MDX ALLOWS COMPLEX, MULTIDIMENSIONAL QUERIES

A HYPERION WHITE PAPER

APRIL 2004
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Introducing MDX to Report Writer Users</td>
</tr>
<tr>
<td>4</td>
<td>Report Layout</td>
</tr>
<tr>
<td>4</td>
<td>Labels and Properties</td>
</tr>
<tr>
<td>4</td>
<td>Member Selection</td>
</tr>
<tr>
<td>5</td>
<td>Sorting</td>
</tr>
<tr>
<td>5</td>
<td>Calculations</td>
</tr>
<tr>
<td>7</td>
<td>Why Use MDX?</td>
</tr>
<tr>
<td>9</td>
<td>API and Application Considerations</td>
</tr>
<tr>
<td>10</td>
<td>MDX Query Execution Order</td>
</tr>
<tr>
<td>14</td>
<td>MDX Calculated Member and Solve-Order</td>
</tr>
<tr>
<td>16</td>
<td>MDX Calculated Members and Security</td>
</tr>
<tr>
<td>17</td>
<td>MDX and Member Ordering</td>
</tr>
<tr>
<td>17</td>
<td>MDX and Dynamic Time Series Members Support</td>
</tr>
<tr>
<td>17</td>
<td>MDX and Sorting</td>
</tr>
<tr>
<td>18</td>
<td>MDX and Advanced Filtering Options</td>
</tr>
<tr>
<td>20</td>
<td>MDX and Suppression of Missing Values in Output</td>
</tr>
<tr>
<td>22</td>
<td>Appendix:</td>
</tr>
<tr>
<td>22</td>
<td>Report Layout Commands</td>
</tr>
<tr>
<td>23</td>
<td>Member Selection Commands</td>
</tr>
<tr>
<td>24</td>
<td>Sorting Commands</td>
</tr>
<tr>
<td>25</td>
<td>Data Ordering and Data Range Commands</td>
</tr>
<tr>
<td>26</td>
<td>Format Commands</td>
</tr>
<tr>
<td>27</td>
<td>Column or Row Calculation Commands</td>
</tr>
<tr>
<td>28</td>
<td>Member Names and Aliases</td>
</tr>
</tbody>
</table>
MDX (Multidimensional Expressions) is a new data extraction mechanism introduced in Hyperion Essbase Analytic Services release 7.0 (herein referred to as “Essbase”), that provides a method for forming complex and robust multidimensional queries.

MDX allows you to:

- Use declarative query as input and set of structures as output
- Transfer more calculations and selections into the query from the client
- Use the same set of APIs for data and metadata queries
- Return cell properties and properties of the members in the same query
- Automate analysis with fewer steps

In this document, we will provide an overview of MDX, its critical implementation aspects, and compare its interface with a report writer language, highlighting similarities and differences as relevant. We will also describe specific details of MDX query execution semantics through examples. A solution section to demonstrate migration of existing report writer-based queries into the MDX syntax is included. All the example queries assume the Sample: Basic database as the queried data source.

MDX (Multidimensional Expressions) is a declarative query language that can be used to extract information from an Essbase database. The output of an MDX query has the form of a cube itself. The query captures both the output specification and the dimensionality of the resulting cube. MDX-related procedural APIs allow a query to be submitted for execution and the navigation of the query result consists of member combinations and corresponding data value. A query and reporting application will use these APIs to format and display the result.¹

Report writer is also a declarative scripting language that can be used to report on an Essbase database. You can combine report writer’s selection, layout, and formatting commands to build reports with different output specifications.

In this section, we will provide a brief overview of the two query interfaces comparing and contrasting some commonly used functions and execution paradigms.
**REPORT LAYOUT**

Report writer commands enable a report to be defined in terms of columns and rows across one or more pages. The equivalent construct in MDX is the axis – columns, rows and pages are considered three of the possible axes. A query may have only one or two axes as well as three or more (the maximum number that can be used within a single query is 64). An optional slicer axis defines a point of view for the query.

MDX does not support functionality similar to the report-writer formatting commands such as PAGEHEADING. The resultant data structure is not a text report, and it is left to the client application to format results for presentation.

**LABELS AND PROPERTIES**

Both MDX and report writer support the ability to display query output using either member name or alias. MDX supports the display and use of both in the same query. As described later, member attributes and cell properties can also be presented to the client through the MDX APIs.

**MEMBER SELECTION**

Report writer member selection commands enable construction of queries based on member (hierarchical) relationships in an outline. MDX supports a much richer set of functions for the same purpose. For example, the member relationship function, descendants, is exposed in the report writer in two variants – Descendants (all members in a sub-tree/sub-hierarchy rooted at the member whose descendants are desired) and Idescendants (same as descendants but includes the rooted member in the result set). The MDX Descendants function, on the other hand, provides several variants that enable a more comprehensive selection.

For example, Descendants (<member>, <layer>, SELF) returns the members of a particular generation or a level in a sub-tree/sub-hierarchy rooted at the <member>. To return the same set of members in report writer, you have to use Descendants (<member>). To return all the members in the sub-tree rooted at the member, use Gen or Lev command to return all the members of a generation or a level of the entire dimension, and, finally, use the Link command with an And operator.

Another example is the MDX Uda function that takes as its argument the root member of a sub-hierarchy in a dimension of interest. The function returns members with the specified UDA in the sub-tree. The report writer UDA command, in contrast, only works for the top-most member of the dimension. As a result, you have to use the Link command to prune out members not in the sub-tree in the dimension of interest.

MDX also provides functions that have no counterparts in the report writer interface, such as the LastPeriods function, which can be used to analyze a sales trend over the last n months starting from a given month. Another example is the ParallelPeriod function used to compare same-period metrics in a prior year. OpeningPeriod and ClosingPeriod can be used to achieve Time Balance functionalities similar to those provided by the custom TB First, TB Last outline tags.

1. For more information on a complete list of MDX functions implemented, please refer to
Unlike dynamic-calc members, calculated members are not considered part of the Essbase outline and will not be returned in functions that return a set of members from metadata. For example, the result of the MDX expression Year.Members includes the member Qtr1, which is a dynamic calc member. In contrast, a MDX calculated member is defined only for the duration of a query. It is not part of the database outline, and metadata functions do not include such members in their result sets.

Consider a calculated member definition such as:

```
MEMBER [Year].[MyQtr] as 'Feb + Mar + Apr'
```

The calculated member MyQtr will not appear in the result of the function Year.Members, which will be resolved based on the database in Essbase. Therefore, a calculated member has to be explicitly specified whenever it is needed in a query.

For example, assume you must create a new type of scenario that will indicate the difference with the last quarter’s value.

```
WITH
MEMBER Scenario.[Qtr to Qtr Variance] AS
'IIF (Is (Year.CurrentMember, Qtr1),
  0,
  Year.CurrentMember - Year.CurrentMember.PrevMember
)
SELECT
  { Scenario.Children, [Qtr to Qtr Variance] } on Columns,
  Year.Children on Rows
WHERE (East, [100], Sales)
```
Note that [Qtr to Qtr Variance] is specified explicitly in the column axis. For Qtr1, there is no previous quarter – so the calculated member formula handles a special case for Qtr1 and evaluates the [Qtr to Qtr Variance] as 0.

The expression IsValid (Year.CurrentMember. PrevMember) returns True if the current member along Year dimension has a previous member at that generation – for the first member, for example January or Qtr1, IsValid will return False. Thus, you can rewrite the same formula so that it works on members of all generations in Year dimension as follows.

```
WITH
MEMBER [Scenario].[Qtr to Qtr Variance] as
'IIF (IsValid (Year.CurrentMember. PrevMember),
   Year.CurrentMember - Year.CurrentMember. PrevMember,
   0)
',
SELECT
{ Scenario.Children, [Qtr to Qtr Variance] } on Columns,
Year.Members on Rows
FROM Sample.Basic
WHERE (East, [100], Sales)
```
The addition of a calculated member in a query does not affect the database outline and hence no database restructuring is needed. In addition, this new member (assuming it is along dense dimensions) does not increase the size of the big block – the unit of data that Essbase server internally needs to allocate in memory for calculating dynamic-calc data values -- and hence has no effect on the dynamic calculation cache.

Readers familiar with the calculation language in Essbase will notice similarities between the behavior of certain calculation functions and corresponding MDX functions.

Comparison of MDX to Essbase calculation language is outside the scope of this document. In the MDX Calculated Members and Security section, we will describe other differences in terms of the security model between a calculated member in an MDX query and a formula attached to a dynamic calc member in the Essbase outline.

In this section we provided an overview of MDX with the report writer functionality as the backdrop. We will now discuss why it may be appropriate to transition your custom query interfaces to use MDX.

WHY USE MDX?

If you are familiar with reports and are wondering why you may want to switch to MDX, this section describes specific concepts in MDX not available in reports.

1. Declarative query as input and set of structures as output

Report writer interface lets you define the query in a declarative fashion – but the report writer output is in text. As a result, if you are developing a custom application using the report writer interface, you have to parse the report output before you can do any application-specific processing. You may have to use special formatting commands, to identify information like member name/alias/page break etc. Providing an application with a spreadsheet grid using the grid API requires a number of API calls that use a number of data structures in order to specify the query, which can be complex when the query is not based on plain, hierarchy operations. The only equivalent of MDX is the Grid API’s free-form APIs such as EssGBeginReport(). However, this works only for limited kinds of reports and will be deprecated in future Essbase releases. MDX supports a powerful expressive language as well as a set of APIs to introspect the contents of the result set.

2. Transfer more calculations and selections into the query from the client

One of the unique aspects of MDX is the ability to write a query that can access sets of members/data values that is completely different from the set of members/values reported in the final query output.

For example, assume you are responsible for the Eastern Region, and you must analyze the budgeted marketing dollars for products in Q2 that had Q1 sales in excess of $1,000. You wish to return cells for Budget and Q2 for East based on data from Actual and Q1 for East.

This query can be written in MDX as follows.
The entire query uses data for the Eastern regions. The \texttt{Filter()} expression prunes out those Level-0 products based on the Actual Sales numbers in Qtr1 – note that the query outputs the Budget numbers for Qtr2, whereas the Actual values for Qtr 1 were used during query execution.

In a subsequent section (MDX Query Execution Order), we will describe in more detail the concept of a query context as it applies to MDX query execution.

1. Same set of APIs for data and metadata queries

The querying syntax as well as output processing interfaces does not change, regardless of whether you are interested in the actual data values or just the metadata combinations. For example, you may want to query for states in the Eastern region, but are not interested in the actual data values that may exist for those states. In this case, you can execute the following query in data-less mode:

\begin{verbatim}
SELECT Children(East) on Columns,
      Children(Colas) on Rows
FROM Sample.Basic
WHERE (Qtr1, Actual, Sales)
\end{verbatim}

The next query is a data query retrieving the actual sales in Qtr1 for Colas in the Eastern-regional states:

\begin{verbatim}
SELECT (Budget, Marketing) on Columns,
      Filter (Product.Levels(0).Members,
               (Qtr1, Actual, Sales) > 1000)
      on Rows
WHERE (East, Qtr2)
\end{verbatim}
Note that in both cases the query uses the same children operator to query for states in the Eastern region. The output processing is also identical. Logically, an MDX-API based client program may be interested in:

a.) finding the number of tuples (known in MDX as members and cross-members) on the column axis (number of states in this case)
b.) finding the tuple at ith position (i.e. get name of the ith state).

Invoking the function EssMdxSetDataLess() before submitting a query for execution tells the server that the query is interested in retrieving just the metadata. Metadata queries may be beneficial in certain types of screening. Note that although a query may be interested in obtaining only metadata, the selection criterion for the metadata itself could be based on data. For instance, a query may retrieve just those Product-Market combinations for which quarter-to-quarter variance is significant.

1. Property support: the ability to return cell properties and properties of the members in the same query

The MDX language includes constructs that can be used to request Grid-API style cell-properties in the query output. Such properties identify whether a cell is read-only, dynamically calculated etc. In addition, MDX has constructs that allow certain member properties to be displayed in the output as well. MDX extends the notion of properties to include user-defined attributes (UDA) and attribute dimensions in Essbase. For example, you can request Color and Size properties for every member of the Product dimension (Color and Size are attributes of product) or if you want to know whether a market is a “Major Market” for every member from Market dimension in the query result. Report writer does not support cell-properties. You can get some of the member properties (e.g. use {IndentGen} and parse the output to figure out the generation number). However, such a task is extremely cumbersome and error-prone in report writer. Neither Grid API nor report writer supports the MDX style of exposing user-defined attributes or attribute dimensions as properties.

2. Automating analysis with fewer steps

You can write MDX queries to automate the production of reports that would otherwise need substantial human intervention. For example, you may want to report on the markets at the regional level – you want the state level details for the regions with a negative variance from last quarter. You might want to report on the states with substantial (30% and above for example) contribution to its region’s sales. The richness of the MDX expression model makes these queries straightforward.

API and Application Considerations

3. MDX is not an end-user interface for churning out end-user reports from the Essbase Server. Report writer has a rich formatting support and you can write a report to output the query result in a presentable fashion. Of course, you can also use the powerful Essbase spreadsheet Add-in interface to design robust Excel spreadsheets to retrieve data from Essbase.

On the other hand, MDX lets the user specify the query and supports a set of APIs to navigate over the query results. The Essbase Multidimensional Access Language (MAXL),
An axis consists of members retrieved from one or more dimensions. Two different axes may not select members from the same dimension for output. However, an axis may internally use a dimension that is already specified as an output dimension in a different axis for computation/member-filtering purposes. In addition to members and cross-members (referred to as tuples in MDX), MDX introduces a new entity called SET. A set is an ordered collection of tuple(s). All tuples within a set must have the same dimensionality and the dimension order of members within each tuple in a set must be identical.

MDX query execution involves independent processing of the set definitions along each axis present in the query and then returning the data points at the intersection of all possible combinations of members (member combinations) from all axis. Let us define MDX context as a calculation context that holds one member from every dimension present in the cube. Contents of MDX context are well defined by the query execution semantics as explained below.

MDX query execution begins with a default MDX context. The context has the dimension root members for all regular dimensions. Hence, the smallest valid MDX query supported in Essbase is “SELECT” which returns the cell-value at the intersection of top members from each dimension (and no members from attribute dimensions). In Sample/Basic, the intersection is (Year, Measures, Product, Market, Scenario).

Recall that an MDX query consists of a data request clause (SELECT), a context clause (FROM) and an optional point-of-view clause (WHERE). The SELECT clause determines the axis dimensions and up to 64 different axes can be specified in a single query. In practice, however, between 2 and 4 axes are more common.

<table>
<thead>
<tr>
<th>MDX Query Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall that an MDX query consists of a data request clause (SELECT), a context clause (FROM) and an optional point-of-view clause (WHERE). The SELECT clause determines the axis dimensions and up to 64 different axes can be specified in a single query. In practice, however, between 2 and 4 axes are more common.</td>
</tr>
</tbody>
</table>

| An axis consists of members retrieved from one or more dimensions. Two different axes may not select members from the same dimension for output. However, an axis may internally use a dimension that is already specified as an output dimension in a different axis for computation/member-filtering purposes. In addition to members and cross-members (referred to as tuples in MDX), MDX introduces a new entity called SET. A set is an ordered collection of tuple(s). All tuples within a set must have the same dimensionality and the dimension order of members within each tuple in a set must be identical. |

4. MDX is not an interface for churning out database exports, or very large production-type reports from the Essbase Server. The report writer interface supports streaming of the result set. Neither the server nor the client needs to hold on to the entire result set. You may potentially write a report to back up a portion of the database without needing the server to go offline. The Grid API (used by the spreadsheet interface) does not stream in the server and in the client. As a result, when a large query is executed, the server has to hold on to the entire result set before sending it to the client.

MDX, on the other hand, supports server-side streaming and as a result the server does not need to hold on to the entire result set. The server is capable of sending partial results to the client. In order to provide easy navigation over the query results, the API accumulates the entire result, before returning control to the caller. Even though the API stores the query results in an optimized compressed form, the client may not have enough memory to hold on to the entire result set.

<table>
<thead>
<tr>
<th>MDX Query Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall that an MDX query consists of a data request clause (SELECT), a context clause (FROM) and an optional point-of-view clause (WHERE). The SELECT clause determines the axis dimensions and up to 64 different axes can be specified in a single query. In practice, however, between 2 and 4 axes are more common.</td>
</tr>
</tbody>
</table>

| An axis consists of members retrieved from one or more dimensions. Two different axes may not select members from the same dimension for output. However, an axis may internally use a dimension that is already specified as an output dimension in a different axis for computation/member-filtering purposes. In addition to members and cross-members (referred to as tuples in MDX), MDX introduces a new entity called SET. A set is an ordered collection of tuple(s). All tuples within a set must have the same dimensionality and the dimension order of members within each tuple in a set must be identical. |

---

**MDX Query Example**

Query:

```
SELECT
(Year, Measures, Product, Market, Scenario)
```

Result:

```
105,522
```
Then the Slicer clause (i.e. singleton set definition in the WHERE clause), if present, is processed. Member(s) corresponding to the dimension(s) present in the slicer override the default context to form the new MDX context. This new context is used for the rest of the MDX query execution. Note that the slicer clause can only specify singleton members from each dimension.

For example, the following query

\[
\text{SELECT} \\
\text{WHERE (EAST, QTR1)}
\]

returns the data point at the intersection of (Qtr1, East, Measures, Product, Scenario). Note that Measures, Product and Scenario are the root members of the respective dimensions not explicitly specified in the query, and are part of the POV by default.

MDX also has an optional WITH clause that can be used to define calculated members and named sets.

The named sets are processed in order in which they are defined in the query. Note that named set evaluations may or may not need the MDX context. For example, if the named set definition is

\[
\text{With} \\
\text{Set Favourite_Colas as 'TopCount (Children ([100]), 2, Sales)'}
\]

the MDX execution does not need the MDX context to resolve the set definition.

In contrast, if the named set definition is the following, then the TopCount function execution needs to use MDX context (since the set construction is a function of data values in the cube).

\[
\text{With} \\
\text{Set Favourite_Colas as 'TopCount (Children ([100]), 2, Sales)'}
\]

If the slicer specifies to use Qtr1, we will get the top 2 children based on Qtr1 sales, whereas if the slicer specifies to use Qtr2, we will get the top 2 children based on Qtr2 sales.

Let us change the above query to add a slicer clause as follows.

\[
\text{With} \\
\text{Set Favourite_Colas as 'TopCount (Children ([100]), 2, Sales)'} \\
\text{Where (EAST, QTR1)}
\]

First, Children ([100]) will be resolved directly from the outline into [100-10], [100-20] and [100-30]. Next, the set will be pruned to include just the top 2 products based on their sales values.

In order to get any cell data values from Essbase, we need the members from all regular dimensions and, optionally, from attribute dimensions. The sales value for [100-10] does not resolve into one data point – representative members from the missing dimension(s) are required and hence are taken from the MDX context.
So the comparison to find top 2 will take place among the data values below:

(QTR1, Sales, [100-10], EAST, Scenario),
(QTR1, Sales, [100-20], EAST, Scenario),
(QTR1, Sales, [100-30], EAST, Scenario)

1. As an optimization the named-set definitions are processed only if they are later referenced in the query. If they’re not actually used, Essbase spends no time building them.

After any necessary named sets are resolved, the individual axis sets (i.e. set definitions along multiple axes) are resolved. The axis set resolution may or may not require MDX context for the same reason explained for named sets.

After all the axes sets are resolved (i.e. all the member combinations/tuples are found for a set), the final retrieval stage pulls data-values for all possible combinations of members (i.e. member combinations) from all the axes, using MDX context, as needed.

Let us next consider a slightly more advanced role that the MDX context plays using a class of functions called Iterative functions.

An iterative function in MDX is a function that works on a set of tuples performing an operation on every tuple in an iterative fashion. For example, the Sum function

\[ \text{SUM} (<\text{input-set}>, <\text{numeric-value-expression}>) \]

evaluates a numeric-value-expression for every tuple in the input set, adds all these values up to return the total. The following are some examples of MDX functions that operate on sets iteratively:

<table>
<thead>
<tr>
<th>SUM</th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>NONEMPTYCOUNT</td>
<td>FILTER</td>
</tr>
<tr>
<td>TOPCOUNT</td>
<td>BOTTOMCOUNT</td>
<td>TOPSUM</td>
</tr>
<tr>
<td>BOTTOMSUM</td>
<td>TOPPERCENT</td>
<td>BOTTOMPERCENT</td>
</tr>
<tr>
<td>ORDER</td>
<td>GENERATE</td>
<td></td>
</tr>
</tbody>
</table>
All the functions listed above take an expression argument like SUM. However, the expression type is different for each function. For example, the ORDER function requires either a numeric or string expression, GENERATE takes a set as the second argument and the FILTER function takes a conditional expression. The expression is numeric for the remaining functions.

Logically, an iterative function works as follows:
1. For every tuple in input set
2. Begin
3. Save the current MDX context (C1)
4. Create new MDX context (C2) overriding the iterating tuples on C1.
5. Evaluate the expression using C2
6. : 
7. : Iterative function specific tasks e.g. cumulatively add the values in case of SUM
8. Restore the saved context (C1) i.e. the current MDX context at this point is C1.
9. End

Now, let us consider an example to illustrate the above logic in steps. The following query sorts the Colas (i.e., Children of [100]) based on Actual Sales in Qtr1 and displays the sorted colas’ Budgeted Sales and Marketing in Qtr2

```
SELECT
CrossJoin ({Budget}, {Marketing, Sales}) on Columns,
Order (Children ([100]), (Qtr1, Actual)) on Rows
WHERE (Qtr2)
```

<table>
<thead>
<tr>
<th>Qtr2</th>
<th>Market</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Budget</td>
</tr>
<tr>
<td></td>
<td>Marketing</td>
</tr>
<tr>
<td>100-30</td>
<td>450</td>
</tr>
<tr>
<td>100-20</td>
<td>1160</td>
</tr>
<tr>
<td>100-10</td>
<td>1800</td>
</tr>
</tbody>
</table>

The ORDER function is specified in the set ‘Children ([100])’ (i.e. [[100-10], [100-20], [100-30]]) to iterate upon.

The expression used for ordering is ‘(Qtr1, Actual)’ – a numeric expression that needs to be computed for each element of the set. The fact that the cell value of (Qtr1, Actual) is required is implicit because it is a numeric expression. Recall that each axis in an MDX query is evaluated independently. Thus, the context at the beginning of the execution (using the above logic) of the ORDER function is the tuple ([Qtr2], [Measures], [Product], [Market], [Scenario]) where Qtr2 is obtained from the slicer clause.

In the first iteration the [Product] is replaced in context with [100-10], the first member in the set. Thus the tuple for the first iteration of ORDER is ([Qtr2], [Measures], [100-10], [Market], [Scenario]).

The numeric value expression is evaluated with this context, and thus the expression translates into the retrieval of the data point ([Qtr1], [Measures], [100-10], [Market], [Actual]).

The second iteration starts with the same MDX Context as the first iteration, namely, ([Qtr2], [Measures], [Product], [Market], [Scenario]). This time, [100-20] will replace [Product] to form the new MDX context.
So the expression evaluation will use ([Qtr2], [Measures], [100-20], [Market], [Scenario]) as the context and thus the expression translates into retrieval of the data point ([Qtr1], [Measures], [100-20], [Market], [Actual]).

The ORDER function finally sorts input tuples based on these values and arranges these tuples in ascending order (the default order).

All other iterative functions operate similarly. After evaluating the expression, the action taken depends on the function. In case of ORDER, the action is to sort tuples in input set based upon values evaluated. In case of SUM, the action is to add up all the value. In case of GENERATE the action is to create an output set by performing a “UNION” of all sets.

Continuing the discussion on current context, it is worthwhile to point out a special function called CURRENTMEMBER (which is no different from the corresponding calculator language function) which takes as input a dimension and is used to refer to the member present in the current MDX context during the course of execution.

The calculated member evaluation is like any other numeric expression evaluation. Depending on the context in which such a member is used, it can be resolved either at axes resolution time or during the final retrieval stage. For example, consider the query:

```
WITH
Member [Scenario].[Revised Budget] as 'Budget * 0.5'

SELECT
{Actual, [Revised Budget]} on Columns,
Children ([@100]) on Rows
FROM (Qtr1, Sales, East)
```

To return ([100-10], [Revised Budget]) cell value, the expression 'Budget * 0.5' will be evaluated in the context of ([Qtr1], [Sales], [100-10], [East], [Scenario]).

To return ([100-20], [Revised Budget]) cell value, the same expression will be evaluated in the context of ([Qtr1], [Sales], [100-20], [East], [Scenario]).

Next, we explore the notion of SOLVE_ORDER, a construct that enables removal of potential ambiguities in a query execution.

A SOLVE_ORDER clause can be specified following the definition of a calculated member to disambiguate the order of evaluation of calculated members in a query that has more than one such definition. We illustrate the use of such a clause by means of an example.

Q1 and M are two calculated members defined on Year and Measures dimensions respectively in the following query:

```
WITH
Member [Scenario].[Revised Budget] as 'Budget * 0.5'

SELECT
{Actual, [Revised Budget]} on Columns,
Children ([@100]) on Rows
FROM (Qtr1, Sales, East)
```
We have a question as to whether to calculate the \((M, Q1)\) cell by the \(M\) formula or by the \(Q1\) formula.

Let us compute the data value at the intersection \((Q1, M, \text{Product, Market, Actual})\) using the query execution rules we have learned thus far. There are two potential ways in which the data value can be computed: , using the definition of the calculated member \(Q1\) or the definition of \(M\).

If the formula for \(Q1\) is evaluated first, then the result will be calculated as follows:

\[
\text{Value of } (Q1, M, \text{Product, Market, Actual}) = \text{Value of } (\text{Jan, M, Product, Market, Actual}) + \text{Value of } (\text{Feb, M, Product, Market, Actual}) + \text{Value of } (\text{Mar, M, Product, Market, Actual})
\]

In this case, the value would be calculated as 55.1 + 55.38 + 55.26 = 165.74

If, however, the formula for \(M\) is evaluated first, then the result will be calculated as follows:

\[
\text{Value of } (Q1, M, \text{Product, Market, Actual}) = \frac{\text{Value of } (Q1, \text{Margin, Product, Market, Actual}) \times 100}{\text{Value of } (Q1, \text{Sales, Product, Market, Actual})}
\]

In this case, the value would be calculated as 52943 * 100 / 95820 = 55.25

As can be seen from the above two examples, there is a need for disambiguation and the solve order clause enables override of the calculation order to achieve the desired end result. In the presence of \texttt{SOLVE\_ORDER} clause, Formula corresponding to the member with highest solve order will be evaluated first. So, if the above query were altered as follows,
then the result will be 55.25, which is desired in this example.

The default SOLVE.ORDER of a calculated member is assumed to be 0.

If multiple members at the intersection have the same SOLVE.ORDER (or no explicit SOLVE.ORDER clause is present), for example,

```mdx
MEMBER [Year].[Q1] as 'Jan + Feb + Mar', SOLVE_ORDER = 10
MEMBER [Measures].[M] as 'Margin * 100 / Sales', SOLVE_ORDER = 10
```

then the tie is resolved using the order of respective dimensions as they appear in the Outline.

Thus, in the above example, since the Year dimension appears before Measures in the Outline, the formula for Q1 will be evaluated first and the resulting value will be 165.74. If you run the example query without specifying the SOLVE.ORDER, both the members are assumed to have SOLVE.ORDER 0 and the tie has to be resolved using dimension information. Hence, the result will be the same in previous case, 165.74.

### MDX Calculated Members and Security

Recall that MDX supports calculated members to define dynamic calculations.

Calculated members are similar to custom formulas that can be attached to dynamic-calc members defined in the outline. The advantage of using a calculated member is that no outline change is required, nor is the cube structure impacted to execute the custom calculation. While calculated members are functionally similar to custom member formulas, there is an important difference in data access privileges between the two.

In the case of dynamic-calc members, the administrator defines the member and the calculation formula associated with the member. Since the administrator is also responsible for enforcing user-level metadata and data security through appropriate filter definitions, it is assumed that the administrator is cognizant of security requirements for the dynamically calculated members as well. Thus, the formula associated with a dynamically calculated member does not enforce metadata or data security members and cell data which is otherwise hidden from the user can be used in the calculation.

In MDX, in contrast, the end-user can define the calculation in a query. The administrator may not know about the formula and the dependent data accesses during query execution. Therefore, the MDX implementation ensures that execution of a calculated member formula only uses data and metadata that the user has access to. The value of any data-point referred by the formula execution that the user has no access to is assumed to be MISSING.

For example, while evaluating a calculated member formula, if the Budgeted Sales for Qtr2 is not accessible to the user (suppressed via a data security filter), it will be assumed to be MISSING.
MDX AND MEMBER ORDERING

Functions such as <IDesc and <Gen in report writer and Descendants() and Members() in MDX return a collection of members. Between a report script and an equivalent MDX query, there is a difference in the order in which members are returned in the result set. Whereas the report writer interface returns members of the dimension tree in an order where all children of members appear before the parent, MDX metadata-based set-functions return members in an order where the parent appears before the children. The MDX function Hierarchize (<set>, POST) allows you to reorder the elements of <SET> in post-order form if desired.

MDX AND DYNAMIC TIME SERIES MEMBERS SUPPORT

Report writer and other interfaces like Grid-API use dynamic time series (DTS) members defined in the outline along with another time dimension member that specifies the latest time-period. MDX does not recognize DTS members defined in the Essbase outline directly. However, MDX can return the same value as returned by requests such as Q-T-D(Aug) (i.e. sum of the values up to month of August in the current Quarter) through functions such as QTD(), YTD(), etc.. These functions take the time dimension member (that one would specify to identify the latest time period in the report writer interface) as an argument. Internally, MDX uses the generation number for the corresponding Essbase member Q-T-D/Y-T-D. For example, the MDX expression QTD(Aug) returns the set of all time dimension members of the generation of Aug up to and including Aug, that fall in the quarter that includes Aug.

We can define a calculated member to return sum/min/max/average of the values for the set returned by the set expressions QTD(), YTD(), etc. Defining a calculated member to Sum over the set returned by MDX QTD() function is logically equivalent to Q-T-D followed by <LATEST latestmember syntax in report writer. For example:

WITH

MEMBER Measures.[QTD Margin] AS 'Sum (QTD(), Margin)' ...

You can alternately use PeriodsToDate (<Layer>, <member>) in place of xTD() functions. For example, if you have a generation named Quarter already in your database, then PeriodsToDate (Quarter, Aug) is equivalent to QTD (Aug). Note that PeriodsToDate works on any dimension whereas the xTD() functions can work only on a Time dimension.

MDX AND SORTING

There are major differences in the way sorting commands work in MDX and report writer. In report writer, there are different commands to sort the members of a dimension and the rows of a report. In MDX, there is one function ORDER (<set>, <expression>) that performs all sorting. It evaluates the input value expression iteratively for every tuple in the set, and the expression can provide names, generation or level numbers, property or cell values to compare. Furthermore, sorting on multiple criteria can be carried out by nesting calls to Order. For example, the following report snippet and MDX snippet are logically equivalent:
However, the report writer sorting commands are stateful. That is, once a sort command is specified in a script, the sorting requirement will be applicable to all subsequent member selection commands in the script. The MDX Order() function, on the other hand, is applied to the input set to the function, and the result of the Order() function is a sorted set. Any other set expressions appearing in the same MDX query will not be effected by the presence of a prior ORDER command. Thus while <SORTNONE functionality is required in report writer language, such a command is meaningless in MDX.

Secondly, report writer sorting commands are applied to the members added with a member command such as <Children, <Descendants, etc. The MDX Order() function works for the entire input set, regardless of how it was created. For example,

```
<SORTMBR Namenames
Nov
<Children Qtr1
```

in report writer returns members in the following order: Nov, Feb, Jan, Mar.

Whereas

```
Order ({ Nov, Children(Qtr1)},
Year.CurrentMember.Member_name)
```

returns the members in the following order: Feb, Jan, Mar, Nov

The MDX expression

```
{ Nov, Order (Children (Qtr1),
Year.CurrentMember.Member_name) }
```

returns the members in the order [Nov, Feb, Jan, Mar], since the Order() function is only applied to the children of Qtr1.

### MDX and Advanced Filtering Options

We will compare report writer commands such as RESTRICT/TOP/BOTTOM /ORDERBY that are used to accomplish filtering with corresponding functionality in MDX.

The report writer retrieves all the data values as specified in the query and then executes the data-related filtering commands on the result. In other words, these commands are post-processing operations in report writer.

On the other hand, equivalent MDX functions are executed as a part of the individual axis processing. Since each of the axis is processed independently, the context of the execution may not be intuitive from report writer user’s perspective. Context processing rules described in section <> apply for MDX execution.

Consider a simple example,

```
SELECT
CrossJoin ({Qtr1, Qtr2}, {Sales})
on Columns,
TopCount (Market.Levels(0).Members, 5) on Rows
From Sample.Basic
```

The set definition along rows will be executed independently of the contents of what is specified on Columns. Since the <numeric
value expression>, which is used to identify the top 5 members along Markets, is not specified, TopCount function will retrieve the value using the current context of execution. In this particular case, the root member along Measures dimension will be used for evaluation, which is different from Sales specified on the columns.

Secondly, each of these report writer commands can be used once in a script in effect. If two or more different commands (TOP/BOTTOM and ORDERBY) are specified, then all of them have to work on the same row dimension grouping. On the other hand, the equivalent MDX functions have no dependency among each other. For example, TOPCOUNT implementation has no assumption about the operations/functions used to create the input set.

Lastly, the report writer TOP/BOTTOM and ORDERBY commands work on a group of members from the innermost row dimension by default, though the grouping dimension can also be input explicitly. The equivalent MDX functions TOP(BOTTOM)COUNT or ORDER have no such assumption about the dimension grouping. For example, the following report will get Top 5 Products for every market.

```<Sym><Column (Scenario, Year) Actual Budget Jan Dec <Top ( 5, @DataCol(4))><Row (Market, Product) <children Market <Idescendants Product !
```

Whereas the MDX expression

```TopCount {
    CrossJoin (Children (Market), Descendants (Product)),
    5,
    (Budget, Dec)
}
```

will return only 5 tuples of (Market, Product) combinations.

In order to get the same result as the above report script i.e. top 5 Products for every Market member, use Generate function as shown in the following MDX query:

```SELECT
CrossJoin ( {Actual, Budget}, (Jan, Dec)) on Columns,
Generate (Children (Market),
    CrossJoin ( {Market.CurrentMember},
    TopCount (Descendants (Product),
        5,
        (Budget, Dec)
    )
) ) on Rows
```

```<Sym><Column (Scenario, Year) Actual Budget Jan Dec <Top ( 5, @DataCol(4))><Row (Market, Product) <children Market <Idescendants Product !
```
The Generate function will iterate through each child of Market, and for each of those will combine (using CrossJoin) that market child with the top 5 descendants of Product.

**MDX and Suppression of Missing Values in Output**

Any multidimensional query language must have first class support to handle processing of missing or non-existent cell values.

The `SUPMISSINGROWS` command in report writer suppresses the display of all rows that contain only #MISSING values.

The equivalent construct in MDX is the `NON EMPTY` clause, which, if present on an axis specification will result in suppression of the tuple(s) that contain only #MISSING values for all possible combination of tuples from other axes.

`SUPMISSINGROWS` only works for dimensions formatted to appear along rows, whereas the `NON EMPTY` clause can be specified for any axis. For example, `NON EMPTY` on columns will check and suppress, if needed, tuples present along the columns. In addition, `NON EMPTY` can be specified for any combination of the axes present in the query.
Secondly, SUPMISSINGROWS logic works on every page in the report. On the other hand, NON EMPTY functionality logically works on all possible member combinations on all axes.

Consider an example,

```
<Page (Market)
<Children Market
<Row
<Idescendants Product
  {SUPMISSINGROWS}
<Column (Scenario, Measures)
  Actual Budget
  Sales    COGS Profit
```

The SUPMISSINGROWS will cause display of only the eligible Products per Market page in the report. (It may so happen that one product suppressed in one page (market) is displayed on another page (market), depending on the data distribution). By contrast, NON EMPTY logically takes into account all possible member combinations on other axes.

For example:

```
SELECT
  Crossjoin ({Actual, Budget}, {Sales, COGS, Profit}) on Columns,
  NON EMPTY Descendants (Product) on Rows,
  Children (Market) on Pages
From Sample.Basic
```
<table>
<thead>
<tr>
<th>REPORT WRITER</th>
<th>MDX</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;COLUMN&lt;br&gt; &lt;PAGE&lt;br&gt; &lt;ROW</td>
<td>On COLUMNS&lt;br&gt; On ROWS&lt;br&gt; On PAGES&lt;br&gt; On CHAPTERS&lt;br&gt; On SECTIONS&lt;br&gt; On AXIS (n) where n is the axis number</td>
<td>Report writer supports up to 3 axes. MDX supports up to 64 axes. The first five axes are named as COLUMNS (0), ROWS (1), PAGES (2), CHAPTERS (3) and SECTIONS (5) respectively. The axes numbering increases from faster to slower moving axes. Operations like suppress missing or filtering can only be done on member combinations along Row in report writer. MDX treats all axes uniformly in the sense that all such operations are valid for sets on any axis.</td>
</tr>
<tr>
<td>&lt;ASYM&lt;br&gt; &lt;SYM</td>
<td>MDX lets you define the set in any way you want. For example, the MDX set expression &quot;((Actual, Sales), (Budget, Profit))&quot; is logically equivalent to the report-script snippet &quot;&lt;ASYM Actual Budget Sales Profit&quot;. Whereas “CROSSJOIN ((Actual, Budget), (Sales, Profit))” along columns will produce a symmetric output like report snippet &quot;&lt;SYM Actual Budget Sales Profit&quot;. MDX does not need to assume symmetric/asymmetric columns, as it deals with the way the set is defined.</td>
<td></td>
</tr>
</tbody>
</table>
### Member Selection Commands

<table>
<thead>
<tr>
<th>REPORT WRITER</th>
<th>MDX</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ALLINSAMEDIM mbrName</td>
<td>mbrName.DIMENSION.members or mbrName.Members</td>
<td></td>
</tr>
<tr>
<td>&lt;ALLSIBLINGS mbrName</td>
<td>mbrName.Siblings or Siblings(mbrName)</td>
<td></td>
</tr>
<tr>
<td>&lt;ANCESTORS mbrName</td>
<td>Attribute (attMbrName)</td>
<td>No equivalent MDX construct available in 7.0</td>
</tr>
<tr>
<td>&lt;ATTRIBUTE attMbrName</td>
<td>&lt;CHILDREN mbrName</td>
<td></td>
</tr>
<tr>
<td>&lt;CURRENCY targetCurrency</td>
<td>mbrName.Children or Children(mbrName)</td>
<td>No equivalent MDX construct available in 7.0</td>
</tr>
<tr>
<td>&lt;DESCENDANTS mbrName</td>
<td>Descendants(mbrName).GENERATION.AFTER</td>
<td>Please see the section on MDX member Ordering.</td>
</tr>
<tr>
<td>&lt;DIMBOTTOM mbrName</td>
<td>mbrName.DIMENSION.Levels(0).Members</td>
<td></td>
</tr>
<tr>
<td>&lt;DIMTOP mbrName</td>
<td>Ancestor(mbrName, mbrName.Dimension.Generations(1))</td>
<td></td>
</tr>
<tr>
<td>GEN name, dimension</td>
<td>dimension.GENERATIONS(genNumber).Members</td>
<td>In case of MDX, the name refers to user defined Generation / Level Name. Default generation/level names like Gen0, Gen1, Lev0, Lev1 are not supported in MDX. Use dimension.Generations(genNumber).Members Or dimension.Levels(levelNumber).Members syntax to select members of a particular generation or level.</td>
</tr>
<tr>
<td>LEV name,dimension</td>
<td>dimension.LEVELS(levNumber).Members</td>
<td></td>
</tr>
<tr>
<td>&lt;GEN(dimName,genNumber)</td>
<td>dimension.GENERATIONS(genNumber).Members</td>
<td></td>
</tr>
<tr>
<td>&lt;LEV(dimName,levNumber)</td>
<td>dimension.LEVELS(levNumber).Members</td>
<td></td>
</tr>
<tr>
<td>&lt;ANCESTORS mbrName</td>
<td></td>
<td>No equivalent MDX construct available in 7.0</td>
</tr>
<tr>
<td>&lt;CHILDREN mbrName</td>
<td>CHILDREN(mbrName, mbrName)</td>
<td>Please see the section MDX and Sorting.</td>
</tr>
<tr>
<td>&lt;DESCENDANTS mbrName</td>
<td>Descendants(mbrName)</td>
<td></td>
</tr>
<tr>
<td>&lt;PARENT mbrName</td>
<td>(mbrName, mbrName.PARENT)</td>
<td>No equivalent MDX construct available in 7.0</td>
</tr>
<tr>
<td>&lt;LATEST mbrName</td>
<td></td>
<td>MDX does not support D-T-S members in ESSBASE. Please see the section MDX and Dynamic Time Series Members Support.</td>
</tr>
<tr>
<td>&lt;LINK (extractionCommand [operator extractionCommand])</td>
<td>UNION, INTERSECTION, EXCEPT, EXTRACT</td>
<td></td>
</tr>
<tr>
<td>&lt;MATCH (&quot;Member&quot;)Gen&quot;</td>
<td>Level&quot;,Pattern&quot;)</td>
<td>No equivalent MDX construct available in 7.0</td>
</tr>
<tr>
<td>&lt;ONSAMELEVELGEN mbrName</td>
<td>mbrName.GENERATION.Members</td>
<td></td>
</tr>
<tr>
<td>&lt;ONSAMELEVELS mbrName</td>
<td>mbrName.LEVEL.Members</td>
<td></td>
</tr>
<tr>
<td>&lt;PARENT mbrName</td>
<td>mbrName.PARENT</td>
<td></td>
</tr>
<tr>
<td>&lt;TODATE (formatString, dateString)</td>
<td>TODATE(formatString, dateString)</td>
<td></td>
</tr>
<tr>
<td>&lt;UDA (dimName, udaStr)</td>
<td>uda(dimName, udaStr)</td>
<td></td>
</tr>
<tr>
<td>&lt;WITHATTR (dimName, &quot;operator&quot;, value)</td>
<td>WITHATTR(member, STRING_LITERAL, [signed_numeric_literal</td>
<td>character_string_literal] )</td>
</tr>
</tbody>
</table>
### Sorting Commands

<table>
<thead>
<tr>
<th>Report Writer</th>
<th>MDX</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;SORTNONE</td>
<td></td>
<td>Equivalent MDX construct not required.</td>
</tr>
<tr>
<td>&lt;SORTASC</td>
<td>ORDER (&lt;set&gt;, &lt;value_expression&gt;, BASC</td>
<td>BDESC )</td>
</tr>
<tr>
<td>&lt;SORTDESC</td>
<td>The value expression has to be written using &lt;dimension&gt;.CURRENTMEMBER,property Where property can be GEN_NUMBER LEVEL_NUMBER MEMBER_NAME, MEMBER_ALIAS</td>
<td></td>
</tr>
<tr>
<td>&lt;SORTGEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;SORTLEVEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;SORTMBRTRANES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;SORTALTNAMES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## DATA ORDERING AND DATA RANGE COMMANDS

<table>
<thead>
<tr>
<th>REPORT WRITER</th>
<th>MDX</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERBY</td>
<td>ORDER (&lt;set&gt;, &lt;numeric_value_expression&gt;, BASC</td>
<td>BDESC)</td>
</tr>
<tr>
<td>RESTRICT</td>
<td>FILTER (&lt;set&gt;, &lt;conditional_expression&gt;)</td>
<td>Please see the section MDX and Advanced Filtering Options.</td>
</tr>
<tr>
<td>TOP</td>
<td>TOPCOUNT (set, index [numeric_value_expression])</td>
<td>Please see the section MDX and Advanced Filtering Options.</td>
</tr>
<tr>
<td>BOTTOM</td>
<td>BOTTOMCOUNT (set, index [numeric_value_expression])</td>
<td>Please see the section MDX and Advanced Filtering Options.</td>
</tr>
<tr>
<td>REPORT WRITER</td>
<td>MDX</td>
<td>COMMENTS</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SUPMISSINGROWS</td>
<td>NON EMPTY</td>
<td>Please see the section MDX and Suppression of Missing Values in Output</td>
</tr>
<tr>
<td>All other report writer format commands like WIDTH, ZERO-TEXT etc.</td>
<td></td>
<td>MDX does not support any formatting commands.</td>
</tr>
</tbody>
</table>
### COLUMN OR ROW CALCULATION COMMANDS

<table>
<thead>
<tr>
<th>REPORT WRITER</th>
<th>MDX</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Column or Row Calculation commands</td>
<td>MDX</td>
<td>MDX does not support any such commands. Calculated members can be used to perform custom calculations in query.</td>
</tr>
<tr>
<td>REPORT WRITER</td>
<td>MDX</td>
<td>COMMENTS</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>OutMbrAlt</td>
<td></td>
<td>Use the Api EssMdxSetMbrIdType and specify the property MEMBER_NAME or MEMBER_ALIAS in DIMENSION PROPERTIES clause for every axis. The current alias table will be used to return the alias.</td>
</tr>
<tr>
<td>OutAltMbr</td>
<td></td>
<td>Use the Api EssMdxSetMbrIdType and specify the property MEMBER_NAME or MEMBER_ALIAS in DIMENSION PROPERTIES clause for every axis. The current alias table will be used to return the alias.</td>
</tr>
<tr>
<td>OutAlt</td>
<td></td>
<td>Use the Api EssMdxSetMbrIdType to set ESS_MDX_MEMBERIDTYPE_ALIAS. The current alias table will be used to return the alias.</td>
</tr>
<tr>
<td>OutAltNames</td>
<td></td>
<td>Not needed in MDX.</td>
</tr>
<tr>
<td>OutAltSelect</td>
<td></td>
<td>No equivalent MDX construct available in 7.0.</td>
</tr>
</tbody>
</table>
FOOTNOTES

1 Note that Hyperion also supports the production of MDX results in XML format through its support for XML for Analysis (XMLA). XMLA opens up new possibilities in the integration of Essbase-held information in an enterprise. This paper will not discuss XMLA; please refer to the XML for Analysis API Reference in the Essbase Deployment Services technical documentation.