Adventures in Groovy – Part 51: Dynamically Changing Storage Properties When Using Hybrid

With hybrid being used more and more there is a need to manage the storage methods of different levels of sparse dimensions. Whether it is a staggered hierarchy or not, getting the storage method from the source can sometimes be challenging. More often times than not, you may want to own it on the Planning side so you can change it at will and not have to go through the typical IT change order process that may take weeks, or even months, to go through the full development cycle.

Managing this manually would not be fun, especially if the hierarchy is loaded more often than monthly. Yes, you could use the Smart View admin option, but it is manual and let's face it, you have a ton going on and you will make mistakes.

In Comes Groovy

With Groovy, a calculation can be written to update metadata. I have talked about this in several other posts, but I am going to walk through a couple specific examples that are for specific situations. I think this will spark some interest of taking this further for situation similar, or completely different.

Reusable Concepts

Before I jump into the situations and examples, there are a couple techniques that will be reused in all the examples. Rather than repeatedly explain them, let me first introduce them.

First, this situation assumes that the storage methods are different for the plan types. This might be more unique, but it is easy to deal with. If this isn't the case, the properties in the example can be change to "Data Storage"

It is always a good idea to start every Groovy script off with the RTPS tag. To understand more about why this is important, read Part 49 This will be used in each example.

```
/*RTPS: */
```

Each example requires methods that have to have the dababase passed to it. The easiest way to get the cube the rule runs on is to use rule.cube. There are other ways to accomplish it, but this is the shortest and most dynamic.

```
List<Member> products = operation.application.getDimension("Product",rule.cube).getEvaluatedMembers("Descendants(Product)", rule.cube)
```

Each example gets the dimension and holds it in a variable. The method requires a pointer to a cube, or cubes. Often it is easier to pass the cubes in the application, rather than one cube, to make sure all artifacts are available and not hard coded. *operation.application.cubes as Cube[]* returns all the plan types as an array of variables that are of type cube.

A note about the parameters that can be used. It is much faster to use the same parameters that are used in planning, like the options in a data map. You CAN use most of the Essbase function. Oracle doesn't recommend them. They are slower, but if you are not iterating and running the request numerous times, I haven't noticed a difference. In this

example, it is executed once, so the performance degradation is minimal.

```
Dimension objDim = operation.application.getDimension('Product',operation.application.cubes as Cube[])
```

To get and set the properties of a member, the toMap method is used. This will return all the properties of the member and I wrote a summary of the use of this method in a prior post found in this post – Part 11 – Accessing Metadata

```
Map<String,Object> memberProps = it.toMap()
```

Lastly, if you aren't familiar with regular expressions, they can be of great use. I have a module dedicated to this in xxxxx. I struggled understanding regular expressions for years. But I promise you, if you take 4 hours and focus on learning them, it will click. To use it in Groovy, using the matches method allows this. Briefly, here are some basic concepts. A ^ means starts with. A $ means ends with. A dot means any character, and following that with an asterisk means many. So .* means one to many characters of any type

```
.matches("^.*Region$") || it.name.matches("^District.*$")
```

Setting All Parents To Dynamic

If you have a smaller hierarchy, one with maybe only a few levels, it might be advantageous to just set all the parents to dynamic. The following script iterates through all the product members and sets every parent to dynamic.

```
/*RTPS: */
List products = operation.application.getDimension("Product",rule.cube).getEvaluatedMembers("Descendants(Product)", rule.cube)
List<Member> lev0Products = operation.application.getDimension("Product",rule.cube).getEvaluatedMembers("ILvl0Descendants(Product)", rule.cube)
```

```
Dimension                    objDim                    =
operation.application.getDimension('Product',operation.applica
tion.cubes as Cube[])
products.each{
try{
       Map<String,Object> memberProps = it.toMap()
       if(lev0Products.contains(it)){
                              memberProps["Data  Storage
(${rule.cube})".toString()] = 'never share'
    }
    else{
                              memberProps["Data  Storage
(${rule.cube})".toString()] = 'dynamic calc'
    }
    objDim.saveMember(memberProps)
}
catch(Exception e) {
    println("Exception: ${e}")
    println it.name
}
}
```

Use Patterns To Set Parent Storage Property

In some situations, there are patterns to the levels of your hierarchy. Maybe you have regional levels that are definable and unique that can be used to set different levels to dynamic. Assume the following naming convention for this example

- Total Products
  - West Region (everything ends in Region)
    - District 1 (everything starts with District)
      - …
        - …

```
/*RTPS: */
// Get every product in the hierarchy
```

```
List<Member>                    products                    =
operation.application.getDimension("Product",rule.cube).getEva
luatedMembers("IDescendants(Product)" , rule.cube)
// Assign the producdt dimension to a variable
Dimension              objDim              =
operation.application.getDimension('Product',operation.applica
tion.cubes as Cube[])
// Loop through each product
products.each{
       // if the product matches these expressions, change the
app setting to dynamic
                  if(it.name.matches("^.*Region$")   ||
it.name.matches("^District.*$")  ||  it.name  ==  'Total
Products'){
              Map<String,Object> memberProps = it.toMap()
         memberProps["Data Storage (${rule.cube})".toString()]
= 'dynamic calc'
         objDim.saveMember(memberProps)
        }
        // otherwise change it to never share
    else{
              Map<String,Object> memberProps = it.toMap()
         memberProps["Data Storage (${rule.cube})".toString()]
= 'never share'
         objDim.saveMember(memberProps)
    }
}
```

More Complicated Possibilities

There are a bunch of other possible needs. Let’s say you have a need to make everything above level 3 dynamic. First, if the hierarchy is staggered, the same level can be a 1 and 5. You would have to decide how to handle that. I would lean toward if it was a level 1 and a 5, I would make it dynamic because that might also mean your level 5 and 9 in that portion of the hierarchy would be a pretty deep hierarchy to make 9 levels dynamic. Every situation is different, and performance would have to be evaluated, but the complexity of identifying how to set the storage in these situations is what

I am trying to explain.

If you want to use patterns, you may also want to ensure that the pattern isn’t replicated at a parent and level 0, so there may be a need to check for both a pattern and the level of the member.

Obviously, there are an infinite amount of possibilities and each one could introduce complexity. Just understand that almost anything can be defined by patterns and levels and can be accomplished, but the level of complexity of your logic or your regular expression may increase.

That’s A Wrap

The bottom line is that we now have the ability to do a lot of things we relied on the source system to do. Or, maybe external scripts were run using Perl, or VBScript, or PowerShell. We can use metadata properties, dynamic levels, any other repeatable pattern that might possibly come up. It is fast and reliable, and completely in our control so we don’t rely on others when it is broken or needs changed.