CONVERTING CALC FORMULAS TO MDX IN AN ESSBASE OUTLINE

Hyperion®

A HYPERION WHITE PAPER

April 2004

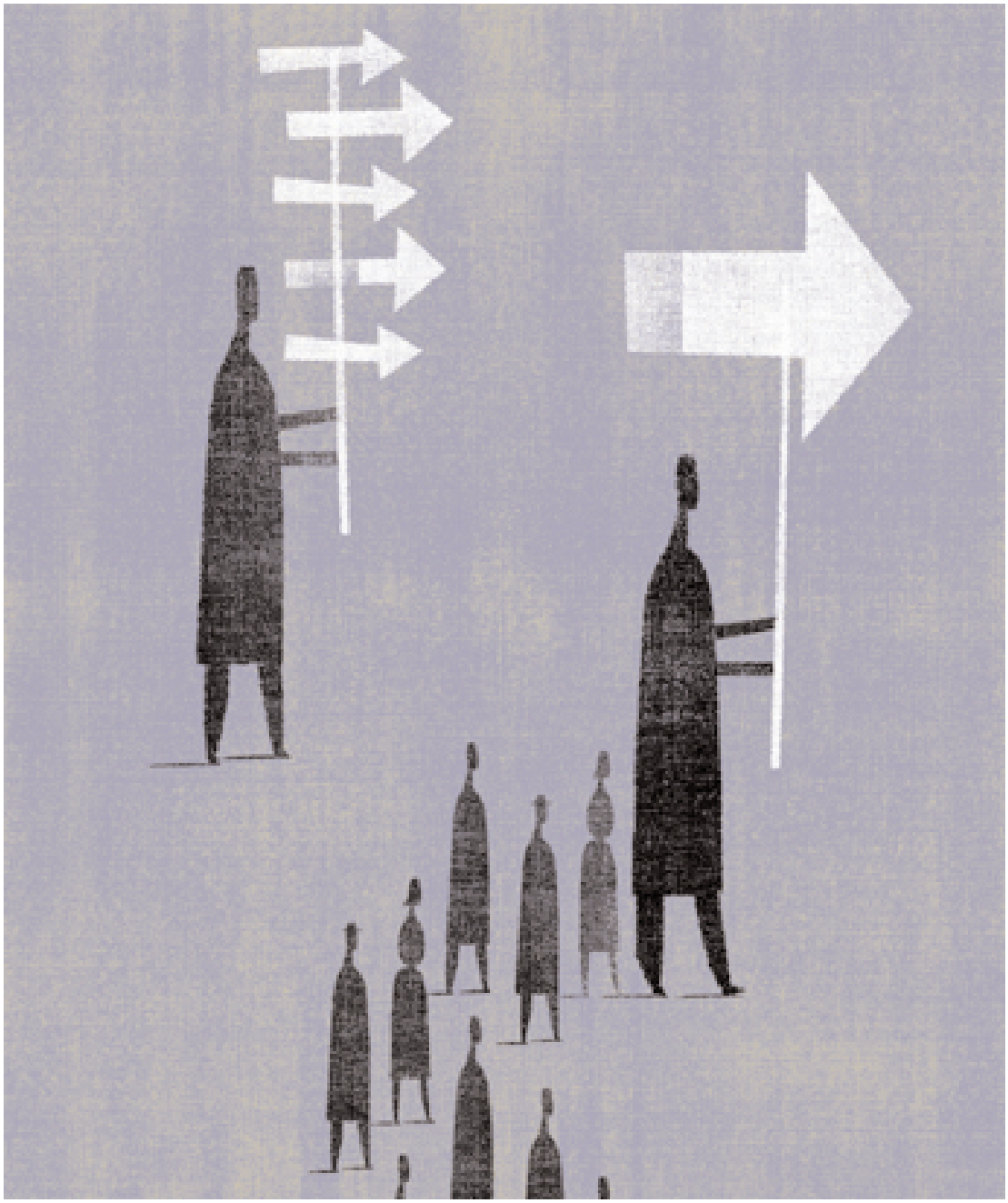

TABLE OF CONTENTS

Hyperion Essbase 7.1 introduces a new storage mechanism to persist a multidimensional cube, called aggregate storage. An aggregate storage database differs from a block storage database in a number of ways. One primary difference relates to member formulas, which are the focus of this document. To write formulas for block storage outlines, Essbase provides a set of calculation functions and operators. For the purpose of this document, this set of functions and operators and the attendant syntax is referred to as the Calc language.

OVERVIEW

The Calc language cannot be used to write member formulas for aggregate storage databases. To write formulas for aggregate storage outlines requires the MDX language. The MDX language and the Calc languages are both defined in the *Technical Reference*. The purpose of the current document is to provide support for rewriting Calc formulas in MDX for outlines that have been migrated from block storage to aggregate storage.

This document provides an overview of the key elements of the calculator and MDX languages comparing and contrasting them, both syntactically and semantically, as appropriate. Function categories in either language are explored in great detail with the intent of enabling the reader to use this document to accomplish the process of converting formulas from a block storage to an aggregate storage outline or vice-versa. Since aggregate storage outline formulas are always computed dynamically, they are comparable to dynamic formulas in the block storage model. Note that the MDX language can be used to describe queries for both aggregate and block storage databases and the calculator language can be used to construct calculation scripts for block storage databases. However, this document addresses neither of these uses of the languages. It focuses just on the member formulas define in the outline.

Before beginning this document you should have basic familiarity with the workings of aggregate storage outlines in Essbase. See Volume IV of the *Database Administrator's Guide*, which discusses all aspects of aggregate storage.

DIFFERENCES BETWEEN BLOCK STORAGE AND AGGREGATE STORAGE OUTLINES

The differences between aggregate storage outlines and block storage outlines affect the writing of member formulas in a number of ways besides the language used:

- While a block storage outline allows formulas to be attached to any member in the outline, aggregate storage outlines require that formulas be attached only to members in the dimension explicitly tagged as accounts.

- The storage characteristics of a member and hence all its associated cells are defined in a block storage outline through dynamic calc (and store) attributes, and stored attributes. Such attributes do not exist in an aggregate storage outline. Upper level members along an explicitly tagged accounts dimension and members with formulas attached to them are always calculated dynamically in such a database.
- Built-in financial calculation functions with time intelligence such as time-balance, time series and expense reporting are available only in block storage outlines.
- In block storage outlines, calculation order is dependent on the order in which members appear in the outline whereas formulas are executed in order of their dependencies in aggregate storage outlines. In addition, calculation order in the event of ambiguity in the evaluation of a cell, and two-pass calculation tags are not required in an aggregate storage outline because formulas can only be attached to accounts dimension members, and all members associated with formulas are calculated dynamically at query time.
- The layout of block storage outlines and the separation of dimensions into dense and sparse has an effect on the semantics of certain calculations, giving rise to concepts such as top-down calculation mode, cell and block calculation mode, and create-blocks on equations. The simplicity of the aggregate storage outlines, which do not separate dimensions into dense and sparse, makes such concepts obsolete.

ELEMENTS OF CALC AND MDX LANGUAGES

The fundamental element in both languages is a *member* in a dimension. A member whose name contains spaces or other special characters is enclosed in double quotes in Calc, whereas MDX requires such names to be enclosed within square braces; for example:

```
[100-10]   /* MDX */.
"100-10"   /* Calc */
```

In Calc, a member that has shared counterparts in the same dimension always refers to the non-shared, tagged version of the member (relevant for the purpose of metadata functions). The concept of shared members does not exist in aggregate storage databases.

SETS

A collection of one member from one or more dimensions is referred to as a cross member in Calc. The same entity is referred to as a tuple in MDX, with the difference that a tuple is an ordered collection of members. This difference is particularly relevant in terms of the next fundamental element of the languages, sets of tuples. Examples of a tuple and a cross member are:

```
([100-10], East) /* MDX tuple */
"100-10"->East /* Calc */
```

A set is an ordered collection of zero, one, or more tuples with the rule that the members appear in the same dimensional order in each tuple. Consequently operations such as the head or tail of a set, current tuple, and *nth* tuple of a set are all meaningful. A set can contain more than one occurrence of the same tuple.

Calc has no corresponding element, though a list comes close to representing a collection of members or cross members. However, there is no rigor to how a list is composed (order or dimensionality requirements) and hence it is not possible to identify the contents of a list by position.

Calc sets can be constructed only by using metadata operations on outline members or by explicit enumeration of members and cross members, referred to as explists. Calc sets do not necessarily subscribe to the definition rules of an MDX set. MDX sets, on the other hand, can also be obtained through data-based operations, referred to as filters.

An explicitly enumerated *explist* in the Calc language can be converted into an MDX set by ensuring that each cross-member in the explist has the same dimensionality (by substituting any missing dimensions in the cross-member by the root dimension member) and that the order of enumeration of members within each cross member is the same.

A cell is identified in both languages by a tuple or a cross member that has representation from every dimension in the outline. When a cell address is incompletely specified, both languages use dimension level members along dimensions that are not represented in a cell address.

In MDX, the difference between a member, and a cell value indexed by a member, are very clear. The MDX Value function is used if a cell value is desired in a particular context. In contrast, most Calc functions implicitly assume that the value of an expression is desired and there is little control over the function behavior. Notable examples are @SHIFT and its variants. See @SHIFT in the Calc to MDX Functions table for more information.

MDX supports set union, intersect and except operations. Calc supports the same set of operations on lists. The rigorous definition of sets requires careful handling of duplicate tuples whereas such requirements must be explicitly handled by application developers in Calc through the use of @REMOVE and @MERGE functions.

The MDX CrossJoin function returns the cross product of two sets, and includes predefined semantics on which set is iterated over faster. The closest analog in Calc is the @RANGE function. However @RANGE operates over a single member and set, unlike CROSSJOIN, which can operate on two sets.

OTHER LANGUAGE ELEMENTS

Both languages include procedural elements in the form of IF-ELSEIF-ELSE constructs, although the syntax differs. MDX exposes the functionality in the form of a IIF construct when there is exactly one IF and one ELSE condition. Nested IF-ELSE-IF logic requires the use of nested IIFs. Additionally, two flavors of a CASE-WHEN-THEN construct are also available in MDX to be used where multiple IF, ELSE clauses are required. Calc has a single IF-ELSEIF-ELSE-ENDIF construct.

While methods (such as Children, Descendants etc) can be invoked syntactically as a function call in both languages, MDX, additionally allows an object-oriented invocation style. For example:

```
Children(East) /* Valid in Calc
and MDX */

East.Children /* Valid in MDX
only */
```

The assignment construct exists only in Calc. Thus in a block storage outline, a formula attached to a member A can influence the values of cells corresponding to a different member B. While the notion of indirect assignment is not relevant to dynamically calculated members, the difference is worth mentioning. MDX formulas return a value only for cells associated with the member.

Calc exposes constructs such as SKIPMISSING, SKIPZERO, SKIPNONE and SKIPBOTH to control how zero and missing cell values in a cube should be treated in certain functions, such as @AVGRANGE. While MDX does not directly support these elements, it is possible to achieve the same result using either the Filter, IIF, or CoalesceEmpty functions. Filter takes a set as input and produces an output set consisting of only those members that satisfy the filter condition.

A dynamically calculated outline member formula is evaluated at query time for every cell requested in the query and associated with the member. Thus the context for the execution of a formula is always a fully specified cell coordinate from the query. The precedence rules and arithmetic rules involving missing and zero cells are identical between MDX and Calc.

The Calc language can be extended through the definition of user defined macros and java functions. The list of native functions available in MDX is much higher and richer than the functions in the Calc language. However, the ability to extend the Calc language enables almost any expression that can be constructed in MDX to be achieved in Calc, although not efficiently. In particular, the ordering semantics of sets is difficult to simulate in Calc. Custom extensions to MDX will, however, be possible in a future release of Essbase.

FUNCTION CATEGORIES

The MDX functions that can be used to construct outline formulas can be categorized based on their return values, as follows:

- Member
- Tuple
- Set
- Number
- Dimension (root member of a dimension)
- Layer (generation or level number/name)
- String
- Boolean.

Calc functions also have similar return types and subscribe to the same categorization, except for return types tuple and set as described earlier.

The functionality available in MDX and Calc is compared from the perspective of a user familiar with Calc and attempting a translation of outline formulas written for block storage outlines to aggregate storage outlines.

FUNCTIONS WITH RETURN TYPE MEMBER

Both languages support outline based operations that return a single member, such as Parent, Ancestor, Lead (Next) and Lag (Prior). Functions dealing with attribute dimensions,

namely, @ATTRIBUTEBVAL, @ATTRIBUTESVAL and @ATTRIBUTEVAL exist only in Calc. MDX exposes attributes as properties. Thus member properties can be used in place of the corresponding Calc functions. See the description of these functions in the "Calc to MDX Functions" table for examples.

@SHIFT in Calc allows elements identified by relative positions in an input list to be returned. Logically, @SHIFT, (@NEXT, and @PRIOR perform a metadata operation, namely, shifting in the right direction, based on the current context of execution and the input set. The result for each of these functions is a cell value.

An important difference between the MDX and Calc languages is illustrated by the @SHIFT function. Calc implicitly assumes, based on the current context of execution, that the value of the shifted member is desired. In contrast, MDX keeps the distinction between members (or sets) and cell values distinct. Thus, the variants of @SHIFT in MDX, namely, the Lead and Lag functions, return a member or a tuple. To obtain the cell value, the Value function must be explicitly specified.

The Calc language allows the defaulting of the range input for functions to be the leaf level members of an explicitly tagged time dimension. This defaulting mechanism does not exist in MDX.

MDX, additionally, has functions such as OpeningPeriod, ClosingPeriod and ParallelPeriod, FirstChild, LastChild, FirstSibling, and LastSibling with no corresponding counterparts in Calc. Alternatively, time balance logic can be implemented in block storage outlines by the use of Accounts tags TBFIRST and TBLAST.

To retrieve a cell value associated with a member, Calc has functions such as @MDPARENTVAL, @MDANCESTVAL and @MDSHIFT. All these functions take as input a member and return the value of a cross member consisting of parent/ancestor/shifted member along all dimensions (based on current member from each dimension) specified as arguments. To retrieve a cell value associated with a member in MDX, you explicitly construct a tuple with overriding parents/ancestors of members then apply the Value function. See the description of @MDPARENTVAL in the "Calc to MDX Functions" table for an example. The function @XREF has a similar purpose except that the value fetched is from a cube identified explicitly as one of the function arguments. MDX does not have the equivalent of an XREF function in this release.

FUNCTIONS WITH RETURN TYPE TUPLE

Although the notion of a cross member exists in Calc, operations to isolate individual members of the cross member are not possible. In contrast, MDX allows specific members to be isolated from a tuple by position using the Item function.

FUNCTIONS WITH RETURN TYPE SET

A rich set of metadata functions that return a set of members based on hierarchical relationships in the outline are present in both languages. In MDX, the resulting set subscribes to ordering and dimensionality rules, whereas in Calc a set is merely a collection of (cross) members in Calc. Most metadata functions in Calc have a counterpart, directly or indirectly, in

MDX, with the exception of the @MATCH function, which returns members from a dimension whose name conforms to a regular expression pattern.

Functions that have a direct counterpart recognizable simply by the same name in either language are listed in the "Calc to MDX Functions" table but are not described in any detail in this document. Certain functions require indirect translation as explained below. For a complete list of functions, both with direct or indirect translation, see the "Calc to MDX Functions" table.

The versions of metadata functions with an i prefix (for example, @ICHILDREN) return the member on which a method is requested along with the resulting members in the same set. In contrast, MDX requires an additional union operation of the member with the result set of the operation. See @ICHILDREN in the "Calc to MDX Functions" table.

The @LSIBLINGS and @RSIBLINGS functions must be translated indirectly into MDX. The following example shows a Calc function and its MDX equivalent:

```
@lsiblings(mbr) /* Calc */

MemberRange(FirstSibling(mbr),
Lag(mbr, 1)) /* MDX */
```

The @RELATIVE function in Calc can be replaced by the MDX Descendants function.

The @XRANGE function is unique to Calc and is an extension of the range functions, : or :: operators that are available in both languages. The :: Calc operator maps to the MemberRange function in MDX. The : Calc operator maps to MemberRange with an additional layer argument. @XRANGE is a multidimensional version of the @RANGE function and creates a result set by identifying the level of the arguments and pruning the result set to include only the argument members and the list of members that are, in terms of outline order, between them.

@CURRMBRRANGE takes as input a range of members (set) and returns different subsets of the input set (with contiguous members) based on other arguments and the relative position in the input set of the current member being calculated. While MDX can support certain variants of this function through the use of the Lag, Lead, and RelMemberRange functions, not all Calc combinations can be readily translated.

Most Calc functions that take a set as input can also take an arbitrarily constructed list of members or cross members, referred to as explists, as input. Such capability makes Calc functions (such as @SUM, @MIN, @MAX. and so on) more flexible than their counterparts in MDX. The explists need to be translated to MDX by fixing the dimensionality of the cross members as needed so they subscribe to the rules of an MDX set.

MDX has a much richer repertoire of metadata functions than Calc. To achieve functionality in Calc, similar to that provided by the MDX metadata functions, may require proce-

dural code in the form of custom defined functions. The following list describes the MDX metadata functions.

The @DESCENDANTS function in Calc maps directly to Descendants in MDX. In addition, MDX has variants of Descendants that enable optionally skipping members from one or more levels.

Operations on a set by using functions such as Distinct (duplicate removal), Head, Tail and Subset are available only in MDX. The Extract function operates on an input set and a number of dimensions and creates a resulting set that has members from those input dimensions only for each tuple.

Operations on two sets, using functions such as Union, Intersect, Except and CrossJoin are available only in MDX. The corresponding list processing functions in Calc do not compare in terms of functionality.

Other sophisticated means of pruning an input set based on the value of an arbitrarily complex numeric expression (an expression that returns a double value when evaluated for each member of the input set) exist in MDX in the form of TopCount and BottomCount functions.

The following Drill functions are available in MDX:

- DrilldownByLayer, (alternate name for the ByLayer function)
- DrilldownMember
- DrillupByLayer
- DrillupMember performs ascendant or descendant operations on one or more input members based on criteria specified by other parameters.

Complex procedural logic requiring outline traversals can be succinctly described using the MDX Generate function. This rich iterative set function applies a specified (member set) method on each element of the input set, and produces an output set. When iterating through the elements of a set, the function CurrentTuple, helps identify the element of the input set currently being operated upon.

MDX has two kinds of set pruning functions. The Filter function uses a Boolean search expression to select input elements to appear in the output. On the other hand, to pick a specified count of members from an input set based on the value of a numeric expression, MDX provides the following functions:

- TopSum
- TopCount
- TopPercent
- BottomSum
- BottomCount
- BottomPercent.

TopPercent, for instance, returns the smallest possible subset of a set for which the total results of a numeric evaluation are at least a given percentage. Elements in the result set are listed from largest to smallest. This function ignores missing values. BottomSum, as another example, returns the smallest possible subset of a set for which the total results of a numeric evaluation is at least a given sum.

Two functions, Order and Hierarchize can be used to change the positions of elements within a set. Order sorts the members of a set in ascending or descending order based on the specified value of a numeric expression to be applied to each element of the set. Hierarchize reorders members within a set based on the hierarchical order of the members in the outline.

While Calc assumes dynamic time series (DTS) members will be captured in the outline and hence has no functions to deal with such members, MDX has explicit support for time series calculations without referencing DTS members in the outline. Since aggregate storage outlines cannot include DTS members, the ability to execute DTS type calculations in MDX formulas is critical. The MDX PeriodsToDate function can be used to accomplish similar functionality.

The appendix section contains examples of Dynamic Time series and Time balance functionality in MDX.

The MDX Count function returns the cardinality of an input set (the number of tuples). There is no analog to this function in Calc.

FUNCTIONS WITH RETURN TYPE NUMBER

The set of numeric functions that accomplish arithmetic and statistical calculations is larger in Calc than in MDX. However, the MDX variations, when they exist, tend to be richer semantically.

In Calc, the mathematical functions come in two variations. In one variation, the functions, which include @SUM, @MIN, @MAX and @AVG, take as input an arbitrary collection of members or cross members (without regard to dimensionality), substituting for missing dimensions from the current calculation context and obtaining the cell values from the cube. Functions in the other variation, which include @SUMRANGE, @MINRANGE, @MAXRANGE and @AVGRANGE, take as input a member or a cross member and a range (collection of members from another dimension) and perform the requested aggregation on cell values obtained by crossing the input member with each element of the range.

MDX rules for a set prohibit the first variation directly. If every element of the arbitrary member collection can be adjusted to subscribe to forming a set before being used as input to the mathematical function, then the translation is possible.

Calc functions optionally take a SKIP argument that specifies how to treat cell values that are missing or zero. Such functionality does not exist in MDX but can be accomplished indirectly using Filter, IIF or CoalesceEmpty constructs as demonstrated in the "Calc to MDX Functions" table.

The Calc COUNT function returns the number of elements in an input set based on the cell values of each element (and has an optional SKIP argument). The MDX counterpart is NonEmptyCount.

Scalar statistical functions, such as @RANK, @STDDEV, @VARIANCE, @MEDIAN and @MODE, which return a single value, as opposed to vector statistical functions, are available only in Calc. Also, expense functions such as @VAR and @VARPER, which determine the difference between two members by taking into consideration any expense tags, are not relevant in aggregate storage outlines.

FUNCTIONS WITH RETURN TYPE DIMENSION

The MDX Dimension function returns the root member of the dimension to which an input member, level, or generation name belongs. Although Calc has no similar function, it is possible to achieve the same result by requesting the ancestor of the member at root level (generation 0).

FUNCTIONS WITH RETURN TYPE LAYER

Functions that return a generation or level number given a member name as input are similar in both languages. Calc, additionally enables generation and level numbers to be obtained based on current context (@CURGEN/LEV). The same result can be accomplished in MDX by using Generation or Level with CurrentMember to identify the input member. For example:

Generation (CurrentMember(dimension)).

FUNCTIONS WITH RETURN TYPE STRING

Calc has four functions that operate on string data types:

- @SUBSTRING
- @CONCATENATE
- @NAME
- @TODATE.

In conjunction with the @MEMBER function the string functions can be used to selectively pick members from an outline.

The @SUBSTRING function returns a contiguous list of *n* characters from an input string. @CONCATENATE pastes together two input strings and the @NAME function converts input argument to a string. @Todate converts date strings to numbers that can then be used in calculation formulas.

While MDX does not support string manipulation functions in the current release, it enables querying properties such as member names, aliases, and attributes. equality and inequality operators can also be used in expressions that accept string types. IIF can return string types.

FUNCTIONS WITH RETURN TYPE BOOLEAN

In Calc, the most common Boolean functions that test for metadata outline relationships are:

@ISANCEST

@ISCHILD

@ISDESC

@ISGEN(LEV)

@ISSIBLING

@ISPARENT

@ISSAMEGEN(LEV)

ISUDA.

While some of these functions exist in MDX others can be implemented through a combination of metadata member-set functions, the set intersection operator, and the Is function, which returns TRUE if two members are identical. For example:

```
@ISDESC(Member) /* Calc */

Count(Intersect({Member.Descenda
nts},

{Member.dimension.CurrentMember}
) = 1  --MDX
```

Alternatively:

```
IsAncestor(Member, Member.dimen-
sion.CurrentMember) --MDX
```

Calc also has *i* prefixed variations of all the above functions (@ISICHILD, @ISIDESC, and so on) which include the specified member in the test. The following examples show a Calc Boolean function and its equivalent in MDX:

```
@ISIDESC(member)/* Calc */

(Count(Intersect({Member.Descenda
nts}, {Member}) = 1

   OR

Is(Member, Member)) /* MDX */
```

Here is another way to write the same functionality in both languages:

```
@ISIDESC (member) /* Calc */

(Is (<member>, <member>.dimen-
sion.CurrentMember)--MDX

  OR

 IsAncestor (<member>,
<member>.dimension.CurrentMember))
```

The Calc IS* functions implicitly assume that the check is for the current member along the same dimension as the argument to the function. MDX, in contrast, requires explicit specification of the current member.

The Calc @ISMBR function takes as input any arbitrary collection of members or cross members and enables membership testing. The same can be accomplished indirectly in MDX using multiple Boolean constructs, Intersect, and Count functions. See the @ISMBR function in the "Calc to MDX Functions" table.

Checking for missing cells can be accomplished in MDX by using the IsEmpty function. For example:

```
(Value <> #MISSING)  /* Calc */

IsEmpty (value_expression) --MDX
```

In Calc, @ISLEV is used to check for leaf-level members. In MDX, the IsLeaf function is used to check for leaf-level members. For example:

```
(@ISLEV(member) <> 0) /* Calc */

IsLeaf(member) --MDX
```

GENERAL STEPS IN TRANSLATING A CALC FORMULA TO MDX

This section provides some general guidelines as well as specific steps for translating Calc formulas to MDX.

Be certain that the application has been redesigned to use an aggregate storage outline. In this regard, make certain that formulas do not reference any block-storage specific outline constructs, such as dynamic time series members, variance functions that rely on expense tagging, or functions that operate on shared members (@RDESCENDANTS). Such constructs are not valid in aggregate storage outlines.

Remember that for an aggregate storage outline, formulas can only be attached to an explicitly tagged accounts dimension.

Rewrite each function in the formulas attached to an explicitly tagged accounts dimension for which a direct counterpart in MDX exists. See the Calc to MDX Functions table for specific information and examples.

Identify functions in the formulas attached to an explicitly tagged accounts dimension for which an indirect rewrite is required using the rules outlined in this document.

Understand the calculation order semantics for the formulas in the block storage outline. Organize the dependent formulas in the aggregate storage outline carefully to achieve the same results as block storage.

If formulas reference custom-defined functions or macros consider rewriting them, if possible, using other MDX functions.

For formulas attached to non-accounts dimensions, create template MDX queries to be executed at query-time. Formulas cannot be created on non-accounts dimensions in aggregate storage outlines.

If an outline Calc formula in a block storage outline involves assignment of values to a stored level 0 member, the translation involves two members in a corresponding aggregate storage outline: one where the data is loaded and another where the corresponding MDX formula can be attached. The current release of Essbase considers members with formulas attached as implicitly 'dynamic'. A dynamic member cannot have data input to it. For example,

```
Net_Written_Car_Years = if
(@ismbr(Coverage) or
@ismbr(Bodily_Injury))

Net_Written_Car_Years-
>Bodily_Injury;

      else

Net_Written_Car_Years;
```

This formula is translated into MDX using an additional member, as follows. Note that `Net_Written_Car_Years_Input` is used to load input data.

```
Net_Written_Car_Years =

IIF (Is
([Coverage].CurrentMember,
[Coverage]) OR

Is ([Coverage].CurrentMember,
[Bodily_Injury]),

(Net_Written_Car_Years_Input,
Bodily_Injury),

(Net_Written_Car_Years_Input);
```

ACCOMPLISHING DYNAMIC TIME SERIES AND TIME BALANCE FUNCTIONALITY IN MDX

The Block Storage outline provides pre-defined tags in the form of TBFirst, TBLast and TBAvg for automatic time balance functionality. Built-in time series members such as YTD, WTD etc can be associated with a specific generation of a named time dimension to obtain time series functionality. Such tags are not available directly in aggregate storage outlines. However, it is possible to achieve similar functionality through slightly different means. This section outlines one such method of achieving both time balance and time series functionality using the sample/basic database when used in aggregate storage mode.

TIME BALANCE FUNCTIONALITY

Recall that before you can use a time balance tag in an outline there must be an Accounts dimension. Normally, the calculation of an upper-level member in the time dimension is based on

either the outline hierarchy or any formula if present on the member. However, if a member in an accounts dimension is marked as TB First, then any upper level member in the time dimension is computed using the TBFirst tag.

For example, in sample/basic block storage outline, data corresponding to member "Opening Inventory" (ignore the formula on opening inventory for the purpose of this discussion and simply focus on the time-balance tag associated with the member) represents the inventory at the beginning of each month. The quarterly value for "Opening Inventory" is equal to the Opening value for the first month in that quarter. Thus, "Opening Inventory" is tagged with TBFirst tag. Similarly, "Ending Inventory" data represents the inventory at the end of each month. The quarterly value for "Ending Inventory" is equal to the ending value for the (last month in the) quarter. Ending Inventory, is thus tagged with TB last. This functionality can be achieved in aggregate storage, for instance, through a combination of UDAs and MDX outline formula as we will demonstrate. Instead of using pre-defined tags, members of accounts dimension that require time-balance tags should be marked with UDAs such as TBFirst and TBLast.

Thus, "Opening Inventory" would be associated with a UDA called "TBFirst" and "Ending Inventory" with the UDA "TBLast". Next, consider creating an additional dimension called "Analytics" dimension which houses all formulas required for the outline and hence is tagged as Accounts dimension in the Aggregate Storage outline. Consider the presence of a member called "Actual" which is used to perform all dataloads. Additionally, consider a member "TB" in this dimension that has the following formula attached to it to achieve time-balance functionality.

```
TB

CASE
WHEN(IsUda([Measures].CurrentMember,
"TBLast"))
THEN (ClosingPeriod
(Year.Levels(0), Actual))
WHEN(IsUda([Measures].CurrentMember,
"TBFirst"))
THEN (OpeningPeriod
(Year.Levels(0), Actual))

ELSE [Actual]
END
```

Thus when cells at the intersection of the member "TB" from the Analytics dimension and any other measure that is tagged with a "Time Balance" User Defined Attribute is retrieved, the time balance computation is performed dynamically at retrieval time.

DYNAMIC TIME SERIES FUNCTIONALITY

Let us consider support for Quarter-to-Date functionality in Sample Basic outline. The functionality is obtained by enabling the Q-T-D member and associating it with the generation containing quarters, namely, generation number 2, containing members Qtr1, Qtr2, Qtr3, and Qtr4. Retrieving Q-T-D member in a query requires specification of a month (level 0 member) as the latest member and results in the calculation of monthly values up to the current month in the quarter. Thus, Q-T-D(May) returns the quarter-to-date values by adding values for April and May.

Let QTD be a member of the newly added "Analytics" dimension in Sample Basic outline in Aggregate Storage mode. In keeping with the requirement that only level-0 members of time dimension can be specified as current time period for Q-T-D, the QTD member intersections with level-0 members of Year dimension alone make sense.

The MDX formula required to be associated with the QTD member for sample/basic is given below.

```
QTD

/* If the member from Year
dimension is at Level 0, then */
CASE  WHEN
IsLevel([Year].CurrentMember, 0)

/* Add all values from the first
sibling of the member to itself */

THEN Sum(
Parent([Year].CurrentMember).Firs
tChild:[Year].CurrentMember,
[Actual])

ELSE  #Missing /* Meaningless to ask for QTD
for any other [Year].dimension member. */

END
```

Alternatively, the functionality represented by DTS member can also be expressed as Sum(PeriodToDate (Year.Generations (n)) where "n" is to be substituted by the generation number to which the DTS member applies. Thus for Q-T-D in the sample/basic database, the value of n=2

TRANSLATION TABLE

The following table lists all functions in the Calc language and their analogs in MDX (and vice versa). Where a direct analog doesn't exist, transformation rules and examples are provided.

CALC	MDX	REMARKS
@ABS	ABS	
@ALLANCESTORS	ANCESTOR	The MDX function returns ancestor at a particular generation. Call this function repeatedly to get ancestors at each generation and union the resulting members. Same as @ANCESTORS since shared members cannot appear in aggregate storage outlines.
@ALIAS	Not required	In MDX, the argument to @ALIAS can be passed as is to the outer function
@ANCEST	ANCESTOR with CURRENTMEMBER as input. CROSSJOIN the result with the optional third argument to @ANCEST function	`@ancest(Product,2,Sales) /* Calc */ Crossjoin ({Sales}, {Ancestor (Product.CurrentMember, Product.Generations (2))}) --MDX`
@ANCESTORS	ANCESTOR	
@ANCESTVAL	ANCESTOR with CURRENTMEMBER as input. CROSSJOIN the result with the optional third argument to @ANCESTVAL function	@ancestval(Product,2,Sales) /* Calc */ (Sales, Ancestor(Product.CurrentMember, Product.Generations (2))).Value --MDX
@ATTRIBUTE	ATTRIBUTE	
@ATTRIBUTEBVAL	[BaseDimension].CurrentMember. AttributeDimension	See “MDX Properties” in the *Technical Reference* for more information. `@attributebval(Caffeinated) /* Calc */ Product.CurrentMember. Caffeinated --MDX`

CALC	MDX	REMARKS
@ATTRIBUTESVAL	[BaseDimension].CurrentMember.AttributeDimension	See "MDX Properties" in the *Technical Reference* for more information. `@attributesval("Pkg Type") /* Calc */ Product.CurrentMember.[Pkg Type] --MDX`
@ATTRIBUTEVAL	[BaseDimension].CurrentMember.AttributeDimension	See "MDX Properties" in the *Technical Reference* for more information. `@attributeval(Ounces) /* Calc */ Product.CurrentMember.Ounces --MDX`

@AVG	If dimensionality of all elements in the input set to @AVG is the same then use Avg. Translate SKIPNONE to INCLUDEEMPTY
	If dimensionality of all elements in the input set to @AVG is not the same, then perform average by explicitly adding the tuples and dividing by the set cardinality.
		Note that the MDX Avg function skips missing cell values by default
		`@avg(SKIPMISSING, @children(East)) /* Calc */ Avg([East].Children)`
		
		If SKIPMISSING is replaced by SKIPNONE, the translation changes to
		`Avg([East].Children, Sales, INCLUDEEMPTY)`
		
		For SKIPZERO, the translation is:
		`Avg([East].Children, IIF(Market.CurrentMember.Value=0, Missing, IIF(Market.CurrentMember = Missing, 0, Market.CurrentMember.Value)))`
		
		For SKIPBOTH, the translation is:
		`Avg([East].Children, IF(Market.CurrentMember=0, #Missing, Market.CurrentMember.Value))`

CALC	MDX	REMARKS

@AVGRANGE	CROSSJOIN(first argument, set created out of second argument). Rest similar to @AVG when dimensionality of all elements of the input set is identical.	`@AVGRANGE (SKIPMISSING, Sales, @children(West)) /* Calc */ AVG(CrossJoin({Sales}, {[West].Children)})--MDX`
		
		If SKIPMISSING is replaced by SKIP-NONE, the translation becomes:
		`AVG({[West].Children)},Sales ,INCLUDEEMPTY) --MDX`
		
		If SKIPZERO is used, then the trans-lation is:
		`Avg([West].Children), IIF(Sales = 0, Missing, IIF(Sales = Missing, 0, Sales)) --MDX`

@CHILDREN	Children	
@CONCATENATE	-	No counterpart
@CORRELATION	-	No counterpart

@COUNT	Use Count if SKIPNONE.
	
	Use NonEmptyCount if SKIPMISSING.
	
	For SKIPZERO, see the example in the next column.
	
	For SKIPBOTH use:
	`Count(Filter(set, value <> 0 && value <> #MISSING))`
		`@COUNT(SKIPMISSING, @RANGE(Sales, Children(Product))) /* Calc */ NonEmptyCount(CrossJoin({Sales}, {Product.Children})) --MDX`
		
		Note that Count counts empty cells by default whereas NonEmptyCount does not
		For SKIPNONE, the translation is:
		`Count(CrossJoin(Sales, Product.Children))--MDX`
		
		For SKIPZERO the translation is:
		`NonEmptyCount(Product.Children, IIF(Sales=0, Missing, IIF(Sales = Missing, 0, sales))) /* MDX */`

CALC	MDX	REMARKS
@CURGEN @CURLEV	Generation(CurrentMember(dimension)) Level(CurrentMember(dimension))	`@CURGEN(year) /* Calc */` `Year.CurrentMember.Generation /* MDX */`
@CURRMBR	CurrentMember	
@CURRMBRRANGE	RelMemberRange	`@CurrMbrRange(Year, LEV, 0, -1, 1) /* Calc */` `RelMemberRange( Year.CurrentMember, 1, 1, LEVEL)`
@DESCENDANTS	Descendants(member)	
@EXP	Exp	
@FACTORIAL	Factorial	
@GEN/@LEV	Generation/Level	
@GEN/LEVMBRS	Layer.Members	
@IALLANCESTORS	Ancestors (note that aggregate storage outlines do not have shared members)	
@IANCESTORS	Same as @IALLANCESTORS since shared members are not present in aggregate storage outlines	
@ICHILDREN	Union(member, member. children)	
@IDESCENDANTS	Union(member, member.descendants)	

@ILSIBLINGS	Union(mbr, MemberRange(FirstSibling(mbr), Lag(mbr, 1)))	`@ILSIBLINGS(Florida) /* Calc */`
		`Union({Florida}, {MemberRange(Florida.FirstSibling,`
		`Florida.Lag(1)}) --MDX`

CALC	MDX	REMARKS
@INT	Int	
@ISACCTYPE	IsAccType	
@ISANCEST	IsAncestor	`@ISANCEST(California) /* Calc */ IsAncestor(Market.CurrentMember, California) --MDX`
@ISCHILD	IsChild	
@ISDESC	IIF (Count(Intersect(*member*.descendants, *descendantmember*)) = 1, TRUE, FALSE)	`@IsDesc(Market) /* Calc */ IIF(Count(Intersect( {Market.Descendants}, {Market.CurrentMember}))= 1, <true-part>, <false-part>) /* MDX */`
@ISGEN	IsGeneration	`@ISGEN(Market, 2) /* Calc */ IsGeneration( Market.CurrentMember,2) --MDX`
@ISIANCEST	IIF(Is(member,ancestormember) OR IsAnest(member, ancestormember), TRUE, FALSE)	The *i* prefix is a simple equality check. For example: `@ISIANCEST(California) /* Calc */ IIF(IS(Market.CurrentMember, California) OR IsAncestor(Market.CurrentMember, California), <true-part>, <false-part>) --MDX`
@ISIBLINGS	Union(member, member. siblings)	Returns a set that includes the specified member and its siblings.
@ISICHILD	IIF(Is(member, childmember) OR IsChild(member, childmember), TRUE, FALSE)	`@IsIChild(South) /* Calc */ IIF(Is(Market.CurrentMember, South) OR IsChild(Market.CurrentMember, South),<true-part>, <false-part>) /* MDX */`

CALC	MDX	REMARKS

@ISIDESC	IIF(Is(member, descmember) OR IsDesc(member, descmember), TRUE, FALSE)	`@IsIDesc(South) /* Calc */`
		`IIF (Is`
		`(Market.CurrentMember,`
		`South)OR`
		`Count(Intersect(`
		`{[South].Descendants},`
		`{Market.CurrentMember}))= 1)`
		`<true-part>, <false-part>)`
		`/* MDX */`

@ISIPARENT	IIF(Is(member, parentmember) OR ISPARENT(member, parent-member), TRUE, FALSE)	`@IsIParent(Qtr1) /* Calc */`
		`IIF(Is(Time.CurrentMember, [Qtr1])`
		`OR IsChild([Qtr1],`
		`Time.CurrentMember),`
		`<true-part>, <false-part>)`

@ISISIBLING	IsSibling(member, sibling-member) OR Is(currentmember, siblingmember)	`@ISISIBLING(Qtr2) /* Calc */`
		`IIF(Is(Time.CurrentMember,`
		`[Qtr2])`
		`OR IsSibling([Qtr2],`
		`Time.CurrentMember),`
		`<true-part>, <false-part>`

@ISLEV	ISLEVEL	

@ISMBR	IIF (Count(Intersect(member-set, member)) = 1, TRUE, FALSE)	Calc allows a collection of members or cross members that do not subscribe to the rules of an MDX set to appear as the second argument. This functionality cannot be easily replicated without enumerating each element of the second set and testing for intersection.
		
		However if the second argument subscribes to MDX set rules then the translation is easier as shown. For example:
		`@ISMBR("New York":`
		`"New Hampshire") /* Calc */`
		`IIF (Count(Intersect(`
		`{MemberRange([New York],`
		`[New Hampshire])},`
		`{Market.CurrentMember})) = 1,`
		`<true-part>, <false-part>)`
		`/* MDX */`

CALC	MDX	REMARKS
@ISPARENT	Use IsChild	`@ISPARENT("New York") /* Calc */ ISCHILD(Market.CurrentMember, [New York]) --MDX`
@ISSAMEGEN/LEV	IIF (member. generation() = CurrentMember(dimension).generation()), TRUE, FALSE)	`@ISSAMEGEN(West) /* Calc */ IIF ( Ordinal(Market.CurrentMember .Generation) = Ordinal(West.Generation), <true-part>, <false-part>) /* MDX * /`
@ISSIBLING	IsSibling	
@ISUDA	IsUda	
@LIST	-	If the memberset does not subscribe to MDX set rules, then explicit enumeration is required. For rangelist use CrossJoin(member, set)
@LN, @LOG/10	Ln, Log, Log10	
@L(R)SIBLINGS		`@LSiblings(Florida) /* Calc */ MemberRange(Florida.FirstSibling, Florida.Lag(1)) –MDX @Rsiblings(Florida) /* Calc */ MemberRange(Florida.Lead (1), Florida.LastSibling) –MDX`
@MATCH		
@MAX	Max	Use Max if argument list is a set. Otherwise rewrite logic using Case constructs by explicit enumeration of the argument list. `@MAX(Jan:Mar) /* Calc */ Max(MemberRange([Jan], [Mar])) --MDX`

CALC	MDX	REMARKS

@MAXRANGE	Max	`@MAXRANGE(Sales,@Children(Qtr1)) /* Calc */`
		`Max(CrossJoin({Sales}, {[Qtr1].Children})) /* MDX */`
		or
		`Max([Qtr1].Children, Sales) --MDX`

@MAXS	Max	`@MAXS (SKIPMISSING, Sales, @Children(Qtr1)) /* Calc */`
		`Max(Filter(Children([Qtr1]), Sales <> Missing)) --MDX`
		
		For SKIPZERO the translation is:
		`Max(Filter (Children([Qtr1]), Sales <> 0)) --MDX`
		
		For SKIPBOTH, the translation is:
		Max(Filter(Children([Qtr1]),
		`Sales <> 0 AND`
		`Sales <> Missing)) -MDX`

@MAXSRANGE	Max	`@MAXSRANGE(SKIPMISSING, Sales, @Children(Qtr1)) /* Calc */`
		`Max(Filter (Children([Qtr1]), Sales <> Missing)) --MDX`
		
		For SKIPZERO the translation is:
		`Max(FILTER(Children([Qtr1]), Sales <> 0)) --MDX`
		
		For SKIPBOTH the translation is:
		`Max(Filter(Children([Qtr1]), Sales <> 0 AND Sales <> Missing)) /* MDX */`

CALC	MDX	REMARKS
@MDANCESTVAL		Find the ancestor along each dimension, cross join the result with the ancestor along the next dimension and so on until all dimensions are covered. Next cross join with the cross member if present. Finally, get the value of the cell. **NOTE:** Crossjoin requires that the tuples or sets of members for the two arguments be from different dimensions, whereas Calc is flexible about the same dimension appearing in the cross member. If the same dimension appears in the cross member as well, then simply skip seeking the ancestor along that dimension. `@MDANCESTVAL(2, Market, 2, Product, 2, Sales) /* Calc */ (Sales, Ancestor(Market.CurrentMember, 2), Ancestor( Product.CurrentMember, 2) ).Value /* MDX */`
@MDPARENTVAL	See “Remarks” for @MDANCESTVAL but replace Ancestor with Parent.	`@MDPARENTVAL(2, Market, Product, Sales) /* Calc */ (Sales, Market.CurrentMember.Parent, Product.CurrentMember.Parent ).Value /* MDX */`
@MDSHIFT	See @NEXT and repeat the same for each dimension that needs to be shifted. Cross join the results from each dimension and get the value of the final tuple. See “Remarks” for @MDANCESTVAL.	

CALC	MDX	REMARKS
@MEDIAN	-	Not available
@MEMBER	-	Not needed in MDX

@MERGE	Union(set1,set2)	If the lists specified as inputs to @MERGE do not subscribe to the rules of an MDX set, then the @MERGE function cannot be translated. The following example assumes that the lists doe subscribe to MDX set rules.
		
		`@Merge(@Children(East),`
		`  @Children(West)) /* Calc */`
		`{Union([East].Children,`
		`  [West].Children)} --MDX`

@MIN	Same rules as @MAX, except use Min instead of Max.	
@MINRANGE	Same rules as @MAXRANGE, except use Min instead of Max.	
@MINS	Same rules as @MAXS, except use Min instead of Max.	
@MINSRANGE	Same rules as @MAXS-RANGE, except use Min instead of Max.	
@MOD	Mod	
@MODE	-	Not supported in MDX
@NAME	-	Not needed in MDX

CALC	MDX	REMARKS

@NEXT	@Next(mbr,[n, range]) returns the nth cell value in the range from the supplied member. The function returns missing value if the supplied member does not exist in the range. If range is not specified, level 0 members of time are used.
	
	MDX does not have an equivalent function for an arbitrary range. However if the range is restricted to members from a specific level or generation, then using NextMember (if n=1) or Lead/Lag will work as shown in the sample translation. This is probably the common case.
		`@Next(Cash) /* Calc */ (NextMember( [Year].CurrentMember,LEVEL), [Cash]).Value /* MDX */`
		Alternatively
		`@Next(Cash, 2) /* Calc */ CrossJoin( Year.CurrentMember.Lead (2, LEVEL), Cash).Value /* MDX */`

@NEXTS	No support	
@PARENT	member.Parent	
@PARENTVAL	Value(Crossjoin(crossmember, Parent(Current Member(dimension)))	`@PARENTVAL(Market, Sales) /* Calc */ ([Sales], [Market].CurrentMember.Parent).Value /* MDX */`
@POWER	Power	

CALC	MDX	REMARKS

@PRIOR	Same as @Next	`@Prior(Cash) /* Calc */ (PrevMember(Year.CurrentMember, LEVEL), [Cash]).Value /* MDX */`
		
		`Alternatively`
		`@Prior(Cash, 2) /* Calc */`
		
		`(Year.CurrentMember.Lag (2, LEVEL), [Cash]).Value /* MDX */`

@PRIORS	No support	

@RANGE	CrossJoin(member, rangeset)	Calc automatically uses level 0 members of time dimension if a range is unspecified. That feature does not exist in MDX, so you must explicitly include the range.
		`@RANGE(Sales, @Children(East)) /* Calc */`
		`CrossJoin({Sales}, {[East].Children}) /* MDX */`

@RANK	No support. This is a vector function	
@RELATIVE		
@REMAINDER	REMAINDER	
@REMOVE	Except(set1, set2)	Translation will work only if set1 and set2 are true MDX sets.
@ROUND	Round	
@SHIFT	See @PRIOR and @NEXT	
@SIBLINGS	Siblings	
@STDEV/P/RANGE	No support	Not supported

CALC	MDX	REMARKS
@SUBSTRING	No support	
@SUM	Sum	Convert each element of the explist to a tuple so collectively the tuples can form a set.
@SUMRANGE	Sum(CrossJoin(member, rangeset))	
@TODATE	ToDate	
@TRUNCATE	Truncate	
@UDA	Uda	
@VAR/@VARPER	Arg1 – Arg2	An aggregate storage outline has no expense tags. So variance functionality defaults to subtraction.
@VARIANCE/ @VARIANCEP	Not supported	
@WITHATTR	WithAttr	
@XREF	No support.	
@XRANGE	No support	

ABOUT HYPERION

FOR MORE INFORMATION, VISIT HYPERION AT WWW.HYPERION.COM

Hyperion is the global leader in Business Performance Management software. More than 9,000 customers – including 91 of the FORTUNE 100 – rely on Hyperion software to translate strategies into plans, monitor execution and provide insight to improve financial and operational performance. Hyperion combines the most complete set of interoperable applications with the leading business intelligence platform to support and create Business Performance Management solutions. A network of more than 600 partners provides the company's innovative and specialized solutions and services.

Named one of the FORTUNE 100 Best Companies to Work For 2004, Hyperion employs approximately 2,600 people in 20 countries. Distributors represent Hyperion in an additional 25 countries. Headquartered in Sunnyvale, California, Hyperion – together with recently acquired Brio Software Inc. – generated combined annual revenues of $612 million for the 12 months ending June 30, 2003. Hyperion is traded under the Nasdaq symbol HYSL. For more information, please visit www. hyperion.com, www.hyperion.com/contactus or call 800 286 8000 (U.S. only).

HYPERION 1344 CROSSMAN AVENUE SUNNYVALE, CA 94089

TEL 408.744.9500 | WWW.HYPERION.COM | FAX 408.744.0400