White Paper

A Best Practice Guide to Developing TM1 Rules

What You'll Learn in This Chapter:

- When to use Rules vs. Consolidations vs. TurboIntegrator
- Best Practice Rules Syntax and Coding Conventions
- Rules Precedence
- Rules and Historical Data
- Recursion and the Stack
- Feeders
- Rules for Statistical Functions

TM1 is an extremely flexible application development platform and includes all of the tools required to develop a complete application from end to end. The flexibility of the environment allows applications to be developed very quickly; application prototypes can be developed in a matter of hours and full applications can be developed in just days or weeks.

The TM1 development environment is also very fast to learn with newcomers to TM1 capable of developing their own applications within just a couple of days of training. The system is very intuitive, especially for those who have developed complex spreadsheet models in the past.

The flexibility described above enables applications to be built quickly and easily, however this level of flexibility can also have some undesirable side effects if projects are not developed or managed well. It is very easy to create "bad" applications that may solve specific business problems, but leave little room for future development and improvement.

To ensure that the TM1 platform is fully leveraged and systems are developed to a high standard there are two key principles that are recommended:

- Implementation of systems that follow best practice development principles
- Management of TM1 projects using a methodology that complements the technology

This document is intended to address the first point; that is to describe an approach to developing TM1 systems using best practice principles.

Why Use Best practice?

Using a best practice approach to developing systems is a way of developing systems that are:

- Efficient
- Consistent
- Reliable
- Easy to understand
- Easy to maintain
- Easy to extend

The intention of best practice is to establish a series of development guidelines and principles that are the result of the collective experience and learning of many TM1 developers over many TM1 projects and systems. Often this is the result of trying many different approaches to solving problems and settling on the approach that is most suitable in the widest variety of circumstances.

Best practice is also a way of making the overall development approach consistent across the entire TM1 system (or systems). This enables the system to be easily understood, maintained and extended by multiple developers with potentially different backgrounds and experience.

It is not the intention of best practice to dictate how problems should be solved, that will always be the creative responsibility of the developer or developers involved in developing TM1 systems. Instead it is a framework of guidelines and principles that allows the creative process of developing systems to result in an efficient and consistent outcome that is easily understood by fellow developers and allows the systems to be easily maintained and extended in the future.

In addition, using best practice allows the ongoing maintenance and development of the systems to happen faster and be more efficient. This is because new developers will be working with a system that is easy to understand and therefore faster to learn. It will also result in the build

up of a library of standardised code that acts as a blueprint for future development.

Introduction

Like most development platforms, there are many different ways of achieving similar outcomes when building TM1 systems, and this includes cube rules. Two different developers presented with the same business problem are unlikely to come up with the exact same design; in some cases their respective designs may be vastly different.

The TM1 rules language allows very complex rules to be created while also allowing the developer a great deal of flexibility. When designing rules, it is not just the calculation that needs to be considered, the query speed of the calculation is also extremely important as is the amount of memory a rule calculated set of values will require.

This white paper presents a best practice approach to developing TM1 rules to strike the right balance between effectiveness, performance and maintainability.

Rules vs. TurboIntegrator

Before describing the best practice approach to developing rules, an important consideration should be made: should a calculation be performed using rules or using a TurboIntegrator process? There are a number of questions to consider:

- **Timing**
  Are calculations required in real time? Generally if calculations are required in real time, which is often the case with budgeting and forecasting applications, then rules will likely be the most appropriate. If calculations are not required in real time it may be more appropriate that they are pre-calculated using a TI process.

- **Performance**
  The TM1 rules engine is extremely efficient, especially when using the skipcheck function to ensure the TM1 sparse consolidation algorithm is turned on. However there is still more overhead for TM1 to calculate and retrieve a rule generated cell than it is to retrieve a simple cell. Using TI, there will be an overhead to calculate a number and storing it, but the cost of that calculation will be experienced only once during the TI execution, subsequent retrieval of the data by the end user will be faster.

- **Calculations on Consolidated Cells**
  Some rule calculations will be required on consolidated cells such as ratios, average prices etc. Where calculations are required on consolidated cells, rules are the only option, TI processes cannot store data against consolidated cells

- **Transparency**
  Rule calculated values are more transparent to the end user as they can trace rule calculations through the TM1 cube viewer. Calculations performed and stored via a TI process are hidden from the user; they cannot trace the calculation logic.

- **Memory Consumption**
  When rules are used to perform a calculation, memory will be required for the rule calculated cells and their associated feeders. The memory for the feeders will always be in use as feeders are fully evaluated on server start-up and when the rules are modified and saved. However, rule calculated cells are “virtual cells” and will only utilise memory when the calculated cells are accessed either directly or via a consolidation which requires their value.

  Alternately, when TI is used to perform and store a calculation, the data is stored as a simple cell in the cube. Simple data cells are physically stored in the TM1 cube and therefore will occupy memory regardless of whether they are accessed or not.

  Which one of these methods uses more memory depends on usage of the calculated cells themselves. If all or most of the calculated cells are accessed and therefore evaluated by the TM1 engine, then more memory will likely be required by rules as there is additional memory overhead to store the associated feeders as well as the rule calculated values. If only a small portion of the calculated cells are accessed, then more memory will likely be required by TI as all cells will require memory regardless of whether they are accessed or not.

Rules vs. Consolidations

Another important decision when performing calculations is whether the calculation should be calculated using rules or using consolidated elements in a dimension hierarchy. Obviously consolidations have some limitations; they are limited to weighted additions and subtractions and cannot use conditional logic. Any calculations that don't fit this model must be performed in rules (or TurboIntegrator). However, calculations that do fit this model should be performed as dimension consolidations for the following reasons:

- Consolidations are "pre-optimised" and calculate much faster than rules
- Consolidations do not require feeding

A good example is splitting a model into "reporting cubes" with stacked time dimensions and "calculation cubes" with a single linear time dimension. Many time dependent calculations such as depreciation schedules, subscription revenue, underwriting earnings, etc require very complex rules in a cube with multiple time dimensions but can be performed much more efficiently by consolidations in a cube with a single time dimension.

Editing Rules

There are a number of different tools for editing TM1 rules as follows:

- Simple rule editor
- Advanced rule editor
- Rule worksheet (.xru file)

The key capabilities of each tool are listed in the table below:

Capability	**Simple Rule Editor**	**Advanced Rule Editor**	**Rule Worksheet**
Version Control	No version control.	No version control.	Versions of rules can be maintained in separate worksheet tabs. Only the active tab will be used when saving and compiling rules. Very easy to revert to previous version of rules if required.
Formatting	Plain text only, no formatting.	Some formatting for keywords and operators.	All formatting options available in Excel.
Find & Replace	No find & replace.	Simple find & replace.	Excel find & replace.
Regionalisation	No options for regionalising.	Rules can be grouped into regions using the **Region** and **End Region** keywords. Code in regions can be collapsed and expanded to make reading the rules easier.	Rules can be grouped using the Excel group and outline functionality. Grouping can be nested multiple levels.
Stability	Stable	Unstable on some versions of TM1.	Stable
Pre-requisites	No other software required.	Microsoft .net framework required.	Microsoft Excel required.
Usability	No ability to save "draft" version of rules if you are working on a large change and want to make an interim save before completing the change.	No ability to save "draft" version of rules if you are working on a large change and want to make an interim save before completing the change.	Rule worksheets can be saved as "draft" versions by using the Excel Save option. This allows interim saves to be made without affecting live rules.

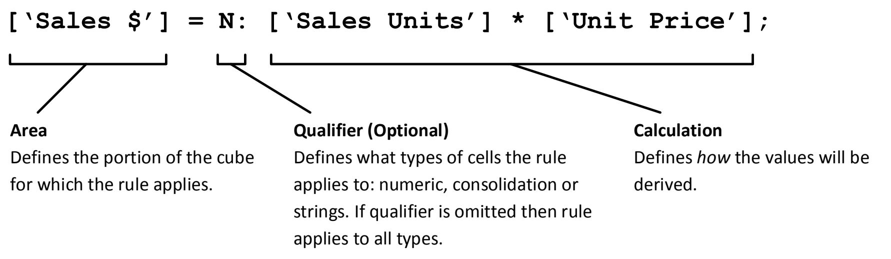
Based on these capabilities, it is recommended to use Rule worksheets for editing rules. The main reason is that rule worksheets are particularly good for maintaining "versions" of rules in separate worksheet tabs. This makes it easy to revert back to an older version if necessary and shows a working history of the changes.

Rules Syntax

There is a lot of flexibility in syntax when writing rules, which can lead to a lot of variation in style from different authors. When writing rules, it is important to strike the right balance between readability, maintainability and performance.

The Basics

TM1 Rules are a truly multi-dimensional language for writing calculations. The syntax for writing rules has some similarities to spreadsheet formulas e.g.:



```
['Sales $'] = N: ['Sales Units'] * ['Unit Price'];
```

Area
Defines the portion of the cube for which the rule applies.

Qualifier (Optional)
Defines what types of cells the rule applies to: numeric, consolidation or strings. If qualifier is omitted then rule applies to all types.

Calculation
Defines *how* the values will be derived.

Notation

There are two types of notation syntax, the shorthand notation which uses square brackets (shown above) and the longhand notation which uses the DB function. For example:

- **Shorthand:**
  ['Budget', '2010', 'Closing Balance']

- **Longhand:**
  DB('General Ledger', 'Budget', '2010', !Month, !Account, 'Closing Balance')

Note that the shorthand notation only requires the elements that are required by the calculation to be specified, in the above example these are Budget, 2010 and Closing Balance. The longhand notation requires all elements to be specified, in the above example these also include Budget, 2010 and Closing Balance as well as the other two dimensions in the cube, in this case Month and Account.

The area (left hand side of equal sign) must use the shorthand notation. The calculation (right hand side of equal sign) can use the shorthand notation, the longhand notation or a combination of both. All inter-cube references must use the longhand notation; intra-cube references can use either notation.

For intra-cube references the shorthand notation should be used where possible as it is more efficient and more compact than the longhand notation. Consider the following two examples which are for a cube "Product Sales" with dimensions: Version, Year, Week, Product, Customer and Sales Measure.

```
['Actual', 'Sales $'] = N:
  ['Sales Units'] * ['Unit Price'];

['Actual', 'Sales $'] = N:
  DB('Product Sales', !Version, !Year, !Week, !Product, !Customer, 'Sales Units') *
  DB('Product Sales', !Version, !Year, !Week, !Product, !Customer, 'Unit Price');
```

These expressions will actually perform the exact same operation but the first uses the shorthand notation and the second uses the longhand notation. Note that the first example is much easier to read and understand due to its more compact form.

However, there are some circumstances where the longhand notation must be used for intra-cube references including:

- **String Rules:**
  The shorthand notation only returns numeric vales. When accessing cells that contain string values the DB function must be used.

- **Rules that require logic to determine element names:**
  The shorthand notation only accepts literal strings and cannot contain operators or function calls; the longhand notation can.

TM1 does not require any specific dimension order when using the shorthand notation so elements may be specified in any order regardless of the dimension order within the cube. However, to enhance readability and maintainability, element names should be specified in the same order as they are contained within the cube. Taking this approach also leads to consistency between similar sets of rules.

As there is no specific order required by the rules engine when using the shorthand notation, TM1 scans the dimensions within the cube when interpreting a rules statement to determine which dimension the specified element belongs to. However, rules can become ambiguous when there are elements from different dimensions in a cube that have the same name. If a rule statement contains an element name that is ambiguous in this way, the rule will not work. The rule may work OK when it is first written as there may be no ambiguity at the time, but as new dimension elements are added the ambiguity may arise. This is difficult to detect until it is discovered by an end user or by someone modifying the rules file or analysing the server log files.

Consider the following rules statement:

```
['1000'] = N: ['Profit Before Tax'] * ['Tax Rate'];
```

In this example, the intention of the rule is to calculate the tax expense and assign the result to element 1000 in the Account dimension. However, what if there is also an element named 1000 in the Cost Centre dimension in the same cube? In this case the rule would not be able to determine whether the rule was intended to assign the result of the calculation to Account 1000 or Cost Centre 1000.

To avoid the possibility of ambiguity TM1 has a longer syntax for the shorthand notation that specifies both the dimension name and the element name. Using this longer notation, the earlier example now becomes:

```
['Account':'1000'] = N: ['Profit Before Tax'] * ['Tax Rate'];
```

Note that element 1000 is now preceded with the dimension name to avoid any ambiguity.

Theoretically, to avoid any possibility of ambiguity throughout a rules file it would be possible to precede every element used in the shorthand notation with its respective dimension name. However, this can lead to an overly verbose rules file where most of the qualified references would likely be unique and not require the prefix. Elements such as "Profit Before Tax" are unlikely to be used in multiple dimensions for example, so it would be overkill to prefix it.

For best practice, it is recommended that elements are prefixed using the longer syntax in the following instances:

- When specifying an element from a dimension that ***has*** a corresponding duplicate in another dimension (in fact the rule won't compile unless you include the prefix).

- When specifying an element from a dimension that ***may have*** a corresponding duplicate in another dimension in the future.

Structure

For elements specified in the area statement it is not necessary to re-qualify them in the expression (unless you specifically want to modify them). Consider the following example:

```
['Actual', 'Sales $'] = N: ['Actual', 'Sales Units'] * ['Actual', 'Unit Price'];
```

In this example, the "Actual" element used in the two expressions on the right hand side of the rules statement is redundant; it is implied from the left hand side and therefore does not need to be specified again. The following syntax achieves the exact same outcome:

```
['Actual', 'Sales $'] = N: ['Sales Units'] * ['Unit Price'];
```

Furthermore, consider a similar example which intends to apply the same rules to multiple elements from the same dimension, in this case "Actual" and "Budget". Using the original syntax two rules statements would be required as follows:

```
['Actual', 'Sales $'] = N: ['Actual', 'Sales Units'] * ['Actual', 'Unit Price'];
['Budget', 'Sales $'] = N: ['Budget', 'Sales Units'] * ['Budget', 'Unit Price'];
```

However, using the revised syntax, only a single rules statement would be required as follows:

```
[{'Actual', 'Budget'}, 'Sales $'] = N: ['Sales Units'] * ['Unit Price'];
```

Note that two elements from the same dimension were specified using the set notation { }. Because the elements specified on the left hand side are implied on the right hand side of the rules statement the rule will apply to both "Actual" and "Budget".

Elements specified on the left hand side of a rules statement are also implied on the right hand side when using the longhand notation. For example consider a similar rule to the previous example but this time the unit price is retrieved from a related price cube:

```
[{'Actual', 'Budget'}, 'Sales $'] = N:
  ['Sales Units'] *
  DB('Product Prices', !Version, !Year, !Week, !Product, 'Unit Price');
```

Note that the !Version reference in the DB function substitutes the respective version from the left hand side of the rules statement.

Qualifiers

As stated earlier, rules can be filtered with N, C and S qualifiers. For example:

```
['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
['Unit Price'] = C: ['Revenue'] \ ['Units Sold'];
```

Rules are seldom required for consolidated cells except in cases such as ratios and average prices. In the second example above, the Unit Price element is calculated at consolidated level as it would not make sense to add up the prices for all sales. The calculation at the consolidated level is effectively overriding the natural consolidation.

In general ***rules should always be limited to leaf cells*** (N: qualifier). Applying the rule to leaf level only and letting TM1's optimised consolidation algorithm do the "adding up" is much more efficient in terms of processing and response time.

Coding Conventions

Avoid Code Repetition

As with any coding language, code repetition leads to additional maintenance overhead and can make code more complex and confusing. This is also true for TM1 rules. Consider the following example:

```
['Version':'Actual', 'Product':'0010', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'];
['Version':'Actual', 'Product':'0011', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'];
['Version':'Actual', 'Product':'0015', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'];
['Version':'Actual', 'Product':'0121', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

All four rules use the same calculation logic but apply to different product codes. If the calculation logic needs to be changed in the future it will need to be changed in four places. Worse still, if only some of the rules are updated then the calculation logic will become inconsistent. In the above case, it is better to use the set notation { } and write the rule only once as follows:

```
['Version':'Actual', 'Product':{'0010', '0011', '0015', '0121'}, 'Revenue'] = N:
  ['Units Sold'] * ['Unit Price'];
```

Now the calculation logic is written only once but applied to four elements from the product dimension. Consider another example:

```
['Customer':'1005', 'Price'] = N: DB('Price', !Year, !Month, 'Coles', 'Wholesale Price');
['Customer':'1110', 'Price'] = N: DB('Price', !Year, !Month, 'Franklins', 'Wholesale Price');
['Customer':'1150', 'Price'] = N: DB('Price', !Year, !Month, 'IGA', 'Wholesale Price');
['Price'] = N: DB('Price', !Year, !Month, 'Other Grocery', 'Wholesale Price');
```

In this example, a number of hardcoded exceptions have been coded into the rule but the logic is virtually identical. Consider the following alternative:

```
['Price'] = N:
  DB('Price', !Year, !Month, AttrS('Customer', !Customer, 'Banner'), 'Wholesale Price');
```

This alternative allows a single rule to be written by using an attribute lookup to determine the type of price to be looked up.

Indentation and Spacing

Like most languages, the TM1 rules engine ignores spaces, blank lines and other whitespace when interpreting and compiling rules. However, using indentation and spacing can greatly assist in making code readable and easy to understand by other people in the future.

Consider the following rule:

```
['Actual',{'CustX','CustY'},'Sales$']=N:['UnitsSold']*DB('Cust Specific
Price',!Year,!Week,!Product,!Customer,'UnitPrice');
```

Note that this rule statement contains no spacing and is written across one line. Due to its length it has automatically wrapped to a new line. A little formatting can greatly improve the readability of this rule. Consider this alternative:

```
['Actual', {'CustX', 'CustY'}, 'Sales $'] = N:
  ['Units Sold'] *
  DB('Cust Specific Price', !Year, !Week, !Product, !Customer, 'Unit Price');
```

The rule has now been split over multiple lines and spacing has been introduced between all of the components of the rule. The second and third lines have been indented to indicate that the rule is continuing across multiple lines. The important thing here is to make the rule as easy to understand as possible to reduce the effort and cost of future maintenance.

Spaces should be used:

- After every comma
- Before and after every operator

Indentation should be used:

- Whenever a rule is written across multiple lines. All lines following the first line should be indented to indicate that the rule is continuing
- When using IF statements that span multiple lines

If function calls have many arguments that make the line too long, it is worth considering using extra lines and indentation to make it clearer. Using our earlier example, it could have been written as follows:

```
['Actual', {'CustX', 'CustY'}, 'Sales $'] = N:
  ['Units Sold'] *
  DB(
    'Cust Specific Price',
    !Year,
    !Week,
    !Product,
    !Customer,
    'Unit Price'
  );
```

Note that the arguments to the DB function have now been broken into separate lines for each argument and they have been further indented to make this clear.

Comments

The only way for someone reading a section of code that does not contain comments to understand its purpose is to work through the code line by line, statement by statement. Even then, the code does not necessarily indicate any assumptions or thoughts that the original developer had at the time of writing the code and can be very time consuming and tedious. In order for code to be easily maintained and extended in the future, it is imperative that any future developers can make sense of what was originally created.

The best way to explain the intention of a section of code is to write a comment in line with the code itself. Comments should be used to describe what the code is trying to achieve and detail any assumptions or other important thoughts that the developer had when writing the code. If necessary, comments can be written across multiple lines so the developer is not restricted when explaining the code.

As with most languages, comments are ignored by the TM1 rules engine and won't have any adverse impact on performance.

Comment should be used:

- Where there are major blocks of code a header comment should be used to clearly highlight its purpose.
- Before each rule (or block of similar rules) to explain what the intention of the calculation is.
- Anywhere else that it might assist future readers of the code to better understand its purpose.

Developers should not be concerned about commenting too much. The more comments you write the better chance the next person has to quickly understand and maintain the code you have written.

If using the Advanced Rules Editor, consider "regionalising" your code using Start Region and End Region statements. This will allow sections of code to be collapsed or expanded in the editor and makes navigating large rules files easier.

Element Names vs. Aliases

TM1 allows the use of either the principal name of an element or an alias in both the area statement and the calculations statement of a rule. Consider the following example:

```
Option 1: ['1000 – Widgets', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'];
Option 2: ['Product':'1000', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

Both of these rules do the same thing, perform a calculation and assign the result to element 1000 in the product dimension. Option 1 uses an alias which has been constructed from the product code and the product description. Option 2 uses the principal element name.

In this example, using the alias actually makes the rule more readable and, given that the alias contains the product description, it is more likely that the alias is unique across all dimensions within the cube and therefore less likely that the element needs to be qualified with the dimension name.

However, using the alias name has a major downside. If the alias for product 1000 is changed in the future then the rule will no longer work. In this example, it is likely that the product dimension is automatically updated via a TI that reads products and their names from a third party sales system. TM1 obviously cannot control what happens in the other system; therefore the product description could change at any time.

For this reason, it is best practice to always use principal element names rather than aliases. Although the rule statement may not be as easy to read, comments can be used to assist in conveying the intention of the rule to the reader.

Division by Zero

A classic problem in any language is that dividing a number by zero results in an undefined return value. Care should be taken to ensure that no rules calculation ever returns an undefined result as this will likely have ramifications for the integrity of the model.

TM1 has an inbuilt capability of dealing with division by zero by using the \ operator rather than the / operator for division. In the event that TM1 encounters a divisor of zero, it automatically returns a zero result for that operation.

In the absence of this capability, rules that contain the division operation would have to be written as follows:

```
['Actual', 'Unit Price'] = C:
  IF(
    ['Sales Units'] = 0,
    0,
    ['Sales $'] / ['Sales Units']
  );
```

Note that an IF statement would have been needed to check for a zero divisor first, and alternate logic used for zero and non-zero divisors. However, with TM1's inbuilt capability the following statement has the exact same outcome:

```
['Actual', 'Unit Price'] = C: ['Sales $'] \ ['Sales Units'];
```

For best practice, the backslash operator \ should always be used when performing division operations.

Position Based Functions

TM1 has a number of inbuilt rules functions that retrieve an element from a dimension based on its relative position in the dimension. These functions should be used with caution as you may write a rule that is based on the structure of the dimension today and assume that the dimension structure remains consistent over time, but this is not something that is guaranteed. In fact some dimensions may be subject to considerable re-organisation over time such as:

- Changes to the order of dimension elements
- New elements being inserted and existing elements being removed
- Changes to dimension hierarchies where new levels may be added between existing levels
- Additional hierarchies added to the dimension

By way of an example, consider a time dimension which has dates at the leaf level. Within this dimension there are two hierarchies, one that consolidates dates into weeks and one that consolidates dates into months. For example, consider the following hierarchies:

First Hierarchy	Second Hierarchy
…	**…**
July	**Week 1**
01/07/2010	01/07/2010
02/07/2010	02/07/2010
…	…
31/07/2010	07/07/2010
…	**…**

Now assume that this dimension is used in a sales cube that contains daily sales units, prices and revenue. Daily sales units are loaded into the sales cube from the main sales system and monthly pricing information is loaded into a separate cube that hold prices by month. In order to calculate revenue the system needs to multiply sales units by price. The first rule will need to retrieve the price from the price cube; the second rule will need to calculate the revenue. It is tempting to write the following rules:

```
['Unit Price'] = N: DB('Price', !Product, ELPAR('Date', !Date, 1));
['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

Note the use of the ELPAR function in the first rule. The ELPAR function returns the parent of a dimension element and takes three arguments: the name of the dimension, the name of the element, and a number indicating which parent to use. It is the last argument that causes this function to potentially act in a different way than intended.

The use of the ELPAR function in the above example will work fine today, because the monthly hierarchy is the first hierarchy in the dimension and the weekly hierarchy is the second hierarchy. Therefore, specifying a 1 as the final argument returns the month name which in turn is used in the DB function call to retrieve the Unit Price from the Price Cube.

However, if the dimension hierarchy is modified in the future and the order of the hierarchies is switched, or if an additional hierarchy is added to the dimension before the existing monthly hierarchy then the ELPAR function will now return a different result. If the ELPAR function returns anything other than a month name, then the DB function that retrieves the Unit Price from the Price cube will not work correctly.

This is why extreme caution needs to be taken when using any functions that return dimension elements based on their relative position within the dimension. In fact, it is generally better to avoid them completely.

In the earlier example, an alternative would have been to create an attribute on the Date dimension called Month, populate this attribute with the month that relates to the date and then use the attribute in the rule. The rules would then read as:

```
['Unit Price'] = N: DB('Price', !Product, ATTRS('Date', !Date, 'Month'));
['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

Note the use of the ATTRS function rather than the ELPAR function. This function will now return the month which relates to the date under consideration, and will always return a month regardless of any future changes to the dimension hierarchy.

Precedence

As specified earlier, the area of the cube that a rule applies to is specified by the left hand side of the rules statement. However, it is possible for there to be multiple rules in a rules file that have overlapping areas. Consider a very basic sales cube as follows:

- **Version:** Actual, Budget, Forecast1, Forecast2, Forecast3
- **Year:** 2010, 2011
- **Month:** Jan, Feb, Mar, ..., Dec
- **Customer:** Cust1, Cust2, ..., CustX
- **Product:** Prod1, Prod2, ..., ProdX
- **Measure:** Units Sold, Unit Price, Revenue

Assume that actual sales units and revenue are loaded from a central sales system but budget and forecast sales are input directly into the cube.

Now consider the following rules:

```
['Actual', 'Revenue'] = N: STET;
['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

The first rule is restricted to the Actual element of the Version dimension; however the second rule does not specify an element from the Version dimension and therefore applies to all elements of the Version dimension. These rules clearly have some overlap as the second rule applies to all elements from the version dimension, including Actual.

TM1 uses an order of precedence to determine which rule actually applies to a given cell in these circumstances. The rule file is processed from top to bottom and once an area of a cube has been assigned a rule all subsequent rules for the same cube area are ignored.

In the earlier example, the rule that contains the Actual element from the Version dimension will be applied first, thereby "claiming" the Actual element for the revenue calculation. Effectively, the second rule will apply to all elements from the Version dimension ***except*** the Actual element.

But what if the order of the rules was reversed as follows?:

```
['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
['Actual', 'Revenue'] = N: STET;
```

In this case, the first rule would apply to all elements in the Version dimension including Actual and the second rule would not work at all. This is because as soon as the Actual element was "claimed" by the first rule it is not possible to be redefined later in the rules file.

It is therefore important to write rules that take this order of precedence into consideration. For best practice, the following approach should be taken:

- Always write rules from the smallest area to the largest area (from most specific case to most general case).
- For more complex rules, consider breaking rule into a series of smaller rules and use the CONTINUE statement.

Rule precedence can be tricky but can also be very useful. Our earlier example has already demonstrated that TM1 can apply a specific rule first and then let the general rule "catch" all remaining cases. Alternatively our earlier example could have been written in a single statement using conditional logic as follows:

```
['Revenue'] = N: IF(!Version @= 'Actual', STET, ['Units Sold'] * ['Unit Price']);
```

However, this is not as efficient as the rules engine now needs to evaluate the conditional logic for every cell that the rule applies to. In addition, the more conditions required by a rule the more complicated the IF statement becomes. Consider the following rule:

```
['Revenue'] = N: IF(!Version @= 'Actual', STET, IF(!Version @= 'Budget', ['Units Sold'] *
['Unit Price'] * ['Budget Growth Rate'], ['Units Sold'] * ['Unit Price']));
```

This statement now has two specific rules and one general rule. The conditional statement has now become much more confusing than the earlier example. The same outcome could have been achieved as follows:

```
['Actual', 'Revenue'] = N: STET;
['Budget', 'Revenue'] = N: ['Units Sold'] * ['Unit Price'] * ['Budget Growth Rate'];
['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

This is a much clearer (and more efficient) way of specifying the rule making it easier to maintain and troubleshoot. Rule precedence should always be used in favour of complicated conditional statements.

It should be noted though that further flexibility is possible by using the CONTINUE statement. If invoked, this allows the querying "transaction" to continue past the current rule statement and check itself off against other statements below it. This can be used as a useful alternative to a complex nested use of the IF statement.

Rule precedence applies only to the ***rule area*** and has no impact on calculation dependency. TM1 will manage calculation dependencies in a similar way that a spreadsheet does. For example, consider the following:

```
['Revenue'] = ['Unit Sales'] * ['Unit Price'];
['Unit Price'] = DB('Prices', !Version, !Year, !Month, !Product);
```

Note that the calculation for revenue uses the unit price element, but the calculation for unit price appears after the calculation for revenue. This will not cause any problems as TM1 will completely evaluate each reference before using it in a calculation and "chase down" all dependencies in the process. The order that rules are specified should always be based on rule precedence and never on calculation dependencies.

STET

The STET statement has already been used in previous examples; however its use needs to be explained in a little more detail.

The STET statement is used to "turn off" rules for given areas of a cube. You might ask "why not just write the rule to exclude the specified area of the cube instead of using STET". The answer to this is that STET allows rules to be written that, when using a combination of STET and rules precedence, can create a much more elegant and maintainable rule. Consider again the following example:

```
    ['Actual', 'Revenue'] = N: STET;
    ['Revenue'] = N: ['Units Sold'] * ['Unit Price'];
```

The first rule uses the STET statement to turn off the rules for Actual Revenue. The second statement then applies the Revenue calculation for all other versions except Actual. If the first rule was not used, the second rule would have to specifically exclude the Actual element from the calculation. This could be done using an IF statement, however this would add complexity to the statement and would slow down processing due to the evaluation of the condition in the IF statement. It could also be done using the Area section of the rules statement as follows:

```
    [{'Budget', 'Forecast1', 'Forecast2', 'Forecast3'}, 'Revenue'] = N:
      ['Units Sold'] * ['Unit Price'];
```

However, this statement has now introduced maintenance overhead by specifying each element from the Version dimension. If a new element gets added to the Version dimension such as Forecast4, it will also need to be manually added to the rule.

By using the earlier STET rule no maintenance would be required as the rule applies to all versions "except" the Actual version.

Therefore, for best practice, if a rule is required for an area of a cube with only a small exception, use STET for the exception and then write a general rule.

Historical Data

One of TM1's greatest strengths is its ability to be used as a planning tool for creating plans, budgets and forecasts. The process of building up plans, budgets and forecasts is generally a cyclical process which occurs on an ongoing basis, such as annual budgets and monthly forecasts. After the TM1 system has been used for several "cycles" some of the data held in the older cycles will become static and never change again. For example, the annual budget for last year may now be locked and a new budget cycle has started for the current year.

During the build up of data within each cycle, rules are commonly used to calculate income and expenses based on input of driver quantities. For example, a sales budget may require the input of unit sales and prices and the revenue figure is calculated by a TM1 rule.

Once the cycle has been completed and the data becomes "static" it is worth considering the conversion of the data within that cycle from rule calculated to static values. There are several advantages to this approach:

- TM1 will retrieve data from historical cycles faster as it no longer has to calculate the data.
- Older cycle specific rules can be removed which leads to less maintenance when modifying rules.
- Data is protected from any "accidental" change to a rule calculation as it is no longer derived from a rule.

The process for converting from rules to static values is relatively straightforward as follows:

- Export all data from the calculated cycle to a text file. Make sure that the export includes rule calculated data.
- Remove rules from the calculated cycle
- Re-import data from the first step into the same cycle

It is also possible to introduce additional "versions" of data as an interim step to ensure data reconciles between the rule calculated and static versions of the data.

Recursion and the Stack

The TM1 rules engine is extremely flexible and allows rules to be written that are "recursive" in nature. Recursive rules are commonly used for calculating opening balances in a cube. The general form is as follows:

- When the system first comes online an initial period is selected and the opening balances are loaded into the "Opening Balance" element within the cube
- The closing balance in each period is calculated as the opening balance at the start of the period plus/minus any movement that occurred during the period and stored against the "Closing Balance" element within the cube
- The opening balance for all periods after the initial period is a rule calculation that replicates the closing balance for the prior period

Consider a very simple example:

	Jan	Feb	Mar	Apr
Opening Balance	1,000	1,460	1,043	771
Movement	460	(417)	(272)	299
Closing Balance	1,460	1,043	771	1,070

In this example, the opening balance for Jan has been loaded into the system and then the opening balance for all subsequent periods is calculated. The Closing Balance element is a consolidated element that adds together Opening Balance and Movement. The rules for this example are as follows:

```
[‘Opening Balance’, ‘Jan’] = N: STET;
[‘Opening Balance’] = N:
  DB(‘Stock’, ATTRS(‘Period’, !Period, ‘Prior Period’), ‘Closing Balance’);
```

The second rule is recursive, it needs to work its way back to Jan to work out the opening balance for any future period. Consider the opening balance for Apr; to calculate this number TM1 would use the following calculation logic:

- **Step 1:** Opening Balance for Apr = Closing Balance for Mar
- **Step 2:** Closing Balance for Mar is a consolidation that includes Opening Balance for Mar, so:
- **Step 3:** Opening Balance for Mar = Closing Balance for Feb
- **Step 4:** Closing Balance for Feb is a consolidation that includes Opening Balance for Feb, so:
- **Step 5:** Opening Balance for Feb = Closing Balance for Jan
- **Step 6:** Closing Balance for Jan is a consolidation that includes Opening Balance for Jan, so:
- **Step 7:** Opening Balance for Jan = 1000

In this very simple example it took 7 steps to work its way to the first period and then had to work its way back through the chain to return the rule calculated result to Apr. Now imagine that our simple example was actually used in an inventory cube that has 10 years of data that is stored for every hour of the day. This would result in up to 10 * 24 * 365 = 87,600 levels of recursion!

Each time TM1 works through a layer of recursion it creates an entry on the “stack” to store where it was up to in the calculation before entering the next layer of recursion. There are limits to the number of layers of recursion TM1 can handle before the system encounters a “Stack Overflow” error. The limit on the TM1 stack differs for different versions and platforms of TM1, however care should be taken so that rules do not result in stack overflows.

The best practice approach to solving this potential problem is to limit the amount of recursion that occurs for any given rule. The best way to do this is to create break point in the rule so that it doesn’t loop infinitely. Generally, the best way to do this is to use a TI process to carry forward the closing balance from one major cycle to the next rather than using a rule. For applications that store data at a monthly level, this breakpoint may be set once per year. For applications that store data at weekly, daily or more granular levels a breakpoint every quarter or month may be more appropriate.

Feeders

Background & History

From the very first version, the fundamental principle that makes TM1 an extremely efficient and scalable system is the ***sparse consolidation algorithm***. This algorithm effectively "skips" all data points that do not have a value so that only those cells that do have values are accessed when computing consolidated cells. This in turn allows query results to be returned to end users extremely quickly.

When cube rules were first introduced to TM1, it greatly improved TM1's computational capability; however it introduced a new challenge for the sparse consolidation algorithm. As rule-calculated cells are "virtual" cells that are not physically stored in the cube, the sparse consolidation algorithm cannot differentiate between rule calculated cells and blank cells.

If the sparse consolidation algorithm was left on, it would skip over rule calculated cells when computing consolidations, therefore returning an incorrect result. If the sparse consolidation algorithm was turned off, it would return the correct result but the system became considerably slower as it now has to check every cell including those with no values.

Clearly, both of these options are undesirable which resulted in the concept of feeders being introduced. Feeders are a way of allowing cubes that contain rules to continue to leverage the performance benefits of the sparse consolidation algorithm, but at the same time ensuring that rule calculated cells were not skipped when consolidated cells were being calculated.

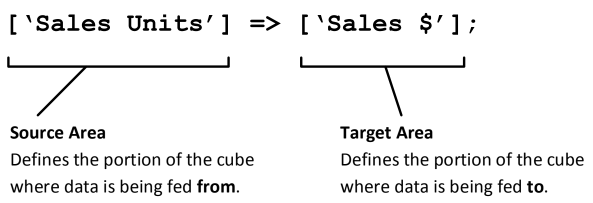
By default, when a new cube is created it uses the sparsity algorithm. However, when a rules file is created for a cube, the system turns off the sparsity algorithm to ensure all rule calculated cells are included when TM1 calculates consolidated cells. It is up to the author of the rules to re-enable the sparsity algorithm using the SKIPCHECK command at the very start of the rules file and to write the appropriate feeders to ensure the correct results are returned.

All cubes that contain cube rules should use feeders. It is only through the use of feeders that the best possible performance of a cube can be realised.

The only exception to using feeders on a cube that contains rules is in cases such as two dimensional system and control cubes.

Basic Syntax

TM1 feeder statements follow a very similar syntax to rules statements e.g.:



```
['Sales Units'] => ['Sales $'];
```

Source Area
Defines the portion of the cube where data is being fed **from**.
Target Area
Defines the portion of the cube where data is being fed **to**.

The theory of feeders is very simple: at its core, a feeder is simply the inverse of a rule. More on this later.

Core Principles

Unlike rules, feeders only ever apply to leaf level cells and never to consolidated cells. However, consolidated elements can be used in the specification of a feeder statement as a shorthand way of specifying all leaf elements within the consolidation. When specifying a consolidated element in a feeder statement the following occurs:

- Feeding ***from*** a consolidation means all leaf descendants of the consolidation will feed if there is a value present. For example if the consolidated element contained four leaf elements but only two of these contained a value, then only those two would feed.

- Feeding **to** a consolidation means that all leaf cells under the consolidation will be fed. For example if the consolidated element contained four leaf elements then all four leaf elements would be fed.

This is an important principle as some people mistakenly believe that it is the consolidation that is doing the feeding or being fed. This can make debugging rules and feeders more difficult if this principle is not well understood.

When a feeder is applied it sets a single byte "flag" or "pseudo data" in a leaf cell to signal that it should be consolidated. This ensures that the sparse consolidation algorithm does not skip this cell when computing consolidated values. Once fed, a cell stays fed until either the server is restarted or the cube is unloaded.

In general, feeders are not required for C level rules. The only exception is when a rule is applied to a consolidated element where it does not have values in any of its child elements.

Underfeeding and Overfeeding

It is the role of the rules author to ensure that calculated cells are fed correctly. Care should be taken to ensure that feeders feed exactly how many cells as is necessary, no more and no less. There can be considerable side effects if a model is not fed correctly.

Underfeeding occurs when some or all of the values that are being calculated are not fed. In most cases this will lead to incorrect results when reviewing consolidated data in the cube. Underfeeding must be avoided, as it undermines the integrity of the model. When writing feeders, you should always start with the inverse of the rule you are feeding to ensure feeding is adequate.

Overfeeding occurs when (a) cells that don't contain rule calculated values are being fed; or (b) when rule-calculated cells that result in a zero value are being fed. Both of these situations should be avoided but especially (a). While overfeeding will not result in incorrect values in the cube it has a detrimental effect on system performance. The time taken to feed cells that don't require feeding is wasted, and all unnecessary feeders occupy memory which is also wasted. Overfeeding can cause a significant explosion in memory.

Coding Conventions

The same coding conventions that were described for rules generally also apply to feeders. Use shorthand notation where possible and always use the principal element name as opposed to an alias. Use comments and spacing to make the feeder as clear as possible to the reader.

In the case of inter-cube feeders, it is good practice to place a comment in the feeders section of the cube being fed as a placeholder so that the reader knows there is a feeder from another cube.

Where it is necessary to feed multiple target areas from a single source area, avoid code repetition:

```
['Sales'] => ['Returns'];
['Sales'] => ['Closing Stock'];
```

by writing a single feeders statement and separating the targets by commas:

```
['Sales'] => ['Returns'], ['Closing Stock'];
```

Conditional Feeders

A conditional feeder is a feeder that will feed different target cells depending on conditions in the feeder statement. A conditional feeder must use the DB( ) function and the conditional logic must reside in one of the arguments to the DB( ) function. For example:

```
['Sales'] => DB( IF( !Product @= '01', 'CatASales', IF( !Product @= '02', 'CatBSales', IF(
!Product @= '03', 'CatCSales', ' ' ))), !Version, !Year, !Week, 'Sales');
```

Note in this example that the conditional logic is contained in the argument for the cube name. However, conditional logic could alternately exist in any of the arguments for the dimension elements depending on the nature of the feeder.

Conditional feeders can be a useful feature when creating TM1 cube rules and can help to avoid overfeeding; however there are some fairly major side effects to using conditional feeders that must be taken into consideration. If you have an alternative to using conditional feeders then use it.

Firstly, by default a feeder only ever fires once. What this means is that at the time the feeder is evaluated it will feed whatever target cells are identified by the conditional statement at that point in time. If the condition subsequently changes the feeder does not re-fire which could potentially result in underfeeding and an incorrect result.

Secondly, conditional feeders add significant overhead to the rule evaluation / feeder processing of saving a rule or loading a cube.

Thirdly, if the condition in the conditional feeder contains a reference to a fed value then the TM1 server can only be started in single-threaded rather than multi-threaded start-up mode. This can significantly add to server start-up times.

There is a parameter (`ReevaluateConditionalFeeders=T`) in the TM1 server configuration file that can be set which will cause conditional feeders to be re-evaluated if one of the components of the condition changes in value. However, bear in mind that there will be a performance overhead when processing rules and feeders as a result.

Calculating Statistical Functions

TM1 rules are limited or unsuitable for advanced statistical calculations (and even many basic ones are less straightforward in TM1 than other applications). For example simple statistical/numerical concepts such as Min, Max, Count and Average, although possible in TM1, are not straightforward to implement.

Modelling statistical calculations with rules can add significant overhead to any model and slow performance down. In most cases pre-calculation is preferable by either: i) loading as values with TI performing calculations or ii) performing the calculations in a database designed and optimised for this purpose and loading as values directly into TM1.

The following example demonstrates the complexity of calculating a simple “average” in TM1.

Example

This example uses a cube that contains information from a survey. The survey asked respondents to rate their satisfaction on a number of areas by entering a numeric score between 1 and 10 for each question. It also asked for information such as their gender, age and postcode.

The cube will be used to slice and dice the information from the survey for analysis purposes. Much of the analysis will made up of things like “what is the average score for question X by all respondents?” or “How does question X score for males vs. females?”. The survey respondents will be grouped using consolidations for their respective gender, an age bracket and a postcode grouping that further consolidates their location into cities, states etc.

In this analysis, a consolidated score for a question does not have much meaning as it is the rating out of 10 that conveys the true meaning. Therefore, instead of consolidating all of the data throughout the hierarchies we need to determine the average score at each consolidation point. The way to do this is by including two additional measures which will be used to record a count of respondents which will then be used to calculate the average.

So the first rule required will be to calculate the counter:

```
['Counter'] = N: IF(['Score'] <> 0, 1, 0);
```

If a score has been entered then it will be counted, otherwise it will not. The rule also needs a “ghost” measure that will be used to store the consolidated result which will then be used to determine the average. The rule for this element will be:

```
['Ghost'] = N: ['Score'];
```

We will leave the “ghost” measure to consolidate naturally. Finally, the rule to calculate the average (which will overwrite the natural consolidation) is:

```
['Score'] = C: ['Ghost'] \ ['Counter'];
```

Only the N: level rules will require feeding, the C: level rule for calculating the average score will not need to be fed as it already has data at leaf level.