

Adventures in Groovy – Part 6: Converting a POV into a Fix

Introduction

One of the fundamental features of a Groovy calculation is the ability to dynamically grab aspects of a grid, and getting the POV is something that is required to dynamically generate an Essbase calculation. There are times when the entire POV is required, times when only members from specific dimensions are needed, and situations where only the rows and columns of the edited cells are used to construct effective fix statements. All 3 of these situations will be discussed.

Use Case (Pulling POV Members)

Many times, the Fix statement of a calculation being built includes all members in a data grid POV.

Code

```
List<String>          povMemberNames          =
operation.grid.pov*.essbaseMbrName
String calcScript = ""
Fix("${povMemberNames.join(' ', "'')}")
  [Calculation]
EndFix;""
```

Breaking It Down

The `operation.grid.pov` provides access to all the dimensions in the POV. From that, dimension names, member names, and other properties that are useful, can be accessed. When followed by a `*`, it returns all the dimensions in the POV to a list. Using the `essbaseMbrName` instructs the function to

return every member in the POV. The `povMemberNames` variable stores a list of all those values.

When building the `calcScript` variable, `${povMemberNames.join("", "'')}` will return the list, delineated with `","`. This would return something like `membername1","membername2","membername3`. This obviously is missing surrounding quotes, which is why it is embedded inside the quotes.

Use Case (Pulling Selective Dimension Members From The POV)

Pulling one, or all of the dimension values in a POV, in individual variables for later use, provides the ultimate flexibility. The following pulls all the members and stores them in a unique variable. Then, any dimension in the POV can be accessed. An easy way to accomplish this is to use the `find` method of a `List` object to filter the elements.

Code

```
List<String> povmbrs = operation.grid.pov
String curYear = povmbrs.find {it.dimName == 'Years'}.essbaseMbrName
String curCompany = povmbrs.find {it.dimName == 'Company'}.essbaseMbrName
Fix((${fixValues(curYear, curCompany)})
  [Calculation]
EndFix;"""
```

Breaking It Down

The first line stores the entire POV collection. That variable can be accessed and used like any Groovy collection. By using the `find` method, items in the collection can be filtered. `povmbrs.find {it.dimName == 'Years'}` will return the `Years` dimension object and one of the properties of that object is the `essbaseMbrName`. Using it will return an `Essbase`

friendly member name.

The “fixValues” method converts strings to “Fix friendly” strings that can be used in an Essbase calc script. Any of the following objects can be inserted into this method.

- AttributeDimension
- AttributeMember
- Dimension
- Member
- PeriodDimension
- RtpValue
- YearDimension

The result is the dimension member name inside a fix statement.

Rows/Columns

Some of the real efficiencies with Groovy stem from the fact that the edited cells can now be determined and a Fix statement can dynamically be generated to isolate longer running calculation on only the rows and columns that have changed. In the following example, there are two methods to get the members.

Code

```
Set<String> periods = []
Set<String> products = []
operation.grid.dataCellIterator({DataCell cell ->
cell.edited}, MemberNameType.ESSBASE_NAME).each { DataCell
cell ->
    periods << cell.periodName
    products << cell.getMemberName("Product")
}
```

```
String calcScript = ""
Fix("${periods.join(' ', ' ')}", "${products.join(' ', ' ')}")
[Calculation]
```

```
EndFix;"""
```

Breaking It Down

If you are unclear about how to iterate through a grid, read *Adventures in Groovy Part 3: Acting On Edited Cells*. Inside the `dataCellIterator`, the example assigns two list objects with the respective members. Since this only touches the cells that have been edited, only the periods and products that have been updated will be stored in the respective variables.

You may be wondering if this is truly the most isolated combination of data. Theoretically, the same product may not have the same edited months. You could further condense changes in the Fix statement by looping through the rows and creating a fix for every row. In this example, Period is dense, so doing this would not change the number of blocks. Depending on the application, taking this extra step might prove more efficient. Rather than complicate this further, we are assuming this isn't required.

Every required dimension (Account, Entity, Period, Scenario, Version, and Year) has its own method. `cell.periodName` returns the Period member of the cell. All the dimensions have the same naming convention.

For custom dimensions, `getMemberName` can be used with the required dimension passed as a parameter. If you want consistency, this method can also retrieve the 6 required dimensions. `cell.getmemberName("DimensionName")`, where `DimensionName` is an actual dimension, returns the respective member in that dimension.

Just like the previous example, add the variable to the Fix statement with a join and it returns the delimited list of members that have been edited.

- `"${periods.join(", ")}"` returns the list of Periods

that have edited cells, and

- “`${products.join(' ', '')}`” returns the rows (or products) that have been edited.

Wrapping Up

One last step that can be added, and should be, is the check to see if any cells have been modified by evaluating the size of one of the lists that is created during the grid iteration. After the iteration is completed, the following can be added to accomplish this. If there are no cells edited, the calculation is stopped at the return line and nothing is sent back to Planning/Essbase to execute.

```
if(products.size() == 0) {
    println("No edited cells found!")
    return
}
```

Joining the examples above, the fix would look like this.

```
String calcScript = ""
    Fix("${povMemberNames.join(' ', '')}", "${periods.join(' ', '')}", "${products.join(' ', '')}", ${fixValues(curYear, curCompany)})
    [Calculation]
EndFix;""
return calcScript
```

Conclusion

If you implement this, you will likely see huge performance improvements. In the client production applications, I have implemented this in, I see at least a 95% improvement in performance. This same logic can be used for Data Maps and data movements from AS0 to BS0 (which we will cover later)

Integrating Groovy Functions Into Calculations Produces Ground Breaking Performance Results

the following are actual results in a production application

