Adventures in Groovy – Part 3: Acting On Edited Cells

Introduction

With the introduction of Groovy Calculations this summer, one of the things I use most, especially for applications with data forms that include a large sparse dimension in the rows with suppression on, is the option to loop through cells and identify only the POV on the cells that have changed. In applications like workforce planning, or product level applications that have hundreds, if not thousands, of possible blocks, isolating only the changed data can have significant impacts on performance. Normally when a data form is saved, the fix includes all level 0 members of the employee dimension and must run the calculations on all of them, regardless of whether employee changed or not. Being able to fix on only a row, or the handful that change, give us a significant advantage in user response.

This post will go into how this is executed, and a few use cases to get you thinking about the possibilities. All of these I have developed and are in a production application.

The Code

Using a grid iterator, with the appropriate parameter, is an extremely easy way to deploy functionality that looks through ONLY the cells that have been changed.

```
  operation.grid.dataCellIterator({DataCell cell -> cell.edited}).each{
  [actions]
 }
```

The cell object, and all its parameters, are available inside

the loop. By adding {DataCell cell -> cell.edited}, the loop is isolated to only cells that have changed. The representative member names, the data value, if it is locked, has attachments, and many other things can be accessed with the example above.

Use Cases

An infinite number of uses are possible, as you are probably already thinking about. If not, the following will probably spark some creativity.

Customizing an Essbase Fix Statement

One of the most significant benefits of this is the ability to be able to dynamically generate a fix statement and filter what is calculated. Although the calculation on the Essbase side isn't improved just by using Groovy, the ability to dynamically write the calculation on only what changed is significant, especially when allocations, data pushes, and other longer running processes are required.

Assuming the rows of a data grid include the dimension Product, and Period is in the columns, the following will create variables that will include only the periods and products that have been updated. These can be used in the string builder that is passed to Essbase. So, rather than @RELATIVE("Product",0) running on all possible products, it can be replaced with "Product 1″,"Product 2".

The following creates list and string variable for Product and Period. Every cell that is updated will add the relative product and period to the list variables. After all the cell values have been read, the two string variables are set to include a unique list of the dimension members, surrounded by quotes, and separated by commas, which are immediately ready to include in the FIX statement.

```
def lstProducts = []
```

```
def lstPeriods = []
operation.grid.dataCellIterator({DataCell   cell   ->
cell.edited}).each{
  lstProducts.add(it.getMemberName("Product"))
  lstPeriods.add(it.getMemberName("Period"))
  }
def strProducts = '"' + lstProducts.unique().join('","') + '"'
def strPeriods = '"' + lstPeriods.unique().join('","') + '"'
```

The string builder would look something like this. In the following example, the other variables are pulled from the POV.

```
def        sScenario=povmbrs.find        {it.dimName
=='Scenario'}.essbaseMbrName
def         sCompany=povmbrs.find        {it.dimName
=='Company'}.essbaseMbrName
def sYear=povmbrs.find {it.dimName =='Year'}.essbaseMbrName

StringBuilder strEssCalc = StringBuilder.newInstance()
strEssCalc <<"""FIX($sScenario,
  $sCompany,
  $sYear,
  $strProducts,
  $strPeriods
)
 Revenue = units * price;
ENDFIX
"""
```

At this point, the strEssCalc value can be passed to Essbase and executed. If only 2 products are changed in 1 month, only those 2 cells would be calculated. If this data form included 12 months and 1,000 products, the calculation would take roughly 1/500th of the time.

Customizing Smart Push

Smart Pushes on forms, depending on the POV, can exceed a threshold of what a user perceives as acceptable performance. In the 17.11 release, Data Maps and Smart Pushes can now

embedded in the Groovy Calculations. The 2 huge benefits to this are that

1. the data that is pushed can be filtered to only the data that changes, decreasing the time of the operation, and
2. the ability control the operation order of when a push runs (for example, calculation, push, calculation, push)

If a data form has a smart push associated to it, it can be accessed and further customized. If not, data maps can also be accessed, customized, and executed.

One thing I learned from the Oracle development team is that the Smart Pushes have a max memory that can be accessed. One Smart Push may never hit that limit if it is isolated enough, but we found issues when multiple Smart Pushes were executed a the same time. We were seeing multiple, and intermediate, failures in the logs. So, it is even more critical to make these pushes as small as possible to eliminate that issue.

If we reference the example above in the customized fix, we expand on that and apply the same filter to the Smart Push. The only addition needed is to encapsulate the strProducts variable in quotes. If nothing is passed, it runs the smart push as it is setup in the form, so operation.grid.getSmartPush("appname").execute() would simply execute the same thing as if the Smart Push was set to run on save.

```
strProducts = """ + strProducts + """
if(operation.grid.hasSmartPush("appname"))
operation.grid.getSmartPush("appname").execute(["Product":strP
roducts,"Period":strPeriods])
```

Validate data

Having the ability to proactively perform data validation is another great addition with Groovy. Rather than running a calculation, and after the Data Form save returning a message telling the user that they have to change something, or

changing it for them, we now can interrupt the data form save and instruct the user to change it before it has any adverse impact on the data. The following will change the cells that violate the validation to red, add a tooltip, stop the form save, and throw an error message. Assume we don't want the user to enter more than 50,000 in salary. This threshold can point to data in the application, but is hard coded below for simplicity.

```
operation.grid.dataCellIterator({DataCell   cell   ->
cell.edited}).each{
  if(it.data > 50000){
    it.addValidationError(0xFF0000, "Salary is outside of the
allowed range.")
  }
}
```

Conclusion

This is just a taste of what can be done. As you can see, with the ability to isolate actions on only the dirty cells, we now have opportunities we haven't had since pre Smart View, and functions are completely new. The impact on performance is game changing and the ability we now have to interact with a user pre and post save is ground breaking to the possibilities.