BY is used.

Defining the criteria to identify the logics to execute

(Only valid for *LOGIC_MODE = 2)

In logic mode 2 (multiple logics), the program will build the names of the logics to execute reading the content of the user-defined property named “LOGIC” in all members being processed for the dimensions CATEGORY and ENTITY.

For example, if the category and entity being processed are ACTUAL and SALESITALY respectively, and these members have, as “LOGIC” property, the values ‘ACT’ and ‘1’ respectively, the logic being executed will be ACT1.LGX.

The dimensions to scan in order to build the name of the logic to execute are by default category and entity, but this default can be overridden with the instruction:

*LOGIC_BY = {dimensions list}

For example, the instruction could say:

*LOGIC_BY = TIME, DATASRC

This will mean that the logic will vary by time-period and by member of the DATASRC dimension.

Remark 1: the order in which the dimensions are written in the instructions will control the sequence in which the content of the 'LOGIC' properties will be concatenated (the module reads the dimension names from left to right). Any number of dimensions can be specified.

Remark 2: blank values for the 'LOGIC' property are acceptable, as long as at least one of the dimensions listed in the instruction has a non-blank value (otherwise the resulting logic name will be blank).

Remark 3: LOGIC_MODE is automatically set to 2 in case the instruction *LOGIC_BY is used.

Remark 4: The logic engine does not set any limit to the number of dimensions by which the logic may vary. However, it is strongly recommended NOT to use more than two dimensions for this purpose, as this could lead to:

- A very high number of logic files to define and maintain
- An excessive fragmentation of the logic execution in many small queries, resulting in unsatisfactory performance

Overriding the name of the property used to identify the logics to execute

A separate instruction can be used to override the default name of the property driving the selection of the logic to use, when the logic is set to vary by the members of one (or more) dimension(s). The instruction is:

*LOGIC_PROPERTY = {property name}

This feature can be useful when the default property name "LOGIC" is already taken by another set of logics. For example the DEFAULT logic could vary by category using the default property LOGIC while a consolidation logic could vary by category using the property CONSOL_LOGIC.

Example:

```
// Content of  DEFAULT.LGF
//-------------------------------------------------------
*LOGIC_BY = CATEGORY

// Content of  CONSOL_LOGIC.LGF
//-------------------------------------------------------
*LOGIC_BY = CATEGORY
*LOGIC_PROPERTY = CONSOL_LOGIC
```

Content of CATEGORY.XLS:

ID	LOGIC	CONSOL_LOGIC
ACTUAL	Default1	ConsLogic1
BUDGET	Default2	ConsLogic2

Write back by difference

When posting values, Everest needs to write in the database just the difference between what is already stored and the new value. The calculation of the difference takes time. If such calculation is performed directly by the Logic Module at the time of executing the logic, it is possible that the subsequent posting time will be reduced. This is what the logic module does by default. To override the default, the following instruction can be entered in a logic file:

*CALCULATE_DIFFERENCE = 0 (default being 1)

When the calculation of the difference is turned off, the write engine is instructed of the change in behavior and takes over the job of calculating (and posting) the difference, so that the final values will be correct anyway. The main reason for using this instruction is that the debug file will show the values exactly as the user expects them to look like in the cube.

***WRITE_TO_FILE = {filename}**

This instruction will write into the specified file a copy of all records to be posted, in a textual, comma-delimited format.

This setting will automatically suppress the writing of the records to the log file, and the results will NOT be written to the cube, even if the simulation mode is left off.

If no path is included in the filename, the DataManager data files path will be used. If no extension is included, a TXT extension will be appended to the file name.

Important remark: this instruction is only supported in “overwrite formula” mode (*logic_mode=1), and can only be entered in the “add formula” text box of the DTS logic task. If written in a logic file directly, it will be ignored.

Writing the results to a different application

With the instruction…:

*DESTINATION_APP = {app name}

…it is possible to redirect the results of a logic execution to be written into an application different from the one against which it was originally executed. For example, when some data are entered into a divisional application, some of these data may need to be also posted into a central application that consolidates the results of different divisional applications.

Example:

*DESTINATION_APP = CentralApplication

It may easily happen that the destination application shares only some of the dimensions of the original application. In this case the missing dimensions can be dropped from the original records with the instruction:

*SKIP_DIM= {dimension name}[,{dimension name},…]

Multiple dimension names can be supplied to the instruction separated by commas, or multiple SKIP_DIM instructions can be entered in separate lines.

If the destination application has dimensions that do not exist in the original application, these can be added to the passed records, using the instruction:

*ADD_DIM {dimension name}={value}[,{dimension name}={value},…]

Multiple dimension names and values can be supplied to the instruction separated by commas, or multiple ADD_DIM instructions can be entered on separate lines.

In addition to the instructions ADD_DIM and SKIP_DIM, the keyword RENAME_DIM can be used, to change name of one or more dimensions. The syntax is:

*RENAME_DIM {dimension name}={value}[,{dimension name}={value},…]

This instruction can be used when data are to be written into an application where a dimension is named with a different ID.

Multiple dimension names and values can be supplied to the instruction separated by commas, or multiple RENAME_DIM instructions can be entered on separate lines.

Example:

*RENAME_DIM ACCOUNT_FLASH= ACCOUNT_MAIN

Here is a more complete example:

```
*DESTINATION_APP = CentralApplication
*SKIP_DIM= PRODUCT,MARKET
*ADD_DIM DATASRC=INPUT
*ADD_DIM CURRENCY=LC
*RENAME_DIM ACCOUNTPM=ACCOUNTMAIN
```

In this example some calculated values are transferred into a central application that is not detailed by product and market, but contains two extra dimensions (datasrc and currency). For these two dimensions the members input and lc will be used. Also, the chart of account is defined in dimension accountmain.

Special 'push logic' instructions

A set of special instructions has been implemented, to allow the user to re-direct the execution of a logic file towards a different data region than the one for which the logic was originally executed.

These instructions can be used to redefine:

The AppSet
The App
The selection of dimension members
The logic to run

Possible uses:

- To enter data in one period and trigger an allocation across all periods of the year
- To enter data in one entity and trigger the elimination logic for the elim-entities
- To modify an exchange rate in the rate cube and trigger a translation in the main cube
- …..

The instructions must be written between a *RUNLOGIC and a *ENDRUNLOGIC command. Here is the full list of supported instructions:

```
*RUNLOGIC
        *APPSET= {AppSet}                       //optional
        *APP = {App}                            //optional
        *LOGIC = {logicname}                    //required
        *DIMENSION {dimname} = {member set}     //optional (one per dim allowed)
*ENDRUNLOGIC
```

All instructions are optional except *LOGIC which is required.

All logic properties that are not redefined with one of these instructions will retain the values they have in the calling logic (for example, if the application is not re-defined, the logic push will be performed against the original application).

The RUNLOGIC sections contained in a logic file will be executed at the END of the entire logic execution, IRRESPECTIVE of their position in the logic relative to all other statements.

Multiple RUNLOGIC sections can be entered in the same logic file. They will be executed in the order in which they are encountered.

RUNLOGIC sections contained in logics called by a RUNLOGIC statement will be ignored (In other words RUNLOGIC calls cannot be nested).

The instruction DIMENSION can be used to redefine the scope of execution in a given dimension. Multiple DIMENSION instructions can be used to redefine the member sets of multiple dimensions in the current data region.

Examples:

```
//execute eliminations in elim entities
//-------------------------------------------------------------------
*RUNLOGIC
        *DIMENSION ENTITY=filter([entity].members,[entity].property('ELIM')='Y')
        *LOGIC=Intco
*ENDRUNLOGIC

//execute currency translation in main app
//-------------------------------------------------------------------
*RUNLOGIC
        *APP=MAIN
        *DIMENSION ENTITY2=                     //blank out scope of invalid dims
        *LOGIC=DefaultTranslation
*ENDRUNLOGIC
```

Remark: For backwards compatibility, the instruction RUNLOGIC can still be written using the original syntax BEGINRUNLOGIC, which however, is not consistent with the overall syntax style, and is not recommended.

Special Keywords in the DIMENSION instruction

The strings passed to the instruction DIMENSION can contain the keyword:

%{dimname}%

For example, if the main logic is being run for entity EUROPE, the following logic push will be executed for all children of EUROPE:

```
*RUNLOGIC
        *DIMENSION ENTITY = [ENTITY].[%ENTITY%].children
        *LOGIC=SomeLogic
*ENDRUNLOGIC
```

Also, if the main logic was run for more than one entity (example: EUROPE and US), the logic push will be performed for the children of each of them, using the following selection in the logic query:

{[ENTITY].[EUROPE].children, [ENTITY].[US].children}

Remark: this works if the calling selection enumerates the members individually as follows:

DIMENSION:ENTITY
EUROPE, US

In case the calling selection contains an MDX statement, this will be treated as one member. The following example would NOT work in the above context

DIMENSION:ENTITY
[WORLD].children

Members sub-names

The keyword %DIMNAME% can be adjusted to return only the prefix or the suffix of a member name.

For example, if the passed member set for the TIME dimension contains the member:

2001.JAN

The keywords:	will return:
%TIME%	2001.JAN
%TIME_PREFIX%	2001
%TIME_SUFFIX%	JAN

Here is an example that uses the prefix keyword to perform an allocation:

```
*RUNLOGIC
        *DIMENSION TIME=descendants([time].[%TIME_PREFIX%.total],99,leaves)*LOGIC=AllocationLogic
*ENDRUNLOGIC
```

Remark: if the user enters data for more than one month for the SAME year, the logic push will be smart enough to process that year only once.

Redefining the Local Currency member

The ID of the local currency member of the currency dimension (typically called “LC” in most of our applications) can be named as desired, but the logic must be informed that its ID has been changed to something different. The instruction to use is:

*LOCAL_CURRENCY= {local currency member}

Example:

*LOCAL_CURRENCY= ValutaLocale // this is Italian!

Remarks:

This instruction must be used in all applications where the local currency member of the currency dimension is named differently from “LC”, even if the local currency member is not explicitly mentioned in the user’s logic formulas, because there are several cases where the logic engine creates queries that make automatic reference to it behind the scenes.

This instruction can be entered just once in the whole logic and does not need to be repeated in each commit section. For a cleaner design, it is recommended to include this instruction in the application constants library APP_CONSTANTS.LGL.

The instruction *USE_UNSIGNEDDATA

When dealing with SQL-based logic, it is sometimes confusing to correctly handle the sign of the values being read and of those to be written in the db. This is because by default the logic engine reads and writes the amounts using the SIGNEDDATA field, which is stored awith a reversed sign when it refers to accounts of type INC or LEQ.

To facilitate the job of the administrators, it is now possible to make the logic read and calculate the amounts as UNSIGNED DATA. Essentially, when this option is turned on, the amounts will be read and written as they appear in WebExcel, and the calculation formulas can be defined thinking about the values in a similar fashion.

The instruction that activates the option is:

*USE_UNSIGNEDDATA

This option is COMMIT-specific and is only available for SQL-based logics.

Warnings:

When this option is turned on the logic engine needs to perform two additional steps during execution: (1) convert all read amounts from signed to unsigned values and (2) convert back all records to be written from unsigned to signed values. These two extra steps have a slight impact on performance and for very heavy logic executions this may be perceivable. In such cases the use of this option could be inappropriate.

The USE_UNSIGNEDDATA option must NOT be used in conjunction with the instruction CALC_DUMMY_ORG (or CALC _ORG), in case the aggregations of the hierarchy are performed on members of the ACCOUNT dimension having different values of the ACCTYPE property.

For example, this logic would generate an INCORRECT RESULT for account #NetIncome:

```
// this logic cannot work
//------------------------------
*XDIM_MEMBERSET ACCOUNT=<ALL>
*USE_UNSIGNEDDATA
*CALC_DUMMY_ORG ACCOUNT=PARENTH1
```

The instruction *PUT( )

This very special instruction can be used to directly write values in a selected region. The syntax is:

*PUT({dimension}={member}[,{dimension}={member}] [EXPRESSION={expression}….)

Example:

*PUT(ACCOUNT=FLAG, INTCO=NONE, CURRENCY=LC, EXPRESSION=123)

This instruction will write the value 123 in account=flag, intco=none and currency=lc for all entities, categories and time periods passed in the selected region.

The instruction can be used to flag a region of data with a value indicating the status of the data. For example, whenever data are entered in a certain entity, the default logic could flag that entity as impacted. Later on, a currency conversion could be executed just for the impacted entities. At the end of its execution, the translation logic could de-impact these entities, by setting their flag account back to zero.

Important remarks:

The *parameters can only take explicit values* and cannot be derived from a dimension property (unless passing through a SELECT instruction)

The *scope* of the records to write is by default defined as all the members passed for the dimensions category, time and entity. This default scope can be modified with the instruction *PUTSCOPE_BY, which is similar to the SCOPE_BY instruction, but is specific to the PUT instruction.

Example:

*PUTSCOPE_BY=CATEGORY, TIME, DATASRC

The *destination member(s)* must be specified for all dimensions. The dimensions defined in the scope derive the correct members from the passed region. All other dimensions must have one member specified in the PUT parameters or by a XDIM_MEMBERSET or similar instruction.

Example:

```
*XDIM_MEMBERSET INTCO=NONE
*XDIM_MEMBERSET CURRENCY=LC
*PUT(ACCOUNT=FLAG, EXPRESSION=123)
```

The *EXPRESSION parameter is optional*. If omitted it will default to 1. This example will generate a value of 1 for account flag:

```
*PUT(ACCOUNT=FLAG)
```

Multiple PUT instructions can be specified in the same commit section. Example:

```
*XDIM_MEMBERSET INTCO=NONE
*XDIM_MEMBERSET ACCOUNT=FLAG

*PUT(CURRENCY=USD)
*PUT(CURRENCY=EURO)
```

The PUT instruction MUST be defined in a *STANDALONE COMMIT section*. It cannot be mixed with MDX formulas or with WHEN/ENDWHEN structures.

The PUT instruction always and automatically enforces a **calculate_difference*=0

The PUT instruction supports the ability to write into an application different from the source. The following is a valid example of logic that runs from a RATE application and writes in the FINANCE application.

```
*destinationapp=finance
*skipdim=inputcurrency
*skipdim=rate
*skipdim=entity2

*adddim intco=non_interco
*Adddim datasrc=%DATASRC% // this is needed if the datasrc is re-directed in the PUT
 *Adddim account=flag
*adddim currency=lc
*adddim entity=dummy

*PUT(datasrc=input)
*PUT(datasrc=adjustment)
```

Running stored procedures from logic

A logic can invoke the execution of an SQL stored procedure, using the instruction:

*RUN_STORED_PROCEDURE={stored procedure name}([{params}])

Example:

```
*RUN_STORED_PROCEDURE=spEliminate(ACTUAL,[2001.JAN])
```

Parameters containing delimiter characters (like 2001.JAN) may need to be passed enclosed between brackets (like [2001.JAN]).

To pass multiple values in a single parameter, it may be appropriate to enclose them between square brackets (or, if they contain a list of members exceeding 128 bytes, between single quotes). However, it will be up to the stored procedure code to support the un-packing of that parameter into its individual members, if appropriate. See this example:

```
*RUN_STORED_PROCEDURE=spCompare( [2001.JAN,2001.FEB],[ITALY,FRANCE])
```

Stored procedures must be written in their own commit section, i.e. they cannot coexist with a WHEN/ENDWHEN structure, nor can be part of any MDX logic. On the other hand, multiple stored procedures can be executed from the same commit section, like in the following example:

*RUN_STORED_PROCEDURE=FirstProcedure('%TIME_SET%')
*RUN_STORED_PROCEDURE=SecondProcedure('%TIME_SET%')

*COMMIT

[account].[#SALES]=Units * Price

*COMMIT

Remarks:

The instruction RUN_STORED_PROCEDURE supports the keyword %APP% as the current application name. This is a common requirement for stored procedures that need to work on different applications.

Support of a log table in a stored procedure

The parameters passed to a stored procedures can include the name of a log table that the stored procedure can fill with whatever information are appropriate. The name of the table is generated automatically by the logic engine and passed to the stored procedure, once the table has been succesfully created. This table will contain a single field named "MSG" that can be filled with any length of text (it is of type NTEXT)

Once the stored procedure has completed execution and has written whatever messages in the log table, the logic engine will read its content and merge it into the normal logic log file, then the table is automatically dropped.

To activate this feature the user must include in the list of parameters passed to the stored procedure the keyword %LOGTABLE%. The logic engine will replace it with the appropriate table name.

Example:

*RUN_STORED_PROCEDURE=spEliminate([%APP%],[ACTUAL], [2001.JAN], [%LOGTABLE%])

Support of blank parameters passed to a stored procedure

A stored procedure will fail if one of the required parameters is blank (null string). This situation may occur, for example, in case one of the parameters is a set of members that was left blank by the user in a DTS prompt, to indicate he wants to process ALL members.

Now the logic engine will automatically trap this situation by replacing any null parameter ([]) with the <NULL> keyword ([<NULL>]). The stored procedure will have to check for parameters with a <NULL> value and take the appropriate action.

Example:

This instruction:

*RUN_STORED_PROCEDURE=spEliminate([], [2001.JAN])

will be converted into:

*RUN_STORED_PROCEDURE=spEliminate([<NULL>], [2001.JAN])

…where <NULL> will have to be interpreted by the stored procedure as (for example) ALL categories.

Passing the selected region to a stored procedure using a table

The parameters passed to a stored procedure can include the name of a temporary table that the logic engine will use to store the members of the selected region for which it was invoked. The name of the table is generated automatically by the logic engine and passed to the stored procedure, once the table has been successfully created and populated with the appropriate information. This table will contain two fields named DIMENSION and MEMBER respectively, and will be populated by the logic engine with one record per each dimension/member combination that have been passed to it by the calling program.

Once the stored procedure has completed execution, the logic engine automatically drops the table.

To activate this feature the user must include in the list of parameters passed to the stored procedure the keyword %SCOPETABLE%. The logic engine will replace it with the appropriate table name.

Example:

*RUN_STORED_PROCEDURE=spEliminate([%SCOPETABLE%])

It is obviously up to the stored procedure to make good use of the information passed to it through this table.

Error handling for stored procedures

Logics running stored procedures are capable to interrupt their execution with an appropriate error message if so instructed by the stored procedure being run.

This feature is automatically activated whenever the stored procedure uses a LOG TABLE to return information to the calling logic (see details above).

What the logic engine will do is to check the content of the log table and look for the presence of the string "***ERROR***" (the word "ERROR" enclosed between asterisks). If such string is found, the logic will abort the execution and return to the user the error message as generated by the stored procedure itself.

The two types of modeling logic

The Logic Module can perform its calculation using two different techniques that we call MDX mode and SQL mode. The MDX mode basically relies on the functionalities of Microsoft MDX language to generate the queries that will return the desired results. The SQL mode issues SQL queries directly against the fact table, and will apply Everest-specific calculation functions to the obtained set of records. The choice of the appropriate mode is left to the user, and is very much a function of the type of calculations that need to be performed. As a rule of thumb, MDX logics are easier to use, more flexible and are more appropriate for complex models and simulations, but do not scale well and are much slower that SQL logics. SQL logics are way more efficient and suit well for currency conversions, allocations and intercompany eliminations, but are less intuitive and harder to use in complex financial models. Nevertheless, given their unquestionable performance advantage, SQL logics are now practically stealing the entire scene of modeling logic.

MDX-based modeling logic

In MDX logics, the Logic Module does not perform the calculations directly, but uses MDX queries that it passes to the Analysis Server to get from it the desired results automatically calculated. Consequently, the formulas that it uses must be written in a correct MDX syntax.

From this point of view, Modeling Logic work very much along the same principles of the Everest dimension formulas. The difference is that the formulas used by the Logic Module are written and executed "outside" the OLAP cube and are not part of the definitions of the cube dimensions. While this method of calculation is not as simple to define than the one used in Everest dimension members formulas, it allows the user to perform calculations with a much greater degree of flexibility. One remarkable side benefit of this approach is that, while Everest dimension formulas are associated with a dimension and, as such, must be valid for all cubes using that dimension, the Logic Module formulas can be different across cubes of the same AppSet

MDX Logic Syntax

As above said, the formulas contained in MDX-based logic files must – almost exactly - comply with the MDX syntax for calculated members. Here is an example:

```
[ACCOUNT].[#GROSSSALES] = -[ACCOUNT].[UNITS]*[ACCOUNT].[INPUTPRICE]
[ACCOUNT].[#COST] = -[ACCOUNT].[#GROSSSALES]*80/100
```

The structure…

{member} = {expression} [, solve_order = n]

…is the normal syntax for calculated members required by MDX queries. The only exceptions are: (1) the equal sign ("=") in place of the "AS" keyword, and: (2) the lack of single quotes around the {expression}.

Some other simpler formats are also supported. For example, the above formulas could be written as follows:

```
#GROSSSALES = -UNITS*INPUTPRICE
#COST = -[#GROSSSALES]*80/100
```

The calculated members must be preceded by a pound (#) sign everywhere in the expression. The pound sign is required at the left of the equal sign as well. If the pound sign is omitted to the right of the equal sign, the formula will refer to the stored value, like in this example, where the account COUNTER is increased by one every time the logic is executed:

```
#COUNTER=COUNTER +1
```

If used to the right side of the formula, the calculated members must be enclosed in square brackets ([#member]), like in this example:

```
#COST = -[#GROSSSALES]*80/100
```

If the dimension name is omitted in the left side of the formula, the account dimension is assumed.

If the dimension name is omitted in the right side of the formula, the dimension is automatically derived by the analysis server, whenever possible, according to the rules of the MDX syntax.

The generated query

As already said, the module inserts the logic formulas in an MDX query, to obtain the desired results. The query is generated automatically, based on:

- the formulas defining the calculated members
- the selection of data passed by the calling program
- the desired format of the query

Following is an example of a query generated by the Logic Module using the above example of logic, against the selection:

Category: ACTUAL
Time: 2000.JAN
Entity: [SalesEurope].children

For those who are interested, the query resulting from the example shown above will look more or less as follows:

```
with
member [ACCOUNT].[#GROSSSALES] AS '-[ACCOUNT].[UNITS]*[ACCOUNT].[INPUTPRICE]'
member [ACCOUNT].[#COST] AS '-[ACCOUNT].[#GROSSSALES]*80/100'
select
non empty {filter([PRODUCT].Members,[PRODUCT].Properties("CALC")="N")} on 0,
non empty {[ACCOUNT].[~GROSSSALES],[ACCOUNT].[~COST]} on 1,
non empty {[SALESEUROPE].CHILDREN} on 2,
non empty {filter([INTCO].Members,[INTCO].Properties("CALC")="N")} on 3
from main
where
([CATEGORY].[ACTUAL],[CURRENCY].[LC],[TIME].[2000.JAN],[MEASURES].[SIGNEDDATA])
```

In particular, the query:

- will put on the 0 axis the calculated members of the calculated dimensions
- will put on some other axis (if more than one member) or in the ‘where’ clause (if just one member) the members passed by the data selection
- will put in the ‘where’ clause the measure signeddata.
- Will put in some axis all non-calculated members of any other dimension (none in the example)

This query returns a set of records that will then be written in accounts GROSSSALES and COST for the specified region.

***ADD…*ENDADD structure**

This structure allows to automatically add to a calculated members a set of members as specified in a comma delimited range. The range can be dynamically derived using a *SELECT( ) instruction (see).

The syntax is:

```
*ADD  {variable} = {set}
        {formula}
*ENDADD
```

Example:

```
*ADD %ACC%=A,B,C,D
        #MYSUM = %ACC%
*ENDADD
```

This will expand into:

#MYSUM = A+B+C+D

Remarks:

The expression to the right of the '=' sign can be a more complex expression like in the following example:

#MYSUM = (-[ACCOUNT].[%ACC%])

Note that the above format enables the user to SUBTRACT the set of members from the calculated one.

Another possibility is to include a fixed part in the sum as follows:

#MYSUM = [ACCOUNT].[FIXED]+ [%ACC%]

The program will use the rightmost '+' sign as delimiter between the fixed portion and the portion to add. The above example will expand as follows:

#MYSUM=[ACCOUNT].[FIXED]+ [A]+ [B]+ [C]+ [D]

The structure supports multiple formulas to be expanded simultaneously, like in this example:

```
*ADD %ACC%=A,B,C,D
        #MYFIRSTSUM = %ACC%
        #MYSECONDSUM = %ACC%
*ENDADD
```

ADD/ENDADD loops executed on empty lists of elements will not cause the logic execution to fail.

Swapping the measures dimensions

With the instruction:

*MEASURES = {dimension}

….the user can tell the Logic Module that the results of the query in the measures dimension will indeed contain information from another dimension.

This functionality can be used whenever a logic calculates members using the measures dimension, for efficiency.

Multi-line MDX formulas

Formulas spanning multiple lines can be specified, but they must be enclosed between the two keywords *BEGIN and *END, like in the following example:

```
*BEGIN
        [ACCOUNT].[#GROSSSALES] =
        -[ACCOUNT].[UNITS]*
        [ACCOUNT].[INPUTPRICE]
*END
```

***USE {filename}**

The USE instruction behaves like the INCLUDE instruction, but only the members that are needed by the calling logic are included. This generates smaller, faster to execute LGX files.

Example of USE instruction:

```
‘----------------------------------------------------------------------------------------------
*USE TranslationMembersFile

[account].[#cost] = -([currency].[lc],[account].[grosssales])/[account].[!End]
[account].[#revenue] = -([currency].[lc],[account].[cost])/[account].[!Avg_End]
‘----------------------------------------------------------------------------------------------
```

the file TranslationMembersFile may contain the definitions of many accounts, but only those for [account].[!End] and [account].[!Avg_End] will be included in the calling logic.

Remarks:

The instruction *USE must be called again after each *COMMIT instruction, if the next section of logic also uses accounts defined in a USEd file. This rule does not apply to *INCLUDEd files.

Used files take the same default path and extension of the included files.

All special instructions (like *COMMIT and other instructions) are ignored, if found in a “USEd” file. We can say that when a logic file is “USEd” by another file, the only thing the calling file will be interested in will be the list of calculated members the “USEd” file contains. This restriction does not apply to “INCLUDEd” files.

All temporary accounts contained in USEd files should always be named with a leading exclamation mark, even if this is not technically required. They could also begin with a pound sign (#), but, in this case, the Logic Module would try to write their resulting values in the cube.

***SELECTCASE / *ENDSELECT structures**

Sometimes the user needs to write formulas containing several nested IIF( ) statements which, in normal MDX syntax, are typically hard to write and impossible to read and debug. To make this job easier, we have implemented the following syntax:

```
*SELECTCASE {expression}
        *CASE {value1}[,{value2},…]
                {formulas}
```

```
        [*CASE {value1}[,{value2},…]
                {formulas}
        [*CASEELSE]
                {formulas}
*ENDSELECT
```

...where:

{expression} is the condition to be evaluated
{value1},.. is the range of comma-delimited results that satisfy the condition for the current case

With such structure the readability of a logic can be significantly improved. For example, the following formulas…:

```
*BEGIN
#A = IIF([ACCOUNT].[E]=1 OR [ACCOUNT].[E]=2,X+Y,
          IIF([ACCOUNT].[E]=3 OR [ACCOUNT].[E]=4,X-Y,X*Y))
*END
#C = IIF([ACCOUNT].[E]=1 OR [ACCOUNT].[E]=2,W+Z,null)
#B = IIF([ACCOUNT].[E]=3 OR [ACCOUNT].[E]=4,W*Z,null)
```

…could be written as follows:

```
*SELECTCASE [ACCOUNT].[E]
   *CASE 1,2
        #A=X+Y
        #C=W+Z
  *CASE 3,4
        #A=X-Y
        #B=W*Z
   *CASEELSE
        #A=X*Y
*ENDSELECT
```

SELECTCASE structures can be nested for as many levels as desired. The following example nests one level of SELECTCASE inside another one:

```
*SELECTCASE [ACCOUNT].[E]
   *CASE 1,2
        *SELECTCASE B
                *CASE 10,20
                        #H=100
                  *CASE 13,14
                        #H=101
                        #K=202
        *ENDSELECT
  *CASE 3,4
        #K=302
*ENDSELECT
```

…this corresponds to the following formulas:

```
*BEGIN
#H = IIF([ACCOUNT].[E]=1 OR [ACCOUNT].[E]=2,
          IIF(B=10 OR B=20,100,IIF(B=13 OR B=14,101,null)),null)
*END
*BEGIN
#K = IIF([ACCOUNT].[E]=1 OR [ACCOUNT].[E]=2,
          IIF(B=13 OR B=14,202,null),IIF([ACCOUNT].[E]=3 OR [ACCOUNT].[E]=4,302,null))
*END
```

Reserved function EVCPN(n)

The logic module supports an implicit function named EVCPN that can be used in MDX formulas like any other user-defined function. EVCPN (Current Period Number) returns the number of the period being executed in the query. For example if you run a query for March and April, March will be assigned the number 1 and April the number 2.

This feature can be used to decide whether a calculated value should be retrieved from the cube or from memory (in this context "memory" means the result of the calculation performed by the query). In the above example, if you need to read a value from February, it can come from the cube (so that February does not get re-calculated) and, if you read it from March or April, you must read the calculated result you have in memory, because the value in the cube has not been updated yet.

Following is an example of how to use EVCPN

```
[ACCOUNT].[~CPN]=EVCPN                    // dummy CPN account

*FUNCTION TLAG(%ACC%,%LAG%)
   iif([ACCOUNT].[~CPN]<(%LAG%+1),
      ([ACCOUNT].[%ACC%], [time].currentmember.lag(%LAG%)),
      ([ACCOUNT].[#%ACC%],[time].currentmember.lag(%LAG%)))
*ENDFUNCTION

[ACCOUNT].[#Z] = TLAG(Z,1) + X – Y
```

The above example enables to read correctly the opening value of account Z without committing each period to the cube in the process.

EVCPN supports a numeric parameter that can be used to restrict the function from calculating the period number beyond a certain number of periods. This could be used to make the calculation of CPN run faster. For example, if you need to test only whether you are in the first period or not (like in the above case), the function does not need to try and assign a period number to all processed periods. It will be enough to assign the value of 2 to all periods beyond the first. To achieve this the logic can be written as follows:

[ACCOUNT].[~CPN]=EVCPN(1)

If you run the calculation from March, the function will expand at run time into something like:

[ACCOUNT].[~CPN]=IIF([TIME].currentmember.name="2001.MAR",1,2)

All periods other than March will have a period number greater than 1, which is all what the function call TLAG(Z,1) needs to know.

Format of MDX queries

While the Logic Module by default generates a multi-axis query, the format of the MDX query can be controlled by the logic using the instruction:

*QUERY_TYPE=0 | 1 | 2

…where:

0 is the default multi-axis type
1 is a row/column type, with multiple crossjoins in rows
2 is a row/column type, with one nonemptycrossjoin in rows

The query type 1 (row/column crossjoin) can be useful when the user wants to check the generated query using Microsoft MDX Samples program. This product, in fact, does not support multi-axis queries. A query type no.1 can simply be copy/pasted from the debug file into the MDX sample UI and executed.

While the type 2 (nonemptycrossjoin) is by far the fastest format, it will only give the desired results in selected cases, and it must be used with care. One example of its use can be found in the Default Translation Logic. This type of query can be controlled using also the instruction *QUERY_FILTER (see).

***QUERY_FILTER={member}**

This instruction enforces a query_type=2 (nonemptycrossjoin) in the MDX logic execution, and causes the search for existing values to be performed on the passed member.

Example:

*QUERY_FILTER=NetProfit

The program will assume the member to be an account, if no dimension is specified. Otherwise, the correct dimension can be explicitly named in the usual MDX format, for example, as follows:

*QUERY_FILTER=[CATEGORY].[ACTUAL]

Breaking the MDX query in multiple queries (when running in *LOGIC_MODE = 2)

When the module runs in multi-logic mode, it automatically tries to group the regions of data sharing the same logic to execute and will try to run the largest possible queries against a given logic, in order to maximize the speed of execution. In many cases, however, the resulting queries will be too large to fit in memory, and the performance of large queries will actually deteriorate significantly, instead of improving.

To prevent this situation, the administrator can use the instruction:

*QUERY_SIZE = 0 | 1 | 2

…where:

2 is the (default) largest size of queries
1 is an intermediate size, and
0 is the smallest practical size

It must be noted that the effects of this instruction will combine with those of the instruction XDIM_MAXMEMBERS, in defining the scope of the queries ultimately being run. The appropriate combination of these two setting should be identified, in order to obtain the best compromise between performance and memory usage.

Breaking the query in multiple queries (when scanning a data file)

When the logic module scans a data file to build a region against which to run, it may as well break the query in many smaller queries, according to the different combinations of members it finds in the file, for the dimension defining the scope (see the "scope_by" instruction). In this situation, similarly to the case of multiple logics (logic_mode = 2), the program can try to maximize the size of the query, according to the setting of the QUERY_SIZE parameter. The behavior of this parameter is the same described for logic_mode = 2.

Support for PERIODIC formulas

If the user uses the following setting for the MEASURES dimension:

*XDIM_MEMBERSET MEASURES=PERIODIC

…the data will be read in PERIODIC view. This will make so that all financial signs will be implied. All formulas will see (and will be applied to) the amounts as they appear in an Excel sheet when the view (measure) PERIODIC is selected. For example the user will be able to write #MARGIN= (REVENUE – COST) and not #MARGIN= (REVENUE + COST).

The Logic module will be smart enough to derive the correct sign for the SIGNEDDATA amounts to be posted. The benefit is that logic formulas might be more readable and easier to maintain, but there will be a small penalty in performance, because the amounts need to be converted back from PERIODIC to SIGNEDDATA at run-time.

Remark: this feature also works in case the appset stores signeddata in YTD format.

Support for YEAR TO DATE formulas

If the user uses the following setting for the MEASURES dimension:

*XDIM_MEMBERSET MEASURES=YTD

…the data will be read in Year-to-date view. All formulas will see (and be applied to) the amounts as they appear in an Excel sheet when the view (measure) YTD is selected. For example, the user will be able to perform a currency conversion applying the rates to the year-to-date values. Similarly to the PERIODIC view, all financial signs will also be implied.

The Logic module will be smart enough to derive the correct sign and amount for the SIGNEDDATA amounts to be posted. Similarly to the PERIODIC view, a small penalty in performance must be expected.

Remark: this feature also works in case the appset stores signeddata in PERIODIC format.

SQL-based modeling logic

Introduction

In an effort to accelerate the execution of modeling logics, a set of instructions has been implemented, that make modeling logic work in an entirely different way. This method of running logic has some fundamental characteristics:

- It runs a SQL query directly against the fact table. This is, in most cases, by an order of magnitude faster than running an equivalent MDX query against the cube.
- It generates new values according to "rules" assigned to the existing values. This reverses the usual approach by which, starting from some formula, we look for values; here, starting from some values, we check for formulas to execute.
- The generated values are "transactional", i.e. multiple values can be generated for the same destination cell of the cube. At the end of the process, all values pointing to the same cell are cumulated into one value to be posted into the cube.

This method of logic execution cannot be applied to all situations, but there are cases where it fits the need quite nicely. The most common cases where this method can be used are currency conversion (translation logics) and inter-company eliminations (consolidation logics).

For example, in a translation all we need to do is to scan the existing records and apply to each of them a factor (the translation rate) to generate the translated amounts.

One important thing to remember is that this method can only read stored values, and cannot be used to access values calculated in the cube. For example, we cannot access any aggregated amount, unless stored using a regular modeling logic. In addition, we can only read the SIGNEDDATA measure. In a periodic appset, for example, we cannot translate the YTD values, because they are not stored.

Another limitation we originally had with this method was that it could not use the results of a prior calculation, unless they were committed first. This limitation has been removed with some enhancements entered in version 4.2 (See the *GO instruction).

Benefits of SQL logic

The main benefit of using this type of logic is purely performance. There is very little that this logic mode can do, that cannot be done with a regular MDX-based modeling logic. Just keep in mind however that this rule, like any rule, has it exceptions.

The WHEN / ENDWHEN structure

What triggers the use of the above described SQL-based logic method is the use of a WHEN / ENDWHEN structure. This structure works in the same way as the SELECTCASE / ENDSELECT structure, but the instructions that it processes are not the usual MDX formulas of a modeling logic, but some *REC( ) statements that generate new records, as below described.

The syntax is:

```
//-------------------------------------------------------------------------------
*WHEN {criteria}
    *IS {valid condition1}[,{valid condition2},…]
    *REC[([FACTOR|EXPRESSION={Expression}[,{dim1}={member},{dim2}=…])]
    [*REC[([FACTOR|EXPRESSION={Expression}[,{dim1}={member},{dim2}=…])]]
        ….
[*ELSE]
        …
        …
*ENDWHEN
//-------------------------------------------------------------------------------
```

Where:

{criteria} is what to test. Typically, it's a property of the current member of a dimension. The syntax is not MDX, but simply:

DimensionName.Property

Example: *WHEN ACCOUNT.RATETYPE

{criteria} can also use the reserved keyword LOOKUP( ), as later described.

{ValidCondition} is one or more values that meet the criteria. They can be enclosed in double quotes if they need to be treated as strings, even if this is not strictly required. If they represent numeric values, the quotes should be omitted.

Example 1: *IS "AVG","END"

Example 2: *IS 10,20,30

If no operator is specified, the *IS clause assumes the presence of an equal sign (*IS = "AVG", "END"). Other types of comparisons are however supported. The following examples represent valid conditions:

*IS > 2
*IS <= 7
*IS <>"ABC"

Remark 1: multiple operators (like "<>" or ">=") must not be separated by any space. One or more blanks can however be inserted between the operators and the value.

Remark 2: If any operator is used, only one value can be passed. (This syntax is invalid: *IS >2,3,4)

Remark 3: Other operators like AND, OR and NOT are not currently supported.

Remark 4: {ValidCondition} can be one (or more) fixed values, or it can be the reserved keyword LOOKUP( ), as later described.

Remark 5: {ValidCondition} can also be a dimension property specified with the simple format:

*IS dimension.property

Following are valid examples:

```
*WHEN ENTITY
*IS INTCO.ENTITY
…

*WHEN ACCOUNT.SCALE
*IS <>CURRENCY.SCALE
…
```

The *REC( ) instruction tells the program what to do once a criteria has been met. Each REC instruction generates ONE new record to be posted to the database. Each source record can generate as many records as desired, even pointing to the same destination cell.

The parameters of the REC( ) function will specify what to modify of the original record. Any dimension member can be modified with the syntax:

{DimensionName}={member}

Example: *REC(CURRENCY=”USD”, ENTITY=”SALESITALY”)

{member} can be enclosed between double quotes and can contain the name of any dimension, enclosed between the percent sign (For example: ENTITY=”IC_%ENTITY%”). (Remark: The dimension name will be replaced with the value of the current member for that dimension, and NOT with just the dimension name). An alternative syntax allows the logic to retrieve the member name from the value of a property of any dimension. In the following example the entity name is read from the “ENTITY” property of the current member of the INTCO dimension:

*REC(FACTOR=-1, ENTITY=INTCO.ENTITY)

The REC() statement can also use the value of a LOOKUP to define a destination member. For example this could be a valid syntax:

*REC(ACCOUNT = LOOKUP(LK1))

This will generate a numeric account ID corresponding to the value retrieved by the LOOKUP.

This syntax also support a (limited) string concatenation functionality, like in this example:

*REC(ACCOUNT = ACC_ + LOOKUP(LK1))

If the value of the lookup is 20, the resulting destination account will be ACC_20

The FACTOR and EXPRESSION instructions

The amount to assign to the new record can be derived from the original amount with either the instruction FACTOR or the instruction EXPRESSION.

The FACTOR is a factor by which the retrieved amount is to be multiplied.

Example: *REC(FACTOR=1/1.55)

In this example the value found in thy current record is multiplied by 1/1.55.

The EXPRESSION is any formula that will result in the new value to post. The formula can include regular arithmetic operators, fixed values and the keyword %VALUE%, representing the original retrieved value.

Example: *REC(EXPRESSION=%VALUE% + 1000)

Both FACTOR and EXPRESSION can also contain the reserved keyword LOOKUP( ), as later described.

Important remark: WHEN / ENDWHEN structures can be nested by as many levels as desired, and in any sequence. For example, the following structure could be a valid one:

```
*WHEN xxx
        *IS "A"
                *REC(…)
                *REC(…)
        *IS "B"
                *REC(…)
                *WHEN yyy
                        *IS "C","D","E"
                                *REC(…)
                        *ELSE
                                *REC(…)
                *ENDWHEN
*ENDWHEN
```

(Note that the indentations are used just for readability purposes, and are not required by the syntax).

A special case of *WHEN criteria is:

*WHEN *

This criteria can be used when there is actually no criteria to test. In this case, the *WHEN statement is only needed to trigger the SQL mode. The correct syntax is:

```
*WHEN *
  *IS *
    *REC(…)
*ENDWHEN
```

The option NOADD for the *REC( ) instruction

There are cases when multiple values found in a source region should generate a certain fixed value for a given destination record and such value should be the same, irrespective of how many source records have been encountered. (An example could be the need to assign a value of 1 to a "flag" account, if ANY account of the selected region has a value). This can now be achieved in a SQL logic by inserting the option NOADD anywhere in the REC statement.

Example:

```
*WHEN ACCOUNT.TYPE
*IS "AST"
        *REC(EXPRESSION=1, NOADD, ACCOUNT=" FLAG_AST")
*ENDWHEN
```

SQL Logic instruction GET( )

As above said, inside a *REC( ) statement of a SQL logic it is possible to assign a FACTOR or an EXPRESSION to the source value in order to calculate a new value. The factor and the expression can include formulas using hard-coded values (factor=1.3) or values retrieved using the LOOKUP function (factor=lookup(avg)).

In addition to the above, the user can recur to the keyword GET( ) , which allows to also use values from some other record within the selected region.

The syntax is:

GET({dimension}={member}[, {dimension}={member}]…)

Where:

{dimension} is a valid dimension name

{member} is a valid dimension member. This can be an explicit member name like "ABC" (in which case it should be enclosed between double quotes), or it can be derived reading the property of the current member of any dimension.

Valid examples are:

```
GET(ACCOUNT="ABC")
GET(ACCOUNT=ACCOUNT.MYPROPERTY)
GET(ACCOUNT=ACCOUNT.ELIMACC,ENTITY=INTCO.ENTITY)
```

For example, assume that the user wants to calculate the account Revenue as follows:

#Revenue = Units * Price //this is the MDX format

The formula to write in SQL logic will be:

```
*WHEN ACCOUNT
*IS "UNITS"
        *REC(ACCOUNT="REVENUE", FACTOR=GET(ACCOUNT="PRICE"))
*ENDWHEN
```

The get statements can also support the concatenation of properties with trailing fixed strings as follows:

GET(dimension=dimension.property + string)

Example:

GET(ACCOUNT=ACCOUNT.ID + .INP)

Important considerations: while this functionality provides the ability to perform much more complex calculations than what previously allowed in SQL logic, it has some intrinsic limitations that should make it preferable to MDX logic only in case it delivers better performance and scalability than what can be obtained with the much more flexible and elegant MDX logic mode.

Here is an explanation of such limitations:

Caveat 1: The value to retrieve is not queried from the database, but it will be searched for in the currently selected record set (If the value is not found, it will be assumed to have a value of zero). This implies that all values required by the logic must be included in the region to process. In the previously described example, the logic will not work correctly unless the account PRICE has been included in the region to scan, with something like:

*XDIM_MEMBERSET ACCOUNT = Units, Price

(This is fundamentally different from the way any MDX logic works: MDX will automatically be able to retrieve from the cube all necessary information, even if not included in the queried region).

Caveat 2: The logic will not be able to use the results of a previously calculated value. The following example will NOT work…:

```
*WHEN ACCOUNT
*IS “UNITS”
        *REC(ACCOUNT=”REVENUE”, FACTOR=GET(ACCOUNT=”PRICE”))
*IS “REVENUE”
        *REC(ACCOUNT=”TAXES”, FACTOR=.5)
*ENDWHEN
```

…Unless the two calculations are separated by a commit statement:

```
*WHEN ACCOUNT
*IS “UNITS”
        *REC(ACCOUNT=”REVENUE”, FACTOR=GET(ACCOUNT=”PRICE”))
*ENDWHEN
*COMMIT
*WHEN ACCOUNT
*IS “REVENUE”
        *REC(ACCOUNT=”TAXES”, FACTOR=.5)
*ENDWHEN
```

A better alternative now available instead of the COMMIT statement is the GO statement (see).

The FLD( ) instruction inside a FACTOR or EXPRESSION

In a FACTOR or EXPRESSION it is also possible to specify a dimension property using the special format:

FLD(dimension.property)

Here is a valid example:

```
*WHEN ENTITY.SCALE
*IS <>””
        *REC(EXPRESSION=%VALUE%*FLD(ENTITY.SCALE))
*ENDWHEN
```

Remark 1: this syntax is a slight departure from the usual format where the dimension property does not need to be enclosed inside a FLD( ) clause. This format has been adopted in order to (1) retain a good performance in the logic execution, (2) simplify the logic validation and (3) allow for future extensions of our functionality.

Remark 2: all the syntaxes that are supported by the FACTOR and EXPRESSION instructions can be combined inside a calculation expression, like in the following (meaningless) example:

```
*REC(EXPRESSION=%VALUE%*FLD(ENTITY.SCALE)+ GET(ACCOUNT=ACCOUNT.MYACC)*5*LOOKUP(XYZ))
```

The LOOKUP / ENDLOOKUP structure

This set of instructions can be used in conjunction with a WHEN/ENDWHEN structure to retrieve ("lookup") some other values that may be needed either in the calculation of the new value or to define some criteria to be evaluated. The lookup can be performed in the current application or into a different application.

The lookup mechanism essentially defines a relationship between the current record being processed and another record in a corresponding user defined record set. For example, in a currency translation, you may want to identify, in the RATE cube, the value of the rate for the current entity, category and period.

The syntax is:

```
*LOOKUP {App}
        *DIM [{LookupID}:] {DimensionName}="Value" | {CallingDimensionName}[.{Property}]
        [*DIM …]
*ENDLOOKUP
```

Where:

{App} is the name of the application from which the amounts are searched

{DimensionName} is a dimension in the lookup app

{CallingDimensionName} is a dimension in the current application

{LookupID} is an optional identifier of the "looked-up" amount. This is only required when multiple values must be retrieved.

Example:

```
*LOOKUP RATE
        *DIM ENTITY2="DEFAULT"
        *DIM SOURCECURR:INPUTCURRENCY=ENTITY.CURR
        *DIM DESTCURR1:INPUTCURRENCY="USD"
        *DIM DESTCURR2:INPUTCURRENCY="EURO"
        *DIM RATE=ACCOUNT.RATETYPE
*ENDLOOKUP
```

In the above example, three different values are retrieved from the INPUTCURRENCY dimension (the rate of the currency of the current entity, the rate of the currency EURO and the rate of the currency USD). Each of these values has been assigned a specific identifier (SOURCECURR, DESTCURR1 and DESTCURR2) that will be used somewhere in the WHEN/ENDWHEN structure.

Any dimension not specified in the lookup instruction is assumed to match with a corresponding dimension in the source application. In the above example, the following instructions have been omitted, because redundant:

```
*DIM CATEGORY=CATEGORY
*DIM TIME=TIME
```

In the following example, a currency translation in the two reporting currencies USD, and EURO is performed.

```
// ------------------------------------------------------------------------------ Get the rates
*LOOKUP RATE
        *DIM ENTITY2="DEFAULT"
        *DIM RATE=ACCOUNT.RATETYPE
        *DIM SOURCECURR:INPUTCURRENCY=ENTITY.CURR
        *DIM DESTCURR1:   INPUTCURRENCY="USD"
        *DIM DESTCURR2:   INPUTCURRENCY="EURO"
*ENDLOOKUP

// ------------------------------------------------------------------------------ Translate
*WHEN ACCOUNT.RATETYPE
    *IS "AVG","END"
        *REC(FACTOR=LOOKUP(DESTCURR1)/LOOKUP(SOURCECURR),CURRENCY=”USD”)
        *REC(FACTOR=LOOKUP(DESTCURR2)/LOOKUP(SOURCECURR),CURRENCY=”EURO”)
    *ELSE
        *REC(CURRENCY=”USD”)
        *REC(CURRENCY=”EURO”)
*ENDWHEN

*COMMIT
// ------------------------------------------------------------------------------
```

Here below I am showing in a different example how a LOOKUP amount can be used to define a WHEN criteria. In such case, what is being tested is an amount (corresponding to a consolidation METHOD) in the lookup cube. (The logic is not very meaningful, but it is a simplified version of a real one. Take it only as an example of valid syntax)

```
// ------------------------------------------------------------------------------ Get the methods and percent consol
*LOOKUP OWNERSHIP
        *DIM INTCO="IC_NONE"
        *DIM PARENT="MYPARENT"
        *DIM MY_METHOD: ACCOUNTOWN="METHOD"
        *DIM IC_METHOD:   ACCOUNTOWN="METHOD"
        *DIM PCON:             ACCOUNTOWN="PCON"
        *DIM MY_METHOD:ENTITY=ENTITY
        *DIM IC_METHOD:  ENTITY=INTCO.ENTITY
        *DIM PCON:            ENTITY=ENTITY
*ENDLOOKUP

*WHEN LOOKUP(MY_METHOD)          // check my method
    *IS 1,2,3
        *WHEN LOOKUP(IC_METHOD) // check the method of the partner
            *IS 1,2,3
                *REC(FACTOR=LOOKUP(PCON), PARENT ="MYPARENT")
         *ENDWHEN
*ENDWHEN
// ------------------------------------------------------------------------------
```

Finally, a LOOKUP keyword can also be used as part of an *IS statement, as shown in the following example:

```
// ------------------------------------------------------------------------------ Get the percent consols
*LOOKUP OWNERSHIP
        *DIM INTCO="IC_NONE"
        *DIM PARENT="MYPARENT"
        *DIM PCON:     ACCOUNTOWN="PCON"
        *DIM IC_PCON:ACCOUNTOWN="PCON"
        *DIM PCON:     ENTITY=ENTITY
        *DIM IC_PCON:ENTITY=INTCO.ENTITY
```

```
*ENDLOOKUP

*WHEN LOOKUP(PCON)
        *IS <= LOOKUP(IC_PCON)
                *REC(FACTOR=-1, PARENT ="MYPARENT",DATASRC="ELIM")
*ENDWHEN
// ---------------------------------------------------------------------------
```

The *WHEN instruction can also take as parameter a property of one of the dimensions of the cube against which a *LOOKUP has been performed, even if such dimension does not exist in the current cube.

In the following example, a currency translation checks for the MD field of the source currency, in order to decide what formula to apply to the rate (Multiply or Divide):

```
//-----------------------------------
// load the rates from the RATE cube
//-----------------------------------
*LOOKUP RATECUBE
        *DIM RATEENTITY="GLOBAL"
        *DIM RATE=ACCOUNT.RATETYPE
        *DIM SOURCECURR:INPUTCURRENCY=ENTITY.CURR
        *DIM USD:INPUTCURRENCY="USD"
        *NEXT
*ENDLOOKUP

//=============================================
// define the translation rule
//=============================================
*WHEN ACCOUNT.RATETYPE
*IS “AVG”,”END”
        // check the multiply or divide property of the currency
        *WHEN INPUTCURRENCY.MD
        *IS "D"
            *REC(FACTOR=LOOKUP(USD)/LOOKUP(SOURCECURR),CURRENCY="USD")
        *ELSE
            *REC(FACTOR=LOOKUP(SOURCECURR)/LOOKUP(USD),CURRENCY="USD")
        *ENDWHEN
*ELSE
        *REC(CURRENCY="USD")
*ENDWHEN
```

Remark: the LOOKUP instructions must obviously define the link between the dimension referenced to in the WHEN statement and one of the dimensions of the current cube. In the above example the logic will understand automatically that it needs to evaluate the MD field of the InputCurrency matching the Currency of the current Entity.

MDX based LOOKUP

The *LOOKUP / *ENDLOOKUP structure normally generates an SQL query, and, as such, is unable to return values calculated by the OLAP cube. To overcome this limitation a different version of the LOOKUP instruction has been implemented. This instruction generates an MDX query and, as a result, allows the retrieval of any value available in the cube.

The syntax is:

*OLAPLOOKUP [{Application name}]

All other instructions specifiable inside the structure remain the same of a regular LOOKUP instruction.

Example:

```
*OLAPLOOKUP FINANCE
        *DIM ENTITY="SALESEUROPE"
        *DIM ACCOUNT="REVENUE"
*ENDLOOKUP
```

The above example will be able to retrieve the REVENUE account of entity SALESEUROPE, even if one or both these members are parent or otherwise calculated by the cube.

This feature, although less efficient than an SQL-based LOOKUP, can be particularly useful for accessing aggregated data that need to be used in the factor of an allocation logic. Here below is an example of allocation logic where the expense HUMAN_RES_EXP incurred by the entity CORP_SERVICES is allocated to all children of EUROPE based on their number of employees (HEADCOUNT):

```
//-------------------------------------------------------------------------------------------
*XDIM_MEMBERSET ENTITY=[ENTITY].[EUROPE].children

*OLAPLOOKUP FINANCE
        *DIM TOT_HC:ENTITY="EUROPE"
        *DIM TOT_HC:ACCOUNT="HEADCOUNT"
        *DIM TOT_HR:ENTITY="CORP_SERVICES"
        *DIM TOT_HR:ACCOUNT="HUMAN_RES_EXP"
*ENDLOOKUP

*WHEN ACCOUNT
*IS HEADCOUNT
        *REC(FACTOR=LOOKUP(TOT_HR)/LOOKUP(TOT_HC),ACCOUNT="ALLOCATED_HR_EXP")
*ENDWHEN
//-------------------------------------------------------------------------------------------
```

The time shift instructions

To simplify the calculation of leads and lags in financial reporting applications, the following new instructions have been implemented for SQL-based logics:

PRIOR
NEXT
BASE
FIRST

The instructions PRIOR and NEXT support an optional numeric parameter. This parameter represents the number of time periods by which the current period must be shifted. If omitted, the functions will assume a time shift of 1 period (forward or backwards). Negative values are accepted (A negative value for a NEXT function corresponds to a positive value for a PRIOR function and vice-versa).

Examples:

```
TIME=NEXT          // In a monthly application this means next month
TIME=PRIOR(3)      // Three periods backwards
TIME=NEXT(-3)      // Same as PRIOR(3)
```

The keyword BASE always represents the last period of prior fiscal year. When the fiscal year is a normal calendar year and the frequency is monthly, the base period of 2004.JUN is 2003.DEC.

The instruction BASE can be useful in YTD applications, where the opening balances need to be retrieved from the last period of prior year.

The keyword FIRST always represents the first period of the current fiscal year. When the fiscal year is a normal calendar year and the frequency is monthly, the base period of 2004.JUN is 2004.JAN.

In case the time shift goes past the boundaries of the TIME dimension, these time shift functions will return no period.

These functions can be used in four ways:

- To re-direct the destination period in a *REC statement

  Example 1: *REC(TIME=NEXT)
  Example 2: *REC(TIME=BASE)

- To retrieve a value from a different period in a *REC statement

  Example 1: *REC(FACTOR=GET(TIME=PRIOR(3)))
  Example 2: *REC(FACTOR=GET(TIME=BASE))

- To add periods to the selected data region in a XDIM_MEMBERSET statement

  Example: *XDIM_MEMBERSET TIME=PRIOR, %TIME_SET%

  In this example, if the first modified period is 2004.APR, the instruction PRIOR will add 2004.MAR to the region to process).

- When the keywords PRIOR, FIRST or BASE are added to a XDIM_MEMBERSET instruction, the time period PRIOR, FIRST or BASE can be also evaluated in a WHEN / ENDWHEN structure, like in the following example:

  *WHEN TIME
  **IS PRIOR
    // ignore
  *ELSE
    *REC(…)
  *ENDWNHEN

  In presence of an XDIM_MEMBERSET containing the PRIOR keyword, like in the above example, the WHEN structure here shown will recognize 2004.MAR as PRIOR period.

Following is an example of logic that performs a carry-forward of account ACCREC, while adding to it the periodic amount from EXTSALES.

```
*XDIM_MEMBERSET TIME=PRIOR,%SET%,%PREFIX%.DEC

*CALC_EACH_PERIOD

*WHEN TIME
*IS PRIOR
        *WHEN ACCOUNT
```

```
        *IS ACCREC
                *REC(ACCOUNT=”OPEACCREC”,TIME=NEXT)
        *ENDWHEN
*ELSE
        *WHEN ACCOUNT
        *IS EXTSALES
                *REC(FACTOR=-1,ACCOUNT="OPEACCREC",TIME=NEXT)
                *REC(FACTOR=-1,ACCOUNT="ACCREC")
        *IS OPEACCREC
                *REC(ACCOUNT=”ACCREC”)
                *REC(ACCOUNT=”ACCREC”,TIME=NEXT)
        *ENDWHEN
*ENDWHEN
```

Improved MEMBERSET definition in the TIME dimension

It is possible to make sure that the member set of the TIME dimension reaches the end of the year, even in case the year does not correspond to the calendar year. To do this the user may write the following instructions:

```
*CALC_EACH_PERIOD
*XDIME_MEMBERSET TIME = %SET%, %YEAR%.DEC
```

The trick is in the recognition of the **%YEAR%** keyword, which will be replaced with the value of the [YEAR] property of the last member in the list of modified periods.

The extension DEC can be modified to match the desired period name (like JUN, if June is the closing month of the fiscal year). The property [YEAR] must contain the calendar year of the closing month of the fiscal year. For example, with a fiscal year ending in JUN, the [YEAR] property of 2005.JUL must be 2006.

The feature will work even if multiple fiscal years are modified simultaneously.

Memory variables (the # prefix)

It is now possible to create intermediate result and assign them to dummy members (like dummy accounts or dummy members of any other dimension). These members can be used as placeholders to store intermediate results that can be used as inputs for subsequent calculations. These values will be automatically skipped at commit time.

Dummy members must be identified with a leading pound (#) sign. For example:

*REC(ACCOUNT = #TEMP)

Account #TEMP does not exist in the account dimension. The generated record can be used somewhere else in the logic, but its value will not be stored in the database.

Example:

```
*WHEN ACCOUNT.FLAG
*IS = Y
        *REC(ACCOUNT=#TEMP)
*ENDWHEN

*GO

*WHEN ACCOUNT
*IS #TEMP
        *REC(FACTOR=GET(ACCOUNT=MULTIPLIER),ACCOUNT=FINAL)
```

```
*ENDWHEN
```

The above technique could be used in an allocation procedure, requiring the calculation of the total value of the coefficient to use in the allocation process. Alternatively, the user could calculate an opening balance amount that does not need to be stored in the db.

Remark: the dummy members generated with this technique may or may not exist in any dimension. Because of this, it may not be possible to make use of properties assigned to them. In this situation, they can only be referenced using their IDs.

Note however that, if the dummy member corresponds to a real member, its properties ARE indeed readable. The following (meaningless) example will work correctly:

```
//=============================================================================
// Reading properties of DUMMY (#) MEMBERS
//=============================================================================
// this logic is able to find the properties CALC and RATETYPE of #ACCREC
// (ACCREC is a real account, generated as DUMMY, i.e. with a  leading #)

*WHEN ACCOUNT
*IS CASH
        *REC(ACCOUNT=#ACCREC)
*ENDWHEN

//--------------------
*GO
//--------------------

*WHEN ACCOUNT
*IS #ACCREC
        *REC(ACCOUNT=# + ACCOUNT.CALC)
        *WHEN ACCOUNT.RATETYPE
        *IS END
                *REC(ACCOUNT=#IS_END)
        *ENDWHEN
*ENDWHEN
```

The XDIM_NOSCAN instruction

When a records set is loaded in memory for processing through a WHEN/ENDWHEN structure, some records are needed as triggers for the activation of some calculation, and some others are just needed to contribute to the calculation.

For example, in a calculation of REVENUE = UNITS * PRICE, the UNITS account is typically the trigger, while the PRICE is only needed to be multiplied by the UNITS, as in the following example:

```
*XDIM_MEMBERSET ACCOUNT=UNITS, PRICE

*WHEN ACCOUNT
*IS UNITS
        *REC(FACTOR=GET(ACCOUNT="PRICE"), ACCOUNT="REVENUE")
*ENDWHEN
```

To simplify this logic, the user could predefine the members that are not needed as triggers, telling the logic that they could be ignored right away, during the scansion of the recordset. In the above example, the instruction could be written as follows:

```
*XDIM_MEMBERSET ACCOUNT=UNITS, PRICE
*XDIM_NOSCAN ACCOUNT=PRICE
```

```
*WHEN *
*IS *
        *REC(FACTOR=GET(ACCOUNT=”PRICE”), ACCOUNT=”REVENUE”)
*ENDWHEN
```

This definition will have two benefits:

- The syntax of the WHEN structure would be simplified
- The logic executes a little bit faster

A more convincing example could be a case where the PRICE is taken from a different CATEGORY, like in the following example:

```
*XDIM_MEMBERSET ACCOUNT=UNITS, PRICE
*XDIM_MEMBERSET CATEGORY=ACTUAL,BUDGET

*WHEN ACCOUNT
*IS UNITS
        *WHEN CATEGORY
        *IS ACTUAL
                *REC(FACTOR=GET(ACCOUNT=”PRICE”,CATEGORY=”BUDGET”), ACCOUNT=”REVENUE”)
        *ENDWHEN
*ENDWHEN
```

…which would simply become:

```
*XDIM_MEMBERSET ACCOUNT=UNITS, PRICE
*XDIM_MEMBERSET CATEGORY=ACTUAL,BUDGET
*XDIM_NOSCAN ACCOUNT=PRICE
*XDIM_NOSCAN CATEGORY= BUDGET

*WHEN *
*IS *
        *REC(FACTOR=GET(ACCOUNT=”PRICE”,CATEGORY=”BUDGET”), ACCOUNT=”REVENUE”)
*ENDWHEN
```

Calculating orgs in memory

If the calculated members generated by a CALC_ORG instruction do not need to be posted to the database, the user can use a variation of the CALC_ORG instruction that will not add these extra records to the database, even when the COMMIT instruction is reached.

The instruction is:

*CALC_DUMMY_ORG {DimensionName} = {OrgProperty}

A valid alternative syntax is:

*CALC_ORG {DimensionName} = {OrgProperty}, DUMMY

This instruction will automatically add a pound sign (“#”) in front of the generated member Ids. As a result these records will be considered memory variables (see above) that will not be posted to the database.

Advanced features of memory variables and dummy orgs

When a memory variable represents a real member of a dimension, the user can still access its properties in a WHEN / ENDWHEN structure.

The logic may use the properties of a valid memory variable in evaluating a WHEN criteria as well as in the definition of a destination member.

The example shown here below uses the property of a memory variable to read an aggregated value from a parent and transfer it into a base level member whose ID is stored in a parent's property:

```
//-------------------------------------------------------------------------------
// create the memory variables (in this case the parents in H1, for example #SALES, #WORLDWIDE1)
*CALC_DUMMY_ORG ENTITY=PARENTH1

// Some parent might have a corresponding input member specified in a property
*WHEN ENTITY.INPUTMEMBER
*IS<>""
        *REC(ENTITY=ENTITY.INPUTMEMBER)
*ENDWHEN
//-------------------------------------------------------------------------------
```

Similarly, it is possible to access the value of a memory variable from a GET instruction, using the syntax shown in the following example:

```
//-------------------------------------------------------------------------------
*XDIM_MEMBER ACCOUNT=SQ_FEET

// create the memory variables
*CALC_DUMMY_ORG ENTITY=PARENTH1

// Calculate the allocation factor of each base member (note # used to identify the dummy parent)
*WHEN ENTITY.ISBASEMEM
*IS="Y"
   *REC(FACTOR=1/GET(ENTITY=# + ENTITY.PARENTH1) * 100, ACCOUNT=PCT_SQ_FEET)
*ENDWHEN
//-------------------------------------------------------------------------------
```

If the base level members have many levels of parents, it is also possible to specify the number of levels to ascend, in the search for the value of a "parent" member. The syntax is:

GET(dimension=dimension.property (number of levels))

Here below a search is performed in a parent that is 4 levels above current member. Note that this example does not use memory variables, because the parents in the stored hierarchy PARENTS1 are actually stored (in fact the function used is CALC_ORG and not CALC_DUMMY_ORG).

```
//-------------------------------------------------------------------------------
*XDIM_MEMBER ACCOUNT=SQ_FEET

// Calculate the parents to store
*CALC_ ORG ENTITY=PARENTS1

// Calculate the allocation factor of each base member having 4 levels of parents above
*WHEN ENTITY.PARENTS1
*IS<>""
   *REC(FACTOR=1/GET(ENTITY=ENTITY.PARENTS1(4)) * 100, ACCOUNT=PCT_SQ_FEET)
*ENDWHEN
//-------------------------------------------------------------------------------
```

Enhanced CALC_ORG and CALC_DUMMY_ORG features

The instructions CALC_ORG and CALC_DUMMY_ORG now support a new multi-line syntax that permits to better control the scope of the calculation.

While, the original single-line syntax is still supported, the following new syntax can also be used:

```
*CALC_DUMMY_ORG
        *ORG {dimension}={property}
        [*NOSCAN]
        [*WHERE {dimension} = {member set}]
        [*WHERE {dimension} = {member set}]
        …
*ENDCALC
```

Example:

```
*CALC_DUMMY_ORG
        *ORG ENTITY=PARENTH1
        *WHERE ACCOUNT = CASH, ACCREC
        *WHERE INTCO = NON_INTERCO
*ENDCALC
```

In the above example the calculation of the hierarchy PARENTH1 for the ENTITY dimension will only be performed for accounts CASH and ACCREC and for the intercompany member NON_INTERCO, even if all account or intercompany members are in memory.

This feature can be extremely helpful in controlling the number of records that will be created by the CALC_ORG or CALC_DUMMY_ORG instructions, in all those cases where only a few elements are actually needed for some dimension.

Important remarks:

The instruction still works only on the records that have been pulled in memory by the logic query, as defined by the XDIM_MEMBERSET or similar instructions. The WHERE clause in the CALC_ORG or CALC_DUMMY_ORG will NOT modify the region accessed by the query, but it will only apply an additional filter to such region.

The WHERE clause only supports the EQUAL (“=”) sign. Other operands like “<>” are not supported for the time being.

In addition, the member set at the right of the equal sign in the WHERE clause must be EXPLICIT. Other syntaxes (like WHERE ACCOUNT = ACCOUNT.GROUP=”Assets” or similar) are currently not supported.

The NOSCAN instruction in CALC_DUMMY_ORG

A NOSCAN instruction is also supported in a multi-line CALC_DUMMY_ORG structure, with the syntax shown in the following example:

```
*CALC_DUMMY_ORG
        *ORG ENTITY=PARENTH1
        *NOSCAN
*ENDCALC
```

This will allow the user to define a logic that does not need to skip the calculated members in the WHEN filters, but can still use the generated values in a GET( ) statement.

(For additional details see the *XDIM_NOSCAN instruction)

The instruction *PRIOR

It may sometimes happen that two different data categories need to be linked in time for some logic calculation. A good example is a BUDGET category that must retrieve the opening balances of the balance sheet from category ACTUAL.

This link can be easily established with the following instruction:

*PRIOR {dimension name} = {member ID}

Example:

*PRIOR CATEGORY = ACTUAL

This instruction will automatically force the retrieval of prior period's values from a different category (in this case ACTUAL).

An alternative syntax allows the dynamic retrieval of the name of prior category reading a property of the current category, like in this example.

*PRIOR CATEGORY = CATEGORY.PRIORCAT

Note that this syntax only makes sense in a case where the logic needs to go backwards in time to retrieve values from past periods. For example, assume the user wants to load in memory the three months preceding the first modified month. This will be normally achieved with the instruction:

*XDIM_MEMBERSET TIME = PRIOR(3),%SET%

However, if the first modified period is FEB, going backwards three months will cross the year-end boundary, and NOV and DEC will be retrieved from last year. Now, if last year's data are not stored in the current category but in a different one, the above instruction will do the job.

It is important to realize a few things:

1) Prior category does not need to be included in the set of categories to load in memory: its data will be automatically loaded in memory if needed (according to the number of periods the logic goes backwards).

2) The values of prior category will be available for processing by the logic, but it is very likely that these should not be modified (last year data might even be locked) and the logic will have to skip them. However, the way to control this is NOT to check the name of the category with something like:

```
// This will not work!!!
*WHEN CATEGORY
*IS ACTUAL
```

The good news is that in reality what needs to be skipped are not the values coming from prior category, but the values coming in ANY of the prior periods, irrespective of whether they belong to ACTUAL or BUDGET (if the first modified period is FEB, all three periods NOV DEC and JAN must not be changed). This condition can be easily tested with these instructions:

```
// This will work!!!
*WHEN TIME
*IS PRIOR
```

3) The logic will always work as if all periods come from current category, even while reading records that come from prior category. For this reason the destination category does not need to be specified in the REC instructions.

```
// This will write in ACTUAL, even if reading a record coming from BUDGET
*REC(ACCOUNT=”#TEMP”,TIME=NEXT(3))
```

4) In lead and lag calculations the most common (and sometimes difficult) decision to take is whether one should get values from the past or push values into the future. This may sometimes depend on the type of formula. In most cases I have found more practical to push values into the future using the existence of values in the past as triggers.

```
// Example
*WHEN ACCOUNT
*IS COLLECTIBLES_AT_THREE_MONTHS
        *REC(ACCOUNT=#COLLECTIONS_AT_THREE_MONTHS, TIME=NEXT(3))
*ENDWHEN
```

Note that this logic should only write in valid periods (<>PRIOR). This is why the logic stores the value in a temporary variable (using a leading pound sign #). The temporary values can then be written only if the period is correct.

```
*WHEN TIME
*IS <> PRIOR
        *WHEN ACCOUNT
        *IS #COLLECTIONS_AT_THREE_MONTHS
                *REC(ACCOUNT=COLLECTIONS_AT_THREE_MONTHS)
        *ENDWHEN
*ENDWHEN
```

5) This instruction will only work as desired if only ONE CATEGORY at a time is modified by the user during data entry (or processed from a Data Manager task). The administrator should somehow make sure that a logic containing this instruction is used correctly.

6) It is important to remember to use the above techniques in conjunction with the *CALC_EACH_PERIOD option. This will enforce the calculation of all periods in the correct sequence.

The instruction *FIRST_PERIOD

This instruction comes as an integration of the above described *PRIOR instruction and can be used whenever the switch between the current category and prior category should not happen at year end, but in a different, user-defined period.

The syntax is:

*FIRST_PERIOD = {period}

For example, say that a rolling forecast is entered in the 12 periods 2005.APR through 2006.MAR, but the preceding values for the months of 2005.JAN through 2005.MAR should be read from a different category.

This result can be achieved combining the following two instructions:

*PRIOR CATEGORY = “ACTUAL” // quotes are optional
*FIRST_PERIOD = ”APR”

An alternative syntax allows the dynamic retrieval of the name of first period by reading a property of the current category, like in this example.

*FIRST_PERIOD = CATEGORY.FIRSTPERIOD

The search for the correct date will be performed scanning all periods preceding the first modified period moving backwards until a time period with the property PERIOD = {period} is met. For example, if {period} is “MAR” and the first modified date is 2005.FEB, the first date will be 2004.MAR, which is the first date where PERIOD=”MAR” going backwards from 2005.FEB.

The keywords POS( )

A special keyword can be used in a WHEN / ENDWHEN structure, when the user needs to compare the position of the current time period relative to a different one. This situation may arise when some logic must change behavior when passing beyond a certain date. For example the first three months of a year may contain actual data, where some accounts are input, and the other months contain budget data, where the same accounts are calculated.

The keyword comes in two formats:

*POS(TIME) // to use in a *WHEN instruction

And

POS({time}) // to use in a *IS instruction

The correct structure is:

*WHEN POS(TIME)
*IS <>= POS({time})

//…..

…where {time} must be an explicit date. For example:

*WHEN POS(TIME)
*IS <>=POS(“2005.MAR”)
//……

To make the date more dynamic an additional instruction has been implemented, as here below explained.

The instruction *FLAG_PERIOD= {period}

..where period must be a valid value of the PERIOD property of the TIME dimension. (For example the date 2005.MAR would typically have PERIOD=”MAR”).

The value of {period} can also be retrieved from a property with the following alternative syntax:

*FLAG_PERIOD= {dimension }.{property}

For example:

*FLAG_PERIOD = CATEGORY.FLAGPERIOD

This instruction creates a value for the **reserved keyword %FLAG_PERIOD%**, which can be used as a replacement string inside the POS( ) keyword as follows:

*WHEN POS(TIME)
*IS >=POS(%PREFIX%.%FLAG_PERIOD%)

Example:

```
//-------------------------------------------------------------------------------------------------
*FLAG_PERIOD=CATEGORY.FIRSTPERIOD

*WHEN POS(TIME)
*IS >= POS(%PREFIX%.%FLAG_PERIOD%)
        *WHEN ACCOUNT
        *IS UNITS
                *REC(FACTOR=GET(ACCOUNT=”PRICE”), ACCOUNT= “REVENUE”)
        *ENDWHEN
*ELSE
        *REC    // this may be needed if clear_destination is  used
*ENDWHEN
//-------------------------------------------------------------------------------------------------
```

Remarks:

Similarly to the *PRIOR instruction, the *FLAG_PERIOD instruction will work dynamically only if just ONE member of the selected dimension (in this example CATEGORY again) is being processed by the logic.

On the other hand, this instruction can also be used in a logic definition where *CALC_EACH_PERIOD is not required.

Recognition of PRIOR(n) periods in * IS statements

In a logic like the following…:

```
*CALC_EACH_PERIOD
*XDIM_MEMBERSET TIME = PRIOR(3), %SET%

*WHEN TIME
*IS PRIOR
        //……
*ENDWHEN
```

…all 3 periods of data preceding the first period in %SET% will be recognized as PRIOR in the statement *IS PRIOR

Example:

If for example, the user modifies 2005.MAR and 2005.MAY, the three period 2004.DEC, 2005.JAN, and 2005.FEB will meet the criteria stated in the *IS line.

This will allow the logic to recognize periods that may have been loaded in memory only with the purpose of calculating some lags in the time dimension. A fair example of how to use this feature is represented by the following logic:

```
//-------------------------------------------------------------------------------------------
*XDIM_MEMBERSET ACCOUNT=REVENUE,PRICE,PAYMENTS

*CALC_EACH_PERIOD

*XDIM_MEMBERSET TIME=PRIOR(3),%SET%,%PREFIX%.DEC

*WHEN ACCOUNT
*IS REVENUE
        *WHEN GET(ACCOUNT="PRICE")
        *IS 0
                *REC(ACCOUNT=#PAYMENTS0,TIME=NEXT(0))
        *IS 1
                *REC(ACCOUNT=#PAYMENTS1,TIME=NEXT(1))
        *IS 2
                *REC(ACCOUNT=#PAYMENTS2,TIME=NEXT(2))
        *IS 3
                *REC(ACCOUNT=#PAYMENTS3,TIME=NEXT(3))
        *ENDWHEN
*ENDWHEN

*GO

*WHEN TIME
*IS<>PRIOR     // prior here means any period before first in %SET%
        *WHEN ACCOUNT
        *IS #PAYMENTS0,#PAYMENTS1,#PAYMENTS2,#PAYMENTS3
                *REC(ACCOUNT=PAYMENTS)
        *ENDWHEN
*ENDWHEN
//-------------------------------------------------------------------------------------------
```

Recognition of NEXT(n) keyword in XDIM_MEMBERSET statement

The instruction XDIM_MEMBERSET, when applied to the TIME dimension, can also handle the keyword NEXT(n), which will allow to extend in the future the set of passed periods to process.

Examples:

```
*XDIM_MEMBERSET TIME=PRIOR,%SET%,NEXT        // add 1 period before and 1 period after

*XDIM_MEMBERSET TIME= %SET%,NEXT(3)          // add 3 periods after
```

The instruction TEST_WHEN

Sometimes it may happen that a condition could be tested only once, because such condition is not dependent on the records being scanned. For example one might want to test that in the selected set of members for a given dimension there is a specific keyword or that a certain cell of the cube has a specific value. Instead of applying the condition to the entire set or records to scan, the condition may be evaluated before the WHEN / ENDWHEN loop, with the following instruction:

*TEST_WHEN({condition})

The {condition} is a string that the logic engine will pass to VB script for evaluation. If the returned value is TRUE the subsequent WHEN / ENDWHEN loop will be processed. Otherwise the entire loop will be skipped.

Example:

```
// skip the loop if Budget is not among the passed categories
*TEST_WHEN(instr("%CATEGORY_SET%","BUDGET")>0)

*WHEN *
//…..
*ENDWHEN

*GO

// skip the loop if account FLAG is zero for a certain time,intco combination
*TEST_WHEN(GET(ACCOUNT="FLAG",INTCO="NON_INTERCO",TIME="2004.JAN")<>0)

*WHEN *
//…..
*ENDWHEN
```

As shown in the above example, the instruction supports the use of the %{dim}_SET% keyword. It also supports the GET instruction, to retrieve a value from the recordset. When one or more GET instructions are used in the evaluation of the condition, it must be remembered that all required dimensions must be specified. The non specified dimensions will default to the values they have in the first record of the source record set. In other words, the user can only omit dimensions that do not vary in the recordset being scanned.

Another syntax supported anywhere in this instruction is the FLD( ) keyword. Here is a valid example:

```
*CALC_EACH_PERIOD                            // handle periods one by one
*XDIM_MEMBERSET TIME=PRIOR, %TIME_SET%       // include prior period to selected dates

// This will test that the evaluated period is not the prior period
*TEST_WHEN(instr("%TIME_SET%","FLD(TIME.ID)")>0)
```

*WHEN….

The TEST_WHEN instruction is specific to the current GO section. If no GO section is specified, it is specific to the current COMMIT section.

Clearing the destination area in SQL logics

In most cases there is no need to perform a clear of the destination area when re-executing a modeling logic that uses SQL queries. SQL logics, in fact, base their computation on the existence of records in the fact table, irrespective of the value assigned to these records. Since values that have been re-set to zero maintain records in the fact table, the execution of the SQL logic will ensure the correct handling of any value, including those that have been set to zero.

This may not be true in all cases. For example, if a value is set to zero and, before the SQL logic is executed, an administrator performs a compression of the fact table, the zeroed-out records will be lost. This could lead to a situation where the logic will not clear the records it generated in a prior pass, until the user re-enters some value in the values that he set to zero. This situation is unlikely to happen when the logic is a default logic (which is executed at the same time data are entered), but it could happen with logics that are executed in a batch mode (like eliminations, allocations or consolidations).

To control these situations, the administrator can make use of a couple of instructions that will enforce a clear of all records existing in the destination region at the time the logic is executed.

WARNING: these instructions MAY LEAD TO A DELETION OF THE INPUT DATA, if used incorrectly. A good understanding of their behavior is required, in order to avoid the risk of serious losses of data in the database.

The instructions to use are:

*CLEAR_DESTINATION
*DESTINATION {DimensionName1}={MemberSet1}
*DESTINATION {DimensionName2}={MemberSet2}
…

The instruction CLEAR_DESTINATION activates the clear mechanism. If not present, the logic will not try to perform any clear.

The second instruction, the DESTINATION instruction, is optional. However, when CLEAR_DESTINATION is used, the DESTINATION instruction MUST be used for ALL the dimensions for which the user wants to be SURE that the correct region is cleared. If not used, the program will try to automatically decide what to clear in each dimension, and this, in some cases, might be incorrect. Following is an explanation that clarifies what could happen:

Case 1: the source and destination regions are the same.

For example, the category is ACTUAL for both the source and the destination regions. The category the program will clear will definitely be ACTUAL. For this dimension, there is no need to specify anything.

Case 2: the source and destination regions are different, as specified by a XDIM_MEMBER instruction.

Example: *XDIM_MEMBER DATASRC=LC TO ELIM

In this case the logic knows that the destination Datasrc member can only be ELIM. The Datasrc the program will clear will definitely be ELIM. In this case too, there is no need to specify anything.

Case 3: the source and destination members are different, as defined by one or more *REC( ) instructions.

This is the case where the logic might have an issue in deciding what to clear. The REC( ) instruction, in fact, has a great deal of power in deciding where to write its output, and could do it on multiple dimensions at the same time. It could say thing like:

```
*REC(CURRENCY=”EURO”)
*REC(ACCOUNT=ACCOUNT.PLUGACCOUNT)
*REC(ENTITY=”IC_%ENTITY%”,CURRENCY=ENTITY.CURR)
```

As a result, for ***all*** the dimensions where the destination region is defined by one or more *REC instructions, it is MANDATORY for the user to explicitly restrict the region that he wants to be cleared.

For example, if a translation logic says:

```
*WHEN ACCOUNT.RATETYPE
*IS "AVG","END"
        *REC(FACTOR=LOOKUP(EURO)/LOOKUP(SOURCECURR),CURRENCY="EURO")
*ELSE
        *REC(CURRENCY="EURO")
*ENDWHEN
```

…the instructions to use will be:

```
*CLEAR_DESTINATION
*DESTINATION CURRENCY=EURO
```

If the DESTINATION for the currency dimension is not specified, the destination region for the currency will be:

LC,EURO

..resulting in the loss of all input data!!!

Remark 1: The members specified in the DESTINATION region can be a list of comma-delimited members (example: *DESTINATION CURRENCY=EURO,USD) or a member set defined with an MDX expression, for example like the following:

```
*DESTINATION CURRENCY=filter([CURRENCY].members,[CURRENCY].properties(“REPORTING”=”Y”)
```

Remark 2: A destination region is created by the logic even when no CLEAR_DESTINATION instruction is used. This is done when the CALCULATE_DIFFERENCE option is active, to calculate the difference between the newly calculated values and the values existing in the database. In this situation, having a destination region that could be broader than needed cannot do any harm, and the user does not need to worry. However, the instruction DESTINATION could still be used, even if no clear_destination instruction is used, simply to optimize the size of the destination region to query, with some benefit to the performance and memory footprint of the logic execution.

The instruction * DESTINATION supports also the “not equal to” operator with the syntax:

*DESTINATION<>{MemberSet}

This operator can be handy to pass to the SQL query smaller lists of valid members that will be more efficiently parsed by the Microsoft SQL engine.

Calculate_difference across applications in SQL logic

The program is able to determine the structure of the records of the destination application, and compare them with the records calculated from the source application, to correctly post the difference.

The calculate_difference option is however disabled in logics of type MDX, when the destination application is different from the source application.

Together with this feature comes the ability to define a *DESTINATION region different from the default, and to enforce a *CLEAR_DESTINATION option, even if the option *DESTINATION_APP has been set.

Intercompany Eliminations in SQL logics

The elimination of the intercompany values is in general based on the hierarchical relationships defined in the entity dimension. This type of relationships can be evaluated using MDX-based logics, because the MDX language understands hierarchies. SQL-based logics are faster than MDX logics, but they are only able to evaluate member names and properties, and no MDX formula can be defined inside their WHEN / ENDWHEN structures.

A special function, able to evaluate hierarchies and to understand parent-child relationships in a tree, has been implemented specifically for SQL-based logics, to enable the elimination of intercompany values using SQL-based logics.

This function is named CPE (for "Common Parent Elimination") and supports the following syntax:

CPE({Entity1} , {Entity2} [, {Organization} [, {ElimProperty} [, {ElimDim}]]])

This function will return the name of the "Elimination entity" immediately below the first "common parent" of {Entity1} and {Entity2}, as found in the specified {Organization}. The "Elimination entity" is an immediate dependant of the common parent having the property {ElimProperty} set to "Y".

Examples:

CPE("ITALY" , "FRANCE")

…should return the name of the elimination entity below EUROPE, for example E_EUROPE

CPE("ITALY" , "US")

…should return the name of the elimination entity below WORLD, for example E_WORLD

Remarks:

The parameters Entity1 and Entity2 are required and can be:

- A specific name enclosed in double quotes (like "Italy")
- A dimension name (like ENTITY, meaning the current member of the ENTITY dimension)
- A dimension name and property combination (like INTCO.ENTITYPROP, meaning the value of the ENTITYPROP property of the current member of the INTCO dimension)

The Organization parameter is optional and must be in the format Hn where n is the organization number (for example H1 or H2, etc. to indicate PARENTH1 or PARENTH2, etc.). If omitted the value H1 will be assumed.

The ElimProperty parameter is optional and can be used to indicate the name of the entity property that defines the "elimination" entities (i.e. the entities where the amounts should be eliminated). If omitted the value ELIM will be assumed.

The ElimDim parameter is also optional. It can be used in case the dimension for which the elimination must be performed is NOT a dimension of type ENTITY. In the following example the user is defining the elimination between Business Units in the BU dimension (here the Inter-CO dimension is in reality an inter-BU dimension).

```
*WHEN CPE(BU, I_BU.BU, , , BU)
*IS <>""
        *REC(…….)
*ENDWHEN
```

The function CPE( ) is only valid in a *WHEN statement. The subsequent *IS statement should verify that a valid elimination entity has been returned. (*****IS <> "”). The returned elimination entity can then be referenced in a subsequent *REC statements using the keyword %CPE%. The following example shows a correct use of the function:

```
*WHEN CPE(ENTITY,INTCO.ENTITY)
*IS <>""
    *REC( ENTITY = %CPE%, FACTOR = -1 )              // eliminate the original account...
    *REC( ENTITY = %CPE%, ACCOUNT = ”PLUG”)          //...into the account plug
*ENDWHEN
```

Note how the name of Entity1 is derived from the current ENTITY member and the name of Entity2 is derived from the ENTITY property of the current INTCO member.

The instruction *XDIM_GETINPUTSET

This instruction filters the members of the selected region for a given dimension according to their compliance with some user-defined criteria that must be met by the values in the database.

XDIM_GETINPUTSET serves the same purpose of XDIM_GETMEMBERSET, but it generates an SQL query instead of an MDX query behind the scenes. This implies that, while it can only be used to check for the value of input cells (i.e. having entries in the fact table), it will, in most cases, deliver a much better performance and scalability.

The syntax is:

```
*XDIM_GETINPUTSET {dimension} [={member set}]
        [*APP={application}]                                  //optional
        [*XDIM_MEMBERSET {dimension} [={member set}]          //as many of these as needed
        [*CRITERIA {expression}]                              //optional
*ENDXDIM
```

The portions between [square brackets] are optional.

Example:

```
*XDIM_GETINPUTSET ENTITY
        *APP OWNERSHIP
        *XDIM_MEMBERSET ACCOUNTOWN=METHOD
        *XDIM_MEMBERSET CURRENCYPARENT=C_GR_FIN
```

```
        *XDIM_MEMBERSET INTCO=TPNONE
        *CRITERIA SIGNEDDATA>71
*ENDXDIM
```

This will filter all entities that in application OWNERSHIP have a value greater than 71 in account METHOD, for currencyparent=C_GR_FIN, etc.

For all the dimensions not specified in the instruction, the search will be performed in the corresponding members of the selected region. For example, the category and period will be those for which the logic was being executed.

Warning: the criteria instruction can be used to filter the members against an acceptable value range. However, currently the instruction is not able to take into account the stored sign of the selected account, and this has to be figured out manually. For example, if the instruction is supposed to filter a REVENUE account with a value greater than 100, the instruction should say:

```
        *CRITERIA SIGNEDDATA<-100          // this means >100 for income accounts
```

…because REVENUE is stored with a negative sign.

Similarly to XDIM_GETMEMBERSET, the XDIM_GETINPUTSET instruction is specific to the COMMIT section it is written in.

Special case: Change of dimension name across applications

The instruction *XDIM_GETINPUTSET can redefine the name of the dimension being filtered, if, when querying a different cube, such dimension has a different name than in the source application.

The syntax supports an optional “AS” statement as follows:

```
*XDIM_ GETINPUTSET {ThatDimension} [ AS {ThisDimension}] [={member set}]
```

Example:

```
*XDIM_GETINPUTSET SOMEENTITY AS ENTITY=[SOMEENTITY].members
        *APP SOMEOWN
        *XDIM_MEMBERSET ACCOUNTOWN=METHOD
        *XDIM_MEMBERSET RPTCURRENCY=GROUP1
        *XDIM_MEMBERSET INTCO=Non_Interco
        *CRITERIA SIGNEDDATA<>0
*ENDXDIM
```

In the above example the members of the ENTITY dimension are extracted from the members of the SOMEENTITY dimension existing in the SOMEOWN cube.

The instruction *GO

In general, multiple sequential WHEN / ENDWHEN structures are processed in sequence for each record found in the source region. As a result, the following example might not work correctly:

```
//-------------------------------------------------------
*WHEN ACCOUNT
*IS UnitsSold
        *REC(FACTOR=GET(ACCOUNT=”Price”), ACCOUNT=Sales)
*ENDWHEN
*WHEN ACCOUNT
*IS Sales
        *REC(FACTOR=.08, ACCOUNT=SaleTaxes)
*ENDWHEN
//-------------------------------------------------------
```

The reason for this logic not working is that the value of account Sales held in memory is not the newly calculated one, but what found in the db before the new value is calculated. Sales, once calculated, must be posted to the db with a commit instruction, and then retrieved for the subsequent calculation of Sales Taxes, as follows:

```
//-------------------------------------------------------
*WHEN ACCOUNT
*IS UnitsSold
        *REC(FACTOR=GET(ACCOUNT=”Price”), ACCOUNT=Sales)
*ENDWHEN
*COMMIT
*WHEN ACCOUNT
*IS Sales
        *REC(FACTOR=.08, ACCOUNT=SaleTaxes)
*ENDWHEN
//-------------------------------------------------------
```

Obviously, a better way to write this logic would be to calculate SalesTaxes at the same time Sales are calculated, as show here:

```
//-------------------------------------------------------
*WHEN ACCOUNT
*IS UnitsSold
        *REC(FACTOR=GET(ACCOUNT=”Price”), ACCOUNT=Sales)
        *REC(FACTOR=GET(ACCOUNT=”Price”) * .08, ACCOUNT=SalesTaxes)
*ENDWHEN
//-------------------------------------------------------
```

However, there are cases of calculations where this is not a practical approach. For these situations, in order to avoid the inefficiencies of a COMMIT step, the instruction *GO can be used. Here is how the above logic would look like:

```
//-------------------------------------------------------
*WHEN ACCOUNT
*IS UnitsSold
        *REC(FACTOR=GET(ACCOUNT=”Price”), ACCOUNT=Sales)
*ENDWHEN
*GO
*WHEN ACCOUNT
*IS Sales
        *REC(FACTOR=.08, ACCOUNT=SaleTaxes)
*ENDWHEN
//-------------------------------------------------------
```

Any GO instruction defines the end of a logic section, more or less like a COMMIT instruction, in the sense that the logic is executed normally up to that point. However, unlike a COMMIT, no posting is performed. All generated results are merged with the original set of source records and the logic re-starts from the beginning of the record set, to process the following WHEN / ENDWHEN structures.

There is no limit to the number of GO instructions that can be inserted within a COMMIT section. However, since each GO generates a little bit of overhead, and requires an additional scansion of the record set, their number should be kept to the minimum.

One important thing to remember is the following: even if at a first glance it may seem that the GO instruction can be used in place of a COMMIT instruction, this is not quite true. All instructions that are COMMIT-specific (like for example XDIM_MEMBERSET) are still COMMIT-specific and not GO-specific. In other words you cannot redefine the data region to process for each GO instruction, but only for each COMMIT instruction. The GO instruction only sets a stop-and-go point between WHEN / ENDWHEN structures of the same COMMIT section, i.e. of the same data region.

The instruction *STORE_ORG

Sometimes it happens that a section of SQL-based logic needs to read the value of a parent member, to perform some subsequent calculation. (For example it may need to store a PL net profit into a BS account). This instruction allows to store in the WB table all parent members of a given dimension as specified in a property like PARENTH1, PARENTH2, etc.

The syntax is:

*STORE_ORG {dimension name} = {property name}

Example:

*STORE_ORG ACCOUNT = PARENTH1

The STORE_ORG instruction must be executed standalone is its own COMMIT section.

Important remarks:

Storing in the WB table records that OLAP will not ever see (parent member are re-calculate on the fly in the OLAP cube) is NOT a sound practice. Not only this may increase dangerously the size of the WB table, especially if the selected organization has many intermediate parents, but future versions of Everest might not tolerate this altogether (future data schemas might fail to process the cube if the fact table contains non-leaf members).

In reality, while it would work with the current version of Everest, duplicating in the database the values of OLAP parents is NOT a recommended way to use this feature. A much wiser approach is to use it store the values of ‘parent-like’ members, defining some generic property called MYPARENTH1 or PARENTS1 or the like, and using it to store members that for the cube are NOT parents in any real hierarchy.

This will also permit to store only the appropriate parents, skipping all non-required intermediate parents that a true org may include for other reasons. For example the user could store in a dummy org defined in property MYORG just the value of Net Income, skipping all other parents like Gross Profit, NPBT, etc. The logic would simply say:

*STORE_ORG ACCOUNT = MYORG

An even better approach is to use the instruction CALC_DUMMY_ORG, as below described. Such feature will enable the user to calculate the values of the parent members without posting them to the database.

The instruction CALC_ORG

A special instruction can be inserted between WHEN / ENDWHEN structures, to calculate the parent members of a selected hierarch. Its syntax and behavior are exactly the same of the STORE_ORG instruction (see above). The main difference from STORE_ORG is that CALC_ORG does not require to be written between COMMIT instructions, and does not enforce a posting of the data to the database, until a subsequent COMMIT is found.

Its syntax is:

*CALC_ORG {DimensionName} = {OrgProperty}

Example:

```
*WHEN…
        // some calculation……..
*ENDWHEN

*CALC_ORG ACCOUNT = PARENTH1

*WHEN…
        // some other calculation……..
*ENDWHEN
```

In the above example the results of the preceding calculations will be included in the calculation of the parents of PARENTH1 org.

The CALC_ORG instruction automatically enforces a double GO instruction (before and after its execution). For this reason no other instruction needs to be inserted between it and the preceding WHEN / ENDWHEN structures, nor the following ones.

The instruction CALC_EACH_PERIOD

The instruction CALC_EACH_PERIOD can be inserted within a commit section, to enforce an orderly calculation of the members of the time dimension. When this instruction is present, the source records are scanned one period at a time, in ascending order from the oldest to the youngest, and the results generated by each time period are merged with the rest of the record set, before the next period is calculated.

With this technique we can more easily and efficiently handle the typical calculations of financial reporting applications, where the results of one period are the inputs for the calculation of the next period.

Example:

```
//-----------------------------------------------------
*CALC_EACH_PERIOD

*WHEN ACCOUNT
*IS OPEN_BALANCE, MOVEMENTS
        *REC(ACCOUNT=CLOSING_BALANCE)
        *REC(ACCOUNT=OPEN_BALANCE,TIME=NEXT)
*ENDWHEN
 //-----------------------------------------------------
```

The above sample logic performs a carry-forward of the closing balance of each period into the opening balance of next period (Note the use of the new instruction NEXT. See below for more details)

The instruction *NO_PARALLEL_QUERY

SQL server automatically tries to run multiple queries in parallel, if possible. In case many k2logic executions are requested to the server simultaneously, this mechanism may end up eating up too much memory on the server, ultimately slowing down the process, or even generating an 'out of memory' error.

This situation can be controlled using the instruction:

*NO_PARALLEL_QUERY

This instruction will suppress the generation of parallel queries on the server. The instruction is specific to the commit section into which it is inserted.

The *JOIN( ) instruction

This instruction is a bit of an experiment in the direction of making our logic easier to define. With this instruction the logic can be linked to the values entered in some field of a user-defined table. Since we still do not provide a UI to support the maintenance of such tables, this feature should be considered experimental only, for the time being.

The syntax is:

*JOIN({tablename} , {dimension.property} [, {tablefield}] [, {selected fields}])

where:

{tablename}	is the name of the user-defined table
{dimension.property}	are the dimension and property that should be used to join the table with the Everest data.
{tablefield}	is the field in the selected tables on which the dimension property is joined
{selected fields}	are the fields in the table that are of interest in the current logic. If more than one, the list of fields must be enclosed in double quotes.

Example:

*JOIN(RulesTable, Account.Rule, Rule, "Destacc, Factor")

The above instruction will tell the logic to join the Everest data with the data contained in a table called RulesTable, performing a join of the property Rule of the account dimension and the field Rule of the RulesTable table. The fields to read in the RulesTable are DestAcc and Factor. Behind the scenes, the generated SQL query will look more or less as follows:

```
select
….., mbrACCOUNT.[RULE] AS [ACCOUNT.RULE],
RULESTABLE.[DestAcc] AS [RULESTABLE.DestAcc],
RULESTABLE.[Factor] AS [RULESTABLE.Factor]
INNER JOIN RULESTABLE on mbrACCOUNT.[RULE] = RULESTABLE.[RULE]
```

The parameters {tablefield} and {selected fields} are optional. If omitted, the {tablefield} is assumed to be the same as the property of the joined dimension, and the {selected fields} are assumed to be all fields in the table. In other words, the above example could have been written as follows:

*JOIN(RulesTable, Account.Rule)

With the above technique an account could be assigned a rule of behavior using a regular property. A separate table of Rules could then be maintained, defining in a generic way what a rule should imply in the execution of a logic.

Once the join has been defined, all fields in the joined table can be used anywhere in the WHEN/ENDWHEN statement to perform the appropriate calculations.

Example:

```
*JOIN(AccountRules,account.rule)

*WHEN *
*IS *
        *REC(FACTOR= AccountRules.Factor, ACCOUNT=AccountRules.DestAcc)
*ENDWHEN
```

Remark: by nature a join will only accept the values of the selected Everest region of data that have in the selected property a value corresponding to the joined field in the joined table. In other words, still referring to the above example, all records not having a valid Rule will be skipped, even if included in the selected region. This could also be considered as a way to filter the members of a dimension without using a XDIM_MEMBERSET instruction.

Support for FAC2 table

In Everest the OLAP cubes support an additional partition called FAC2. This extra MOLAP partition is used to allow for a fast cleanup of the Writeback (WB) table in large applications where the WB table increases its content quickly. The data source for this partition is an additional fact table similar to the regular fact table, called tblFAC2{AppName}. The trick is that, when the WB table needs to be emptied and its records should be moved in the fact table, the records can be moved into the (smaller) FAC2 table, and only the partition pointing to the FAC2 table is processed. This task is much faster than a full optimization of the cube and will only keep the application off-line for a very short period of time.

Everest supports the FAC2 table by default, and so does K2logic, when in presence of a version 4 of the database. In addition, K2logic is also able to support a FAC2 table in version 3, checking a parameter defining the presence of the FAC2 table in any give application. The parameter is HAS_FAC2 and should be set to YES in the application parameters accessible through the portal.

Writing directly in the FAC2 table

A special instruction can be entered in logic, triggering the posting of the results directly into the FAC2 table, if desired (see “Support for FAC2 Table” for more details on this table). The instruction is:

*WRITE_TO_FAC2

If this instruction is found in a given COMMIT section, the records to be posted will end up directly into the FAC2 table.

This mechanism can be helpful when a logic execution may generate a significant amount of new records, leading to a very fast increase in the content of the WB table, which in turn would cause a significant degradation of performance in all I/O activities.

Important remark: At the end of the logic execution a processing of the FAC2 partition will be required, in order to make the results of the logic available to the cube. This can be done adding an appropriate task at the end of the DTS package that executed this type of logic. Alternatively the PROCESS_FAC2 instruction can be used (see below).

The *PROCESS_FAC2 instruction

This instruction can be used to trigger the processing of the FAC2 partition of Everest cubes after a commit directly into the FAC2 partition has been performed. The instruction applies to a single commit section.

Example:

```
*WRITE_TO_FAC2
*PROCESS_FAC2

WHEN *
*IS *
   *REC(…..)
*ENDWHEN

*COMMIT
```

Technical note: Filtering the WB table on SOURCE field in SQL queries

During database optimization processes performed without putting the system off-line (something we do trough the FAC2 table), reading data from the Write back (WB) table may lead to incorrect results, unless these data are filtered for zero values in the SOURCE field.

K2logic SQL queries against the WB table are now automatically filtered for SOURCE=0, whenever the FAC2 table exists in the application.

Technical note: the format of the SQL query

The queries generated by SQL logic use a format that has proven scalable in larger installations involving many concurrent users. This format relies on a temporary table that is created and destroyed on the fly by the logic engine. Handling a temporary table has a minimal impact on smaller queries (a fraction of a second) but significantly improves the performance of concurrent and/or larger ones.

If desired, the query format can be simplified, and made it similar to the one used by prior versions of Everest (V4). This “older” format can be triggered using either one of the following methods.

1) From the portal add an application parameter named LOGIC_QUERY_MODE and give it the value USE_OLD.

2) Add the instruction *USE_OLD in all commit sections of your logics that should use the older query format.

In some special circumstances, the users asked to be able to modify the way SQL logic accesses the db. For these cases we have given them the option to build their own custom view they can store in SQL and instruct the logic engine to use such view.

The custom view must be called **ViewFact{Application name}**, like for example ViewFactFinance, and it may be enabled using either one of the following methods.

3) From the portal add an application parameter named LOGIC_QUERY_MODE and give it the value USE_VIEW.

4) Add the instruction *USE_VIEW in all commit sections of your logics that should use the older query format.

Enforced security and data status validation

Since 4.2 SP1, the logic engine supports native validation of user's security and data approval status for the data being modified. This means that the current user will not be able to modify data that are locked or which he does not have write access to.

This behavior has been introduced under request from various users that considered the original behavior (no security or status validation) incorrect. However, since this is a change to a behavior that some other existing users relied upon, we decided to give them the option to use logic the way they used to in previous versions.

To re-instate the original behavior the user can now set the following APPLICATION PARAMETERS to "Y" (Yes):

LOGIC_IGNORE_STATUS

LOGIC_IGNORE_SEC

The instructions *IGNORE_STATUS and *BLOCK_STATUS

By default the logic engine performs a check of the status of the data being written, and aborts the entire operation if any record should go in a region whose status is greater than zero.

This behavior can also be more accurately controlled with one of the following instructions:

*IGNORE_STATUS

This instruction is specific to a given COMMIT section and can be used in place of the application parameter LOGIC_IGNORE_STATUS (see above) to override the status validation just for a given portion of a logic.

*BLOCK_STATUS > n

This instruction can be used to control what status level is acceptable for posting. For example if the user enters this instruction:

*BLOCK_STATUS > 2

…the posting will succeed if the new records are being posted in a region where the status is 0 or 1 or 2, but it will abort if the status is 3 or higher.

The default value for this option is *BLOCK_STATUS > 0

This instruction is specific to a given COMMIT section

The instructions *IGNORE_SECURITY

By default the logic engine performs a check of the security of the region being passed to it, automatically dropping the members that are not accessible by the user running the logic.

This behavior can be more controlled with the following instruction:

*IGNORE_SECURITY

This instruction is specific to a given COMMIT section and can be used in place of the application parameter LOGIC_IGNORE_SEC (see above) to override the security validation just for a given portion of a logic.

Valid combinations of instructions, and other considerations

Instructions spanning multiple COMMIT sections

When a logic is broken in different execution sections through the insertion of one or more *COMMIT instructions, it can be considered in all practical terms like a set of independent logics executed in sequence.

However, some of the available instructions will apply to the entire logic file as a whole, and some others will be specific to the section in which they are entered. It is important to know the behavior of each instruction vis-à-vis the commit statement.

- **Instructions that must only be entered once in the entire logic file**

*FUNCTION {functionname}({Param1}[,{Param2}…]) = {Function Text} (*)
..or…
*FUNCTION / *ENDFUNCTION structure (*)
*INCLUDE {filename}(Param1,Param2,….) (wherever appropriate)
*LOCAL_CURRENCY
*LOGIC_BY = {dimensions list}
*LOGIC_PROPERTY = {property name}
*LOGIC_MODE = 0 | 1 | 2
*MEMBERSET({variable}, {member set in MDX format})
*PROCESS_EACH_MEMBER = {dimensions list}
*QUERY_SIZE = 0 | 1 | 2
*RUNLOGIC / ENDRUNLOGIC (once per pushed logic)
*SCOPE_BY = {dimensions list}
*SELECT ({variable}, {What}, {From}, {Where})
*SUB / ENDSUB
*SYSLIB {Includedfile} (Param1,Param2,….)
*XDIM_REQUIRED = {dimensions list}

- **Instructions that are specific to a section**

*ADD_DIM {dimension name}={value}[,{dimension name}={value},…]
*ADD_TABLE
*CALC_DUMMY_ORG {dim name}={property name}
*CALC_EACH_PERIOD
*CALC_ORG {dim name}={property name}[,DUMMY]

*CALCULATE_DIFFERENCE = 0 | 1
*CLEAR_DESTINATION
*COMMIT_EACH_LEVEL = {Dimension name}
*COMMIT_EACH_MEMBER = {Dimension name}
*COMMIT_MAXMEMBERS
*DESTINATION {Dimension name}={MemberSet}
*DESTINATIONAPP = {app name}
*GO
*JOIN({tablename} , {dimension.property} [, {tablefield}] [, {selected fields}])
*LAST_MEMBER {Dimension name} = {Member }
*LOOKUP / ENDLOOKUP
*MEASURES = {dimension}
*NO_PARALLEL_QUERY
*OLAPLOOKUP [{Application name}]
*PROCESS_FAC2
*PUT( )
*PUTSCOPE_BY
*QUERY_FILTER = {member set}
*QUERY_TYPE = 0 | 1 | 2
*REC( )
*RENAME_DIM {dimension name}={value}[,{dimension name}={value},…]
*RUN_STORED_PROCEDURE={stored procedure name}([{params}])
*SKIP_DIM= {dimension name}[,{dimension name},…]
*STORE_ORG
*TEST_WHEN
*USE {filename}
*USE_UNSIGNEDATA
*WRITE_TO_FAC2
*WHEN / ENDWHEN
*XDIM_ADDMEMBERSET {dimension} = {members set}
*XDIM_DEFAULT {Dimension name} = {Members Set}
*XDIM_FILTER {Dimension name} = {Members Set}
*XDIM_GETINPUTSET / *ENDXDIM structure
*XDIM_GETMEMBERSET / ENDXDIM structure (****)
*XDIM_MAXMEMBERS {Dimension name} = {Number of members}
*XDIM_MEMBER {Dimension name} = {Member1} [TO {Member2}]
*XDIM_MEMBERSET {Dimension name} = {Members Set} (**)
*XDIM_NOSCAN {Dimension name} = {Members Set}

Instructions that can only be entered in the "add formula" text box of K2DTSRunLogic

*WRITE_TO_FILE = {filename}

Important remarks:

(*) If multiple instances of the same user-defined function (*FUNCTION) are entered in the logic file, the FIRST occurrence will prevail on the subsequent ones. This behavior can be used to redefine a function using the "add formula" text box in the DTSRunLogic task.

(**) The instructions XDIM_MEMBERSET and XDIM_GETMEMBERSET will behave in a special way if written in the "add formula" textbox: the generated set of members will not apply to the current section of logic up to the next commit, but to all sections in the logic. In other words, these two instructions, when used in the "add formula" text box, will re-define the initial selection of members for the entire logic. All other *XDIM instructions found inside individual logic sections will still behave normally (they will redefine the selection for the current section only), but they will use as default selection the selection defined by the XDIM instruction contained in the "add formula" text box. .

- ***Valid instructions in SQL logic***

WHEN and LOOKUP instructions can be run from the DEFAULT modeling logic as well as from any other logic executed from a Data Manager tasks. Also, they can also appear in multiple COMMIT sections within the same logic. However, only one LOOKUP instruction can be inserted in a given COMMIT section. On the other hand, as many WHEN/ENDWHEN structures as desired can be inserted in the same COMMIT section, either nested or not nested.

Inside a given COMMIT section, the LOOKUP and WHEN instructions can be combined with most of the other currently valid logic instructions. Here is, in alphabetical order, the list of the instructions that could make sense in the context of a WHEN section:

*ADD_DIM
*ADD_TABLE
*CALC_DUMMY_ORG {dim name}={property name}
*CALC_EACH_PERIOD
*CALC_ORG {dim name}={property name}[,DUMMY]
*CALCULATE_DIFFERENCE
*CLEAR_DESTINATION
*DESTINATION
*DESTINATION_APP
*FOR / *NEXT
*FUNCTION
*GO
*INCLUDE
*JOIN
*LAST_MEMBER
*LOCAL_CURRENCY
*LOOKUP / ENDLOOKUP
*MEMBERSET
*NO_PARALLEL_QUERY
*OLAPLOOKUP
*PROCESS_EACH_MEMBER
*PROCESS_FAC2
*RENAME_DIM
*SCOPE_BY
*SKIP_DIM
*SELECT
*SUB / ENDSUB
*SYSLIB
*TEST_WHEN
*USE_UNSIGNEDATA
*WRITE_TO_FILE
*WRITE_TO_FAC2
*XDIM_ADDMEMBERSET
*XDIM_DEFAULT
*XDIM_FILTER
*XDIM_GETINPUTSET / *ENDXDIM
*XDIM_GETMEMBERSET / *ENDXDIM
*XDIM_MAXMEMBERS
*XDIM_MEMBER
*XDIM_MEMBERSET
*XDIM_NOSCAN
*XDIM_REQUIRED

All other instruction would not make sense in a COMMIT section containing a WHEN instruction. Here is the list of such invalid instructions:

*ADD / *ENDADD
*BEGIN / *END
*COMMIT_EACH_LEVEL
*COMMIT_EACH_MEMBER
COMMIT_MAXMEMBERS (*)
*LOGIC_MODE
*LOGIC_BY
*LOGIC_PROPERTY
*MEASURES
*PUT( )
*PUTSCOPE_BY
*QUERY_FILTER
*QUERY_SIZE
*QUERY_TYPE
*RUNLOGIC / ENDRUNLOGIC
*RUN_STORED_PROCEDURE
*SELECTCASE / *ENDSELECT
*STORE_ORG
*USE

(*) in SQL logic the sets of 'maxmembers' are already committed individually by default, and the instruction would be redundant.

Last but not least, it must be remembered that any MDX calculation defined in a section containing a LOOKUP or WHEN instruction, will be ignored.

Localization

The Logic module fully supports localization. If used in conjunction with the appropriate resource DLL file (its name is EvResSvrLgc.DLL), all the returned messages will show up in whatever language the resource file has been compiled with.

Remarks:

Many messages returned by the Logic module are generated directly by Microsoft SQL server or Analysis server. Those messages will remain in the language of whatever version of those products has been installed on the server (normally English).

All messages written in the debug file will remain in English. We did not feel like those messages were worth translating, as most of them represent untranslatable scripting instructions, MDX and SQL queries, and logs of posted records.

Although the latest version of the logic module is intended to work with the currently commercial release, it remains fully backwards compatible, and will still work correctly with any prior version of Everest. In case no resource file is found, the Logic module is still able to generate its messages correctly, albeit in English only (On these grounds, the English version of the logic module could even be installed without a resource file).

The EvDTSRunLogic task

As previously mentioned, the product comes with a DTS version of the Logic Module that can be used from within the Data Manager module.

A very good reason to support modeling logic execution as a DTS task is that there are many cases in which some calculations can only be executed in a batch mode, after the appropriate data have been entered into the cube. For example, an elimination procedure is only worth being executed after all intercompany

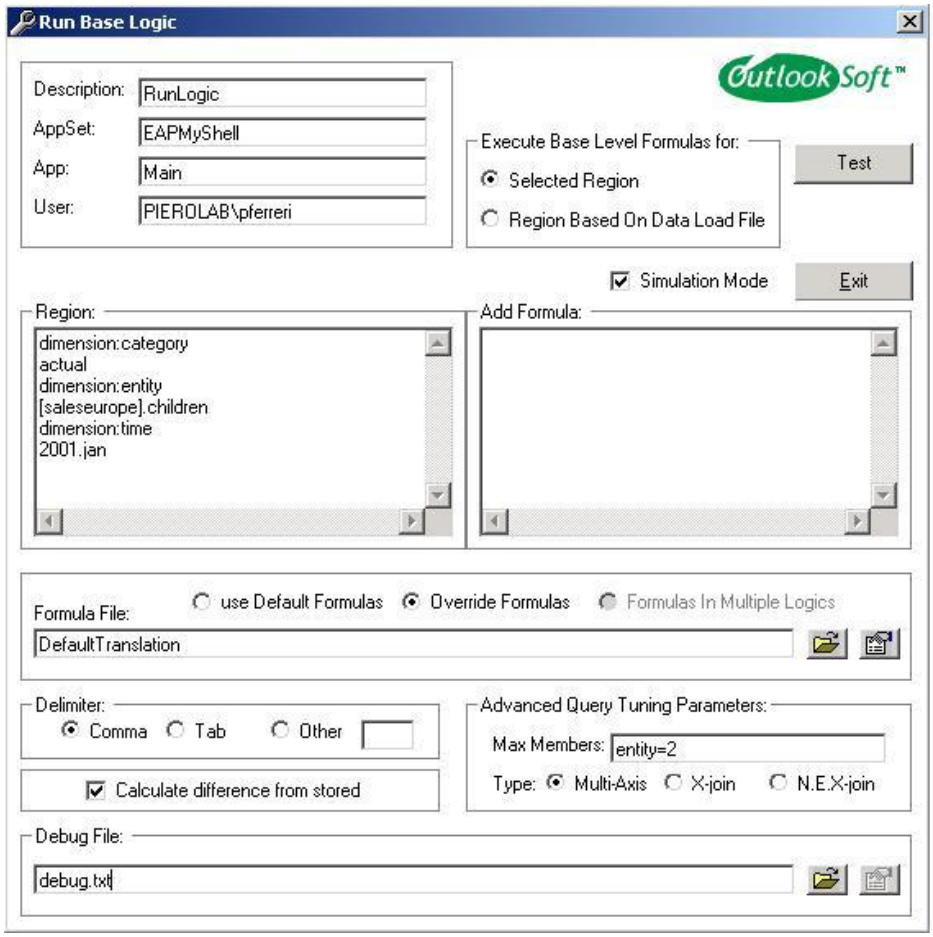
declarations have been entered by all companies, or a currency conversion must be re-executed after some exchange rates have been modified. For performance reasons, it may also be wiser to wait until all correct data have been entered, before running a given modeling logic.

Another valid reason why we may need this task is that Data Manager packages that import data writing directly into the fact table will not trigger any logic execution after data have been imported. For applications where the execution of the default logic is required at the end of this type of imports, the user needs to include a call to EvDTSRunLogic as last step after the cube has been processed, to perform the correct calculations.

The EvDTSRunLogic user interface

The UI of EvDTSRunLogic task supports a fair number of settings that can be used to control the action of the task. Most of these settings represent a duplicate of some of the instructions that can be written directly in the logics to execute (see above). If these instructions exist in the logic, the settings shown in the UI will be overridden. The settings of the UI can be considered defaults that explicit logic settings could override.

Here is a view of the main screen of EvDTSRunLogic, and a description of all the user-definable settings:



Description, Appset, App and User

These are the usual fields required by all our DTS tasks in order to be able to login into one of our applications.

Execution Mode

The module can be run against

- a selected region
- a selected file of data

In case the option for running against a region is chosen, the textbox to enter the desired data selection is shown, like in the above example.

Region

A region (selection) of data must be entered in the format we use for other Data Manager tasks, that is:

DIMENSION: {dimensionname}
{members set}

The member set in this case can also be represented by an MDX expression like:

DIMENSION: entity
[SalesEurope].children

Region in data file

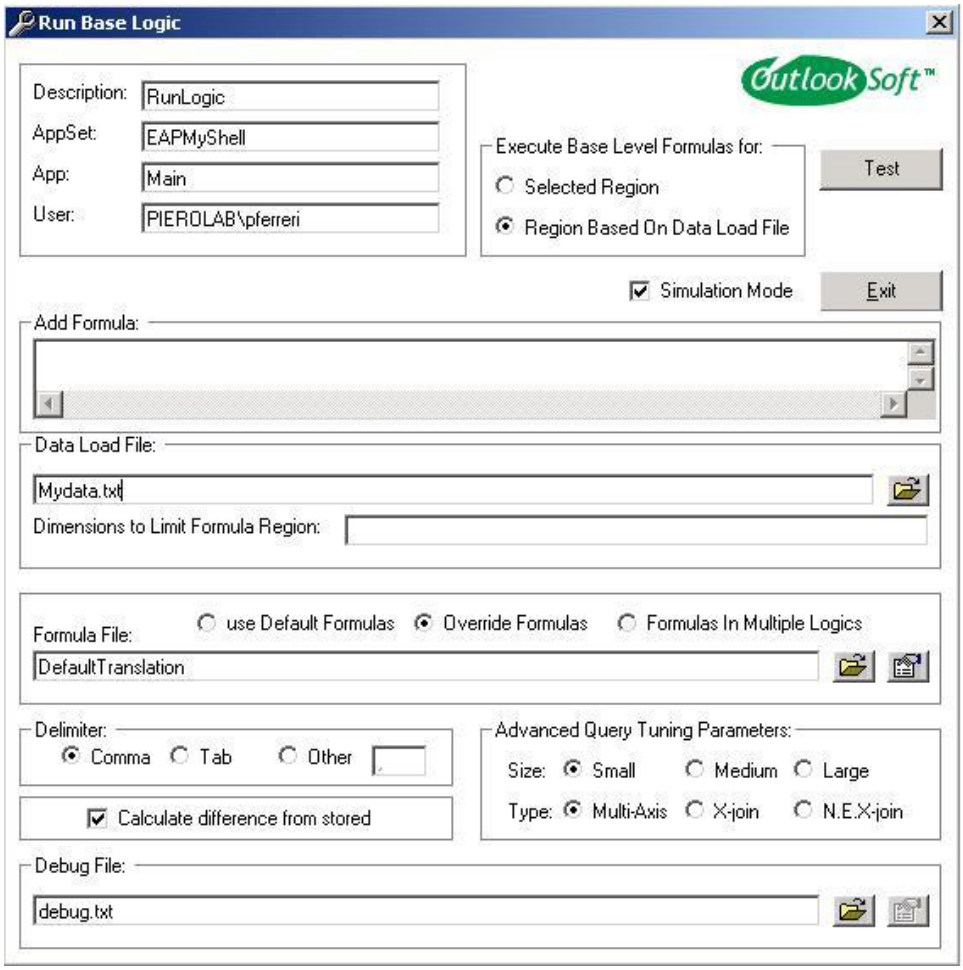
Although not evident, an acceptable syntax for the region is also the name of a file containing the text defining the selection. For example:

C:\Everest\Webfolders\NEWSHELL2\Main\DataManager\SelectionFiles\MySelection.ESF

This syntax can be used in packages that use the reserved keyword %SELECTIONFILE% (see Data Manager documentation for an explanation of this keyword).

Data file

In case the data file mode is selected, the user can pass the name of a data file to scan, in order to build the correct query. When this mode is selected, the main screen will look as follows:



The data load file the Logic Module will scan in order to decide where to execute the logic must be in the usual ready-to-load format that Data Manager would accept: a first record with the dimension names followed by one record per cell, like in this example:

CATEGORY,TIME,PRODUCT,ACCOUNT,ENTITY,CURRENCY,AMOUNT
ACTUAL,2000.JAN,OTHER,ACCPAY,SALESJAPAN,LC,-807.990550994873
ACTUAL,2000.JAN,OTHER,ACCPAY,SALESJAPAN,LC,-865.849018096924
ACTUAL,2000.JAN,OTHER,ACCPAY,SALESJAPAN,USD,-807.990550994873

Dimensions to limit formula region

In data file mode, another field is shown to the user. This field can be used to control the criteria the Logic Module should use to restrict the scope of the generated query. If for example, the user enters in this field "CATEGORY,TIME", the logic will be executed only for all different category-time combinations found in the load file (in the above example only for actual-2000.jan).

If the field is left blank, the Logic Module will default to:

"CATEGORY, TIME, ENTITY, PRODUCT, CURRENCY".

Use default formulas or override formulas

These options allow the user to control the logic to run. "Use default formula" will default to the logic DEFAULT.LGX. In case "override formulas" is selected, the user has the possibility to enter the name of a logic of his choice.

Omitting the file extension will default it to LGX. Omitting the file path will default it to:

{webfolders}\{appset}\AdminApp\{App}\

Formulas in Multiple Logics

The last radio button in the above picture activates the multi-logic mode, by which different logics can be assigned to different regions of data.

When this option is selected, the UI displays a text box where to enter the names of the dimensions to use in order to derive the name of the logic files to use.

In this logic mode, the Logic Module will scan the data file for all members belonging to the dimensions entered in the displayed text box, and will search for the value of a property called “LOGIC”, and will use that property to build the logic name.

For example, if in the text box the user enters “ENTITY”, and the load file contains data for entity SalesItaly, and SalesItaly has in the “LOGIC” field the value “LOGIC_XYZ”, the logic used against the data of entity SalesItaly will be LOGIC_XYZ.LGX.

Remark: if multiple dimensions are entered, the contents of the “LOGIC” fields will be concatenated in the user-entered order, to build the name of the logic file.

Empty Logic

In “override formulas” mode, the DTSRunLogic task must be given the name of a valid logic to execute, otherwise it will return an error. However, there may be cases when the entire set of executable instruction is defined in the Formula Script property, and no logic file is really needed. In this situation the user should build a logic file containing no instruction, and pass the name of this logic to the DTSRunLogic task. To avoid this complication, the keyword EMPTY can be passed, as the name of the logic to execute, even if no such logic exists.

If the name of the passed logic is EMPTY, the logic module will not look for a logic named EMPTY.LGX, but it will assume that a logic with no executable instruction is to be used.

Simulation Mode

When this check box is activated, the logic will be executed but the resulting values will not be posted to the cube. This mode can be used to debug the logic without impacting the stored values.

Delimiter

In “data file” mode, this parameter can be used to specify the delimiter used in the file to scan. Both in “data file” and in “region” mode, the field is used internally by the module to delimit the generated records.

The default delimiter is a comma.

Advanced query tuning parameters

These settings can be used to control the size and the format of the generated queries.

The size of the query is only relevant in case the “data file” mode is selected, and should be kept to “large” for a faster execution. Smaller sizes could be selected only if the large queries take too much memory. The selection of the size of the query will (should) have no impact on the results of the logic calculations.

The type of query can also have a beneficial (or negative) impact on performance, with the exception of the NonEmptyCrossJoin mode, that should only be used in special cases as already above described.

The field “max members” allows the user to restrict the number of members to process in a query. If the number is exceeded, the query is broken in multiple smaller queries. The syntax is:

{dimension} = {number of members}[,{dimension} = {number of members}]

If this fields is left blank, the default value will be ENTITY=1.

This field corresponds to the XDIM_MAXMEMBERS instruction available in the logic script.

Add Formula

When the user selects a logic type different from default (“override formula” mode), an additional piece of logic could be written directly in the “Add formula” text box. This piece of logic will be added at the top of the selected logic, and the resulting combined logic will be processed as one piece

This feature can be used in Data Manager to create “dynamic logics” that can be modified at run time by the user, passing the correct answers to some prompt activated by Data Manager task ModifyPkg.

For example, the user might be prompted for a percentage to use in the calculations, or the name of a file to include in the logic, or the type of rate to use for a given currency conversion, etc.

Important remarks:

If the text box contains expressions that must be transformed into executable instructions by the validation process (for example a FOR/NEXT loop), this can be enforced writing the name of the logic to execute with the extension LGF. The presence of the LGF extension will force a validation of the logic at run time. The generated LGX file will be kept in memory and executed without being saved to disk.

The instructions XDIM_MEMBERSET and XDIM_GETMEMBERSET, will behave differently, if written in the “add formula” textbox: the generated set of members will not apply to the first section of logic only

(up to the first commit), but to all sections. In other words, these two instructions, when used in the "add formula" text box, will re-define the selection of members for the entire logic.

Difference from stored

This option can be selected to speed up the posting of the calculated values into the database. When a calculated value is sent to the Everest data-write engine, Everest in reality needs to write in the database just the difference between what is already stored and the new value. The calculation of the difference takes time. If such calculation is performed directly by the Logic Module at the time of executing the logic, it is possible that the subsequent posting time will be reduced.

This option defaults to "yes".

Debug File

In case a valid debug file name is entered in this field, the module will save in such file an audit of its execution, showing the details of all the generated queries and the resulting calculated values. In case the file cannot be created (because invalid or in use by another process) the logic will run normally, but no debug file will be created.

If no path is entered, this will be defaulted to the TempFiles directory for the current user.

Ability to activate or disable a logic execution from DTS

The custom task EvDTSRunLogic.DLL supports a property named RUNTHELOGIC. This property can be set to true ("1") or false ("0") using a TASK( ) instruction from the K2ModifyPkg. Through this property the user can decide at run-time whether a logic should be executed or not, for example, after the completion of an import task.

Example:

In the script of K2ModifyPkg, the following instructions would let the user decide to run a RunLogic task, after importing a file of input data.

```
PROMPT(CHECKBOX,%RUN%,"Run the default logic after import")
TASK(K2RUNLOGIC,RUNTHELOGIC,%RUN%)
```

(In the above example the new PROMPT(CHECKBOX) instruction is used. In prior versions of Everest the simpler PROMPT(TEXT) instruction could be used. The user should manually enter 0 or 1 in the textbox).

Remark: The property RunTheLogic does not need to be explicitly set to true, to enable the logic execution. Any package that ignores this property will run correctly, as if this property did not exist. This makes the feature compatible with older packages.

Appendix

*ADD / *ENDADD	X		X	
*ADD_DIM	X		X	X
*ADD_TABLE	X			X
*BEGIN / *END	X		X	
*BLOCK_SECURITY	X		X	X
*CALC_DUMMY_ORG	X			X
*CALC_EACH_PERIOD	X			X
*CALC_ORG	X			X
*CALCULATE_DIFFERENCE	X		X	X
*COMMIT	X		X	X
*COMMIT_EACH_LEVEL	X		X	X
*COMMIT_EACH_MEMBER	X		X	X
*COMMIT_MAXMEMBERS	X		X	
*CLEAR_DESTINATION	X			X
*DESTINATION	X			X
*DESTINATION_APP	X		X	X
*FOR / *NEXT		X	X	X
*FUNCTION		X	X	X
*FUNCTION / *ENDFUNCTION		X	X	X
*IGNORE_SECURITY	X		X	X
*INCLUDE		X	X	X
*GO	X			X
*JOIN	X			X
*LAST_MEMBER	X		X	
*LOCAL_CURRENCY		X	X	X
*LOGIC_BY		X	X	X
*LOGIC_MODE	X		X	
*LOGIC_PROPERTY		X	X	X
*LOOKUP / ENDLOOKUP	X			X
*MEASURES	X		X	
*MEMBERSET	X		X	X
*NO_PARALLEL_QUERY	X			X
*PROCESS_EACH_MEMBER		X	X	X
*PROCESS_FAC2	X		X	X
*PUT( )	X		X	X
*PUTSCOPE_BY	X			
*OLAPLOOKUP	X			X
*QUERY_FILTER	X		X	
*QUERY_SIZE	X		X	
*QUERY_TYPE	X		X	
*REC( )	X			X
*RENAME_DIM	X		X	X
*RUNLOGIC / ENDRUNLOGIC		X	X	X
*RUN_STORED_PROCEDURE	X			
*SCOPE_BY		X	X	X
*SKIP_DIM	X		X	X
*SELECT ( )		X	X	X
*SELECTCASE / *ENDSELECT	X		X	
*SUB / ENDSUB		X	X	X
*SYSLIB		X	X	X
*TEST_WHEN( )	X(**)			X
*USE		X	X	
*USE_UNSIGNEDDATA	X			X
*WHEN / ENDWHEN	X			X
*WRITE_TO_FILE		X	X	X
*WRITE_TO_FAC2	X		X	X
*XDIM_ADDMEMBERSET	X		X	X
*XDIM_DEFAULT		X	X	
*XDIM_FILTER	X	X (*)	X	X
*XDIM_GETINPUTSET / *ENDXDIM	X		X	X
*XDIM_GETMEMBERSET / *ENDXDIM	X		X	X
*XDIM_MAXMEMBERS	X		X	X
*XDIM_MEMBER	X	X (*)	X	X
*XDIM_MEMBERSET	X	X (*)	X	X
*XDIM_NOSCAN	X			X
*XDIM_REQUIRED		X	X	X

(*) Global when entered in the Formula Script of the DTS task
(**) Actually by each GO within a COMMIT