Adventures in Groovy – Part 31: Benefits of Running Logic in Groovy (Bypass Essbase)

Say what? An application where no calculations are performed with the Essbase calculation engine?

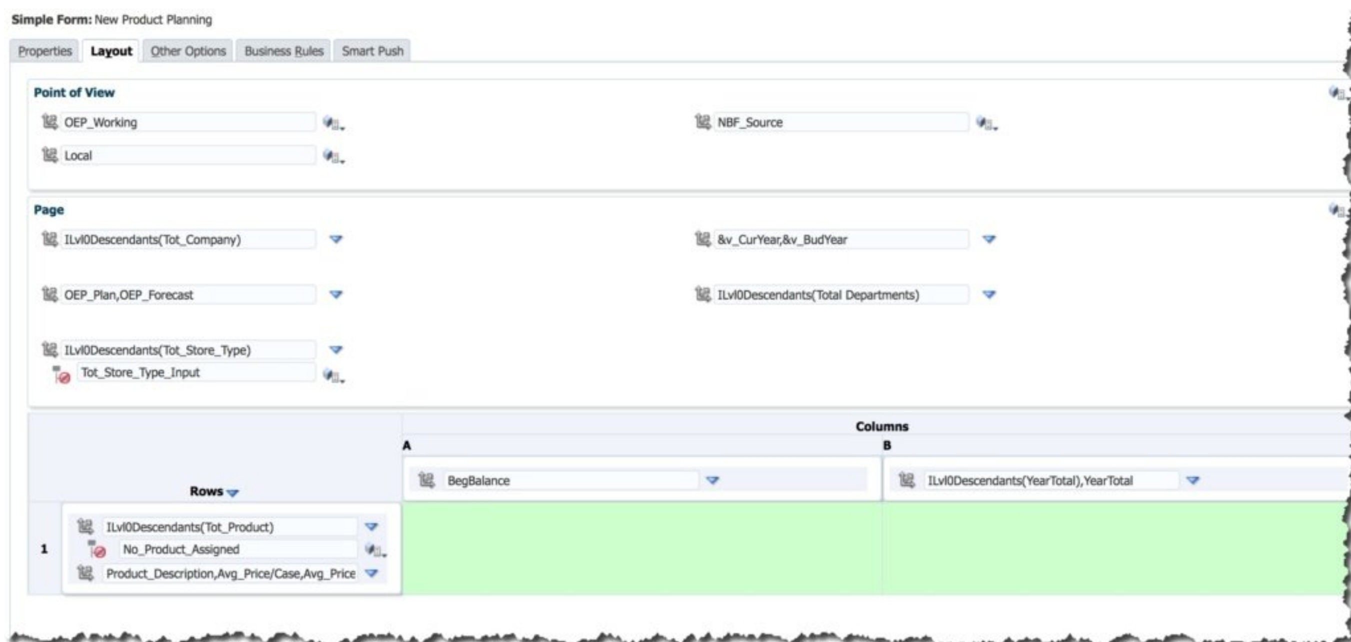
Business logic, currency, and every other calculation can be performed within Groovy outside of Essbase. I don't know how I feel about this yet, but the thought of building this in calculations on an ASO cube is intriguing. The elimination of consolidating BSO cubes and data movements with Smart Pushes is also interesting. Some of my coworkers have built entire applications this way with great success and speed. I wanted to replicate this on a small scale to identify the pros and cons of this methodology, and here is what I found.

In my adventures, I found that performance was similar. Sometimes it was quicker, sometimes a little slower. I did not do large allocations, but I have done account calculations and currency conversion. I have done it on form save as well as globally. Complexity varies. If you are a Groovy expert, it likely seems less complex. If you are an Essbase developer, it might seem more complicated. Code can be reused by developing functions and placing them in scripts, just like Essbase scripts can be embedded into rules.

One thing I did find appealing for a user is that change history is complete! If you have dabbled with this, you will know that the change history only includes edited data by a user. So, calculated accounts don't show any history. When the calculations are done in Groovy, the results are stored in a grid, and saved to the database. This mimics a user updating the values and reflects in the change history. This, I think, is pretty cool.

Another benefit is that calculated data in a grid is reflective of the inputs. With prompts, or when data isn't validated, the form shows the results of the calculations before the data is saved in the form. I have been asked by many if it is possible to show the results real time before save (like an adhoc form with formulas built in Excel). This is now possible if you get creative.

Before we jump into the code, it might make sense to understand what dimensions/members are in the POV.



Code Introduction

The following calculation is replicating what an Essbase calculation is performing.

1. Performs simple math to calculation revenue and costs – mostly simple multiplication and division
2. Looks up the currency rates for the Entity selected and applies the exchange rate to the account if it is identified as an account that should be converted (ignores statistical accounts, for example)
3. Converts the account from ProdRev to the representative account in the P&L plan type.

4. Submits the calculated results to both the ProdRev BSO and ASO (in a real world situation, you may decide to exclude the BSO cube and connect the form and write the data directly to the ASO cube)

As previously mentioned, there are a couple way to do this. This method runs post save and pulls the necessary data from the cube and created two GridBuilders to submit the calculated accounts back to the databases. It would likely be more productive to pull the data from the grid the user entered it, assuming there were no other data points that were needed to calculate all the accounts. This would reduce the performance by one retrieve. This also connects to the BSO cube, and this may not even be in play if all the calculations were executed in Groovy and the forms connected to the ASO cube. So, I welcome all questions and constructive feedback, but understand this is one way to do this and it will be used to demonstrate the functionality. It doesn’t necessarily represent the best way to architect the plan types.

The data entry form is below. Some of the dimensions are hidden in the POV or Page Header. The columns just include the Period dimension. The rows, and this is important to understand because of the way the code loops, contain the vendors and the accounts.

New Product Planning

Company	Years	Scenario	Department	Store_Type
North Illinois	2018	Plan	ACCESSORIES	No Store Type

Actions · Save · Refresh · Close · Data · Ad hoc · Format

		Assumptions	January	February	March	April	May	June	July	August	September	October
Rocketfish??? - 4-Port 4K HDMI Switch Box - Black (5055100)	Description		0	0	0	0	0	0	0	0	0	0
	Unit Price		79.99	79.99	79.99	79.99	79.99	79.99	79.99			
	Avg Unit $ Inp		35.00	35.00	35.00	35.00	35.00	35.00	35.00	0.00	0.00	
	Units		25,000	25,000	25,000	25,000	25,000	25,000	25,000	0	0	
	Net Sales		1,999,750	1,999,750	1,999,750	1,999,750	1,999,750	1,999,750	1,999,750	0	0	
	COGS		586,250	586,250	586,250	586,250	586,250	586,250	586,250	0	0	
	Gross Profit Stage 1		288,750	288,750	288,750	288,750	288,750	288,750	288,750	0	0	
	ProdRev Level 1 %		14.4%	14.4%	14.4%	14.4%	14.4%	14.4%	14.4%			
	ProdRev Level 1 & Inp		33.0%	33.0%	33.0%	33.0%	33.0%	33.0%	33.0%	0.0%	0.0%	
	Customer Satisfaction Discount		586,250	586,250	586,250	586,250	586,250	586,250	586,250	0	0	
	Gross Profit after COGS		0	0	0	0	0	0	0	0	0	
	Gross Profit after COGS %		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%			
	ProdRev Level 2 % Inp											
	Overstock Discount											
	Customer Satisfaction Discount											
	Multi-Purchase Discount											
	Net Gross Profit		0	0	0	0	0	0	0	0	0	
	Net Gross Profit %		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%			
NBF2	Description	aaaaaaa										
	Unit Price											
	Avg Unit $ Inp											
	Units											
	Net Sales											
	COGS											
	Gross Profit Stage 1											
	ProdRev Level 1 %											

Explaining the Code

On with the show! The first piece includes a couple functions that will be referenced in loops. I find repetitive logic like this placed in function to be more readable, but it certainly isn’t required.

```
// This function returns the POV member with or without quotes
// This is one operation I want to look into as it may not be
required or may be executed better
def getPovMbr(member, boolean quotes = false) {
  if(quotes == true)
    return '"' + operation.grid.pov.find{it.dimName==member}.essbaseMbrName + '"'
  else
    return operation.grid.pov.find{it.dimName==member}.essbaseMbrName
}
// This function returns a 1 if the value is 0.  The primary
use of this is currency
// exchange.  In this example, the planners didn't enter 1 for
a USD to USD conversion,
// so somewhere a 1 needed to be assumed.
def checkZero(double dVal=0) {
  if(dVal == 0) {
```

```
    return 1
  }
  else {
    return dVal
  }
}
// This returns a true or false as to whether the account has a UDA identifying that
// account as one that either does or doesn't get converted.
boolean convertCurrency(def account)
{
  Dimension AccountDim = operation.application.getDimension("Account")
  Member AccountMbr = AccountDim.getMember(account)
  def memberProps = AccountMbr.toMap()
  if(memberProps['UDA'].toString().contains('IgnoreCurrencyConversion'))
    return false
  else
    return true
}
```

The next piece is setting each POV member in a variable. If you have read some of my other posts, there are better ways to do this. For the sake of simplicity and trying to explain other concepts, I have chosen to do it this way.

```
// Set each POV to its own variable
String sCompany = operation.grid.getCellWithMembers().getMemberName("Company")
String sVersion = operation.grid.getCellWithMembers().getMemberName("Version")
String sScenario = operation.grid.getCellWithMembers().getMemberName("Scenario")
String sYear = operation.grid.getCellWithMembers().getMemberName("Years")
String sMaterialGroup = operation.grid.getCellWithMembers().getMemberName("Material_Group")
String sChannel = operation.grid.getCellWithMembers().getMemberName("Channel")
```

```
String sSource = operation.grid.getCellWithMembers().getMemberName("Source")
```

At this point, the administrative stuff is out of the way. The first thing this example does functional is to get the currency exchange rates for the month of the Entity selected. This would be a little more complex if it were converting income statement and balance sheet accounts because it would need to distinguish the difference so the correct rate (average or ending) was applied.

```
// Define the map to hold the appropriate rate.  This will be a map that has an entry for each month, with the value equal to
// the conversion rate - Jan:.878, Feb:.899, ...
def currencyRates = [:]
// Build currency for the company by retrieving the correct POV from the plan type
Cube lookupCube = operation.application.getCube("ProfRev")
DataGridDefinitionBuilder builder = lookupCube.dataGridDefinitionBuilder()
builder.addPov(['Years', 'Scenario', 'Version','Channel','Material_Group','Source','Vendor','Currency','Account'],[[sYear],[sScenario],[sVersion],['No_Channel'],['No_Material_Group'],['Input'],['No_Vendor_Assigned'],['Local'],['End_C_Rate']])
builder.addColumn(['Period'], [ ['ILvl0Descendants("YearTotal")']])
builder.addRow(['Company'], [ [sCompany] ])
DataGridDefinition gridDefinition = builder.build()

// Load the data grid from the lookupCube to the Map
lookupCube.loadGrid(gridDefinition, false).withCloseable { dataGrid ->
  dataGrid.dataCellIterator().each{ rate ->
currencyRates.put(rate.getMemberName('Period'),checkZero(rate.data))
  }
}
```

Now that we have a map of the currency rates, the next piece

will create the grids that will submit the calculated results back to the cubes. Remember, in this example, the data is being entered to a plan type that is a BSO cube. This will submit the results back to the BSO cube, but also to the corresponding reporting (ASO) cube. Since this is done post save (and we will assume post Smart Push), the data needs to be update in both places. Is this the best and most efficient way to do this? Probably not. Again, take it for what it is – an education on bypassing the Essbase calculations!

```
// Get list of vendors that are in the rows
def listVendors = operation.grid.rows.headers*.essbaseMbrName.collect {vendor, account -> return vendor}.unique()

// Create a list of calculated members that need to be in the grid
def listAccounts = ["Regular_Cases","Net_Sales","prodRev_Level_1","Cost_of_Sales_without_Samples","prodRev_Level_2","Depletion_Allowance_Manual_Chargeback"]

//prodRev Grid Setup
Cube prodRevCube = operation.application.getCube("prodRev")
Cube rprodRevCube = operation.application.getCube("rprodRev")

DataGridBuilder prodRevGrid = prodRevCube.dataGridBuilder("MM/DD/YYYY")
DataGridBuilder rprodRevGrid = rprodRevCube.dataGridBuilder("MM/DD/YYYY")

prodRevGrid.addPov(sYear,sScenario,sVersion,sChannel,sMaterialGroup,sSource,sCompany)
rprodRevGrid.addPov(sYear,sScenario,sVersion,sChannel,sMaterialGroup,sSource,sCompany,'MTD')
prodRevGrid.addColumn('Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec')
rprodRevGrid.addColumn('Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec')
```

The next piece is replicating what would be in a typical

Essbase calculation. This is the business logic and currency conversion.

```
// Define the currencies to be updated and sent back to the database
def lstCurrency = ["Local","USD"]
// loop through the grid the user entered their updates.  This will loop through each vendor, then loop
// through each account for each vendor
listVendors.each{ vendor ->
  listAccounts.each{ account ->
    // Create two variables to hold the calculated results for local and USD
    def sValues = []
    def sValuesUSD = []
    // Create two variables to hold all the monthly calculated results to be added to the grid to submit
    List addcells = new ArrayList()
    List addcellsUSD = new ArrayList()
     // Run for 12 months - this would likely need to be dynamic for a form that included
    // the option for forecast and plan
['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec'].each{cMonth ->
    // Calculate all the accounts
    double setValue
      switch (account) {
         case "Net_Sales":
            setValue = operation.grid.getCellWithMembers("Avg_Price/Case_Inp",cMonth.toString(),vendor.toString()).data * operation.grid.getCellWithMembers("Regular_Cases",cMonth.toString(),vendor.toString()).data
           break
         case "prodRev_Level_1":
            double netSales = operation.grid.getCellWithMembers("Avg_Price/Case_Inp",cMonth.toString(),vendor.toString()).data * operation.grid.getCellWithMembers("Regular_Cases",cMonth.toString(),vendor.toString()).data
            setValue = netSales *
```

```
operation.grid.getCellWithMembers("prodRev_1_%_Inp",cMonth.toS
tring(),vendor.toString()).data
           break
         case "Cost_of_Sales_without_Samples":
                                       double netSales =
operation.grid.getCellWithMembers("Avg_Price/Case_Inp",cMonth.
toString(),vendor.toString()).data *
operation.grid.getCellWithMembers("Regular_Cases",cMonth.toStr
ing(),vendor.toString()).data
                          double prodRev1 = netSales *
operation.grid.getCellWithMembers("prodRev_1_%_Inp",cMonth.toS
tring(),vendor.toString()).data
           setValue = netSales - prodRev1
           break
         case "prodRev_Level_2":
                                       double netSales =
operation.grid.getCellWithMembers("Avg_Price/Case_Inp",cMonth.
toString(),vendor.toString()).data *
operation.grid.getCellWithMembers("Regular_Cases",cMonth.toStr
ing(),vendor.toString()).data
                                 setValue = netSales *
operation.grid.getCellWithMembers("prodRev_2_%_Inp",cMonth.toS
tring(),vendor.toString()).data
           break
         case "Depletion_Allowance_Manual_Chargeback":
                                       double netSales =
operation.grid.getCellWithMembers("Avg_Price/Case_Inp",cMonth.
toString(),vendor.toString()).data *
operation.grid.getCellWithMembers("Regular_Cases",cMonth.toStr
ing(),vendor.toString()).data
                          double prodRev1 = netSales *
operation.grid.getCellWithMembers("prodRev_1_%_Inp",cMonth.toS
tring(),vendor.toString()).data
                          double prodRev2 = netSales *
operation.grid.getCellWithMembers("prodRev_2_%_Inp",cMonth.toS
tring(),vendor.toString()).data
           setValue = netSales - prodRev1
           break
         default:
                                                 setValue =
operation.grid.getCellWithMembers(account.toString(),cMonth.to
```

```
String(),vendor.toString()).data
           break
      }
       // Update the variables that will be used to create the
grid rows
      sValues.add(setValue)
      addcells << setValue
      // Convert Currency if account should be converted
      if(convertCurrency(account) == true){
sValuesUSD.add(currencyRates[cMonth].toString().toDouble()  *
setValue)
                                          addcellsUSD     <<
currencyRates[cMonth].toString().toDouble() * setValue
      }
      else
      {
        sValuesUSD.add(setValue)
        addcellsUSD << setValue
      }
    }
    // After all 12 months are traversed calculated, add local
and USD to the grid
prodRevGrid.addRow([account.toString(),vendor.toString(),'Loca
l'],addcells)
prodRevGrid.addRow([account.toString(),vendor.toString(),'USD'
],addcellsUSD)
rprodRevGrid.addRow([account.toString(),vendor.toString(),'Loc
al'],addcells)
rprodRevGrid.addRow([account.toString(),vendor.toString(),'USD
'],addcellsUSD)
  }
}
```

The last piece is to submit the grids. The following will write to the log the number of cells accepted and rejected.

```
// After all vendors and account have been calculated, submit
the grid to the respective database
DataGridBuilder.Status status = new DataGridBuilder.Status()
DataGridBuilder.Status rstatus = new DataGridBuilder.Status()
DataGrid grid = prodRevGrid.build(status)
DataGrid rgrid = rprodRevGrid.build(rstatus)
```

```
println("Total number of cells accepted: $status.numAcceptedCells")
println("Total number of cells rejected: $status.numRejectedCells")
println("First 100 rejected cells: $status.cellsRejected")
prodRevCube.saveGrid(grid)
println("Total number of cells accepted: $rstatus.numAcceptedCells")
println("Total number of cells rejected: $rstatus.numRejectedCells")
println("First 100 rejected cells: $rstatus.cellsRejected")
rprodRevCube.saveGrid(rgrid)
```

So…

Why Do this?

This would add complexity for those who don’t know Groovy. It is a completely different thought process. But there are benefits.

- Elimination of the BSO application creates a simpler model.
- There is no requirement of data synchronization between the BSO and ASO cube.
- Users will see cell history for all data that was changed, not just the data changed by a user directly.

That is not to say there aren’t drawbacks, because there are. I am sure if you are experienced, you are already asking questions about clearing data, large allocations, and several other necessary components.

Is It Worth It?

Honestly, I don’t know yet. This is a complete paradigm shift in our world. The thought of calculating everything in Groovy and not having a BSO plan type is just completely foreign to me. The exploration of this option and documenting here surely proves my interest is peaked! Large allocations seem to be

slower than BSO Essbase calculations. Everything else seems to be as quick or quicker. I think in the short term, having both gives us the best of both worlds, even with the additional need to synchronize the data between the two. Long term? Who knows. I never thought I would even be entertaining it, so I am not closing my mind to the possibility.

What do you think?