MWSUG 2016 – Paper BB34-SAS

Data Analysis with User-Written DS2 Packages

Robert Ray and William Eason, SAS Institute Inc., Cary, NC

ABSTRACT

The DATA step and DS2 both offer the user a built-in general purpose hash object that has become the go-to tool for many data analysis problems. However, there are occasions where the best solution would require an object specifically customized for the problem space. The DS2 package syntax enables the user to create objects that can be linked to form structures in memory such as trees or lists that can then be traversed and manipulated. For data that can describe many more states than actually exist, dynamic structures such as this can provide a compact way to represent and analyze the data. The SAS® In-Database Code Accelerator enables these custom packages to be deployed in parallel on massive data grids. Further, with the use of the SAS Macro facility, it is possible to create a package template that facilitates code reuse in a strongly typed system.

INTRODUCTION

SAS DATA step programming has traditionally been limited to a few built-in data elements such as variables, variable arrays, temporary arrays, and, more recently, the hash object. The introduction of the hash object gave the user a powerful dynamic structure that quickly became one of the most used tools in the DATA step toolbox. The hash object allows the user to store selected variables in memory and quickly access a subset using a single or compound key. As flexible as the hash is, it is still a *symmetric* structure where the dimensions of each element are fixed at initialization. While the number of hash entries can grow with the data, the size of each hash element is fixed. Most modern languages have the ability to define an arbitrary combination of fundamental data types into custom structures or objects and include the ability to create links between two or more of these objects to create complex structures in memory. User-created DS2 packages give this same power to the SAS programmer, which is particularly useful when the structure can be used to efficiently represent fundamentally sparse data. If sparse data is represented as an array, the array is primarily filled with zero or missing data points that occupy memory but are not needed to represent the problem space. To avoid inefficient memory utilization, SAS procedures such as PROC FREQ and PROC SUMMARY use compact linked structures such as trees or hashes to efficiently aggregate statistics in memory.

In this paper we will show how DS2 packages can be used to create linked in-memory structures in general, and then we will apply that to a specific data analysis task. The example code can be easily modified to work in your particular problem space. In addition, we will show how to extend specific code to be more generic through the use of template techniques with SAS macros.

DS2 PACKAGE SYNTAX BY EXAMPLE

One of the useful features of DS2 is its package syntax that enables the user to create compound structures and assign actions (methods) to these structures. In essence, the user is creating objects in the language sense although DS2 is not a full object-oriented language. A DS2 package cannot inherit attributes or methods from another package but it can contain references to other packages and therefore can support composition (has-a) relationships. While DS2 does not offer inheritance we will see that in a practical sense, this limitation does not prevent us from achieving code reuse through the principal of genericity.

Let's start with a very simple list package, *myList*,that holds a specific element called *myElement* followed by a small data program that will exercise the two packages.

```
proc ds2;
/*-- Define a simple list element --*/
package myElement/overwrite = yes;
   dcl package myElement next;
   dcl int d;

   /*-- Custom constructor with parameter --*/
   method myElement( int d );
      this.d = d;  /*-- Use “this” to differentiate d’s --*/
   end;

   method print();
      put d=;
   end;
   /*-- A default delete method is implicitly defined --*/
endpackage;

/*-- Define a package to hold the elements --*/
package myList/overwrite=yes;
   dcl package myElement front;
   dcl package myElement back;

   method add( package myElement element );
      if null( front ) then do;  /*-- first element --*/
         front = element;
         back = element;
      end;
      else do;  /*-- link in new element --*/
         back.next = element;
         back = element;
      end;
   end;

   method printFront();
      if ^null( front ) then front.print();
      else put 'Front is NULL';
   end;

   method printBack();
      if ^null( back ) then back.print();
      else put 'Back is NULL';
   end;

   /*-- A custom destructor - called when references go to zero --*/
   method delete();
      dcl package myElement cur next;
      cur = front;
      /*-- Explicitly empty the list --*/
      do while ( ^null( cur ) );
         next = cur.get_next();
         cur.print();
         cur.delete();
         cur = next;
      end;
   end;
endpackage;
run;
```

```
data _null_;
   method run();
      dcl int i;
      dcl package myList ml();  /*-- Instantiated --*/
      dcl package myElement me;   /*-- Reference only --*/
      do i = 1 to 10;
         me = _new_ myElement(i); /*-- Constructor with parms --*/
         ml.add(me);
      end;
      ml.printFront();
      ml.printBack();
   end;
enddata;
run;
quit;
```

Notice how the *myElement* package declares a reference to itself, *next,* as the first element of its internal structure. This is the foundation of building linked structures in memory: a package's ability to reference itself. Next it declares the content that the list will hold, in this case a single integer called *d*. Two methods are defined.The first *myElement(int d)* has the same name as the package, which means it is the *constructor* for the package. It is not necessary to define a constructor, but, if you do, it can take initializing parameters such as the value of *d* in this case. The second method, *print(),* simply calls *put* on *d* to display the value of the element in the SAS log. A default destructor is supplied by the compiler if a user-written *delete()* method is not included.

Next we have the *myList* package that includes two elements of type *myElement*, *front* and *back*. With those and the *add()* method, you have a simple list structure. To illustrate the list there is a *printFront()* and *printBack()* methods which in turn calls the *print()* method of the *myElement* package. Finally, there is a custom *delete()* method, which is recommended whenever one package holds references to other packages, which could lead to a circle of references. When the execution leaves the run() method, delete() is implicitly called on the instance of *myList* held by *ml*, which in-turn explicitly calls delete on all the instances of *myElement* in the list. We will discuss instance management further in the section.

So there we have a simple but very specific list. DS2 is a strongly typed language that is good for producing code that has predictable run-time behavior but, being strongly typed also works against code genericity and reuse. Later in the paper we will discuss techniques for improving the reusability of fundamental algorithms such as lists by creating the effect of package templates.

PACKAGE INSTANCE MANAGEMENT

It is important to understand the difference between a package instance and a package reference (variable). An instance is the in-memory realization of a package, while a variable references (contains the memory address of) an instance. Anytime you declare an instance of a DS2 package with an instantiating declaration – one that ends with parenthesis and possibly enclosing arguments – you are creating an *instance* of the package in the declaration. The other mechanism to create an instance is using the _NEW_ operator with the package name along with a parameter list. When an instance is created, memory for the instance attributes along with hidden support structures is allocated and associated with the instance reference. If you assign another instance to the reference, the original instance will be abandoned. For example, consider the code below:

```
dcl package myElement me(0);   /*-- Abandoned instantiation --*/
dcl package myList ml();    /*-- Instantiated --*/
   do i = 1 to 10;
      me = _new_ myElement(i); /*-- Constructor with parms --*/
      ml.add(me);
   end;
```

You will notice that the declaration of *me* includes “(0)”, which means that an instance was assigned at that point. Later, that instance is replaced with the instance created in the loop with the _NEW_ operator. In this case, one instance of *myElement* was abandoned when *I= 1*. The DS2 run time will recover these instances once it determines that there are no active references to the instance. This reference counting memory recovery scheme was added in the SAS 9.4M3 release. When the reference count of an instance goes to zero, its *delete()* method (or the default destructor) will be called on that instance. If the instance contains instance references such as is the case with the *next* field of our *myElement* package, the *delete()* method will be called on those elements. As you can imagine, if you had a very long list of elements, the function call stack limit could be exceeded. Therefore it is important that a custom delete method be created for packages that could hold this type of instance list. You might be thinking that it would be elegant to simply create a delete method in the *myElement* that would call *delete()* on *next*. For long lists, that could also result in call stack overflow.

You might ask “What if another package also held a reference to a list element?” Instance reference is actually by indirection. For each instance, there is a small instance “handle.” When *delete()* is called explicitly on a package instance, the actual instance memory is freed and the reference to the memory in its associated handle is set to NULL. Any further operations on that instance from other references will result in a “NULL package reference error” message being sent to the log and a program halt. This scenario is demonstrated by the code below:

```
proc ds2;
data _null_;
method run();
   dcl package myList m1 m2();
   m1 = m2; /*-- m1 and m2 now reference the same instance --*/
   m2.printFront();
   m1.printBack();
   m2.delete(); /*-- instance body is gone, handle set to NULL --*/
   m1.printFront(); /*-- m1 is now null, so this will cause an error --*/
end;
enddata;
run;
quit;
```

Package Instance Size

Let’s talk about DS2 package instance size in memory. Currently (SAS 9.4M3 and earlier), the instance model for DS2 is fairly dense. For example, instances of *myElement* in the code above will actually occupy around ***2k*** bytes of memory. The fixed overhead does not vary based on the number of private attributes declared and includes things such as private memory pools for each instance. This is an important concept to understand before embarking on building complex structure of package instances. Memory savings over traditional data structures such as arrays will be evident only for very sparse data. There is currently a project underway to minimize the instance overhead so that complex structures composed of small packages will require significantly less memory. The same instance of *myElement* in the new model will consume around ***0.2k*** bytes. This ten-fold reduction in instance overhead will expand the scenarios where lists of custom packages create the smallest memory footprint.

DATA ANALYSIS PROBLEM

The hypothetical data analysis problem we will explore is as follows. The user has transactions records for customer visits to their website. Each record has the customer ID, timestamp, and an activity code indicating which activity on the website the customer selected. There are hundreds of thousands of customers and tens of thousands of possible activities, but any one customer will historically engage in less than 100 different activities. The objective of the analysis is to generate frequency counts of activities for each customer so that a *top-N* list of activities can be generated. We also need to understand the activity pattern for each customer – what are the *first-N* activities in order for each customer– which gives insight into navigation patterns and how they relate to overall *top-N* list. In doing so, we hope to collect data that will help correlate initial navigation patterns with eventual browsing and buying patterns.

Custom Ordered-Set Package

What we need is a succinct way to capture the activity code and accumulated frequencies while at the same time maintain the order of occurrence. For this, we choose to create a custom DS2 package that implements aspects of a typical ordered-set collection class where there is a set of unique elements (keyed by activity code) that can be traversed in one or more criteria. Each element of the set also contains a cumulative frequency and the earliest start-time for a particular activity code. We will need to be able to reorder the set by frequency and start-time at some point, so this ordered-set will need a "reordering" method as well. We will also need a simple list class that can manage free customer activity instances between BY-groups.

Below are the initial lines of a customer activity package that will be embedded in a generic collection "node." As you can see from the *node* package, a tree algorithm has been used to implement the set. In this case, it will be an AVL height-balanced binary tree. The OSet structure will actually maintain two ordered states with each insert: order by key ascending and order of arrival. The AVL tree provides an efficient key lookup during the date accumulation phase to maintain uniqueness and is used to reorder the data for the final output phase.

```
package CustomerActivity;
   dcl int activityCode;
   dcl int frequency;
   dcl double startTime;
<…>
package CustomerActivtyNode;
   dcl package CustomerActivity key;
    /*-- For the list --*/
   dcl package Node next;
   /*-- For the AVL --*/
   dcl package Node left right;
<…>
package CustomerActivityList;
   dcl package CustomerActivityNode front back;
   <…>
package CustomerActivityOSet;
   dcl package CustomerActivtyNode head;
   dcl package CustomerActivtyNode front back;
   dcl int comparisonType;
<…>
```

Using this set of packages we will build our custom solution to navigate a large data space using package instances to represent discrete data values. The strategy will be to use BY-group processing on the data based on customer ID and timestamp. We will reuse a single instance of our Ordered-set (*OSet*) package for each BY group and "reset" it between customers to minimize the frequency of instance creation. For each BY group, the frequencies for each unique activity will be accumulated by searching the list and updating the value. Naturally as the list grows, linear searching will be prohibitively expensive, which is why our *OSet* class leverages a balanced binary tree to achieve lookup in O(l*og N*) time. Below is an illustration of the links built inside the *OSet* package to maintain both an order-of-arrival list and a sorted binary tree.

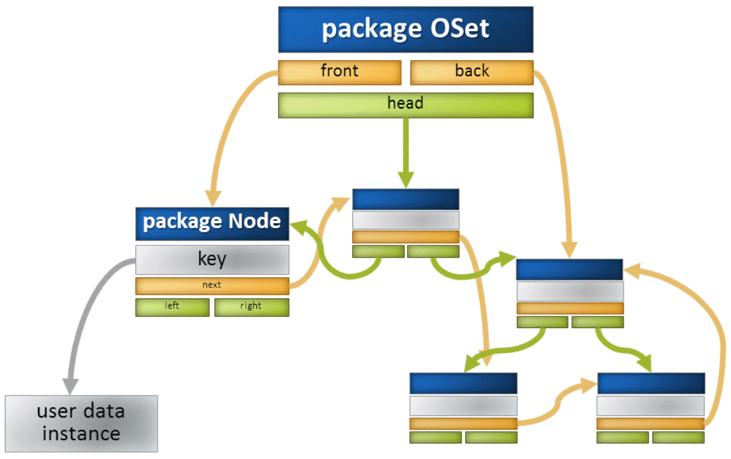

Figure 1. Ordered-Set Internal Structure. Orange Links Are List Links. Green Links Are Tree Links.

One of the required methods for a node-key is *compare()*, which is called at every step of tree traversal to establish the desired order. Below is a code segment for the *CustomerActivity* package showing part of the *compare()* method. A return value of -1 means that *this* should precede the parameter key *ca* whereas 1 means that *ca* should precede *this*. A return value of 0 means that *this* and *ca* are the same key value. Only the “0” comparison type can and must return ‘0’, meaning two keys are equal– which is required for the initial set build where each activity node must be unique. Reordering the set by frequency or start time must include tie-breaking and *never* return 0 because 0 indicates duplicate, which would invalidate the set after it has been established.

```
package CustomerActivity;
   dcl int activityCode;
   dcl int frequency;
   dcl double startTime;
   /*-- user written set/get required for use inside package now --*/
   method getFrequency() returns int;
      return self.frequency;
   end;
   method getActivityCode() returns int;
      return self.activtyCode;
   end;
   method getStartTime() returns double;
      return self.startTime;
   end;
   method compare(package CustomerActivity ca, int type) returns int;
      dcl int caInt;
      dcl double caDbl;
      if type = 0 then do;
         caInt = ca.getActivityCode();
         if this.activityCode > caInt then return -1;
         if this.activityCode < caInt then return 1;
         return 0;
      end;
```

```
        if type = 2 then do;
           caInt = ca.getFrequency();
           if this.frequency > caInt then return -1;
           if this.frequency < caInt then return 1;
           caInt = ca.getActivityCode(); /*-- tie break on activity --*/
           if this.activtyCode < caInt then return -1;
           return 1;  /*-- never return zero for this comp type --*/
        end;
        <…>
```

Code Acceleration and Distributed BY-Group Processing

This solution will be run in a THREAD program so it can leverage one of the SAS® In-Database Code Accelerator products, which means that BY groups will be distributed across many separated nodes during execution. An important concept to understand with distributed BY-group processing is that the distribution process might hash the combined values of all BY variables taken together. This means that when multiple BY variables are in play such as *state* and *city*, that “Nevada / Las Vegas” might not occur on the same grid node as “Nevada / Reno”. Logic that depends on traditional, single-threaded execution where *first.city* is nested inside *first.state* cannot be counted on to “see” all the cities of Nevada in alphabetical order. However, using the single BY variable *State* will ensure that all the cities in Nevada will be clustered together in a single node/BY-group but in no specific order. Likewise, we cannot use both *customerID* and *startTime* as BY variables if we intend that all the rows belonging to a specific customer are clustered together on the same node/process. We must use the single BY variable, *customerID,* and therefore cannot assume any ordering of records with regard to *startTime*. For this reason, we will need to reorder our list of activities twice on output, once by *frequency* and again by *startTime*.

The Main Processing Thread Program

Now we are going to look at some segments of the primary data processing thread program. Below are the declarations for the primary data processing thread program. Placing the logic in a thread program enables it to process the data in parallel and also makes the code eligible for code acceleration. By writing logic that is compatible with a program that sees only part of the input data, you are assured that the program will translate to large scale distributed database systems.

```
thread ActivityAnalysis / overwrite = yes;
   /*-- Primary storage structure - Ordered-set --*/
   dcl package CustomerActivityOSet activities;
   /*-- Simple lists to manage data nodes --*/
   dcl package CustomerActivityList freeList;
   /*-- An iterator to traverse the OSet --*/
   dcl package CustomerActivityIterator acti;
   /*-- The user defined data element of the OSet --*/
   dcl package CustomerActivity act find_result;
   dcl int activityCode frequency;
   dcl double startTime;

   /*-- output variables --*/
   vararray int time_activity_[&n];
   vararray double time_[&n];
   vararray int freq_activity_[&n];
   vararray int freq_[&n];
   dcl bigint customerID;
```

```
method init();
   /*-- initialize OSet comparison type 0 --*/
   activities = _new_ CustomerActivityOSet();

   /*-- create empty free list to hold “recycled” activity nodes --*/
   freeList = _new_ CustomerActivityList();
end;
```

With two of our packages initialized, we are now ready to enter the implicit SET loop of the RUN method. Remember that the data has three fields, *CustomerID*, *activityCode*, and *startTime*. The logical steps will be as follows:

1. Allocate or reuse an activity node.
2. Initialize the node with data from the PDV.
3. Insert the node in the OSet
   a. If it is new, it will be added and the return value will be null.
   b. If it is not new, accumulate the data into the existing node and recycle this one.

Notice that the *CustomerActivityOSet* is initialized with the default comparison type code of ‘0’, which is the only one valid for inserts. The other comparison codes will be used to reorder by the cumulative frequency or the activity start time.

```
method run();
   /*-- read data in one customer at a time --*/
   set ActivtyLog;
   by customerID;

   /*-- Pull activity node off free list --*/
   act = freeList.removeElement();

   /*-- If list was empty. Create new activity --*/
   if null(act) then
      act = _new_ CustomerActivity(activityCode, 1, startTime);
   else
      /*-- load new variables into existing activity node --*/
      act.resetActivity(activityCode, 1, startTime);

   /*-- Insert the customer activity. Returns null if key is new --*/
   find_result = activities.insert(act);

   /*-- If we found an existing entry --*/
   If ^null(find_result) then do;
      /*-- Recycle the unneeded CustomerActivity --*/
      freeList.addElement(act);
      /*-- Accumulate frequency --*/
      find_result.frequency = find_result.frequency + 1;
      /*-- Keep earliest startTime for a given activity --*/
      if startTime < find_result.startTime then
         find_result.startTime = startTime;
      end;
   /*-- end of data load loop --*/
```

At the end of each *customerID* BY group, we will generate the output of interest which is two lists of activity codes: one based on top N frequencies (with a tie-break of activity code) and a second based on earliest activates – the first N activities. To do this, we use the *reOrder* method of our *OSet* package. During data accumulation, the *OSet* package is building its unique collection based on the *activityCode*, and this is determined by the *comparisonType* attribute of the *OSet*. By changing the comparison type field during *reOrder()*, we can establish new ordering criteria. A custom iterator is used to traverse the *OSet* after each reordering.

```
      /*-- data output --*/
      if last.customerNumber then do;
         /*-- re-order activities by frequency --*/
         activities.reOrder(2);

         /*-- Iterate OSet for top-N frequencies --*/
         acti = _new_ CustomerActivityIterator(activities);
         act = acti.visitNext();
         i = 1;

         do while (^null(act) && i <= &n);
            freq_[i] = act.frequency;
            freq_activity_[i] = act.activityCode;
            act = acti.visitNext();
            i = i + 1;
         end;
         acti.delete();

         /*-- repeat the process in order of startTime --*/
         activities.reOrder(5);

         acti = _new_ CustomerActivityIterator(activities);
         act = acti.visitNext();
         i = 1;
         do while (^null(act) && i <= &n);
            time_[i] = act.startTime;
            time_activity_[i] = act.activityCode;
            act = acti.visitNext();
            i = i + 1;
         end;
         acti.delete();

         /*-- Output PDV and reset the OSet and recycle its nodes --*/
         output;
         freeList.appendFromOSet(activities);
      end;
   end;
 endthread;
```

There is nothing in the data program other than the SET statement invoking the thread program. The data are processed in parallel with one or more customer BY groups being computed on each parallel thread.

PACKAGE GENERICITY

At the heart of this solution is the *CustomerActivityOSet* package that employs the AVL balance binary tree algorithm discovered in 1962 by the Russian mathematicians, G. M. Adelson-Velisky and E. M. Landis, as described by Donald Knuth in *The Art of Computer Programming, Volume 3: Sorting and Searching*. The algorithm is compact and elegant, but it is not necessarily simple to code and is definitely not something that you would want to code over and over. In this paper we have shown package nesting with *CustomerActivity* being an element of the *CustomerActiveyNode*, which is the basic element of both the *CustomerActivityList* and the *CustomerActivityOSet*. Because DS2 is a strongly typed language, making another list/ordered-set package to hold another type of data would require editing both of the container packages. Although DS2 does not currently have formal template syntax, a simulation of a C++ *template* or Java *generic* can be implemented for DS2 by using a SAS macro for code that executed in the DS2 procedure. In the context of PROC DS2, the SAS macro will expand the source before it is sent to the DS2 compiler. At execution time, the DS2 language will be running outside the domain of the SAS macro processor so that runtime interaction with the macro system such as SYMPUT and SYMGET are *not* supported. The SAS macro-based solution will work for any DS2 code that is launched from PROC DS2 on a SAS client such as the SAS® In-Database Code Accelerator.

At the top of thread program, the following macros are called to generate the packages used.

```
%generateOSetPackages(CustomerActivity);
```

As you might imagine, the macro substitutes the name of the data element package to build type-matching node, list, and ordered-set packages. Using this technique allows investment in complex code to be reused in a type-safe way and thus achieve a greater degree of language genericity.

```
%MACRO generateOSetPackages(dataPackageName);
/*-- define the node package used by the sets --*/
package &dataPackageName.Node;
   dcl package &dataPackageName key;
   /*-- tree pointers --*/
   dcl package &dataPackageName.Node left right;
   /*-- linked list --*/
   dcl package &dataPackageName.Node next;
   <…>
package &dataPackageName.List;
   dcl package &dataPackageName.Node front back;
   <…>
package &dataPackageName.OSet / overwrite=yes;
   /*-- tree head --*/
   dcl package &dataPackageName.Node head;
   /*-- linked list front/back --*/
   dcl package &dataPackageName.Node front back;
   /*-- comparison type - default 0 --*/
   dcl int comparisonType;
   <…>
package &dataPackageName.NodeStack/overwrite=yes;
   dcl package &dataPackageName.Node stackTop;
   <…>
package &dataPackageName.Iterator/overwrite=yes;
   dcl package &dataPackageName.NodeStack ns();
   dcl package &dataPackageName.Node current;
   <…>
```

The code for the Ordered-Set package is not included in this paper but can be found in the external SAS Knowledge Base at this link: http://support.sas.com/kb/57/803.html

CONCLUSION

The ability of a DS2 package to hold a reference to another instance of itself or other package types enables the user to create custom complex structures of instances in order to represent irregular or sparse data patterns. These structures are built one instance at a time by linking instances to other instances so that meaningful data patterns can be expressed succinctly and that traditional structured programming algorithms can be applied to sets of DS2 instances. The example showed a package that was similar in many ways to the built-in hash package but the ability of the OSet package to leverage comparison methods allows it to behave in very customizable ways. Along with this flexibility comes responsibility to manage instance lifetime when the structures include potential rings of instance links.

Moreover, the current minimum instance size of DS2 packages implies that the total memory required to support an instance-based data structure could substantially exceed the same data held in a more traditional structure such as an array or a hash package. A current project to reduce the minimum instance size will make structures in DS2 much more efficient. Because DS2 is a strongly typed language, reusing complex algorithms can be difficult without a formal *template* or *generic* syntax. The SAS macro preprocessor can be used to emulate a formal template facility and thereby extend code genericity and thereby code reusability. This paper illustrates these concepts with an example of a generic ordered-set package, which can be found in the external SAS Knowledge Base at this link: http://support.sas.com/kb/57/803.html.

REFERENCES

Knuth, Donald E. 1998.*The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd ed. Reading, MA: Addison-Wesley.

RECOMMENDED READING

Ghazaleh, David. 2016. “Exploring SAS® Embedded Process Technologies on Hadoop®.” *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available http://support.sas.com/resources/papers/proceedings16/SAS5060-2016.pdf.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Robert Ray
William Eason
500 SAS Campus Drive
Cary, NC 27513
SAS Institute, Inc.
robert.ray@sas.com
will.eason@sas.com