Paper 1976-2018

Advanced Programming Techniques Using the DS2 Procedure

Viraj Kumbhakarna, MUFG Union Bank N.A.

ABSTRACT

DS2 is a SAS® proprietary programming language appropriate for advanced data manipulation. In this paper, we explore advantages of using PROC DS2 procedure over the data step programming in SAS®.

We explore various programming techniques within PROC DS2 to resolve real world SAS programming problems such as use of hash objects for performance improvement, support for calls to FCMP user defined functions and subroutines within DS2, and use of powerful and flexible matrix programming capability within DS2. We explore executing queries in-databases using FED SQL and explore use of embedded FedSQL to generate queries in run-time to exchange data interactively between DS2 and supported database. This allows processing data from multiple tables in different databases within the same query thereby drastically reducing processing times and improving performance. We explore use of DS2 for creating, bulk loading, manipulating and querying tables in an efficient manner.

We will compare and contrast traditional SAS vs. DS2 programming techniques and methodologies. How certain programming tasks can be accomplished using PROC DS2 at the cost of slightly higher added complexity to code but with huge performance benefits? When does it make most sense to use of DS2 and perform performance benchmarking between using traditional programming techniques vs. PROC DS2 to perform statistically intensive calculations in SAS.

INTRODUCTION

In getting started with DS2 programming, one must know methods and packages. In general, methods are like functions, procedures and subroutines and the methods of object oriented languages like C++ or Java. A method can be thought of as a module that contains a sequence of instructions to perform a specific task. DS2 methods can exist only within a data program, thread program or package.

Thus methods enable users to break up a complex problem into smaller modules. Smaller modules are easier to design implement and test, thereby allowing users greater flexibility in code development. Use of smaller modules allows code reusability and can greatly shorten development time and help standardize repetitive or business specific programming tasks. Also, modular programming allows enhanced code readability and results in easier understanding of code by testers and other programmers.

The DS2 packages are the core constructs of DS2 programming. DS2 packages are language constructs that bundle the variables and methods into named objects that can be stored and reused by other DS2 programs. The primary benefit of a package is rendered from the reusability of the set of methods it offers

DS2 package can be compared to a ‘class’ in object-oriented languages like C, C++ etc. It acts as an extensible program-code template for creating objects, providing initial values for state (member variables) and implementation of behavior (member functions or methods). Thus, one can develop packages that approximate the encapsulation and abstraction of the object-oriented classes.

In computer programming, encapsulation is used to refer to the one of the two related but distinct notions and sometimes to the combination thereof:

- A language mechanism for restricting direct access to some of the objects components
- A language construct that facilitates he bundling of data with the methods (or other functions) operating on the that data

Encapsulation is one of the fundamentals of OOP (object-oriented programming). It refers to the building of data with the methods that operate on the data. Encapsulation is used to hide the values or state of a structured data objects inside a class, preventing unauthorized parties direct access to them.

Abstraction is a technique for arranging complexity of computer systems. It works by establishing a level of simplicity on which a person interacts with the system, suppressing the more complex details below the current level. The programmer works with an idealized interface (usually well defined) and can add additional levels of functionality that would otherwise be too complex to handle. For example, a programmer developing a code that involves building a logistic regression model may not be interested in the way numbers are represented in the underlying hardware (e.g. whether numbers are 32 bit or 64 bit integers), and where these details have been suppressed it can be said that they were abstracted away, leaving simply numbers with which programmers can work with.

A DS2 packages bundles data and methods into a named object that can be stored and reused by other DS2 programs. Although, DS2 packages do not hide their data or methods (there is no concept of public or private), packages can be designed to abstract behavioral details. In such a package, the methods define the object and enable controlled manipulation of the data.

There are two types of packages in DS2:

1. Predefined packages:

These packages encapsulate common functionality that is useful to many customer solutions (for example, Hash and Hash iterator I used for manipulating hash objects and matrix package is used for matrix operations on data structures). Predefined packages are part of DS2

2. User-defined packages:

These are the packages that one can create by defining one's own methods for reuse.

A package is defined by a package programming block. A package begins with the PACKAGE keyword and ends with the ENDPACKAGE keyword.

A user stores methods created by them in user-defined packages. Packages can be thought of as libraries of one's methods. Any type of method can be saved in a package. Once user stores methods in a package (using the PACKAGE statement), one can access them by creating an instance of the package with only a DECLARE statement or with the _NEW_ operator.

```
declare package package-name instance-name;

instance-name = _new_ package-name();
```

Alternatively, one can use the condensed constructor syntax:

```
declare package package-name instance-name();
```

Please find an example of a simple user-defined package called Simple_Interest. It contains a method that calculates the simple interest on an investment.

A= P (1 + rt)

Where P is the Principal amount of money to be invested at an Interest Rate R% per period for t Number of Time Periods

```
proc ds2 ;
      package banking /overwrite=yes;
            method simpleInterest(  double P,
                                    double R,
                                    double t) returns double;
                  return P*(1 + (r *t)) ;
            end;
      endpackage;
run;
quit;
```

In the example below, the simple interest is calculated using the method simpleInterest from the package banking that was created in the earlier example. First the package is declared and instantiated, then the simpleInterest method is called and the result is assigned to 'A'.

```
proc ds2;
      data _null_;
            dcl double A ;
            method init();
                  dcl package banking bkg();
                  A = bkg.simpleInterest(1000,1.3,2);
                  put 'Simple Interest= ' A;
            end;
      enddata;
run;
quit;
```

The SAS log below displays the calculated value of SimpleInterest corresponding to the variable 'A'.

```
207  proc ds2;
208     data _null_;
209        dcl double A ;
210        method init();
211           dcl package banking bkg();
212           A = bkg.simpleInterest(1000,1.3,2);
213           put 'Simple Interest= ' A;
214        end;
215     enddata;
216  run;
Simple Interest=  3600
NOTE: Execution succeeded. No rows affected.
217  quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time          0.07 seconds
      cpu time           0.06 seconds
```

THE HASH PACKAGE

The SAS hash object as its name implies, is a based on certain variety of hashing. Hashing is a group of table search algorithms based on the concept of direct addressing. Hash objects have been in used in SAS for quite some time now, use of hash objects has shown to be much more efficient than the existing data lookup-methods mostly because the data is loaded in the memory and results in faster retrieval.

A SAS hash object is a hash table supplied with tools to control and manipulate it in the DATA step during its execution (run) time. In particular:

- The hash table can be created, loaded, emptied, and deleted.
- The table is temporary: Once the DATA step has stopped execution, it ceases to exist. Thus, it cannot be reused in any subsequent step. However, its content can be saved in a SAS data file or external data base.
- The table resides completely in memory (RAM, main storage, etc., depending on the OS semantics). This is one of the factors making its operations fast, but it's also what limits the amount of data it can contain.
- Just like any table-like object, it contains rows (hash entries) and columns (hash variables).
- Each entry must have at least one key column and one data column. The key variables make up the key portion of the table, and the data variables - the data portion.
- Because the table's look-up mechanism is based on hashing, it supports such standard key-search-based table operations as search, insert, update, retrieve, and delete in constant, or O (1) time. The latter means that their execution time does not depend on the number of entries in the table.

- It supports the enumeration operation in O (N) time. It means that the time required to list all the entries is directly proportional to the number of entries, N.
- It supports, all on its own, input/output (I/O) operations. That is, the hash object can read from and write to a SAS data file (or any other data base structure properly linked via a SAS libref). These operations are independent from both the file names listed in the DATA statement and from the DATA step's own I/O statements (such as SET, MERGE, OUTPUT, etc.).

You can picture the hash object as a black-box device you can manipulate from inside the DATA step to ask it for high-performance data storage, update, retrieval, and I/O services using the object's methods and operators. You can also ask for other information (e.g. the number of entries) using its attributes (Dorfman, 2016, Pg. 1).

HASH PACKAGE IN DS2

DS2 provides a predefined hash package for the programmer's perusal. The hash and hash iterator in DS2 enables users to quickly and efficiently store, search and retrieve data based on unique lookup keys. The hash package keys and data are variables. Key and data value can be directly assigned constant values, values from a table, or values can be computed in expression. The hash package stores and retrieves data based on a unique lookup keys. Depending on number of unique lookup keys and size of the table, the hash package can be significantly faster than a standard format lookup or an array.

In order to use a DS2 hash package, one needs to define and construct and instance instantiate) of the hash package. After the hash package instance is created, one can perform few of the many tasks listed below:

- Store and retrieve data
- Replace and remove data
- Generate a table that contains the data in the hash package.

Defining and Creating a Hash Package Instance

To create an instance of a hash package, a user should provide keys, data, and optional initialization parameters about the hash instance to be constructed. A hash package instance can be defined in two ways: 1) Defining Hash Instance By Using Constructors and 2) Defining a Hash Instance Through Method Calls.

Defining a Hash Instance By Using Constructors

A constructor method is used to instantiate a hash package and initialize the hash package data. There are three different methods for creating a hash package instance with constructors.

1. Create a partially defined hash instance

   DECLARE PACKAGE HASH instance (hashexp, {'datasource'|'\{sql-text\}'}, 'ordered', 'duplicate', 'suminc', 'multidata');

   Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).
   The escape character (\ ) before the bracket indicates that the bracket is required in the syntax. The key and data variables are defined by method calls. Optional parameters can be specified in the DECLARE PACKAGE statement as shown above, in the _NEW_ operator, by method calls, or a combination of any of these. A single DEFINEDOEN method call completes the definition.
2. Create a completely defined hash instance with specified key and data variables

   DECLARE PACKAGE HASH instance (\[keys\],\[data\] [,hashexp, {'datasource'|'\{sql-text\}'}, 'ordered', 'duplicate', 'suminc', 'multidata']);

   The key and data variables are defined in the DECLARE PACKAGE statement, which indicates that the instance should be created as completely defined. No additional initialization data can be specified with subsequent method calls.

3. Create a completely defined hash instance with specified key variables (a keys-only hash instance). There are no data variables.

   DECLARE PACKAGE HASH instance(\[\][,hashexp,{'datasource'|'\{sql-text\}'}, 'ordered', 'duplicate', 'suminc', 'multidata']);

   They key and data variables are defined in the DECLARE PACKAGE statement which indicates that the instance should be created as completely defined. No addiitnaonl initialization data can be specified using subsequent method calls.

Defining a Hash Instance By Using Method Calls

If hash instance is partially defined during the construction of the instance then the instance can be further defined through the calls to the following methods

KEYS – Defines Key variable for a hash package using a variable list

DEFINEKEY- Defines key variables for a hash package using implicit variables

DATA – Specifies the data variables to the stored in the hash package using a variable list

DEFINEDATA – Defines data variables for the hash package using implicit variables

DUPLICATE – Determines whether to ignore duplicates when loading into hash package.

HASHEXP – Defines the hash package's internal table size. The size of hash table is 2^n

ORDERED – Specifies whether or how the data is returned ordered by key-value with a hash iterator package or OUTPUT method

MUTIDATA - Specifies whether multiple data items are allowed for each key

SUMINC - Specifies a variable that maintains a summary count of hash package keys.

DEFINEDONE – Indicates that all key and data definitions are complete

Please find below an example of hash instance, h, defined using the method calls.

```
data _null_;
      declare package hash h(0, 'testdata');
      method init();
            h.keys([key]);
            h.data([data1 data2 data3]);
            h.ordered('descending');
            h.duplicate('error');
            h.defineDone();
      end;
enddata;
```

Defining Key and Data Variables in Hash

The hash package uses unique lookup keys to store and retrieve data. The keys and data variables are used by user to initialize hash package by using dot notation method calls.

Keys and data variables can be defined in following three ways:

- Use the variable methods, DEFINEDATA and DEFINEKEY

```
/* Keys and data defined using the implicit variable method */
declare package hash h();
h.definekey('account_id');
h.definedata('total_sales');
h.definedone();
```

- Use the variable list methods, DATA and KEYS

```
/* Keys and data defined using the variable list methods */
```

```
declare package hash h();
h.keys([account_id]);
h.data([total_sales]);
h.definedone();
```

- Use key and data variables lists specified in the DECLARE PACKAGE statement

```
/* Keys and data defined using the variable list constructors */
declare package hash h([k],[d]);
```

If an instance of the hash package is not completely defined at construction that is keys and data variables are not specified at construction, you must call the DEFINEDONE method to complete initialization of the hash instance. Key variables must be a DS2 built-in type (character, numeric, or date-time). Data variables can be either a DS2 built-in type or a built-in or user-defined package type.

Hash Initialization Data Provisioning

As discussed, hash object needs three inputs – keys, data and initialization parameters. Following optional initialization parameters can be provided to hash package:

- Internal table size (hashexp) where size of hash table is 2n
- Name of table to load (datasource) or FedSQL query to select data to load from.
- Whether data is returned in ordered by key-variables or not (ordered option)
- Whether duplicate key values are ignored or not when loading a table (duplicate option)
- Name of variable that maintains summary count of hash package keys (suminc)
- Whether multiple data items are allowed per key value or not (multidata)

Performing a table lookup using Hash in DS2

Let us consider a practical application of using Hash package in DS2 to perform data lookup on a smaller table to read and merge data with a larger table one key values. The required variables from the smaller table will be loaded in the resultant table only for matching key variables, thereby performing a match-merge operation similar to what one can get using traditional SAS methods such as by using Set and Index variable or by using a merge statement and use of By variables for matching based on keys.

In order to enumerate the match merge operation using Hash, we will create two tables – Employee and Salary. Employee table contains an ID variable corresponding to an Employee ID and Employee Initials.

```
proc ds2;
data emp /overwrite=YES;
  dcl char(8) ID ;
  dcl char(3) INITIALS;
  method init();
      dcl integer i;
      dcl integer j;
      do i = 1 to 20;
        ID = put(i, BEST8.);

        do j=1 to 3;
          substr(INITIALS,j)=byte(int(65+26*ranuni(0)));
        end;
        output;
      end;
  end;
  enddata;
run;
```

```
quit;
```

The following dataset is generated as a result of executing above code:

Dataset: Emp

Obs	ID	INITIALS
1	1	FBN
2	2	SQO
3	3	YPS
4	4	RHO
5	5	HCA
6	6	ZJA
7	7	VAQ
8	8	YZE
9	9	XFC
10	10	HIX

Figure 1. Employee dataset generated using above code

The Salary table is created such that it contains two variables the Employee ID and employee Salary information. The do loop is used to create ID variable and the ranuni function is used to generate random numbers for generating sample salary information.

```
proc ds2;
data salary(overwrite=yes);
      dcl char(8) ID;
      dcl double SALARY ;
      method init();
            dcl integer i;
            do i = -20 to 20;
                  ID = put(i, BEST8.);
                  SALARY = int(ranuni(0)*10000);
                  output;
            end;
      end;
      enddata;
run;
quit;
```

The following dataset is created using the above code:

Dataset: Salary

Obs	ID	SALARY
1	-20	6860
2	-19	7151
3	-18	847
4	-17	3105
5	-16	7298
6	-15	1086
7	-14	9426
8	-13	2581
9	-12	3950
10	-11	5010

Figure 2. Salary dataset generated using above code

We will perform a match merge operation using the ID variable as keys to merge the two tables and create a resulting table containing the required data variables from both the tables only on the matching key variables. This results in a match merge operation which is much more efficient as compared to the other match merge methods since the entire table is loaded in memory there by resulting in much faster processing speeds.

```
proc ds2;
/*load EMP table into the hash package */
      data emp_salary(overwrite=yes);
            declare char(8) ID;
            declare char(8) INITIALS;
            declare package hash h(8,'emp');
/* define EMP table variable ID as key and INITIALS as data value */
            method init();
                  rc = h.defineKey('ID');
                  rc = h.defineData('INITIALS');
                  rc = h.defineDone();
            end;
/* use the SET statement to iterate over the EMPLOYEE table using */
/* keys in the SALARY table to match keys - ID in the hash package */
            method run();
                  set salary;
                  if (h.find() = 0) then output;
            end;
            enddata;
      run;
quit;
```

The following dataset is created using the above code:

Dataset: Emp_Salary

Obs	ID	INITIALS	rc	SALARY
1	1	FBN	.	8004
2	2	SQO	.	631
3	3	YPS	.	2780
4	4	RHO	.	2314
5	5	HCA	.	9390
6	6	ZJA	.	8315
7	7	VAQ	.	2870
8	8	YZE	.	8152
9	9	XFC	.	1984
10	10	HIY	.	2143

Figure 3. Emp_Salary dataset after match-merge operation

Please note, only the rows matching on key variable ID from both the datasets are output to the resultant dataset. The result is corresponding to that of an inner join between two tables from an SQL query. The

rc.find() method can be used to determine if a match is found or not. The FIND method returns a zero value if the key is found in the hash table and a nonzero value of the key is not found.

THE FCMP PACKAGE

FCMP stands for SAS Function Compiler. The SAS function compiler enables users to create, test and store SAS functions, CALL routines, and subroutines before one can use them in other SAS procedures or DATA steps.

PROC FCMP allows users to build one's own custom functions, CALL routines and sub-routines using the programming syntax similar to that of a DATA step. The SAS PROC FCMP procedure has a syntax that is slightly variant from that of DATA step statements and SAS allows programmers flexibility to use most SAS programming features in functions and CALL routines that are created by PROC FCMP. Programmers can call FCMP functions and CALL routines from the DATA step. Use of FCMP procedure allows programmers flexibility to develop complex code and allows reusability since the call routines are packaged and bundled together as long as the programmer has access to the storage library location.

FCMP procedure uses SAS language complier to execute SAS program. The compiler subsystem generates machine language code for computer on which SAS is running. Programmers can use the functions and subroutines created in PROC FCMP procedure with the DATA step, the WHERE statement, the Output Delivery System (ODS) and with several different procedures (refer the Base SAS 9.4 Procedures Guide for more details).

FCMP PACKAGE IN DS2

FCMP package is a subset of SAS functions, CALL routines and sub routines stored in a single SAS dataset containing FCMP functions which can be called in other DATA or PROC steps. On the other hand a DS2 packages are language constructs that bundle the variables, respective data types and methods into named objects that can be stored and reused by other DS2 programs.

DS2 supports calls to functions and subroutines that are available or are created with the FCMP procedure through an FCMP package.

Please note, a PACKAGE statement is required for all user-defined packages and for the FCMP package that is supplied by SAS. The hash, hash iterator, logger, matrix and SQLSTMT packages since they are system defined packages, do not require an explicit PACKAGE statement.

SAS provides an FCMP package that supports calls to the FCMP functions and subroutines from within the DS2 language

Construct FCMP Package Instance

Programmers can create FCMP package by using the LANGUAGE=FCMP and TABLE=options in a PACKAGE statement. After the package is created, there are two ways to construct an instance of FCMP package:

Using Constructor Syntax

FCMP package instance can be created using the DECLARE PACKAGE statement along with its constructor syntax

```
declare package fcmp area();
```

Using _NEW_ operator

FCMP package instance can also be constructed using DECLARE PACKAGE statement along with the _NEW_operator.

```
declare package fcmp area;
area = _new_ fcmp();
```

Creating an FCMP Package and instantiating in DS2

Please find below an example on ways to create an FCMP package and calling the created routine in DS2 program. Consider a scenario where we want to create a library of our own functions to calculate are of various geometrical objects.

In order to create reusable code, we will create our own functions to calculate the area of various geometrical figures using the FCMP procedure and bundle these functions for re-use in the DS2 procedure. We will store the newly created FCMP package named as fcmparea and call the FCMP function in the DS2.

We use the PROC FCMP procedure to create an FCMP package 'fcmparea' which contains various FCMP functions to calculate area of a square, a rectangle and a triangle respectively. The package is created in the current directory and can be referenced using the base libname

```
libname base '.';
* proc fcmp function to calculate areas of geometrical figures;
proc fcmp outlib = base.fcmparea.package1;
      function square(side);
            return (side*side);
      endsub;

      function rectangle(length,breadth);
            return (length*breadth);
      endsub;

      function triangle(base,height);
        return (0.5*(base*height));
      endsub;
run;
```

We construct an instance of the FCMP package for use in DS2 by defining a DS2 package via which the FCMP functions will be called. See code below to define the DS2 package.

```
* define the ds2 package thru which the fcmp functions will be called;
proc ds2;
      package pkg / overwrite=yes
      language='fcmp'
      table='base.fcmparea';
run;
quit;
```

The following DS2 program instantiates and calls the methods defined in the previous PACKAGE statement. We create test data for enumerating how the area will be calculate for sample values of sides created randomly for enumeration purposes. We create two sides and assign values from 1 to 10 and multiple of 2 times 1 to 10 for side1 and side2 respectively. We call the FCMP functions to calculate area of square, rectangle and triangle for each of these sides respectively.

```
* call fcmp thru the ds2 wrapper package;

      proc ds2;
      data _null_;
            dcl package pkg geometry();
            dcl double side1 side2 area_square area_rectangle area_triangle;
            method init();
                  do side1 = 1 to 10;
                        side2=2*side1;
                        area_square=geometry.square(side1);
                        area_rectangle=geometry.rectangle(side1,side2);
                        area_triangle=geometry.triangle(side1,side2);
```

```
                    put side1= area_square=;
                    put side1= side2= area_rectangle=;
                    put side1= side2= area_triangle=;
                end;
        end;
        enddata;
        run;
quit;
```

Please note, the results can be seen from the SAS log which shows the functions have been applied correctly.

```
side1=1 area_square=1
side1=1 side2=2 area_rectangle=2
side1=1 side2=2 area_triangle=1
side1=2 area_square=4
side1=2 side2=4 area_rectangle=8
side1=2 side2=4 area_triangle=4
side1=3 area_square=9
side1=3 side2=6 area_rectangle=18
side1=3 side2=6 area_triangle=9
side1=4 area_square=16
side1=4 side2=8 area_rectangle=32
side1=4 side2=8 area_triangle=16
side1=5 area_square=25
side1=5 side2=10 area_rectangle=50
side1=5 side2=10 area_triangle=25
side1=6 area_square=36
side1=6 side2=12 area_rectangle=72
side1=6 side2=12 area_triangle=36
side1=7 area_square=49
side1=7 side2=14 area_rectangle=98
side1=7 side2=14 area_triangle=49
side1=8 area_square=64
side1=8 side2=16 area_rectangle=128
side1=8 side2=16 area_triangle=64
side1=9 area_square=81
side1=9 side2=18 area_rectangle=162
side1=9 side2=18 area_triangle=81
side1=10 area_square=100
side1=10 side2=20 area_rectangle=200
side1=10 side2=20 area_triangle=100
NOTE: Execution succeeded. No rows affected.
701  quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           0.11 seconds
      cpu time            0.11 seconds
```

Output 1. Output of DS2 package showing calculation of area for various geometrical figures

In this manner, one can use the FCMP library functions created in prior projects also in the DS2 procedure. One simply needs to instantiate the FCMP library as specified in the above example and the programmer can use the user defined reusable functions in the DS2 programming.

Advantages of FCMP Functions and CALL Routines

PROC FCMP enables programmers to write functions and CALL routines using DATA step syntax. There are multiple advantages to writing one's own user-defined functions, including but not limited to:

- Function(s) are centralized and needs to be changed only once if an update or enhancement needs to be made to the function. All programmers referring to the appropriate FCMP library will be able to benefit from the change without having to update each individual code, ensuring version control
- Function or CALL routine makes a program much easier to read, write and modify
- Functions or CALL routines are independent and not affected by downstream code changes i.e. any change to the functionality of the function does not necessarily impact any other processes except the FCMP package.
- FCMP functions are reusable and allow programmers greater flexibility to define repetitive functions in centralized location. Any program having access to data set where function routine is stored can call the routine.

THE SAS® SQLSTMT PACKAGE

The SAS® SQLSTMT package provides a way to pass FedSQL statements to a DBMS for execution and to access the resulting output set returned by the DBMS. Programmer can perform SAS software equivalent of the ANSI SQL Data Definition Language (DDL) statements such as create, modify or delete tables using FedSQL.

SAS® FEDSQL

The SAS® FedSQL is SAS propriety implementation of ANSI SQL:1999 core standard. FedSQL provides a scalable, threaded, high performance way to access, manage and share relational data in multiple data sources. SAS in the background when possible optimizes FedSQL queries with large multithreaded algorithms to resolve large scale operations. FedSQL provides a common SQL syntax across various data sources thereby providing programmers a vendor-neutral SQL dialect thereby allowing programmers to submit queries without having to worry about syntax specific to the various DBMS sources.

FedSQL Program Execution

FedSQL programs can be executed in multiple ways using SAS software as shown below:

- Using FedSQL procedure using Base SAS
- From a JDBC, ODBC or OLE DB client by using SAS Federation Server
- From a Base SAS language interface by using SAS Federation Server LIBNAME engine and PROC SQL pass-through facility.
- From a SAS DS2 wrapper program
- From the SPD Server using the SQL pass-through facility or ODBC or JDBC client.

Advantages of using FedSQL

FedSQL provides programmers with multiple advantages especially if one is interacting with the other DBMS sources outside of SAS® PROC SQL procedure. See below for the list of advantages of using FedSQL:

- FedSQL conforms to the ANSI SQL:1999 core standard, thereby allowing it to process queries using the standard syntax which is same across other standard DMBS sources which conform to the same standard.
- FedSQL supports many additional datatypes than the previous SAS SQL implementations which was earlier limited to only two datatypes the SAS character and the SAS numeric thereby allowing greater precision during data transfer between external databases and traditional data

sources access through SAS®/ACCESS. FedSQL connects to data sources and translates the target data source definitions to the appropriate data types within FedSQL thereby allowing much greater precision during calculations.

- FedSQL handles federated queries. A federated query is one that accesses data from multiple data sources and returns a single result. As compared with a traditional DATA step or SQL procedure, a SAS®/ACCESS LIBNAME engine can access only the data for its intended data source.
- The FedSQL language can create data in any of the supported data sources, even if the target data source is not represented in any query. This enables users to store data in the data source that most closely meets the needs of one's own application.

Federated Queries

Federated query is a unique feature offered by the FedSQL procedure. A federated query is one that accesses data from multiple data sources, possibly even from various DBMS sources and/or a combination of SAS datasets and DBMS sources and returns a single result set. The data remains stored in the data source. For e.g. in the query below the data is requested form an Oracle data source and a SAS dataset:

```
libname mydata base 'U:\Personal\SAS\SASGF_2018\Data\';
libname myoracle oracle path=ora11g user=xxx pwd=xxx schema=xxx;

proc fedsql;
      select * from mydata.customer
      where exists (select * from myoracle.sales
      where product.prodid=sales.prodid);
quit;
```

SQLSTMT PACKAGE IN DS2

The SQLSTMT package provides a way to pass FedSQL statements to a DBMS for execution and access result set returned by the DBMS. If FedSQL statements selects rows from a data source, the SQLSTMT package provides methods for interrogating the rows returned in a result set.

In DS2, when the SQLSTMT instance is created, the FedSQL statement is sent to the FedSQL language processor which in turn sends the statement to the DBMS to be prepared and stored in the instance. The instance can then be used to efficiently execute the FedSQL statement multiple times. With the delay of the statement prepare until run time, the FedSQL statement can be built and customized during execution of the DS2 program.

For enumeration purposes, consider a sample SAS dataset 'Customer' containing variables Identifier (ID), and quarterly sales information – q1, q2, q3 and q3.

```
libname mydata base "U:\ Personal\SAS\SASGF2018\Data\";

data mydata.customer;
  format  ID 8.  q1 q2 q3 q4 8.;
  do i = 1 to 10;
        ID = put(i, BEST8.);
        q1 = int(ranuni(0)*10000);
        q2 = int(ranuni(0)*10000);
        q3 = int(ranuni(0)*10000);
        q4 = int(ranuni(0)*10000);

        output;
  end;
  drop i ;
```

```
run;
```

Please find the output of the sample dataset created below:

Dataset: customer

Obs	ID	q1	q2	q3	q4
1	1	9773	7006	399	5113
2	2	838	9233	1416	9997
3	3	8514	128	83	2117
4	4	1214	663	7723	1504
5	5	9465	6008	9234	1952
6	6	5257	3569	1474	3923
7	7	7596	4810	98	9412
8	8	2685	2052	8586	9815
9	9	1742	1853	4139	5481
10	10	7488	8361	7274	3603

Figure 4. Customer dataset before update

Please find below an example of using SQLSTMT to insert values in a SAS dataset. The following program inserts few additional customers (total of 5 additional) in the original dataset and updates the sales information for the same.

```
proc ds2;
  data _null_;
    dcl double x;
    dcl double y;
    dcl double z;
    dcl double w;
    dcl double u;
    dcl package sqlstmt s('insert into mydata.customer
                           (ID, q1, q2, q3, q4)
                           values (? , ? , ? , ? , ?)
                          ',[x y z w u]);

      method init();
        do i=11 to 15;
          x = put(i, BEST8.);
          y = int(ranuni(0)*10000);
          z = int(ranuni(0)*10000);
          w = int(ranuni(0)*10000);
          u = int(ranuni(0)*10000);
          s.execute();
        end;
      end;

  enddata;
run;
quit;
```

From the above example you can observe how a SQLSTMT instance can be invoked from within DS2. The resulting output table is updated with the additional rows:

Dataset: customer

Obs	ID	q1	q2	q3	q4
1	1	9773	7006	399	5113
2	2	838	9233	1416	9997
3	3	8514	128	83	2117
4	4	1214	663	7723	1504
5	5	9465	6008	9234	1952
6	6	5257	3569	1474	3923
7	7	7596	4810	98	9412
8	8	2685	2052	8586	9815
9	9	1742	1853	4139	5481
10	10	7488	8361	7274	3603
11	11	8584	9756	7372	9387
12	12	7027	1474	1115	2500
13	13	7363	1790	9430	3335
14	14	3393	2199	3046	1796
15	15	9050	1052	1752	824

Figure 5. Customer dataset after insert operation

Declaring and Instantiating an SQLSTMT Package

Programmers can use DECLARE PACKAGE statement to declare the SQLSTMT package. Programmers can create a variable to reference the instance of the package during package declaration. In the example below, '**s**' is the variable used to reference the constructed package instance.

```
dcl package sqlstmt s('insert into mydata.customer
                       (ID, q1, q2, q3, q4)
                       values (? , ? , ? , ? , ?)
                      ',[x y z w u]);
```

There are three ways to construct an instance of an SQLSTMT package.

1. Using constructor syntax

   Please find below two syntax forms for instantiating a package using DECLARE PACKAGE statement along with its constructor syntax:

   DECLARE PACKAGE SQLSTMT variable [('sql-txt' [, \[parameter-variable-list\])];

   DECLARE PACKAGE SQLSTMT variable [('sql-txt' [, connection-string])];

2. Using _NEW_ operator

   Please find below two syntax forms for instantiating package using DECLARE PACKAGE statement along with _NEW_operator:

   DECLARE PACKAGE SQLSTMT variable;

   variable = _NEW_ SQLSTMT ('sql-txt' [,\[parameter-variable-list\]);

   DECLARE PACKAGE SQLSTMT variable;

   variable = _NEW_ SQLSTMT ('sql-txt' [, connection-string]);

   NOTE: The DECLARE PACKAGE statement does not construct the SQLSTMT package instance until the _NEW operator is executed. The SQL statement prepare does not occur until the _NEW_ operator is executed.

3. Without using SQL text

DECLARE PACKAGE SQLSTMT variable();

variable = _NEW_ SQLSTMT ();

With the _NEW operator, the sql-text can be a string value that is generated from an expression or a string value that is stored in a variable.

If the DECLARE statement includes arguments for construction within its parentheses (and omitting arguments is valid for the SQLSTMT package), then the package instance is allocated. If no parentheses are included, then a variable is created but the package instance is not allocated.

Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

Specifying a Connection String

A connection string defines how to connect to data, it identifies query language to be submitted as well as information required to connect to data or respective DMBS or other data sources.

Programmers can specify connection string during declaration and instantiation of an SQLSTMT package. Connection string parameter is primarily designed for use with SAS Federation server.

If connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 or DS2 procedure by using the attributes of the currently assigned libref.

Executing FedSQL Statement

The EXECUTE method executes the FedSQL statement and returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists when an SQL UPDATE or DELETE statement does not affect any rows.

When the FedSQL statement is executed, the values of the bound variables are read and used as the values of the statement's parameters.

An SQLSTMT instance maintains only one result set. The result set from the previous execution, if any, is released before the FedSQL statement is executed.

FedSQL statement executes dynamically at runtime and because the statement is prepared at run time, it can be built and customized dynamically during the execution of the DS2 program.

Accessing Result Set Data

FETCH method returns the next row of the data from the result set. A status indicator is returned. Zero is returned for a successful execution; 1 is returned if there is an ERROR; 2 is returned of there is no data (NO DATA). The NODATA condition exists if the next row to be fetched is located after the end of the result set.

If variables are bound to the result set columns with the BINDRESULTS method or by the FETCH method, then the fetched data for each result set column is placed in the variable bound to that column. If the variables are not found to the result set columns, the fetched data can be returned by the GETtype methods. For character data, programmers can all the GETtype method repeatedly until all of the result set column data is retrieved. For numeric data, programmers can call the GETtype method only once to return result set column data. Subsequent method calls result in a value 2(NODATA) for the rc status indicator.

A SQLSTMT instance maintains only one result set. The CLOSERESULTS method automatically releases the result set when the FedSQL statement is executed or deleted.
A run-time error occurs if the FETCH method is called before the FedSQL statement is executed.

Comparing SQLSTMT Package and the SQLEXEC Function

See below for table comparing the SQLSTMT package and SQLEXEC function.

SQLSTMT Package	SQLEXEC Function
applicable when FedSQL statements are executed multiple times	applicable when a FedSQL statement is executed only once
allocates, prepares, executes, and frees a FedSQL statement dynamically at run time	allocates, prepares, executes, and frees a FedSQL statement dynamically at run time
supports the passing of parameters	does not support the passing of parameters
produces a result set	does not produce a result set
supports run-time SELECT query generation	cannot be used with a SELECT statement
similar to the Java Database Connectivity (JDBC) PreparedStatement class	similar to the SQL EXECUTE IMMEDIATE statement or the JDBC Statement.executeUpdate(String) method

Table 1. SQLSTMT Package vs SQLEXEC function

INVOKE DS2 PACKAGE METHODS USING FEDSQL

The FedSQL language supports ability to invoke user-defined DS2 package methods as functions in the SELECT statement. This is an important feature as a user can invoke SAS user-defined functions while reading from DBMS data sources other than SAS i.e. Oracle, DB2 etc. Imagine the flexibility that is rendered to the programming when one can execute user defined specific SAS functions directly on the data from external DBMS sources.

Please find a sample code enumerating this functionality below:

```
proc ds2;
  package adder / overwrite =yes;
    method add( double x, double y ) returns double;
        return x + y;
      end;
  endpackage;
  data numbers / overwrite = yes;
    dcl double x y;
      method init();
        dcl int i;
        do i = 1 to 10;
          x = i; y = i * i;
            output;
        end;
      end;
  enddata;
  run;
quit;

proc fedsql;
  select x, y, work.adder.add( x, y ) as z from work.numbers;
quit;
```

In the above example, we create and submit a DS2 package method named Add created within the Adder package. Further, we execute the FedSQL code to invoke the user defined method Add to add two numbers from the numbers dataset. The numbers dataset is a test dataset we create containing sample

data, i.e. tow numeric variables ‘x’ and ‘y’. The Add method is invoked via FedSQL to calculate an additional variable ‘z’.

COMPARISON OF SAS DATA STEP AND PROC DS2

PROC DS2 shares its core components with a SAS data step. Although, there are also differences between a DS2 procedure and a SAS data step.

SIMILARITIES BETWEEN DS2 AND DATA STEP

DS2 and the DATA step share many language elements, and those elements behave in the same way:

- SAS formats.
- SAS functions.
- SAS statements such as DATA, SET, KEEP, DROP, RUN, BY, RETAIN, PUT, OUTPUT, DO, IF-THEN/ELSE, Sum, and others.
- DATA step keywords are included in the list of DS2 keywords.

You can perform most DATA step tasks using DS2:

- process variable arrays, multi-dimensional arrays, and hash tables
- convert between data types
- work with expressions
- calculate date and time values
- processing missing values

DIFFERENCES BETWEEN DS2 AND DATA STEP

In a data step the executable code resides in the DATA and PROC step, whereas the executable code resides in the methods. There is no concept of scope in a data step, whereas in DS2, variables that are declared in methods have local scope and all others have global scope. There are three types of global scope in DS2: 1) Data program, 2) thread program and 3) package. Declaring variables is not required in a data step. The variables are created on the fly by assignment. In DATA step, data type of the variable is determined by the context it is first used. All variables have global scope. In DS2, variables need to be explicitly declared using the DECLARE statement, which also determines data type and scope attribute of the variables. The data types supported in data step are mainly character and numeric. Numeric data is signed fractional, limited to bytes and has approximate precision. Character data is fixed length. In PROC DS2, almost all ANSI SQL data types are supported. Numeric types of varying sizes and precision. Character data types can be fixed or variable length. DS2 also supports ANSI date, time and timestamp data types, but can also process SAS date, time and datetime values using conversion functions. SQL language statements cannot be written in a SAS data step, they are available in PROC SQL. Whereas SQL select statements can directly be written in and used as input for a DS2 SET statement. In addition the SQLSTMT predefined package provides a way to pass SQL statements to a DBMS for execution and to access the result set returned by DBMS

WHEN TO USE DS2

DS2 offers many benefits in terms of added flexibility for programming, reusability of code, centralizing user defined functions etc. but programmers don’t necessarily have to convert all DATA step programs to DS2. Developing DS2 programs is most beneficial to programmers when the programmers:

- Require the precision that results from using the new supported data types
- Benefit from using the new expressions or write methods or packages available in the DS2 syntax
- Need to execute SAS FedSQL from within the DS2 program
- Execute outside a SAS session, for example, in-database processing on Hadoop or Teradata, in SAS Viya, or the SAS Federation Server
- Take advantage of threaded processing in products such as the SAS In-Database Code Accelerator and SAS Enterprise Miner

CONCLUSION

In conclusion, we have observed that DS2 is a very powerful language. It allows a great deal of additional functionality such as support for different data types allowing for greater precision in data processing, threaded application processing resulting in faster processing speeds in on a machine with multiple cores as well as within massively parallel processing databases. It offers support for in-database processing at the disposal of an application developer and accepts embedded FedSQL which allows users to connect to multiple tables within disparate databases within a single query and extract data for processing. DS2 is a powerful language. The DS2 language shares core features with the DATA step. However, capabilities of DS2 extend far beyond those of the DATA step.

REFERENCES

SAS Institute Inc. 2017. SAS® 9.4 DS2 Programmer's Guide. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2016. SAS® 9.4 DS2 Language Reference, Sixth Edition. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2014. SAS® 9.4 FedSQL Language Reference, Third Edition. Cary, NC: SAS Institute Inc.

Dorfman, Paul M. 2016. Fundamentals of the SAS Hash Object, Proceedings of SESUG 2016

ACKNOWLEDGMENTS

I would like to profusely thank John Hagen and Mike Zeffiro for supporting my work at MUFG Union Bank. I would also like to thank my manager Uma Karanam and my colleagues Thomas Billings and Baskar Anjappan for encouraging me to write and publish a paper. I would also thank my employers MUFG Union Bank for providing me with an opportunity for attending the conference.

RECOMMENDED READING

- *Mastering the SAS DS2 Procedure, Mark Jordan, SAS Institute 2016*
- *The DS2 Procedures: SAS Programming Methods at Work, Peter Eberhardt, SAS Institute, 2016*

CONTACT INFORMATION

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

Name: Viraj R Kumbhakarna
Enterprise: MUFG Union Bank N.A.
E-mail: vkumbhakarna@gmail.com

DISCLAIMER

The contents of the paper herein are solely the author's thoughts and opinions, which do not represent those of MUFG Union Bank N.A. MUFG Union Bank N.A. does not endorse, recommend, or promote any

of the computing architectures, platforms, software, programming techniques or styles referenced in this paper.