Adventures in Groovy – Part 22: Looping Through Member Descendants

There are a lot of reasons one might loop through children in a Groovy Calculation. On my journeys with Groovy, I have run into a few roadblocks where this has been helpful. Some of these were related to limits in PBCS. Looping through sets of members allowed us to get around some of the limitations.

- Running Data Maps and Smart Pushes have limits on the amount of data they can push when executed from a business rule (100MB).
- Using the clear option on Data Maps and Smart Pushes has limits on the length of the string it can pass to do the ASO clear (100000 bytes).
- The Data Grid Builders have limits on the size of the data grid that can be created (500,000 cells before suppression).

All 3 of these situations create challenges and it is necessary to break large requests into smaller chunks. An example would be running the methods on one entity at a time, no more than *x* products, or even splitting the accounts into separate actions.

Possibilities

Before going into the guts of how to do this, be aware that member functions in PBCS are available. Any of the following can be used to create a list of members that can be iterated through.

- Ancestors (i)
- Children (i)
- Descendants (i)

- Children (i)
- Siblings (i)
- Parents (i)
- Level 0 Descendants

More complex logic could be developed to isolate members. For example, all level 0 descendants of member *Entity* that have a UDA of *IgnoreCurrencyConversion* could be created. It would require additional logic that was covered in Part 11, but very possible.

Global Process

In this example, *Company* was used to make the data movement small enough that both the clear and the push were under the limits stated above. The following loops through every *Company* (Entity dimension) and executes a Data Map for each currency (local and USD).

```
// Setup the query on the metadata
Cube cube = operation.application.getCube("GP")
Dimension companyDim = operation.application.getDimension("Company", cube)
// Store the list of companies into a collection
def Companies = companyDim.getEvaluatedMembers("ILvl0Descendants(Company)", cube) as String[]

// Create a collection of the currencies
def Currencies = ["Local","USD"]

// Execute a Data Map on each company/currency
for (def i = 0; i < Companies.size(); i++) {
 def sCompanyItem = '"' + Companies[i] + '"'
  for (def iCurrency = 0; iCurrency < Currencies.size(); iCurrency++){
        operation.application.getDataMap("GP Form Push").execute(["Company":Companies[i]
```

On Form Save

When there are large volumes of data that are accessed, it may not be realistic to loop through all the children. In the case of a product dimension where there are 30,000 members, the overhead of looping through 30,000 grid builds will impact the performance. However, including all the members might push the grid size over the maximum cells allowed. In this case, the need to iterate is necessary, but the volume of products is not static from day to day. Explicitly defining the number of products for each loop is not realistic. Below creates a max product count and loops through all the products in predefined chunks until all 30,000 products are executed.

```
def slist = [1,2,3,4,5,6,7,8,9,10]
// Define the volume of members to be included for each process
int iRunCount = 4
// Get the total number of items in the Collection
int iTotal = slist.size()
// Identify the number of loops required to process everything
double dCount = (slist.size() / iRunCount)
int iCount = (int)dCount
// Identify the number of items in the last process (would be be <= iRunCount)
int iRemainder = (slist.size() % iRunCount)

//Run through each grouping
for (i = 0; i <iCount; i++) {
 // loop through each of the members up to the grouping (iRunCount)
 for (ii = 0; ii < iRunCount; ii++) {
  // Do the action
  // slist[i * iRunCount + ii] will give the item in the list to use as needed
  print slist[i * iRunCount + ii] + " "
 }
}

// Run through the last group up to the number in the group
```

```
 for (i = 0; i < iRemainder; i++) {
  // Do the action
  print slist[iCount * iRunCount + i] + " "
 }
```

A Wrap

So, the concept is pretty simple. Now that we have the ability to do things like this outside of typical planning functions really opens up some possibilities. This example ran post-save, but what about pre-save? How about changing the color of cells with certain UDAs? What about taking other properties managed in DRM that can be pushed to Planning that would be useful? How about spotlighting specific products for specific regions that are key success factors in profitability?

Do you have an idea? Take one, leave one! Share it with us.