

A Brief Introduction to DS2

Mark Jordan, SAS Institute, Inc.

ABSTRACT

DS2 is a SAS programming language available in Base SAS 9.4+ and all versions of SAS Viya. The language is designed for simple, safe parallel processing using an intriguing combination of SQL and traditional DATA step syntax. This paper shows some of the benefits of processing in DS2, both in SAS9 and SAS Viya. With DS2, you can work with database high-precision numeric values, such as BIGINT and DECIMAL, without loss of precision. You can achieve a significant boost in processing speed by converting a CPU-bound SAS DATA step to DS2 for execution in base SAS. And finally, I'll demonstrate how amazingly fast the process can be when executed in CAS on SAS Viya.

INTRODUCTION

I learned about DS2 and began writing the first SAS training course for DS2 programming in 2013. I quickly developed a fondness for this amazingly versatile language. This paper is intended to acquaint you with some of the most useful features in DS2, including:

- Basic DS2 programming syntax
- Similarities to and differences from traditional DATA step programming.
- Thumb rules for choosing DS2 or DATA step
- How to access and manipulate database table data at full precision when processing ANSI data types like DECIMAL, BIGINT, and VARCHAR.
- How to convert DATA step programs into DS2 threads to enable parallel processing.

WHAT IS DS2?

DS2 is a modern base SAS programming language designed to provide advanced data manipulation techniques using an intriguing mix of SAS DATA-step-style code and SQL syntax. Although the structure is a bit different, DATA step features, like the DATA, assignment, PUT, and RUN statements are carried forward.

WHY LEARN TO PROGRAM IN DS2?

DS2 provides several "superpowers" not available in other SAS languages. For example, most SAS languages can process only 8-byte floating point numbers or fixed-width character data, but DS2 handles most ANSI data types natively. When reading data from a database table, most SAS languages rely on the SAS/ACCESS engine to retrieve the database data and translate database data types to fixed-width character or double-precision floating point numeric values before the data can be processed in SAS. This can sometimes result in a loss of precision. Because DS2 can natively process most ANSI data types, DS2 bypasses the SAS/ACCESS engine, instead using a special driver to access database tables and process the data directly in SAS. This allows DS2 to process high-precision numeric data types, like DECIMAL and BIGINT, without losing precision.

It's also easy to extend DS2's capabilities, and to securely share those extensions with others. In DS2 you can write custom methods. Methods are named blocks of executable code that can be used to build more complex programs and work much like DATA step functions and call routines. Collections of DS2 methods can be stored in SAS libraries as packages of encrypted source code. When a DS2 package instance is included in a DS2 program, any of the package's methods can be executed simply by referencing it using dot notation.

And because it was built with parallel processing in mind, DS2 includes simple, safe syntax that enables processing multiple rows of data simultaneously. On the SAS Compute Server, the Base SAS DATA step reads and processes each row sequentially. Using threads, DS2 can process several rows of data in parallel. A single DS2 read-write thread distributes blocks of data to multiple DS2 compute threads. The compute threads process rows of data in parallel, returning processed rows to the read-write thread for output. But DS2 can also run in many massively parallel processing (MPP) environments, providing exceptional flexibility. Using the SAS In-Database Code Accelerator Embedded Process (SAS EP) software installed on supported databases, DS2 code can be sent to the SAS EP on each database node. Then, each node lifts the portion of the data stored there into memory, processes the data, and writes the processed data back to disk in parallel. This process brings the code into the data instead of extracting data and bringing it back to the Compute Server for processing - nothing returns to the SAS session except the log. Data movement and the associated latency is eliminated, providing an excellent performance boost. Finally, in SAS Viya, DS2 can run natively in Cloud Analytic Services (CAS), where the is not only distributed, but already resident in memory. Each node processes its portion of the data in parallel. And because the data persists in memory, subsequent programs can operate on the data again without having to reload it from disk. Many processes run exceptionally fast in this environment, and DS2 is no exception.

WHEN SHOULD I WRITE MY PROGRAMS IN DS2?

Consider using DS2 when the application meets one or more of these criteria:

- On the SAS Compute Server, CPU bound processes are good candidates for threaded processing. If your DATA step program is computationally complex, it might benefit from refactoring in DS2. Check the log after the program runs. If the CPU time and real time are about the same, the process is probably CPU bound.
- If your DATA step reads from and writes to tables in a massively parallel processing (MPP) database where the SAS In-Database Code Accelerator is installed, you should try running the process in-database with DS2.
- Programs that read from and write to tables in CAS are also good candidates.
- If you require the precision that the DS2 data types offer or want to leverage your library of DS2 packages to reduce development time, DS2 might well be your choice for coding.

But remember, not every process runs faster in DS2. If your program doesn't meet one or more of these criteria, traditional DATA step might be a better choice.

DS2 provides syntax for three types of programs, written in clearly defined blocks of code.

Data blocks are the core of the DS2 language and are used to manipulate data and create reports.

Thread blocks create and store DS2 threads. Threads are data blocks stored in SAS libraries that can be used for parallel processing.

Package blocks create and store packages of user defined DS2 methods that can be used to add functionality to future data and thread blocks.

PROC DS2 uses RUN-group processing. A RUN-group consists of one or more DS2 action statements followed by a RUN statement. After PROC DS2 is invoked, you can submit multiple RUN-groups without restarting the procedure. The QUIT statement indicates that we have finished submitting DS2 code. For PROC DS2, the RUN and QUIT statements are required syntax. Enterprise Guide and SAS Studio automatically submit RUN and QUIT statements after every SAS program you run, but the SAS Windowing Environment does not.

DS2 PROGRAM BLOCKS

DS2 data blocks are the core of the DS2 language and are used to manipulate data and create reports.

A data block begins with a DATA statement and ends with an ENDDATA. The data block contains the declarative and executable statements required to produce the desired result. All executable statements must be inside of a method block. DS2 methods are named blocks of executable code. The method definition begins with a METHOD statement and ends with an END statement. All executable statements in a DS2 program must be part of a method definition. A combination of built-in system methods and user-defined methods can be used to produce the desired results. Note that user-defined methods can accept parameters, but system methods do not. Let's take a tour of the system methods.

System Methods Control Data Block Flow

INIT

The INIT method is called when the DS2 program begins executing and executes each included statement only once. At the END statement, the INIT method automatically calls the program's RUN method and stops executing.

RUN

The RUN method begins execution when called by the INIT method and executes each included statement sequentially. By default, the RUN method's END statement includes implicit OUTPUT and RETURN statements, causing a record to be written to the result set and returning execution to the top of the RUN method for another iteration. This looping continues until all input data is exhausted, at which point the RUN method calls the TERM method and stops executing.

TERM

The TERM method begins execution when called by the RUN method and executes each included statement sequentially once and only once. At the END statement, the TERM method terminates the DS2 program execution.

Every DS2 data or thread block must include all three system methods to properly function. When a DS2 program is compiled, the compiler notes any undefined system methods, and supplies a NULL method definition, consisting of the appropriate METHOD and END statements. It is possible to write a DS2 data block without explicitly coding any of the system methods and have that program run without error. However, such a program would never actually execute any code.

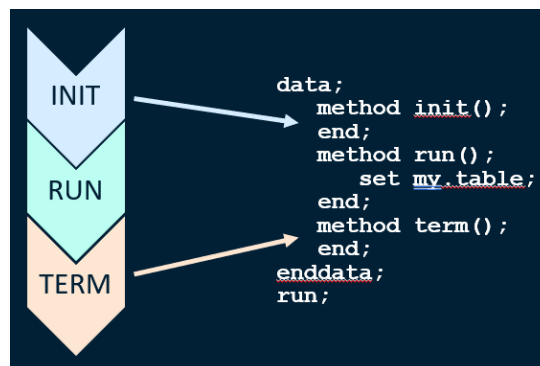


Figure 1 - System methods in a data block

DECLARATION STATEMENTS

Declaration statements must be placed in your code before any executable statements. The section of code between the DATA, PACKAGE or THREAD statement and the first method block is often referred to as the "declaration section". Declarations statements in the declaration section are **global** declarations and affect the entire program block. **Local** declaration statements apply only to the method in which they appear and must appear within the method block before the first executable statement. The DECLARE statement is the only local declaration statement permitted in DS2.

The syntax for the DECLARE statement is:

```
DECLARE|DCL data-type variable-list <HAVING LABEL 'string' | FORMAT | INFORMAT>;
```

The DECLARE statement is used to declare variables. It can be abbreviated as DCL. The data type must be declared first, followed by the name or names of the variables being declared. Optionally, a HAVING clause is used to assign a label, format or informat to the variables of type CHAR or DOUBLE.

Examples:

```
/* Declare three DOUBLE variables formatted dollar12.2*/
dcl double Var1 Var2 Var3 having format dollar12.2;

/* Declare a high-precision fixed point numeric variable */
dcl decimal(35,5) Var1;

/* Declare a fixed-width character variable labeled 'My Text'*/
dcl char(25) Var1 having label 'My Text';
```

In DS2 programs, variables are either explicitly declared using a DECLARE statement or implicitly declared in a SET statement which retrieves column metadata from the input table. Any other variables referenced in a DS2 program are undeclared variables. Undeclared variables are created as either CHAR or DOUBLE based on context. If context isn't available, the default type is DOUBLE. By default, undeclared variables produce warnings in the SAS log.

Many other declaration statements are functionally identical in DATA step and DS2 data blocks, but in DS2 they are all global statements and produce syntax errors if included in a method definition.

DS2 DATA BLOCK STRUCTURE

Let's look at the components of a DS2 data block. The program begins with a DATA statement followed by the declaration section. Method definitions follow, may include local variable declarations, and must include all executable code. The data block ends with an ENDDATA statement. When executing in the Compute Server using PROC DS2, the RUN statement is required to trigger execution.

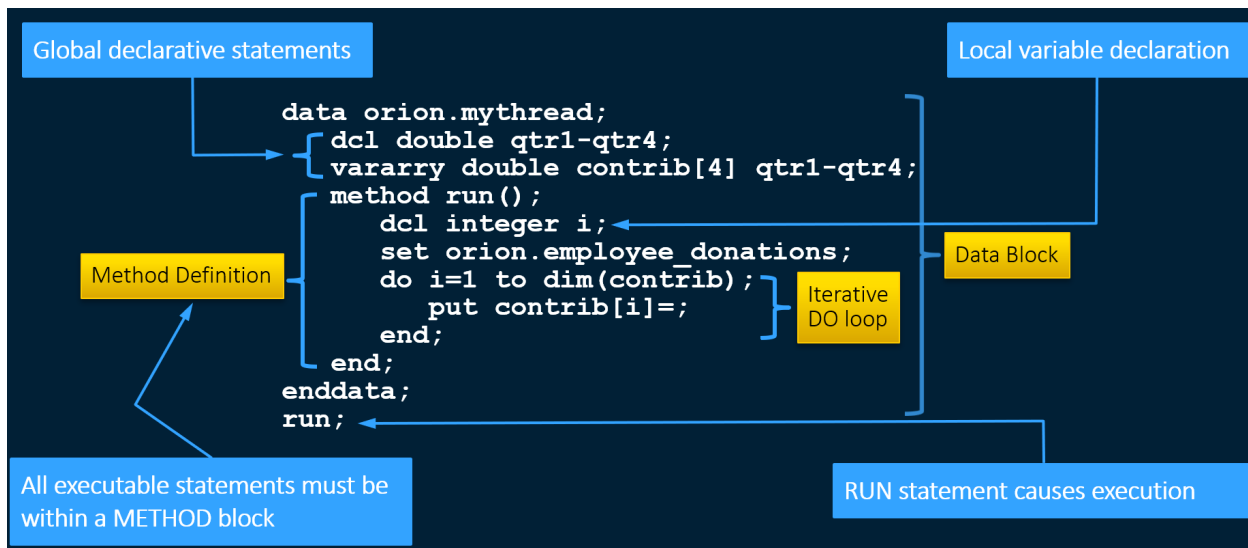


Figure 2 - DS2 data block structure

DS2 PACKAGES

DS2 packages are collections of methods and data elements known as *attributes*. They provide a straightforward way to extend the capabilities of the DS2 language. to securely share proprietary algorithms throughout the organization. SAS ships several pre-defined packages with DS2. If you are familiar with object-oriented programming techniques, DS2 packages can be used like "classes" in other languages, providing many of the benefits associated with object-oriented coding.

Creating a package is simple using a DS2 package block. The program begins with a PACKAGE statement followed by the declaration section, where the attributes are declared. The package methods definitions follow and an ENDPACKAGE statement completes the block. Once again, if you are using PROC DS2 to run your program, the RUN statement is required to trigger execution. Executing a package block merely stores the code – none of the code is executed.

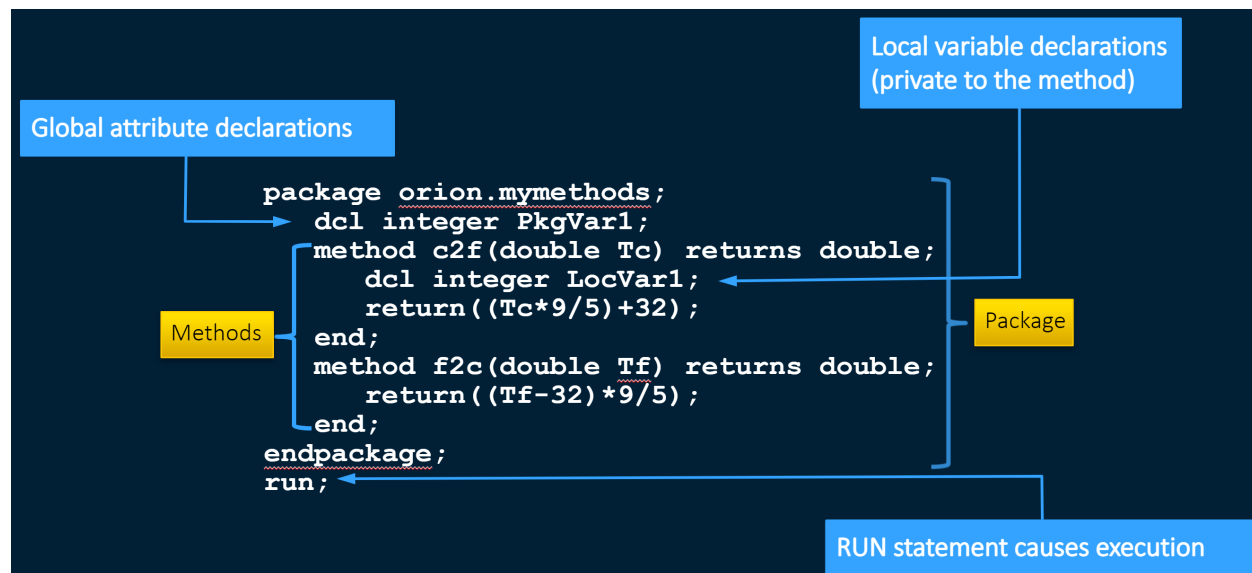


Figure 3 - DS2 package block structure

DS2 THREADS

Threads are DS2 programs stored in SAS libraries that are used to process rows of data in parallel. Thread blocks are remarkably like data blocks and have the same syntax requirements. But threads are also like packages in that they store program code that can be reused in subsequent programs.

A thread block begins with a THREAD statement followed by the declaration section. The method definitions may include local variable declarations and must include all executable code. Like a data block, each thread block requires all three system methods, and the compiler provides a null method block for any system method not explicitly coded. The thread block ends with an ENDTHREAD statement, and the RUN statement triggers execution in PROC DS2. Like a package block, executing a thread block merely stores the code – the stored code does not execute.

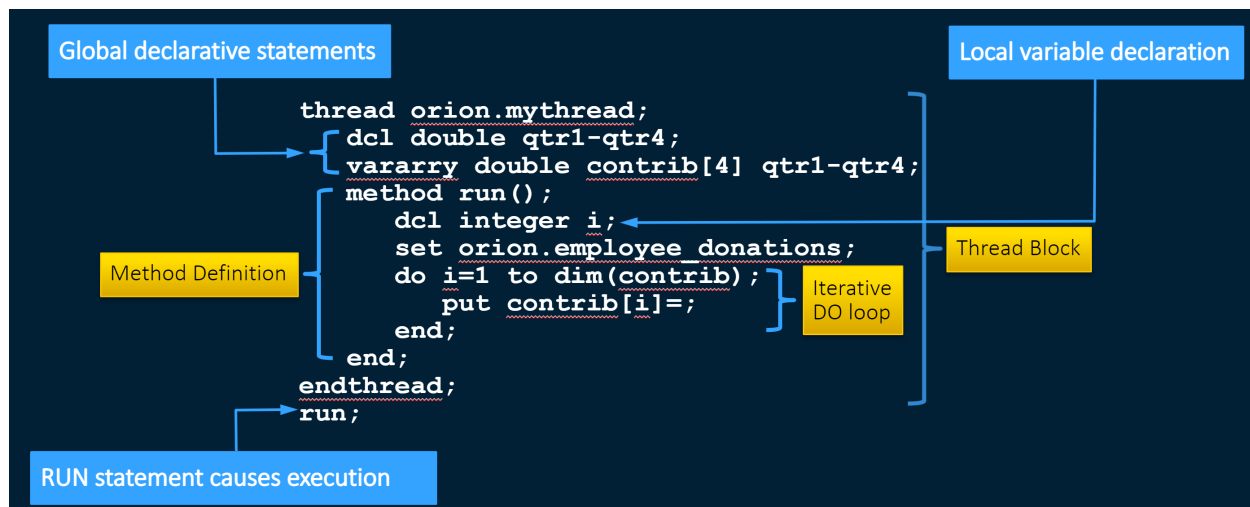


Figure 4 - DS2 thread block structure

USER-DEFINED METHODS

Implementing reusable algorithms is easy in DS2 with user-defined methods. We pass values to a method using parameters. Parameters come in two types: standard and IN_OUT. Standard parameters accept a value expression. Instead of a provided value, IN_OUT parameters require passing in a variable name. The method retrieves data from and can write data back to the variable specified for an IN_OUT parameter, much like a DATA step call routine. It's acceptable to use both types of parameters in a single method definition.

Optionally, a method may also return a value, but only one value, much like a DATA step function. The data type of the value to be returned is specified on the METHOD statement with the RETURNS option, and the actual value is returned using a RETURN statement within the method block. The RETURN statement is the executable statement that returns a value to the calling process. User-defined methods execute only when called and can be executed as many times as desired within a program.

STANDARD PARAMETERS

Standard parameters work much like parameters for SAS functions and serve as input only. Standard parameters may scalar or array values. The term scalar indicates a variable that holds a single value, while array designates a multi-dimensional data structure containing more than one value of the same data type. VARARRAY is also valid for standard method parameters. In the following example, the method **addone** accepts one scalar standard parameter. When the program calls method **addone**, any valid numeric expression can provide the value for the standard parameter, including constants, variables, or mathematical expressions. We provide the expression `B-2`, which resolves to 1, and the value is passed to method **addone**. The method then adds 1 to the value of **X**, and storing the result in local variable **Y**. The RETURN statement delivers the value of internal variable **Y** to the assignment statement calling the method, which assigns the value to variable **A**:

```

dcl int A B;
method addone(int x) returns int;
  dcl int y;
  y=x+1;
  return y;
end;
method init();
  B=3;
  A=addone(B-2);

```

```

end;
method addone(in_out int X);
    x=x+1;
end;

```

IN_OUT PARAMETERS

An IN_OUT parameter will only accept a variable name as input for the method call. Actual values, such as constants or results of expressions, are not valid here. In this example, the **addone** method has one IN_OUT parameter of type integer. Internally, references to **X** will point to the memory location for the variable identifier provided in the IN_OUT parameter. The variable may be a scalar or array, and the data type must match the type specified in the method definition. A variable array created with the VARARRAY statement is not valid as an IN_OUT parameter. This example is a copy of the previous program with the standard parameter changed to an IN_OUT parameter:

```

dcl int A B;
method addone(IN_OUT int x) returns int;
    dcl int y;
    y=x+1;
    return y;
end;
method init();
    B=3;
    A=addone(B-2);
end;

```

Execution generates an error:

```

ERROR: Compilation error.
ERROR: In call of addone: argument 1 is 'in_out'; therefore, it must be an
identifier instead of an
expression.

```

In the following example, the value of global integer A is set to 1 by the expression 3-2. The identifier A is then passed as the IN_OUT parameter when the **addone** method is called. Method **addone** assigns local identifier **X** to the same memory location identified by global identifier **A**. Because the current value of **A** is 1, within method **addone** the value of **X** is also one. When the method adds 1 to **X** and assigns the resulting 2 back to **X**, the value of global variable **A** is also 2. No assignment statement required when calling the method.

PROGRAM BLOCK VARIABLE SCOPE

Variables declared in the global declaration space or implicitly declared by a SET statement are *globally declared* variables and have global scope within the program block. For data and thread blocks, global-scope variables appear in the Program Data Vector (PDV) and are available to all methods in the block.

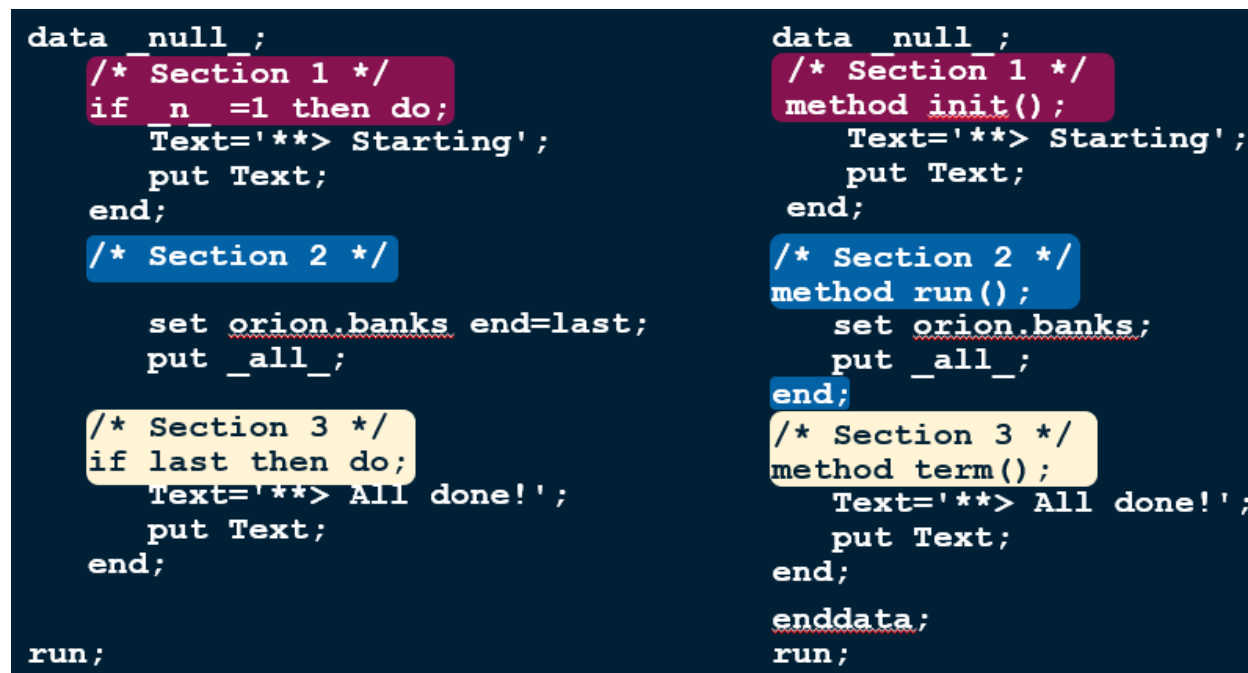
Variables not introduced using a SET or DECLARE statement are undeclared variables. Undeclared variables also have global scope.

Variable declared within a method block are *locally declared* variables and have local scope. Local-scope variables are private to the method in which they were declared and can only be accessed within the method block. Local-scope variables never appear in the PDV. It is convenient and a good practice to declare DO loop index variables local to the method which contains the DO loop code.

CONVERTING A DATA STEP TO DS2

Let's examine how a DATA step might best be converted to a DS2 data block. In figure 5 below, Section 1 of the DATA step code executes only once, during the first iteration of the DATA step. This code is well-suited for the INIT method in a DS2 data block. The code in Section 2 should iterate over the input data, so it is best suited for the RUN method. And finally, the code in section 3 executes only once, just before the DATA step execution terminates, so in DS2, this code is best executed using the TERM method.

Note: The SET statement END= option is not supported in DS2



```
data _null_;
  /* Section 1 */
  if _n_ =1 then do;
    Text='**> Starting';
    put Text;
  end;
  /* Section 2 */
  set orion.banks end=last;
  put _all_;
  /* Section 3 */
  if last then do;
    Text='**> All done!';
    put Text;
  end;
run;
```

```
data _null_;
  /* Section 1 */
  method init();
  Text='**> Starting';
  put Text;
  end;
  /* Section 2 */
  method run();
  set orion.banks;
  put _all_;
  end;
  /* Section 3 */
  method term();
  Text='**> All done!';
  put Text;
  end;
  enddata;
run;
```

Figure 5 - DS2 data block system method selection when converting DATA step code

In practice, it's easier to let PROC DSTODS2 do the preliminary work to convert DATA step code to DS2. PROC DSTODS2 reads a file containing a single SAS DATA step program and creates an output file containing the equivalent DS2 data block. Use the IN= option to specify the fully qualified location of the file containing the DATA step program to be converted. The OUT= parameter specifies the name of the file that will contain the generated DS2 code, and optionally, you can specify the location to which the output file will be written using the OUTDIR= option.

During conversion, any code that can't be converted is included in the output file as a comment. A little cleanup and reformatting are usually necessary to produce a final, production ready DS2 data block.

Example code:

```
proc dstods2 in ="&path/demos/ds203d02a.sas"
             out="ds203d02a_ds2.sas"
             outdir="&path/demos/";
run;
```

Note: The trailing path delimiter is required when specifying the OUTDIR= option.

CONVERTING A DS2 DATA BLOCK TO A THREAD

Now, one of the joys of DS2 processing is the ability to process data in multiple threads, using stored thread blocks executed from a data block. To convert this DS2 data block to a thread:

```
data;
  dcl package my_sas.pkgBankMethods bank();
  dcl double valueInitial valueFinal having format dollar12.2;
  keep Employee_ID, Employee_Name value;;
  dcl int _count;
  keep Employee_ID, Employee_Name value;;
  method run();
    dcl int duration;
    set orion.employees;
    valueInitial=Salary*.1;
    Duration=year(today())-year(employee_hire_date);
    valueFinal=bank.interest(ValueInitial,.04, Duration);
  end;
  method term();
    put '*** Thread' _threadid_ 'of' _nthreads_ 'processed'
        _count_ 'rows on' _hostname_;
  end;
enddata;
```

Change the DATA statement to a THREAD statement and the ENDDATA statement to an ENDTHREAD statement.

```
thread my_sas.thRetirement;
  dcl package my_sas.pkgBankMethods bank();
  dcl double valueInitial valueFinal having format dollar12.2;
  keep Employee_ID, Employee_Name value;;
  dcl int _count;
  keep Employee_ID, Employee_Name value;;
  method run();
    dcl int duration;
    set orion.employees;
    valueInitial=Salary*.1;
    Duration=year(today())-year(employee_hire_date);
    valueFinal=bank.interest(ValueInitial,.04, Duration);
  end;
  method term();
    put '*** Thread' _threadid_ 'of' _nthreads_ 'processed'
        _count_ 'rows on' _hostname_;
  end;
endthread;
```

Running this program stores the thread in the **my_sas** library. Like DS2 packages, threads are stored in tables as encrypted source code.

To use a thread in a data block, first declare an instance of the thread. Here we declare an instance of the stored thread **my_sas.thRetirement** identified as **ret**. The SET FROM statement executes the specified thread instance. While executing, each thread has a separate PDV identical to the data block PDV.

EXECUTING DS2 THREADS IN BASE SAS

In base SAS, the default behavior is to execute a single copy of the thread block, but you can use the `THREADS=` option to specify the number of threads to execute. Executing more threads than the available CPUs can handle, known as over-threading, will negatively affect performance. The `SYSNCPU` macro variable stores the number of CPUs available to the SAS session and provides a good starting point for threading in base SAS. Tweaking this number slightly can sometimes improve performance.

Example Code:

```
data;
  dcl thread my_sas.thRetirement ret;
  method run();
    set from ret;
  end;
enddata;
run;
```

EXECUTING DS2 THREADS IN CAS ON SAS VIYA

To run your DS2 code in CAS, all assets required to execute the program must be in CAS, including source tables, result tables, threads, and packages. Generally, you can just copy your packages to CAS with a utility like PROC CASUTIL. Threads originally executed on the Compute Server read from tables available to the Compute Server. Modify the thread block that created the thread to read from a caslib table and save the thread to a caslib. The data block that executes the threads must be modified to use the thread stored in CAS and to write any result sets to caslib tables. In CAS, default behavior is to run DS2 threads distributed to all available CAS nodes, so to maximize threading, delete the `THREADS=` option from the SET FROM statement.

When running the data block from PROC DS2 on the Compute Server, use the PROC DS2 `SESSREF=` option to indicate the CAS session which should run the DS2 code and submit the program.

RECOMMENDED READING

- [A Brief Introduction to DS2.zip](#) (SAS program files used to create the presentation)
- [SAS® DS2 Language Reference](#)
- [SAS® DS2 Programmer's Guide](#)
- *Mastering the SAS® DS2 Procedure: Advanced Data Wrangling Techniques, Second Edition*

CONTACT INFORMATION

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

Mark Jordan
Email: Mark.Jordan@sas.com
X: @SASJedi
Author Page: <http://support.sas.com/jordan>

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.