Paper BB-10-2014

DS2: The New and Improved DATA Step in SAS®

Rajesh Lal, Experis US Inc., Portage, MI

ABSTRACT

The DATA step is a fundamental building block of Base SAS language or any SAS program. It is characterized by implicit loop of SET statement, reading and writing dataset observations, implicit global variable declaration and access to a huge library of SAS functions and ability to use inbuilt or user-defined formats.

DS2 is a SAS proprietary programming language that is used for data manipulation and data modeling applications. It shares the core features of the DATA step and extends far beyond that. It includes several features that are not available in the DATA step like: variable scoping, user-defined methods, ANSI SQL data types, user-defined packages, and programming structure elements. This paper discusses the features, basic components, usage and examples of DS2.

INTRODUCTION

DS2 is a powerful programming language that is used for advanced data manipulation and data modeling applications. DS2 features extend far beyond the DATA step by adding variable scoping, user-defined methods, ANSI SQL data types and user-defined packages. User-defined packages provide modularity and data encapsulation. DS2 makes it possible to insert SQL directly into the SET statement, blending the power of two powerful data manipulation languages.

There are additional features of DS2 that are not available with the DATA step, such as predefined and user-defined methods and packages. DS2 also includes additional data types, ANSI SQL data types, and programming structure elements. In addition, the DS2 SET statement has been enhanced to accept ANSI SQL: 1999, enabling SQL to preprocess input tables. This feature effectively combines the power of the DS2 and SQL languages.

DS2 is particularly suited for the programs/applications that:

- require the precision that new supported data types offer
- benefit from using the new expressions or write methods or packages
- need to execute SAS FedSQL from within the DS2 program
- execute outside a SAS session, e.g. on High-Performance Analytics Server or the SAS Federation Server
- take advantage of threaded processing in products such as the SAS In-Database Code Accelerator, SAS High-Performance Analytics Server, and SAS Enterprise Miner

DS2 AND DATA STEP

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

Formats, functions, statements like 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.

Most of the DATA step tasks can be performed using DS2: variable arrays, multi-dimensional arrays, hash tables, expressions, date and time values, conversion between data types.

There are many differences between DS2 and the DATA Step, which are summarized in the table below:

Feature	DATA Step	DS2
Variable Scope	No concept of scope. All variables are global within the DATA step.	Variables that are declared in a method have local scope. All other identifiers have global scope. Global scope can be in data program, thread program or package.
Variable declaration	Variables are not explicitly declared. Variables are created by assignment. The data type of a variable is determined by the context of how it is first used.	Variables are declared using the DECLARE statement, which also determines the data type and scope attributes of the variable. Variables can be declared by assignment. Variable declaration strict mode can be enforced by setting the system option DS2SCOND=ERROR or the PROC DS2 option SCOND=ERROR.
Keywords and reserved words	No reserved keywords.	Keywords are reserved words.
Quotation marks	Single or double quotation marks can delimit a character constant.	ANSI SQL quoting standards are followed. Single quotation marks delimit a character constant. Double quotation marks delimit an identifier.
PUT statement	Supports column and line parameters.	Column and line parameters are not supported. Dot notation parameters are not supported.
Programming paradigm	Executable code resides in the DATA step and PROC step.	Executable code resides in methods.

DS2 LANGUAGE COMPONENTS

DATA TYPES

DS2 supports many of the ANSI SQL data types that are native to the data sources that SAS supports, which are not available in Base SAS. DS2 has fixed and variable length character data types, numeric data types of different sizes for storing fractional, integer and binary numbers and ANSI date, time and timestamp data types. For details please refer to Appendix 1.

VARIABLES

Variables in DS2 are 1-256 characters in length and follow the naming convention similar to DATA step variables. Variable data types are assigned either implicitly or explicitly depending on how they are declared. Variables in DS2 can be declared in three ways:

1. Explicit declaration by using the DECLARE statement

   The DECLARE statement associates a data type with each variable in a variable list or an array. If the DECLARE statement is used outside a method, a global variable is created. If the DECLARE statement is used within a method, a local variable is created. Within a method, DECLARE statements must precede method statements. Otherwise, an error occurs.

2. Implicit declaration by using a SET statement

   The SET statement reads the column information for each specified table. For each column in each table, the SET statement creates a global variable in the DS2 program with the same data types as those of the column.

3. Implicit declaration by using an undeclared variable in a programming block

If you use a variable without declaring it, DS2 assigns the variable a data type. By default, a warning is sent to the SAS log. The data type for an undeclared variable on the left side of an assignment statement is determined by the data type of the value on the right side of the assignment statement. If the data type of the value on the right side of the assignment statement is numeric or NULL, then type DOUBLE is assigned to the left side variable. Otherwise, the data type of the value on the right side is assigned to the left side variable.

To control how DS2 handles an undeclared variable, DS2SCOND system option or the SCOND option can be used.

CONSTANTS

A constant is a number, character string, binary number, date, time, or timestamp that indicates a fixed value. Some examples DS2 constants are shown here:

107

'SAS Visual Analytics'

date '2014–01–01'

b'10011001'

x'FFE3546F'

A point to be noted is that a sequence of characters enclosed in double quotation marks is not a character constant, it is an identifier.

EXPRESSIONS

An expression is made of up of operands, and optional operators, that form a set of instructions and that resolves to a value. An operand can be a single constant or variable, or it can be an expression. Operators are the symbols that represent a calculation, comparison, or concatenation of operators.

The following are examples of DS2 expressions:

a = b * c

"col1"

s || 1 || z

a >= b**(c - 8)

system.put(a*5,hex.)

PROGRAMMING BLOCKS AND SCOPE

A programming block defines a section of a DS2 program that encapsulates variables and code. Programming blocks encourage the creation of modular, reusable code. In addition, a programming block defines the scope of identifiers within that block. In DS2, it is possible for variables to have the same name and data type, as long as they have different scope. Table 1 summarizes the characteristics of DS2 programming blocks and scope (each of these blocks will be described later):

Table 1

Block	Delimiters	Scope of variable declared at top	Included in…	Existence
Data program	DATA... ENDDATA	Global within the data program. Variables referenced by SET statement also have global scope.	PDV	Duration of the data program
Package	PACKAGE... ENDPACKAGE	Global scope within the package.	None	Duration of the package instance.
Thread program	THREAD... ENDTHREAD	Global scope within the thread program. Variables referenced by SET statement also have global scope.	Thread output set	Duration of the thread program instance, but they can be passed to the SET FROM statement in the data program

Method	METHOD... END	Local scope. Method names have global scope within the enclosing block.	None	Duration of the method call.
DO loop	DO... END	Not applicable.	None	None

METHODS

Methods are basic program execution units. A method is a sub-block of a data program, package, or thread program. Method names have global scope within the enclosing block. Variables that are declared at the top of this programming block have local scope. Local variables are not included in the PDV and exist for the duration of the method call. Any parameters and any variable declarations in the method body are local to the method. In DS2, all program code must reside in some method.

System methods have a preset meaning in DS2. There are three system methods: INIT, RUN, and TERM. These methods cannot be overloaded. There is one optional system method that is used only with threads: SETPARMS. Table 2 lists and summarizes the purpose of each DS2 system method.

Table 2

System Method	Execution Details	Purpose
INIT( )	Automatically executes one time, as the first method of a program.	INIT( ) is a good place to initialize global program variables. Most global variables are not initialized by the system. However, the system does initialize predefined variables, such as _N and _N_, and variables that are used in Sum and RETAIN statements.
RUN( )	Automatically executes after INIT( ) completes.	The RUN( ) method is the functional equivalent of the DATA step. That is, if your RUN( ) method contains a SET statement, the method runs as an implicit loop. You can also use RUN( ) to read rows from a thread program using the SET FROM statement.
TERM( )	Automatically executes one time, as the last method of a program.	As the name implies, TERM( ) is where final processing takes place, before the program exits. TERM( ) automatically resets global variables to uninitialized values, except: predefined variables, such as _N and _N_, accumulator variables that were used in Sum statements, variables that were used in a RETAIN statement and package variables.
SETPARMS( )	Executes one time, when called from a data program, to initialize the values of a parameterized thread.	SETPARMS( ) initializes the values of a parameterized thread. Because only parameterized thread programs require this, SETPARMS( ) is the only system method that must be called. *Note:* Do not write a SETPARMS( ) method in your thread program. The system supplies the method for you. Do not call SETPARMS( ) more than once. The initialization only works the first time.

PACKAGES

A DS2 package is a collection of methods and variables that can be used in DS2 programs. A DS2 package supports a set of related tasks and is designed for reuse. By appropriately defining and using packages, modularity and data encapsulation can be achieved. There are two types of packages: User-defined packages and predefined packages.

LEARNING BY EXAMPLES

USING PROC DS2

When using the DS2 procedure to write a program, place the code within the following framework:

```
options ds2scond=error;
proc ds2;
...DS2 statements...
```

```
run;
quit;
```

FIRST DS2 PROGRAM

The following program writes “My first DS2 program” to the SAS log:

```
proc ds2;
data _null_;
    method init(); /* System method INIT()*/
           declare varchar(50) message; /* method (local) scope */
           message = 'My first DS2 program';
           put message;
    end;
enddata;
run;
quit;
```

The following is written to the log:

```
My first DS2 program
```

Let’s try to follow the above program. Within PROC DS2 and QUIT, we have written a DATA…ENDDATA block which defines the system method INIT. The variable MESSAGE is declared as VARCHAR and has local scope because it is declared in the INIT( ) method. The INIT( ) system method automatically runs first in a DS2 data program. Single quotation marks delimit the character constant.

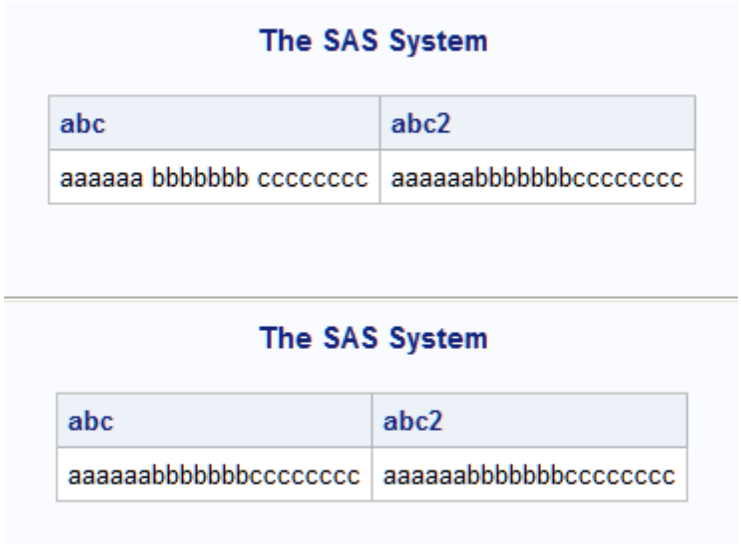
SCOPE

This example shows the use of CHAR and VARCHAR types and their operators and functions.

```
proc ds2;
data;
    dcl char(24) abc abc2;
    method init();
           dcl char(8) a b c;
           a = repeat('a',5);
           b = repeat('b',6);
           c = repeat('c',7);
           abc = a || b || c;
           abc2 = trim(a) || trim(b) || c;
    end;

enddata;
run;
data;
    dcl char(24) abc abc2;
    method init();
           dcl varchar(8) a b c;
           a = repeat('a',5);
           b = repeat('b',6);
           c = repeat('c',7);
           abc = a || b || c;
           abc2 = trim(a) || trim(b) || c;
    end;
enddata;
run;
quit;
```

In this example, the variables A, B, and C are locally scoped to the INIT method. That is, their value is not seen outside of the INIT method and is not output to the result table.



The SAS System

abc	abc2
aaaaaa bbbbbbb cccccccc	aaaaaabbbbbbbccccccccc

The SAS System

abc	abc2
aaaaaabbbbbbbccccccccc	aaaaaabbbbbbbccccccccc

Figure 1. SAS output

PACKAGE

This program creates GREETING package with overwrite option. The GREETING package has two constructors: a default constructor and one that accepts an argument. This program uses multiple instantiation of packages, without creating a data set.

```
proc ds2;
    /* GREETING - User-defined package that writes a message to the SAS log */
    package greeting /overwrite=yes;
           dcl varchar(100) message; /* package (global) scope */
           FORWARD setMessage;
           /* greeting() - default constructor */
           method greeting();
                  setMessage('This is the default greeting.');
           end;
           /* greeting(MESSAGE) - constructor */
           method greeting(varchar(100) message);
                  setMessage(message);
           end;
           method greet();
                  put message;
           end;
           method setMessage(varchar(100) message);
                  /* Must use THIS. to distinguish global */
                  /* variable MESSAGE from parameter named MESSAGE. */
                  this.message = message;
           end;
    endpackage;
run;
/* data program */
data _null_;
    /* All package instances have global scope in the data program. */
    dcl package greeting
           g0() g1('Hello World!') g2('What''s new?') g3('Good-bye World!');
    /* init() - automatically runs first in the data program.*/
    method init();
           g0.greet();
           g1.greet();
           g2.greet();
           g3.greet();
end;
```

```
enddata;
run;
quit;
```

The following is written to the log:

```
This is the default greeting.
Hello World!
What's new?
Good-bye World!
```

PARALLEL PROCESSING IN DS2

DS2 supports parallel execution of a single program that can operate on different parts of a table. This type of parallelism is classified as Single Program, Multiple Data (SPMD) parallelism. In DS2, it is the responsibility of the programmer to identify the program statements that can operate in parallel.

For programs that are CPU bound, using a thread program on Symmetric Multiprocessing (SMP) hardware can improve performance. For programs that are either CPU or I/O bound, Massively Parallel Processing (MPP) hardware can improve performance.

The following program demonstrates how a thread creates data and passes variables to the data program:

```
options DS2SCOND=ERROR;
proc ds2;
    /* thread program - Creates data in a loop */
    thread work.t (double d) /overwrite=yes;
           dcl int x;
           dcl double y;
           method init();
                  dcl int i; /* local - not included in the output data set */
                  do i = 1 to 9;
                         x = i;
                         y = i * 2.5 + d;
                         put 'THREAD: i=' i ' x= ' x ' y= ' y;
                         output; /* output variables include X and Y */
                  end;
           end;
           method term();
                  put 'THREAD TERM (_ALL_):';
                  put _all_;
           end;
    endthread;
run;
/* data program - Reads data from a thread program */
data;
    dcl thread work.t t;
    dcl double answer total;
    method init();
           t.setparms(1.25); /* initialize parameter of thread */
           put 'INIT (_ALL_):';
           put _all_;
    end;
    method run();
           set from t threads=2; /* input variables include X and Y */
           answer = x + y;
           total+answer; /* Sum statement syntax implicitly retains TOTAL */
           put ' x= ' x ' y= ' y ' answer= ' answer ' total= ' total;
    end;
    method term();
           put 'TERM: (_ALL_)';
           put _all_;
    end;
enddata;
run;
```

```
quit;
```

The following is written to the log:

```
INIT (_ALL_):
total=0 answer=. x= y=. _N_=1
THREAD: i= 1 x= 1 y= 3.75
THREAD: i= 1 x= 1 y= 3.75
THREAD: i= 2 x= 2 y= 6.25
THREAD: i= 3 x= 3 y= 8.75
THREAD: i= 4 x= 4 y= 11.25
THREAD: i= 5 x= 5 y= 13.75
THREAD: i= 6 x= 6 y= 16.25
THREAD: i= 7 x= 7 y= 18.75
THREAD: i= 8 x= 8 y= 21.25
THREAD: i= 2 x= 2 y= 6.25
THREAD: i= 9 x= 9 y= 23.75
THREAD: i= 3 x= 3 y= 8.75
THREAD TERM (_ALL_):
THREAD: i= 4 x= 4 y= 11.25
d=1.25 x= y=. _N_=1
THREAD: i= 5 x= 5 y= 13.75
THREAD: i= 6 x= 6 y= 16.25
x= 1 y= 3.75 answer= 4.75 total= 4.75
THREAD: i= 7 x= 7 y= 18.75
THREAD: i= 8 x= 8 y= 21.25
THREAD: i= 9 x= 9 y= 23.75
THREAD TERM (_ALL_):
d=1.25 x= y=. _N_=1
x= 2 y= 6.25 answer= 8.25 total= 13
x= 3 y= 8.75 answer= 11.75 total= 24.75
x= 4 y= 11.25 answer= 15.25 total= 40
x= 5 y= 13.75 answer= 18.75 total= 58.75
x= 6 y= 16.25 answer= 22.25 total= 81
x= 7 y= 18.75 answer= 25.75 total= 106.75
x= 8 y= 21.25 answer= 29.25 total= 136
x= 9 y= 23.75 answer= 32.75 total= 168.75
x= 1 y= 3.75 answer= 4.75 total= 173.5
x= 2 y= 6.25 answer= 8.25 total= 181.75
x= 3 y= 8.75 answer= 11.75 total= 193.5
x= 4 y= 11.25 answer= 15.25 total= 208.75
x= 5 y= 13.75 answer= 18.75 total= 227.5
x= 6 y= 16.25 answer= 22.25 total= 249.75
x= 7 y= 18.75 answer= 25.75 total= 275.5
x= 8 y= 21.25 answer= 29.25 total= 304.75
x= 9 y= 23.75 answer= 32.75 total= 337.5
TERM: (_ALL_)
total=337.5 answer=. x=9 y=23.75 _N_=19
```

The parameterized thread accepts a value that must be initialized by the data program using the SETPARMS( ) system method. The OVERWRITE=YES table option enables the thread program to be overwritten. Note that the THREAD statement syntax requires the '/' (slash character) syntax. Because the data program specifies two threads, the thread program runs in two separate threads in a single process. Please note that this thread program produces one set of output variables per thread. Because threads run asynchronously, the order of processing is unpredictable. In the data program, the global accumulator variable TOTAL is implicitly retained because of the total+answer; Sum statement syntax.

MODULARITY, ENCAPSULATION, AND ABSTRACTION IN DS2

When you use DS2 methods and packages, you approach writing SAS programs differently than you do when programming in Base SAS. These DS2 language constructs follow a more structured-programming and object-oriented approach.

In general, DS2 methods are like functions, procedures, subroutines, and the methods of object-oriented languages such as C++, 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 you to break up a complex problem into smaller modules. Such modules are easier to design, implement, and test. Code reuse can shorten development time and help standardize often-repeated or business-specific programming tasks. Also, modular programming enhances readability and understandability by testers and other programmers.

DS2 packages enable encapsulation and abstraction of behavior. DS2 packages are similar to classes in object-oriented languages. However, a DS2 package can also be used as a bucket of useful but unrelated methods and variables, if that meets your needs.

A DS2 package 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 and private), packages can be designed to abstract behavioral details. In such a package, the methods define the object and enable controlled manipulation of package data.

LIMITATIONS AND CONSIDERATIONS

You do not necessarily have to convert your DATA step programs to DS2. You need to weigh in the advantages of using DS2 with additional complexity of creating and maintaining DS2 programs. In general, if a problem is simple enough to be solved by writing a Base SAS program, go with Base SAS.

CONCLUSION

DS2 is a powerful language that can help a programmer write re-usable, dynamic and low maintenance programs. With the introduction of DS2, Base SAS has given the power and flexibility to a programmer that is beyond that of the DATA step. By using DS2 methods and packages, you approach writing SAS programs differently than you do when programming in Base SAS. DS2 language constructs follow a more structured-programming and object-oriented approach.

REFERENCES

- SAS Online Documentation
- "I Object: SAS® Does Objects with DS2" by Peter Eberhardt and Xue Yao. http://support.sas.com/resources/papers/proceedings14/1283-2014.pdf
- "Using Base SAS® to Extend the SAS® System" by Mark L. Jordan http://support.sas.com/resources/papers/proceedings14/SAS013-2014.pdf

APPENDIX 1 – DATA TYPES IN DS2

Data Type	Description
BIGINT	To store a large, signed, exact whole number, with a precision of 19 digits.
BINARY(*n*)	fixed-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
CHAR(*n*)	stores a fixed-length character string,
DATE	stores a calendar date. A date literal is specified in the format *yyyy-mm-dd*: a four-digit year (0001 to 9999), a two-digit month (01 to 12), and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as `1975-09-24`. DS2 complies with ANSI SQL:1999 standards regarding dates. However, not all data sources support the full range of dates.

DECIMAL|NUMERIC(*p*,*s*)	stores a signed, exact, fixed-point decimal number, with user-specified precision and scale.
DOUBLE	stores a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.
FLOAT(*p*)	stores a signed, approximate, single-precision or double-precision, floating-point number. The user-specified precision determines whether the data type stores a single-precision or double-precision number. If the specified precision is equal to or greater than 25, the value is stored as a double-precision number, which is a DOUBLE
INTEGER	stores a regular size signed, exact whole number, with a precision of ten digits.
NCHAR(*n*)	stores a fixed-length character string like CHAR but uses a Unicode national character set
NVARCHAR(*n*)	stores a varying-length character string like VARCHAR but uses a Unicode national character set,
REAL	stores a signed, approximate, single-precision, floating-point number.
SMALLINT	stores a small signed, exact whole number, with a precision of five digits
TIME(*p*)	stores a time value. A time literal is specified in the format *hh:mm:ss[.nnnnnnnnn]*; a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value.
TIMESTAMP(*p*)	stores both date and time values. A timestamp literal is specified in the format *yyyy-mm-dd:hh:mm:ss[.nnnnnnnnn]*: a four-digit year 0001 to 9999, a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value
TINYINT	stores a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127. Integer data types do not store decimal values; fractional portions are discarded.
VARBINARY(*n*)	stores varying-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If **`varbinary(10)`** is specified and the binary string uses only five bytes, only five bytes are stored in the column.
VARCHAR(*n*)	stores a varying-length character string, where *n* is the maximum number of characters to store. The maximum number of characters is not required to store each value. If **`varchar(10)`** is specified and the character string is only five characters long, only five characters are stored in the column.

ACKNOWLEDGMENTS

I would like to thank Brian Varney for reviewing this paper and providing valuable comments.

CONTACT INFORMATION

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

Name: Rajesh Lal
Enterprise: Experis US Inc.

Address: 5220 Lovers Lane, STE 200
City, State ZIP: Portage, MI 49002
Work Phone: 269-553-5147
Fax: 269-553-5101
E-mail: rajesh.lal@experis.com
Web: www.experis.com