THE POWER TO KNOW.

SAS® Viya™ 3.1: DS2 Programmer's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Viya™ 3.1: DS2 Programmer's Guide*. Cary, NC: SAS Institute Inc.

SAS® Viya™ 3.1: DS2 Programmer's Guide

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2016

3.1-P1:ds2pg

Contents

Part 1

Introduction

Chapter 1

Introduction to the DS2 Language

Introduction to the DS2 Language

DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with SAS Visual Data Mining and Machine Learning and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages.

In addition, DATA step logic can be transformed to run in the SAS Embedded Process where DS2 is supported and the DATA step is not.

The DS2 procedure enables you to submit DS2 language statements from a SAS Studio session. For more information about PROC DS2, see *SAS Viya Data Management and Utility Procedures Guide*. In addition, you can use the runDS2 action to prepare and execute a DS2 program. For more information about the runDS2 action, see *SAS Cloud Analytic Services: System Programming Guide*.

Note: Because the DS2 language can be used with many data sources, the terms row, column, and table are used to describe the data elements. When you compare this to SAS DATA step terminology, a row corresponds to an observation, a column corresponds to a variable, and a table corresponds to a data set.

Running DS2 Programs

You can submit DS2 programs in one of the following ways.

- In SAS Studio using the DS2 procedure. The DS2 procedure can be used to run DS2 code in Viya or in the CAS server. A single PROC DS2 step can contain several DS2 programs.

  For more information, see "DS2" in *SAS Viya Data Management and Utility Procedures Guide*.

- In SAS Studio using the runDS2 action in the CAS server. The runDS2 action is used in conjunction with the CAS procedure.

  Note: Unless you are using Python or Lua, it is recommended that you use PROC DS2 to submit DS2 code to the CAS server.

  For more information, see "Run program" in *SAS Cloud Analytic Services: System Programming Guide* and Chapter 18, "DS2 in CAS," on page 191.

Supported Data Sources

DS2 can access the following data sources:

- Hadoop (Hive)
- Impala
- ODBC-compliant databases (such as Microsoft SQL Server)
- Oracle
- PC Files
- PostgreSQL
- SAS data sets
- Teradata

Note: The following data sources are not supported:

- Aster
- DB2 for UNIX and PC operating environments
- Greenplum
- Informix
- Memory Data Store (MDS)
- MySQL
- Netezza
- OLEDB
- SAP (Read-only)
- SAP HANA
- SAS Scalable Performance Data Engine (SPD Engine) data set
- SQL Server
- Sybase
- Sybase IQ
- Vertica

Intended Audience

The information in this document is intended for the following users who perform in these roles:

- **Application developers** who write the client applications. They write applications that create tables, bulk load tables, manipulate tables, and query data.
- **Database administrators** who design and implement the client/server environment. They administer the data by designing the databases and setting up the data source metadata. That is, database administrators build the data model.
- **SAS programmers** who want or need to take advantage of the features of the DS2 language.

When to Use DS2

You do not necessarily have to convert your DATA step programs to DS2. Typically, DS2 programs are written for applications that carry out the following actions:

- 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
- take advantage of threaded processing

Syntax Conventions for the DS2 Language

Typographical Conventions

Type styles have special meanings when used in the documentation of the DS2 language syntax.

UPPERCASE BOLD
identifies DS2 keywords such the names of statements and functions (for example, PUT).

UPPERCASE ROMAN
identifies arguments and values that are literals (for example, FROM).

italic
identifies arguments or values that you supply. Items in italic can represent user-supplied values that are either one of the following.

- nonliteral values assigned to an argument (for example, ALTER=*alter-password*).
- nonliteral arguments (for example, KEEP=(*column-list*).

If more than one of an item in italics can be used, the items are expressed as *item* [, …*item*].

monospace
identifies examples of SAS code.

Syntax Conventions

SAS Viya: DS2 Language Reference uses the Backus-Naur Form (BNF), specifically the same syntax notation used by Jim Melton in *SQL:1999 Understanding Relational Language Components*.

The main difference between traditional SAS syntax and the syntax that is used in the DS2 language reference documentation is in how optional syntax arguments are displayed. In traditional SAS syntax, angle brackets (< >) are used to denote optional syntax. In DS2 language syntax, square brackets ([ ]) are used to denote optional syntax and angle brackets are used to denote non-terminal components.

The following symbols are used in the DS2 language syntax.

::=
This symbol can be interpreted as “consists of” or “is defined as”.

<>
Angle brackets identify a non-terminal component (that is, a syntax component that can be further resolved into lower level syntax grammar).

[ ]
Square brackets identify optional arguments. Any argument that is not enclosed in square brackets is a required argument. Do not enter square brackets unless they are preceded by a backward slash (\), which denotes that they are literal.

{ }
Braces provide a method to distinguish required multi-word arguments. Do not enter braces unless they are preceded by a backward slash (\), which denotes that they are literal.

|
A vertical bar indicates that you can choose one value from a group. Values that are separated by bars are mutually exclusive.

…
An ellipsis indicates that the argument or group of arguments that follow the ellipsis can be repeated any number of times. If the ellipsis and the following arguments are enclosed in square brackets, they are optional.

\
A backward slash indicates that the next character is a literal.

The following examples illustrate the syntax conventions that are described in this section. These examples contain selected syntax elements, not the complete syntax.

```
1 SET 2 <table-reference> […  [<table-reference>] [INDSNAME=variable];
   [3 BY [DESCENDING] 4 column [5 ...[DESCENDING] column];
   6 <table-reference>::=
   {table (table-options)} 7 | 8 \{sql-text 8 \}
```

1 SET is in uppercase bold because it is the name of the statement.

2 <table-reference> is in angle brackets because it is a non-terminal argument that is further resolved into lower level syntax grammar. You must supply at least one <table-reference>.

3 BY and DESCENDING are in uppercase roman because they are literal arguments. DESCENDING is in square brackets because it is an optional argument.

4 *column* is in italics because it is an argument that you can supply.

5 The square brackets and ellipsis around the second instance of *column* indicate that you can repeat this argument any number of times as long as the arguments are separated by commas.

6 The <table-reference>::= non-terminal argument syntax is read as follows: A <table-reference> consists of a table name and table options or embedded SQL text.

7 The vertical bar (|) indicates you can supply either *table* [*table-options*] or *sql-text*, but not both.

8 The backslash (\) before the braces around *sql-text* indicate that those braces are literals and must be entered.

Part 2

DS2 Concepts

Chapter 2

DS2 Programming Semantics

Basic DS2 Program Syntax

A DS2 program consists of a list of declarations followed by a list of method statements. Here is an example of a simple declare list:

```
declare int x;
declare double d;
```

Here is an example of a simple method statement list:

```
method init();
end;

method run();
end;

method term();
end;
```

Combining the two lists creates a simple DS2 program.

```
declare int x;
declare double d;

method init();
end;

method run();
end;
```

```
method term();
end;
```

Although a DS2 program is typically more complex, this simple program contains several syntactic elements:

Keywords:

- DECLARE
- DOUBLE
- METHOD
- END

Identifiers:

- x
- d
- INIT
- RUN
- TERM

Lexical Separators:

- (
-)
- ;

The program illustrates how to declare an identifier, either in a DECLARE statement or in a METHOD statement. It also illustrates how the high-level structure of a DS2 program consists of a sequence of variable declarations followed by a sequence of METHOD statements. The next section explains what these terms mean in DS2.

Basic DS2 Program Semantics

Variable Declaration Statements

A variable declaration allocates memory space and identifies that memory with an identifier, called the variable name. The declaration, either explicitly or implicitly, allocates memory for the variable and designates the type of data that can be saved at that memory location. In DS2, you declare variables by using the DECLARE statement. A DECLARE statement performs the following actions:

- assigns an identifier to a memory location. That identifier becomes the variable name.
- designates the type of data that the variable can hold.
- allocates a specified amount of memory to the variable.

More than one variable can be declared in one DECLARE statement. For more information, see the "DECLARE Statement" in *SAS Viya: DS2 Language Reference*.

Methods

Methods are basic program execution units. In DS2, the method structure is used to group related program statements in one syntactically identifiable location. The group of statements in the method can then be easily invoked, or executed, multiple times.

DS2 methods are similar to functions, in languages such as C, and methods in Java. In addition, the label target of a LINK statement is functionally similar to a rudimentary method.

A method defines a scoping block. Therefore, any parameters and any variable declarations in the method body are local to the method.

A **type signature**, or simply **signature**, is defined to be the ordered list of the method's parameter types. If any two method definitions have the same name, but different type signatures, the method is **overloaded**. An error occurs if two method definitions have the same name and same type signature.

Note: You cannot overload a method based on CHAR and NCHAR data types alone if session encoding requires multiple bytes. If a session encoding requires multiple bytes per character, for example, UTF-8, then CHAR and NCHAR are identical types and both use NCHAR. Consequently, the two method definitions would be seen as the same.

There are two types of methods in DS2: *system* methods, and *user-defined* methods.

A DS2 program can contain the following three system methods:

```
method init();
   end;
method run();
   end;
method term();
   end;
```

Every DS2 program will contain, either implicity or explicitly, these three methods. If you do not define any one of these methods in a DS2 program, the DS2 compiler will create an empty version of it (like those above). These methods are meant to provide a more structured framework than the SAS DATA Step implicit loop concept. The entire DATA Step program is included in the implicit loop. In DS2, the implicit loop is represented by the RUN method, with the INIT and TERM methods providing initialization and finalization code, respectively.

When a DS2 program executes, here are the results:

1. The INIT method runs. Any initializations take place.
2. Variables in the program data vector which have not been retained will be set to the appropriate missing values. For more information, see the “RETAIN Statement” in *SAS Viya: DS2 Language Reference*.
3. The RUN method executes.
4. Execution control then depends on the status of any input statement in the RUN method. Currently, the only input statement in DS2 is the SET statement. If the RUN method meets one of these conditions, then processing proceeds to Step 5. Otherwise, processing proceeds to Step 2 so that the RUN method can execute again:
   - No input statements
   - An input statement that has completed execution

5. The TERM method executes, and any final statements execute.

The INIT, RUN, and TERM methods must be defined without any parameters and without a return value. If you specify a parameter for the INIT, RUN, or TERM methods, an error will occur.

If you do not specify an OUTPUT statement in the DS2 program, the DS2 compiler will provide one with no parameters that executes at the end of the RUN method.

If you attempt to call the INIT, RUN, or TERM method directly from a DS2 program, an error will occur.

User-defined methods can be created by enclosing statements that you would like executed one or more times within METHOD and END statements. For more information about user-defined methods, see the "METHOD Statement" in *SAS Viya: DS2 Language Reference*.

Note: When using PROC DS2, DS2 programs are delimited by RUN statements. If additional DS2 code is found after a RUN statement, this code composes a new, distinct DS2 program from the DS2 program before the previous RUN statement.

Scope of DS2 Identifiers

Programming Blocks

A programming block is a section of code that begins and ends with and ordered pair of keywords. The following keywords create programming blocks:

- DATA...ENDDATA
- PACKAGE...ENDPACKAGE
- THREAD...ENDTHREAD
- DO...END
- METHOD...END

In this documentation, these terms are used for programming blocks.

- A data programming block or **data program** refers to code bounded by DATA… ENDDATA statements.
- A package programming block or **package** refers to the stored library of variables and methods bounded by PACKAGE…ENDPACKAGE statements. The variables and methods of a package can be used by DS2 programs, threads, or other packages.
- A thread programming block, or **thread program**, refers to a stored program that is bounded by the THREAD…ENDTHREAD statements. The thread program can be called by the SET FROM statement in a DS2 program or package.
- A DO programming block, or **DO loop**, refers to a subblock of programming statements that are bounded by the DO and END statements.
- A method programming block or **method block** refers to a subblock of programming statements that are bounded by the METHOD and END statements.

Some blocks can be nested. In this example, there is one data program, defined by the DATA and ENDDATA statements, and three nested method blocks, defined by the three method statements.

```
data _null_;
   declare int x;

   method init();
      declare double d;
   end;

   method run();
   end;

   method term();
   end;
enddata;
```

A variable declared in the outermost programming block is called a **global** variable, or a variable having **global scope**. A variable declared in any nested block is called a **local** variable, or a variable having scope that is local to that block. DS2 also assigns global scope to undeclared variables. In the preceding example, X is a global variable, and D is a variable that is local to the nested INIT method.

Note: A DS2 program can have multiple subprograms followed by an optional data program. The following restrictions apply:

- There can be only one data program and the data program must be the last subprogram.
- The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

Variable Lookup

When a variable is referenced, DS2 will always search for the variable's declaration beginning in the block of the reference. Then, if it is not found there, it will search successively in any outer containing blocks or program. In this example, any reference to X in the INIT method will refer to the global declaration of X.

```
declare int x;

method init();
end;
```

Because methods are blocks, they can contain declarations themselves. In this example, any reference to X in the INIT method will refer to the local declaration of X, but any reference to X in the RUN method will refer to the global declaration of X.

```
declare int x;

method init();
   declare int x;
end;

method run();
end;
```

Definition of Scope

Scope can be considered an attribute of identifiers. Identifiers can refer to a number of program entities: method names, functions, data names, labels, or program variables. This section uses program variables as examples, but any identifier is subject to scoping rules.

Scope describes where in a program a variable can be accessed. Global variables have global scope and are accessible from anywhere in the program. Local variables have local scope and are accessible only from within the program or block in which the variable was declared.

In DS2, a variable is accessible only as long as program execution is taking place within the scope of the variable. That is, the values of variables are accessible only when a statement in the scope of the variable is actively executing.

In the following example, the variable X is in scope only while the INIT method is executing. Neither the RUN or the TERM methods can refer to it.

```
data;
   declare int x;    /* global x in global scope */

   method init();
      x = 5;          /* global x assigned 5 */
   end;

   method run();
   end;

   method term();
   end;

enddata;
```

Each variable in any given scope must have a unique name, but variables in different scopes can have the same name. When scopes are nested, if a variable in an outer scope has the same name as a variable in an inner scope, the variable within the outer scope is hidden by the variable within the inner scope. For example, in the following program two different variables share the same name, X. Global variable X has global scope, and local variable X has local scope. Within the local scope of method INIT, local variable X hides global variable X. Therefore, the assignment statement assigns 5 to local variable X.

```
data;
    declare int x;    /* global x in global scope */

    method init();
       declare int x;  /* local x in local scope */

       x = 5;          /* local x assigned 5 */
    end;

    method run();
    end;

    method term();
    end;
```

```
enddata;
```

Variable Lifetime

The lifetime of a variable is the time during which the variable exists. Global variables exist for the duration of the program. Local variables exist for the duration of the block in which the variable was declared. The value of a global variable will be set to a missing or null value before entry into the RUN method unless that global variable appears within a RETAIN statement in the current program block.

In the following example, the variable X exists only while the INIT method is executing and the variable Y exists for the duration of the data program.

```
data;
   declare double y;

   method init();
      declare double x;
   end;

enddata;
```

During a variable's lifetime, it can be overshadowed by a locally declared variable of the same name, as in this example:

```
declare int x;

method init();
   declare int x;
end;

method run();
end;
```

Although the global variable X has lifetime for the entire program, it is not directly accessible from the INIT method because of the local declaration of X in the INIT method.

Chapter 3

DS2 Variables

Introduction to DS2 Variables

The properties of DS2 program variables are that they have a name, a scope, and a data type.

A name, or identifier, is one or more tokens, or symbols, that is given to a variable. Names are discussed in Chapter 6, “DS2 Identifiers,” on page 39.

Variables can have either global or local scope depending on where the variable is declared. For more information, see “Variable Scope” on page 25.

Variable data types are assigned either implicitly or explicitly depending on how they are declared. For more information, see “Variable Declaration” on page 20.

Note: The term “data type” includes any data type attributes such as precision, character set encoding, and length. For complete information about data types, see Chapter 5, “DS2 Data Types,” on page 33.

Variable Declaration

There are three ways to declare a variable and its data type:

- 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.

  For more information, see "Variable Scope" on page 25 and "DECLARE Statement" in *SAS Viya: DS2 Language Reference*.

- 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.

  For more information, see "Reading Data Using the SET Statement" on page 150 and "SET Statement" in *SAS Viya: DS2 Language Reference*.

- 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.

  The data type for an undeclared variable on the right side of an assignment statement is DOUBLE.

  Note: A best practice is to declare every variable. By doing so, you can avoid data type mismatches among data sources.

  To control how DS2 handles an undeclared variable, you can use the DS2SCOND system option or the SCOND option on the DS2 procedure:

Table 3.1 *Settings to Control Variable Declaration*

DS2SCOND/SCOND Setting	Effect on Variable Declaration
WARNING	Declaration by assignment occurs. Warning messages are written to the SAS log. This is the default behavior.
NONE	Declaration by assignment occurs. No messages are written to the SAS log.
NOTE	Declaration by assignment occurs. A note is written to the SAS log.

DS2SCOND/SCOND Setting	Effect on Variable Declaration
ERROR	Declaration by assignment does not occur. An error message is written to the SAS log. This is also known as variable declaration strict mode.

For more information, see the "DS2SCOND= System Option" in *SAS Viya: DS2 Language Reference* and "DS2" in *SAS Viya Data Management and Utility Procedures Guide*.

The following example shows how to define variables explicitly and implicitly, or by assignment. FIRSTNAME and LASTNAME are declared as variables with the data type of CHAR(20). PNUM is declared by assignment. DS2 assigns PNUM a data type based on the value on the right side of the assignment, which is CHAR(11):

```
data phonenums;
   dcl char(20) firstName lastname;
   method init();
      firstName='Sam';
      lastName='Alesski';
      pnum = '19192223454';
      … more DS2 statements …
   end;
enddata;
```

Variable Lists

Overview of DS2 Variable Lists

A DS2 variable list is a collection of DS2 variables. Many DS2 statements and table options use variable lists for specification of sets of variables. For example, the KEEP, DROP, and VARARRAY statements can use variable lists for the specification of the set of variables to keep, drop, or reference. Here are some examples:

```
keep name address city state zip phone;
 drop rep1-rep5;
 vararray int grades[*] assignment: quiz: exam:;
```

A variable list can also provide a convenient mechanism for specifying variables of interest to a DS2 method or a package.

DS2 supports the following forms of variable lists.

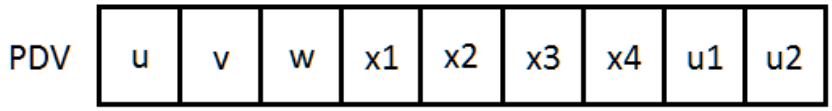
- name variable list
- numbered range variable list
- name range variable list
- name prefix variable list
- type variable list
- special name variable list

For more information, see "Types of Variable Lists" on page 22.

The different forms of variable lists can be mixed within a single variable list. For example, the following is a valid variable list.

```
u x1-x3 u:
```

Assuming the program data vector illustrated below, the above variable list would expand to u x1 x2 x3 u u1 u2. Note that a single variable can be referenced multiple times in a variable list expansion.



Types of Variable Lists

Name Variable Lists

A name variable list is simply a list of variable names.

```
location date pressure temperature
```

Numbered Range Variable Lists

A numbered range variable list expands to reference global variables with a specified prefix and a numeric suffix ranging between two specified numbers. The numbered range variable list has the following syntax.

*prefixn*1-*prefixn*2

n1 and *n2* represent the beginning and ending of the range, inclusive. Variable names are constructed during the numbered range list expansion by concatenating the prefix with each number in the numbered range.

For example, the numbered range variable list **`x1-x5`** expands to **`x1 x2 x3 x4 x5`**. The numbered range variable list **`score10-score5`** expands to **`score10 score9 score8 score7 score6 score5`**.

Name Range Variable Lists

A name range variable list expands to reference all global variables whose variable definition occurred between the definition of two specified variables, inclusive. The name range variable list has the following syntax.

*var*1--*var*2

`sales_jan--sales_mar`is an example.

DS2 maintains a seen list of defined variables. As variables are defined, the variables are added to the seen list in their order of definition. For example, consider the following statements.

```
declare double reg1_id reg2_id reg1_rev reg2_rev;
declare double reg1_exp reg2_exp total_rev total_exp;
keep reg1_rev--reg2_exp;
```

The seen list after executing the DECLARE statements would be **`reg1_id reg2_id reg1_rev reg2_rev reg1_exp reg2_exp total_rev total_exp`**. Therefore, the variable list **`reg1_rev--reg2_exp`** in the KEEP statement expands to **`reg1_rev reg2_rev reg1_exp reg2_exp`**.

Name range variable lists rely on the order of variable definition, that is, **x--a** includes all variables in order of variable definition from **x** to **a** inclusive.

Name Prefix Variable Lists

A name prefix variable list expands to reference all global variables that begin with a specified prefix. The name prefix variable list has the following syntax.

prefix:

An example of a name prefix variable list is **sales:**, which expands to all variables whose names begin with "sales", such as **sales_jan**, **sales_feb**, and **sales_mar**.

Type Variable Lists

A type variable list expands to reference all global variables of a specified type. The type variable list has the following syntax.

data-type:

An example of a type variable list is **smallint int**, which expands to all variables of type SMALLINT or INT. The following types are supported by type variable lists.

- TINYINT
- SMALLINT
- INTEGER
- BIGINT
- REAL
- FLOAT (matches DOUBLE and FLOAT)
- DOUBLE (matches DOUBLE and FLOAT)
- BINARY (matches BINARY and VARBINARY)
- CHAR (matches CHAR, VARCHAR, and CHARACTER)
- NCHAR (matches NCHAR and NVARCHAR)
- CHARACTER (matches CHAR, VARCHAR, and CHARACTER)
- DATE
- TIME
- TIMESTAMP

Special Name Variable Lists

A special name variable list expands to reference a specific group of Read-only, global variables. The _ALL_ variable is supported. _ALL_ references all global variables in the DS2 program.

For more information, see "Predefined DS2 Variables" on page 26.

Expansion of Variable Lists

Variable lists expand to reference global scalar variables that match the variable list type. Local variables are never included in a variable list expansion.

In the following example, variable **x** is a global variable and variable **y** is local to method INIT. The **x:** name prefix variable list in the KEEP statement expands to

reference only variable **x1**. Local variable **x2** is not included in the variable list expansion. Therefore, variable **x2** is not written to the table **example**.

```
data example;
   declare double x1;
   keep x:;

   method init();
      declare double x2;
   end;
enddata;
run;
```

Variable list expansion considers all global variables regardless of where the variable is defined in the program. In the following example, _ALL_ in the KEEP statement expands to reference global variables **x1**, **x2**, and **x3**. Therefore, variables **x1**, **x2**, and **x3** are written to table **example**.

```
data example;
   declare double x1;
   keep [_all_];
   declare double x2;

   method init();
      x3=17;
   end;
enddata;
run;
```

Creating Named Variable Lists

The VARLIST statement creates a named variable list that can be used in multiple DS2 statements. Here is an example.

```
varlist vars [x1-x5 u v w];

method run();
   compute(vars);
   pkg.doStuff(vars);
end;
```

Note that the VARLIST statement is limited to the global scope of the DS2 package or program. The VARLIST statement cannot be used to create a local variable list.

For more information, see the "VARLIST Statement" in *SAS Viya: DS2 Language Reference*.

Unnamed Variable Lists

DS2 provides a mechanism for specification of an anonymous, or unnamed, variable list as part of an expression of another statement. The unnamed variable list has the following syntax.

[*variable-list*]

An example of an unnamed variable list is **compute([x y z]);**.

In this method expression, an unnamed variable list referencing variables **x**, **y**, and **z** is created and passed as the single argument to the **compute** method.

If unnamed variable list syntax is used in an expression, then an unnamed variable list is created in the global scope referencing the variables in *variable-list*. An unnamed variable list is inaccessible from statements other than the statement in which the unnamed variable list was specified. Therefore, it cannot be reused in other expressions.

Passing Variable List Arguments

A DS2 variable list can be passed as an argument to a DS2 method, to a DS2 function, or to a method of a user-defined package. DS2 variable lists are always passed by reference and cannot be passed by value. A DS2 variable list argument can either be a named variable list that is created with a VARLIST statement, or it can be an anonymous variable list. Here is an example.

```
varlist scores [assignment: quiz: exam:];

method init();
   myMeans.by([gender]);
   myMeans.var(scores);
end;
```

Use the following syntax to specify a variable list parameter for a DS2 method.

VARLIST *parameter-name*

Here is an example.

```
method m(varlist list1, varlist list2);
     ...
  end;
```

Variable Scope

Global Variables

A variable with global scope, a global variable, is declared in one of three ways: in the outermost programming block of a DS2 program, using a DECLARE statement, implicitly declared inside a programming block using a SET statement, or implicitly declared inside a programming block by using an undeclared variable. Variables with global scope can be accessed from anywhere in the program and exist for the duration of the program. Global variables can be used in a THIS expression in any program block. For information about declaring DS2 variables, see “Variable Declaration” on page 20. For information about using the THIS expression, see “THIS Expression” on page 63.

Local Variables

A variable with local scope, a local variable, is declared within an inner programming block, such as a method or package, by using a DECLARE statement. Variables with local scope are known only within the inner programming block where they are declared. For more information, see “Programming Blocks” on page 14.

Global and Local Variables in DS2 Output

Only global variables, by default, are included in the output. Local variables that are used for program loops and indexes do not need to be explicitly dropped from the output. Local variables are always created at the start of a method invocation, such as an iteration of the implicit loop of the RUN method, and are destroyed at the end of each invocation. Therefore, it is not recommended to use local variables as accumulator variables in the RUN method.

All global variables are named in the program data vector (PDV). The PDV is the set of values that are written to an output table when DS2 writes a row.

Example of Global and Local Variables

The following program shows both global (A, B, and TOTAL) and local variables (C):

```
data;
   dcl int a; 1 /* A is a global variable */
   method init();
      dcl int c; 2 /* C is a local variable  */
      a = 1; 3
      b = 2; 4 /* B is undeclared so it is global */
      c = a + b;
      this.total = a + b + c; 5
   end;
enddata;
run;
```

1 A is a global variable because it is declared in the outermost DS2 program.

2 C is a local variable because it is declared inside the method block, METHOD INIT ().

3 Because A is a global variable, it can be referenced within the method block, METHOD INIT().

4 Because B is not declared in METHOD INIT(), it defaults to being a global variable. DS2 assigns B a data type of DOUBLE. B appears in the PDV and the output table.

5 THIS.TOTAL simultaneously declares the variable TOTAL as a global variable with the data type of DOUBLE and assigns a value to it based on the values of A, B, and C.

Predefined DS2 Variables

Predefined Method Variables

Predefined variables are variables that are automatically declared within a method block and discarded when the method is complete. The values of predefined variables are retained from one iteration of a RUN method to the next. These variables are added to the program data vector but are not written to the table being created. Predefined variables are temporary and are not saved with your data. Predefined variables are

available in the method block, and you can use them just like any variable that you declare yourself.

Two predefined variables are created:

_N
: is initially set to 0 by the INIT method. Each time the RUN method executes, the value increments by 1. The data type for _N is BIGINT. This is a Read-only variable; you cannot assign a value to _N.

_N_
: is initially set to 1 by the INIT method. Each time the RUN method executes, the value increments by 1. That is, the first time the RUN method executes, the value of _N_ is 1. The value indicates the number of times that the program has looped through the method. The variable can be used to count rows, but it is a good row counter only when one record is read per iteration. The data type for _N_ is BIGINT. You can assign a numeric value to _N_, but in the next iteration of the RUN method, the value reverts to the value assigned by the program

Although predefined variables are not output variables, you can use the PUT statement with the _ALL_ argument to print the values of predefined variables to the SAS log.

The following simple DS2 program illustrates using the PUT _ALL_ statement to print the values of the _N and _N_ variables to the SAS log.

```
data inp /overwrite=yes;
   dcl double a;
   method init();
      a = 1; output;
   end;
enddata; run;

data;
   method init();
     put 'init' _n=; put _ALL_;
   end;
   method run();
     set inp;
     put 'run' _n=; put _ALL_;
   end;
   method term();
      put 'term' _n=; put _ALL_;
   end;
enddata; run;
```

Log 3.1 *SAS Log with _N and _N_ Variable Values*

```
   .
   .
   .
NOTE: Execution succeeded. One row affected.
init _n=0
a=. _n_=1
run _n=1
a=1 _n_=1
term _n=2
a=1 _n_=2
```

DS2 Thread Variables

The following automatic variables are used for subsetting a problem across DS2 threads. These automatic variables are also useful for providing context when you are debugging with PUT statements.

- _HOSTNAME_
- _NTHREADS_
- _THREADID_

For more information, see "Automatic Variables That Are Useful in DS2 Threading" on page 146.

Chapter 4

DS2 Constants

Definition of a Constant

A constant is a number, character string, binary number, date, time, or timestamp that indicates a fixed value. Here are some examples of DS2 constants:

```
107
'Trends in Business'
date '2008-01-01'
b'10011001'
x'FFE3546F'
```

Numeric Constants

A numeric constant is a negative or positive numeric value that is either an integer constant or a fractional constant.

integer constant
: is a numeric value without a fractional component, that is, a whole number. An integer constant value is stored as an exact numeric. An integer constant without the N or n suffix is a BIGINT, INTEGER, SMALLINT, or TINYINT data type value. An integer constant with the N or n suffix is a DECIMAL or NUMERIC data type value. An integer constant has the following forms where *integer* is a sequence of one or more digits, 0 through 9:

  integer

  *integer***N**|**n**

The following values are valid integer constants:

```
0
124
124N
+124n
-124n
```

fractional constant
is a numeric value that has a fractional or decimal component. A fractional constant value is stored as either an exact numeric or an approximate numeric. A fractional constant without the N or n suffix is a REAL or DOUBLE data type value that is stored as an approximate numeric. A fractional constant with the N or n suffix is a DECIMAL or NUMERIC data type value that is stored as an exact numeric up to the maximum precision of the data type. Approximate fractional constants support standard notation or scientific (E) notation. Exact fractional constants support only standard notation. A fractional constant has the following forms where *integer*, *fraction*, and *exponent* is a sequence of one or more digits, 0 through 9:

integer.

integer.**N**|**n**

*integer***.***fraction*

.*fraction*

*integer***.***fraction***E**|**e**[+|-]*exponent*

*integer***E**|**e**[+|-]*exponent*

.*fraction***E**|**e**[+|-]*exponent*

*integer***.***fraction***N**|**n**

.*fraction***N**|**n**

The following values are valid fractional constants:

```
0.
124.0
+124.0
-57.33
.5
1E100
99.99e4
-.33e-2
-57.33n
12345678901234567890123456789O.12N
```

Character Constants

A character constant is a sequence of characters enclosed in single quotation marks and can be written using the following formats:

'*character-string*'

n'*character-string*'

For character strings that contain national characters, use the NCHAR constant, **n**'*character-string*'. National characters can take multiple bytes of storage.

Character and NCHAR constants can include a newline for character constants that span multiple lines.

Note: A sequence of characters enclosed in double quotation marks is not a character constant, it is an identifier. For more information, see "Delimited Identifiers" on page 40.

The following character constants are valid:

```
'PHONE'
n'STÄDTE'
'Phone
Number'  /* constant that spans more than one line */
```

Note: To have a single quotation mark(') or apostrophe (') in a string, use an additional quotation mark. Here is an example.

```
'Isn''t life beautiful'
```

Binary Constants

Binary constants can be written as either a bit string or as a hexadecimal string using the following formats:

b'*bit-string*'

x'*hexadecimal-string*'

bit-string is a sequence of binary digits, 0 and 1.

hexadecimal-string is a string of hexadecimal characters, 0 - 9 and A - F.

Both *bit-string* and *hexadecimal-string* must be enclosed in single quotation marks.

DS2 converts both the bit constant and the hexadecimal constant to their binary equivalents for use in the DS2 program. Binary and hexadecimal constants are padded with zeros to a byte boundary.

The following character constants are valid:

```
b'11100011'
x'FF143E99'
```

Date and Time Constants

Date and time constants can be written as date, time, or timestamp strings using the following formats:

date '*yyyy-mm-dd*'

time '*hh*:*mm*:*ss*'

time '*hh*:*mm*:*ss*.*fraction*'

timestamp '*yyyy-mm-dd hh*:*mm*:*ss*'

timestamp '*yyyy-mm-dd hh*:*mm*:*ss*.*fraction*'

yyyy is a four-digit year.

mm is a two-digit month.

dd is a two-digit day.

hh is a two-digit hour.

mm is a two-digit minute.

ss is a two-digit second.

fraction is a sequence of numbers that represents a fraction of a second.

All date and time constants must be enclosed in single quotation marks. Year, month, and day must be separated by a hyphen. Hours, minutes, and seconds must be separated by colons. Seconds and fraction of a second must be separated by a period. In the timestamp constant, a space separates the date from the time.

The length of a date constant must be ten characters, which includes the hyphens that separate the year, month, and day.

The length of the time constant without the fraction must be eight characters, which includes the colons that separate the hours, minutes, and seconds. The length of the time constant with the fraction can vary with the fraction of a second.

The length of the timestamp constant without the fraction must be 19 characters, which includes the hyphens and colons to separate the date and time components. The length of the timestamp with the fraction can vary with the fraction of a second.

Here are some examples of date and time constants:

```
date '2008-01-01'
time '11:59:59'
timestamp '2007-09-30 02:33:31.59'
```

Constant List

A constant list is a grouping of numeric and character constants that are enclosed in parentheses, separated by commas or blanks, that can be used in simple array assignments and in IN expressions. An example of a constant list is **`(2, '33', 2*-5, 2*(1, '3'))`**.

A constant list can be recursive and they can contain other constant lists. Individual list items can be repeated by using an iterator. An iterator is a number followed by the asterisk (*). The iterator indicates the number of times to repeat the constant or constant list that follows the asterisk. In the example above, the constant list item **`2* -5'`** results in **`-5, -5`**, and the list item **`2*(1,'3')`** results in **`1, '3', 1, '3'`**.

The above constant list **`(2, '33', 2*-5, 2*(1, '3'))`** is equivalent to **`(2, '33', -5, -5, 1, '3', 1, '3')`**.

A constant list has no data type nor does it have scalar values.

Chapter 5

DS2 Data Types

What Are the Data Types?

A data type is an attribute of every column that specifies what type of data the column stores. The data type is the characteristic that identifies a piece of data as a character string, an integer, a floating-point number, or a date or time. The data type also determines how much memory to allocate for the column's value.

The following table lists the sets of data types that are supported by DS2. Note that not all data types are available for table storage on each data source.

Table 5.1 *DS2 Data Types*

Data Type	Description
BIGINT	stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values; fractional portions are discarded.
BINARY(*n*)	stores 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.

Data Type	Description
CHAR(*n*)	stores a fixed-length character string, where *n* is the maximum number of characters to store. The maximum number of characters is required to store each value regardless of the actual size of the value. If **`char(10)`** is specified and the character string is only five characters long, the value is right padded with spaces.
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. For example, dates between 0001-01-01 and 1582-12-31 are not valid dates for a SAS data set or an SPD data set.
DECIMAL|NUMERIC(*p*,*s*)	stores a signed, exact, fixed-point decimal number, with user-specified precision and scale. The precision and scale determines the position of the decimal point. The precision is the maximum number of digits that can be stored to the left and right of the decimal point, with a range of 1 to 52. The scale is the maximum number of digits that can be stored following the decimal point. Scale must be less than or equal to the precision. For example, **`decimal(9,2)`** stores decimal numbers up to nine digits, with a two-digit fixed-point fractional portion, such as 1234567.89.
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	stores a signed, approximate, double-precision, floating-point number. Data defined as FLOAT is treated the same as DOUBLE.
INTEGER	stores a regular size signed, exact whole number, with a precision of ten digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded. *Note:* Integer division by zero does not produce the same result on all operating systems. It is recommended that you avoid integer division by zero.
NCHAR(*n*)	stores a fixed-length character string like CHAR but uses a Unicode national character set, where *n* is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and support all international characters.

Data Type	Description
NVARCHAR(*n*)	stores a varying-length character string like VARCHAR but uses a Unicode national character set, where *n* is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.
REAL	stores a signed, approximate, single-precision, floating-point number.
SMALLINT	stores a small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767. Integer data types do not store decimal values; fractional portions are discarded.
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. For example, the time 6:30 a.m. is specified as `06:30:00`. When supported by a data source, the *p* parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.
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. For example, the date and time September 24, 1975 6:30 a.m. is specified as `1975-09-24 06:30:00`. When supported by a data source, the *p* parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.
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.

Data Type Characteristics

Numeric Data Types

Numeric data types store numbers (for example, quantities and currency values). The choice of a numeric data type depends on the type of number that is being stored and on how the number will be used.

Characteristics to consider when choosing a numeric data type include the following:

- whether you want to use exact numeric data types or approximate numeric data types. Exact numeric data types such as BIGINT, DECIMAL, INTEGER, NUMERIC, and TINYINT, represent a value exactly. Approximate numeric data types, such as REAL, DOUBLE, and FLOAT, do not store the exact values that are specified for many numbers. For many applications, the tiny difference between the specified value and the stored approximation is not noticeable unless exact numeric behavior is required.
- whether you want to store whole numbers or decimal numbers. Note that for integer data types, if a decimal value is inserted, the fractional portion is discarded. For example, if 2.7 is inserted, the stored value is 2.
- how to store numbers with decimal points. Floating-point format is a number in which the decimal point is not fixed. A floating-point data type is used for longer decimal values and for quickly calculating a large range of numbers. For the floating-point data types, if you enter an integer that is too large, an error occurs. Enter the value using floating-point notation. Fixed point is a method for storing a number in which the decimal points line up. The fixed-point data type DECIMAL is used for exact precision of fractional components, such as money.
- a data type's precision. The precision for a data type specifies the total number of digits that a number can contain. The more digits, the higher the precision for the value. If a value is out of range, an error occurs. For example, for the BIGINT data type, an error occurs for a value that is larger than 9,223,372,036,854,775,807.

  To avoid errors or incorrect results, you must consider the results when performing operations on numeric values, particularly for the integer data types. The INTEGER data type has a precision of 10 digits and a range from –2,147,483,648 to 2,147,483,647. If you multiply two large integers (for example, 33432*79879) and the result is larger than 2,147,483,647, an overflow error occurs. If you expect a result that is larger than the data type's precision or range, you can assign the result as a larger data type such as DOUBLE or BIGINT, or you can enter the expression as a double constant (for example, 33432.0*79879.0), not an integer, in order to force the expression to be evaluated as a double precision, floating-point number.

Character Data Types

DS2 provides several character data types that store character string (text) data. Character data types can contain alphabetic characters, numeric digits 0 through 9, and other special characters.

Note: If a character string includes a number, DS2 automatically converts it to a numeric type and uses that number in any calculation.

Each character data type provides a parameter for specifying the maximum number of characters.

The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character (for example, UTF-8), then CHAR and NCHAR are identical types and both use NCHAR.

You can specify the character set encoding information for CHAR and VARCHAR data types in a DECLARE, METHOD, or VARARRAY statement. The default encoding depends on your operating system and locale. For a complete list of character set encoding values, see "Character Sets for Encoding in NLS" in the *SAS Viya National Language Support: Reference Guide*.

Note: Trailing blanks are always ignored even for columns that have the VARCHAR data type.

Note: The ENCODING= system option is not supported when you are reading or updating SAS data sets. DS2 converts a given ENCODING= value to the least common denominator encoding when multiple encoding values are specified with the CHARACTER SET syntax. The ENCODING= value that is used can differ from what really is created in the data set. This conflict can cause potential transcoding errors.

Note: DS2 follows ANSI SQL standards for data casting. For example, if you cast a variable with a CHAR data type as a new variable with a VARCHAR data type, the contents of the CHAR variable are transferred to the VARCHAR variable if it fits. If the CHAR variable does not fit, it is truncated regardless of whether it is a whitespace character. Here is an example.

```
char(8) u = 'eight';
cast( u as varchar(6) ) as v
```

u has a value of '**eight**'. **v** has a value of '**eight**'. You would need to use the TRIM function to eliminate the white space.

Date and Time Data Types

DS2 supports several data types for the specific purpose of storing dates and times. The date and time ranges are data source specific.

Define Data Types for a Column

When defining a data type, use the data type keywords for the data types supported by DS2.

Keep in mind that for data to be stored, the data type must be available for data storage in that data source. Although DS2 supports several data types, the data types that can be defined for a particular table depend on the data source, because each data source does not necessarily support all of the DS2 data types. In addition, data sources support variations of the standard SQL data types. That is, a specific data type that you specify might map to a different data type and might also have different attributes in the underlying data source. This occurs when a data source does not natively support a specific data type, but data values of a similar data type can be converted without data

loss. For example, to support the INTEGER data type, a SAS data set maps the data type definition to SAS numeric, which is a DOUBLE.

For details about data source implementation for each data type, see "Data Type Reference" in *SAS Viya: DS2 Language Reference*.

Note: SAS Cloud Analytic Services currently supports CHAR, VARCHAR, and DOUBLE data types for table storage on each data source. These data types support missing values.

In addition, the CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source or the specified precision or scale is not within the data source range. For information about the CT_PRESERVE= connection argument, see *SAS Federation Server: Administrator's Guide*.

Error Messages That Use the DOUBLE and REAL Data Types

When error messages contain numeric values for DOUBLE and REAL data types, these values are written to the log using the BEST*w*. format. The BEST*w*. format is the default format for writing numeric values. When you use BEST*w*., SAS chooses the format that provides the most information about the value according to the available field width.

Because SAS uses the BEST*w*. format, the DOUBLE and REAL data type values might not be exactly the same as the values that you use in your programs. BEST*w*. rounds the value, and if SAS can display at least one significant digit in the decimal portion, within the width specified, BEST*w*. produces the result as a decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it.

Chapter 6

DS2 Identifiers

Overview of Identifiers

An identifier is one or more tokens, or symbols, that name programming language entities, such as variables, method names, package names, and arrays, as well as data source objects, such as table names and column names.

The DS2 language supports ANSI SQL:1999 standards for both regular and delimited identifiers.

Regular identifiers are the type of identifiers that you see in most programming languages. They are **not case-sensitive** so that the identifier **`Name`** is the same as **`NAME`** and **`name`**. Only certain characters are allowed in regular identifiers.

Delimited identifiers are case-sensitive only for identifiers that require double quotation marks, that is, table and schema names. Other delimited identifiers are not case-sensitive. Delimited identifiers allow any character and must be enclosed in double quotation marks. Variable names **`"Name"`**, **`"NAME"`**, **`Name`**, **`name`**, and **`NaMe`** all represent the same variable.

By supporting ANSI SQL:1999 identifiers, the DS2 language is compatible with data sources that also support the ANSI SQL:1999 identifiers.

Note: Identifiers for SAS and SPD data sets are limited to 32 characters.

Regular Identifiers

When you name regular identifiers, use these rules:

- The length of a regular identifier can be 1 to 256 characters.

- The first character of a regular identifier must be a letter or underscore. Subsequent characters can be letters, digits, or underscores. Note that you cannot begin a regular identifier with two underscores.
- Regular identifiers are case-insensitive.

The following regular identifiers are valid:

```
firstName
lastName
_phonenum
phone_num1
```

Letters in regular identifiers are stored internally as uppercase letters, which allows letters to be written in any case. For example, an unquoted, input column name of **phone_num1**, **Phone_Num1**, or **PHONE_NUM1** is displayed in the output as uppercase, **PHONE_NUM1**. Identifiers that do not contain special characters or spaces and are enclosed in double quotation marks are treated as regular identifiers as if the quotation marks were not present.

Note: Each data source has its own naming conventions, all of which are supported by the DS2 language. When your program contains identifiers specific for a particular data source, you must follow the naming conventions for that data source. For more information, see the topic on naming conventions for your data source in *SAS/ACCESS for Relational Databases: Reference*.

Delimited Identifiers

When you name delimited identifiers, follow these rules:

- The length of a delimited identifier can be 1 to 256 characters.
- Begin and end delimited identifiers with double quotation marks.
- Delimited identifiers consist of any sequence of characters, including spaces and special characters, between the beginning and ending double quotation marks.
- The only delimited identifiers that are case-sensitive are table and schema names. Other delimited identifiers are not case-sensitive.

A string of characters enclosed in double quotation marks is interpreted as an identifier and not as a character constant. Character constants can be enclosed only in single quotation marks.

The following delimited identifiers are valid:

```
"x&y&z"
"Ü1"
"(area)phone_num"
"a**B"
```

You can use delimited identifiers for terms that might otherwise be a reserved word. For example, to use the term “date” other than for a date declaration, you would use it as the delimited identifier “date”. Here is an example.

```
/* In DATA step, there are no reserved words for variables. */
/* So, this doesn't cause an error.  However, how do you    */
/*  use such a variable in DS2?                             */
data a;
   date = mdy(8,14,2012);
```

```
run;

/* This program gives an error because the reserved word DATE */
/*  is used when a variable is expected.   */
proc ds2;
data b(overwrite=yes);
   method run();
      set a;
      if month(date) = 8 then
         put 'August';
      else
         put 'Not August';
   end;
enddata;
run; quit;

/* One solution is to quote the reserved word */
proc ds2;
data b(overwrite=yes);
   method run();
      set a;
      if month("date") = 8 then
         put 'August';
      else
         put 'Not August';
   end;
enddata;
run; quit;


/* Another solution is to rename the variable on input and output */
/*  to avoid the reserved word in DS2 code.                       */
proc ds2;
data b(overwrite=yes rename=(sas_date="date"));
   method run();
      set a(rename=("date"=sas_date));
      if month(sas_date) = 8 then
         put 'August';
      else
         put 'Not August';
   end;
enddata;
run; quit;
```

A warning is issued for tables that are created with delimited column names that are then referenced in DS2 programs that are submitted to data sources that are not case-sensitive. Data sources that are not case-sensitive remove the quotation marks and treat the column name as not delimited.

A delimited identifier is similar to name literals in the SAS DATA step language.

Note: Each data source has its own naming conventions, all of which are supported by the DS2 language. When your program contains identifiers specific for a particular data source, you must follow the naming conventions for that data source. For more information, see the topic on naming conventions for your data source in *SAS/ACCESS for Relational Databases: Reference*.

Referencing a Macro Variable in a Delimited Identifier

To reference a macro variable in a delimited identifier, use the SAS macro function %TSLIT, which overrides the need for double quotation marks around the literal string and puts single quotation marks around the input value. For example, the following statement includes the %TSLIT function to specify the macro variable, %profit:

```
put %tslit(PROFIT: &profit);
```

The %TSLIT macro function is stored in the default autocall macro library. The syntax is as follows:

%TSLIT(<*literal-text* >*macro-call*);

Note: If you do not specify *literal-text* and a null value is passed to the macro variable reference, a warning message `Argument 1 to function TRANSLATE referenced by the %SYSFUNC or %QSYSFUNC macro function is out of range.` is written to the SAS log.

Support for Non-Latin Characters

The DS2 language supports only Latin characters for regular identifiers. To use non-Latin characters, the identifier must be delimited using double quotation marks.

Chapter 7

How DS2 Processes Nulls and SAS Missing Values

DS2 Modes for Nonexistent Data

Many relational databases such as Oracle and DB2 implement ANSI SQL null values. Therefore, the concept of null values using DS2 is the same as using the SQL language for databases that support ANSI SQL. It is important to understand how DS2 processes SAS missing values because data can be lost.

Note: Only DOUBLE or CHAR data types can have missing values. Data types other than DOUBLE or CHAR can have only null values.

Note: SAS Cloud Analytic Services currently supports CHAR, VARCHAR, and DOUBLE data types for table storage on each data source.

Because there are significant differences in processing null values and SAS missing values, DS2 has two modes for processing nonexistent data: ANSI SQL null mode (ANSI mode) and SAS missing value mode (SAS mode).

There are two ways to set the behavior of nonexistent data:

- DS2 code that is submitted to PROC DS2 processes the data using SAS mode. PROC DS2 provides the ANSIMODE option that enables you to process data using ANSI mode.
- DS2 code that is submitted using the DS2 runDS2 action processes the data using SAS mode. The DS2 runDS2 action provides the “nullBehavior”:“ANSI” parameter that enables you to process data using ANSI mode.

In most instances, no mode change is necessary to process nonexistent data. You can be in any mode and still operate on null and missing values together.

The following are instances of when you might want to change the mode:

- when a client application processes SAS data sets and the mode for nonexistent data is in ANSI mode
- when the processing of SAS data sets is complete and the client application is ready to return to ANSI mode

CAUTION:

If the mode is not set for the desired results, data is lost. In ANSI mode, when DS2 reads a SAS numeric missing value from a SAS data set, it converts the SAS missing value to an ANSI null value. If the ANSI null value is then written to a SAS data set, DS2 converts the ANSI null value to the SAS numeric missing value (.). If the SAS numeric missing value from the input data set is a special numeric missing value, such as .A, the .A is lost during the conversion to and from ANSI null, and the SAS numeric missing value (.) is written to the output data set. In the following example, column **x** is of data type DOUBLE. The example illustrates how a SAS special numeric missing value (.A in the fourth row) is transformed to the SAS numeric missing value (.) by a DS2 program run in ANSI mode.

```
/* assume input data set indata contains */
x
100
.
.A

/* Run the following program using ANSI mode */
proc ds2 ansimode;
data outdata;
   method run();
      set indata;
   end;
enddata;
run;
quit;

/* the output dataset outdata contains */
 x
 100
 .
 .
```

In SAS mode, when DS2 reads an ANSI null value of data type CHAR from an ANSI data source, it converts the ANSI null value to a SAS character missing value (blank-filled character string). If the SAS character missing value is then written to an ANSI data set, the output CHAR column value is a blank-filled character string rather than the ANSI null value. In the following example, column **x** is of data type CHAR(5). The example illustrates that the ANSI null value (in the second row) is transformed to a blank-filled string (' ') by a DS2 program that is run in SAS mode.

```
/* assume input data set indata contains */
 x
 'abc  '
 null

/* run the following program using SAS mode */
proc ds2;
data db.outdata;
   method run();
```

```
      set db.indata;
   end;
enddata;
run;
quit;

/* the output dataset outdata contains */
 x
 'abc  '
 '     '
```

For information about how to set DS2 in ANSI mode or SAS mode, see the SAS documentation for your client environment.

Differences between Processing Null Values and SAS Missing Values

Processing SAS missing values is different from processing null values and has significant implications in these situations:

- when filtering data (for example, in a WHERE clause, a HAVING clause, a subsetting IF statement, or an outer join ON clause). SAS mode interprets null values as SAS missing values, which are known values, whereas ANSI mode interprets null values as unknown values.
- when submitting outer joins in ANSI mode, internal processing might generate nulls for intermediate result tables. DS2 might generate SAS missing values in SAS mode for intermediate result tables. Therefore, for intermediate result tables, nulls are interpreted as unknown values in ANSI mode and in SAS mode, missing values are interpreted as known values.
- when comparing a blank space, SAS mode interprets the blank space as a missing value. In ANSI mode, a blank space is a blank space, it has no special meaning.
- DS2 interprets a null as a SAS missing value in these cases:
  - in SAS mode, a null is used in a computation or assignment involving a floating-point or fixed-width character value
  - a null is passed to a SAS format or function

Note: If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.

The following are attribute and behavior differences between null values and SAS missing values:

Table 7.1 *Attribute and Behavior Differences between ANSI SQL Null Values and SAS Missing Values*

Attribute or Behavior	ANSI SQL Null value	SAS Missing Values
internal representation	metadata	floating-point or character

Attribute or Behavior	ANSI SQL Null value	SAS Missing Values
evaluation by logical operators	is an unknown value that is compared by using three-valued logic, whose resolved values are True, False, and Unknown. For example, `WHERE col1 = null` returns **UNKNOWN**.	is a known value that when compared, resolves to a Boolean result
collating sequence order	appears as the smallest value	appears as the smallest value

Reading and Writing Nonexistent Data in ANSI Mode

Many relational databases such as Oracle and DB2 implement ANSI SQL null values. Therefore, the concept of null values using DS2 is the same as using the SQL language for databases that support ANSI SQL. It is important to understand how DS2 processes SAS missing values because data can be lost.

SAS missing value data types can be only DOUBLE or CHAR. Therefore, only the conversion for these data types is shown. The following table shows the value returned to the client application when DS2 reads a null value or a SAS missing value from a data source in ANSI mode:

Table 7.2 *Reading Nonexistent Data Values in ANSI Mode*

Column Data Type	Nonexistent Data Value	Value Returned to the Application
DOUBLE	., ._, or .A-.Z *	null
DOUBLE	null	null
CHAR	'␣' **	'␣' **
CHAR	null	null

* The value ._ (period followed by an underscore) or .A–.Z represent a special numeric missing value. When SAS prints a special missing value, it prints only the letter or underscore.

** The value '␣' is a blank space between single quotation marks, which in ANSI mode, is a blank space, not nonexistent data.

This next table shows the value stored when nonexistent data values are written to data sources in ANSI mode:

Table 7.3 *Storing Nonexistent Data in ANSI Mode*

Column Data Type	Nonexistent Data Value	Value Stored in the SAS Data Set	Value Stored in the ANSI SQL Null Supported Data Source
DOUBLE	null	.	null
CHAR	null	'␣' *	null
CHAR	'␣' *	'␣' *	'␣' *

* The value '␣' is a blank space between single quotation marks, which in ANSI mode, is a blank space, not nonexistent data.

In ANSI mode, a blank space is always interpreted as a blank space and not as nonexistent data. Also, in ANSI mode, SAS missing values are converted to ANSI null values at input or assignment.

Reading and Writing Nonexistent Data in SAS Mode

When the client application uses SAS mode, nonexistent data values are treated like SAS missing values in the DATA step.

The following table shows how nonexistent data values are read in SAS mode:

Table 7.4 *Reading Nonexistent Data Values in SAS Mode*

Column Data Type	Nonexistent Data Value	Value Returned to the Application
DOUBLE	., ._, or .A-.Z *	., ._, or .A-.Z *
DOUBLE	null	.
CHAR	'␣' **	'␣' **
CHAR	null	'␣' ** ***

* The value ._ (period followed by an underscore) or .A–.Z represent a special numeric missing value. When SAS prints a special missing value, it prints only the letter or underscore.

** The value '␣' is a blank space between single quotation marks, which in SAS mode, is nonexistent data.

*** When the SET statement encounters a null for a fixed-width character string, the string contains blank characters for the length of the string.

The next table shows how missing data values are written to a data source in SAS mode:

Table 7.5 *Writing Nonexistent Data Values to a SAS Data Source in SAS Mode*

Column Data Type	Nonexistent Data Value	Value Stored in the SAS Data Set	Value Stored in ANSI SQL Null Supported Data Source
DOUBLE	., ._, or .A	., ._, or .A-.Z	null
CHAR	'␣'*	'␣'*	'␣'*

* The value '␣' is a blank space between single quotation marks, which in SAS mode, is nonexistent data.

In SAS mode, ANSI null values for type DOUBLE or CHAR are converted to SAS missing values at input or assignment.

Testing for Null Values

DS2 provides the NULL function to test for a null value. The NULL function has one argument, which can be an expression. If the expression is null, the function returns `1`. If the expression is not null, the function returns `0`.

This example shows a test for a null value:

```
if null(numCopies) then put 'Number of copies is unknown.'
   else put 'Number of copies is' numCopies;
```

The NMISS function returns the number of null and SAS missing numeric values. The NMISS function requires numeric values and works with multiple numeric values, whereas NULL works with only one value that can be either numeric or character.

For more information, see the "NULL Function" in *SAS Viya: DS2 Language Reference* and the "NMISS Function" in *SAS Viya: DS2 Language Reference*.

Testing for Missing Values

DS2 provides the MISSING function to test for a null value. The MISSING function has one argument, which can be a numeric or character expression. If the expression is null, the function returns `1`. If the expression is not null, the function returns `0`.

The NMISS function returns the number of null and SAS missing numeric values. The NMISS function requires numeric values and works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

For more information, see the "MISSING Function" in *SAS Viya: DS2 Language Reference* and the "NMISS Function" in *SAS Viya: DS2 Language Reference*.

Chapter 8

DS2 Type Conversions

Type Conversion Definitions

binary data type
: refers to the BINARY and VARBINARY data type

character data type
: refers to the CHAR, VARCHAR, NCHAR, and NVARCHAR data types

coercible data type
: a data type that can be converted to multiple data types, not just a character data type

date/time data type
: refers to the DATE, TIME, and TIMESTAMP data types

non-coercible data type
: a data type that only can be converted to a character data type

numeric data type
: refers to the DECIMAL, DOUBLE (or FLOAT), REAL, BIGINT, INT, NUMERIC, SMALLINT, and TINYINT data types

standard character conversion
: if an expression is not one of the character data types, it is converted to a CHAR data type.

standard numeric conversion
: if an expression has a coercible, non-numeric data type, it is converted to a DOUBLE data type.

Overview of Type Conversions

Operands in an expression must be of the same general data type, numeric, character, binary, or date/time, in order for DS2 to resolve the expression. When it is necessary, DS2 converts an operand's data type to another data type, depending on the operands and operators in the expression. This process is called **type conversion**. For example, the concatenation operator (||) operates on character data types. In a concatenation of the character string `'First'` and the numeric integer 1, the INTEGER data type for the operand 1 is converted to a CHAR data type before the concatenation takes place.

When an operand data type is converted within the same general data type, the operand data type is promoted. Operands with a data type of SMALLINT and TINYINT are promoted to INTEGER, and operands of type REAL are promoted to DOUBLE. Type promotion is performed for all operations on SMALLINT, TINYINT, and REAL, including arguments for method and function expressions.

Numeric and character data types are coercible. The BINARY, VARBINARY and the date/time data types DATE, TIME, and TIMESTAMP, are non-coercible and only can be converted to one of the character data types.

When DS2 evaluates an expression, if the data types of the operands match exactly, no type conversion or promotion is necessary and the expression is resolved. Otherwise, each operand must go through a standard numeric conversion or a standard character conversion, depending on the operator.

The results of a numeric or character expression are based on a data type precedence. If both operands have different types within the same general data type, the data type of the expression is that of the operand with the higher precedence, where 1 is the highest precedence. For example, for numeric data types, a data type of DOUBLE has the highest precedence. If an expression has an operand of type INTEGER and an operand of type DOUBLE, the data type of the expression is DOUBLE. A list of precedences can be found in the topics that follow, if applicable, for the different types of expressions.

For a table showing all type conversions, see Appendix 2, “DS2 Type Conversions for Expression Operands,” on page 215.

Type Conversion for Unary Expressions

In unary expressions, such as `+1` or `-444`, the standard numeric conversion is applied to the operand. The following table shows the data type for unary expressions:

Table 8.1 *Data Type Conversion for Unary Expressions*

Expression Type	Expression Data Type
Unary plus	same as the operand or DOUBLE for converted operands
Unary minus	same as the operand or DOUBLE for converted operands

Expression Type	Expression Data Type
Unary not	INTEGER

Type Conversion for Logical Expressions

In logical expressions, such as `a & b` or `(a ~= start) | (f = finish)`, the standard numeric conversion is applied to each operand. The following table shows the precedence used to determine the data type of the expression, where 1 is the highest precedence and 3 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

Table 8.2 *Data Type Conversion for Logical Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	DOUBLE	DOUBLE
2	BIGINT	BIGINT
3	all other numeric data types	INTEGER

Type Conversion for Arithmetic Expressions

In arithmetic expressions, such as `a<>b` or `a + (b * c)`, the standard numeric conversion is applied to each operand.

The following table shows the precedence used to determine the data type of arithmetic expressions for the addition, subtraction, multiplication, and division operators, where 1 is the highest precedence and 3 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

Table 8.3 *Type Conversion for Addition, Subtraction, Multiplication, and Division Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	DOUBLE	DOUBLE
2	DECIMAL, NUMERIC	DECIMAL, NUMERIC
3	BIGINT	BIGINT
4	all other numeric data types	INTEGER

The following table shows the data type for arithmetic expressions that use the min, max, and power operators:

Table 8.4 *Data Type Conversion for the Min, Max, and Power Operator Expressions*

Operator	Operator Data Type	Expression Data Type
min or max	DOUBLE	DOUBLE
min or max	DECIMAL, NUMERIC	DECIMAL, NUMERIC
min or max	BIGINT	BIGINT
**	all numeric data types	DOUBLE

Type Conversion for Relational Expressions

In relational expressions, such as `x <= y` or `i > 4`, the standard conversion that is applied depends on the operand data types. The data type of the expression is always INTEGER, as shown in the following tables.

Table 8.5 *Data Type Conversion for Relational Expressions except IN Expressions*

Order of Data Type Resolution	Data Type of Either Operand	Standard Conversion	Expression Data Type
1	any numeric data type	numeric	INTEGER
2	CHAR, NCHAR	character	INTEGER
3	DATE, TIME, TIMESTAMP	none, data types must match	INTEGER
4	NVARCHAR, VARCHAR	none, error returned	not applicable

Table 8.6 *Data Type Conversion for IN Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	non-numeric	DOUBLE
2	DOUBLE	DOUBLE
3	DECIMAL, NUMERIC	DECIMAL

Precedence	Data Type of Either Operand	Expression Data Type
4	all other types that are not BIGINT	BIGINT

Type Conversion for Concatenation Expressions

In concatenation expressions, such as **a || b** or **x !! y**, the standard character conversion is applied to each operand. The following table shows the precedence used to determine the data type of the expression, where 1 is the highest precedence and 2 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

Table 8.7 *Data Type Conversion for Concatenation Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	if either is type NCHAR	NCHAR
2	CHAR	CHAR

Chapter 9

DS2 Expressions

What Is an Expression?

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 either a calculation, comparison, or concatenation of operators.

Here are some examples of DS2 expressions:

```
a = b * c
"col1"
s || 1 || z
a >= b**(c - 8)
system.put(a*5,hex.)
```

Types of Expressions

Overview of Expressions

The basic type of expression is a primary expression. Complex expressions combine expressions and operators. Other expressions invoke a DS2 construct, such as a method expression or a function expression. The system expression and the THIS expression refer to expressions that are global in scope. The IN expression returns a Boolean result based on whether the result of an expression is contained in a list.

Expression kinds commonly refer to a segment of code. For example, an AND expression refers to the AND operator and the operands that it processes. A binary expression refers to a binary constant or a hexadecimal constant such as **x'ff00effc'**.

Whether an expression is simple or complex, invokes a construct, or is global in scope, expressions of all kinds have a value and a data type.

Primary Expression

In their simplest form, primary expressions are numbers, character strings, binary and hexadecimal constants, literal values, date and time values, identifiers, and null values, as in these primary expressions:

```
a
"var_a"
5.33
'Company'
date '2007-08-24'
```

The following table shows basic primary expressions, their data type(s), and an example:

Table 9.1 *Data Types and Examples of Primary Expressions*

Type of Expression	Short Description	Data Type	Example
Array	Groups same type data	Same type as individual items in the array	**a[5, b+c]**
Binary	Binary and hexadecimal constants	VARBINARY	**x'FE'** **b'01000011'**

Character	Character string	CHAR, VARCHAR	**'New Report'**
			`a='Stock';`
			`b='Report';`
			`c=a|| b;`
			`*`

National Character	National Character string	NCHAR, NVARCHAR	**n'New Report'**
			`a=n'Stock';`
			`b=n'Report';`
			`*`

Type of Expression	Short Description	Data Type	Example
Dot	System, THIS, and package method expressions	The resolved type of the expression	**system.put(x,5.)** **this.s** **p.calc(2,6,9)**
Date / time	Date and time values	DATE, TIME, TIMESTAMP, or DOUBLE	**date '2007-01-01'**
Identifier	Provides a name for various language elements or is a keyword	The declared or default type. The default is DOUBLE.	**a** **"part1"** **IN**
Integer	Integer numbers	INTEGER, BIGINT	**123**
New	Instantiates a package method	The resolved type of the expression	`a=_new_ package_name();` *
Numeric	Integer, real, and floating point numbers, or a missing or null value	DOUBLE, FLOAT, REAL	**5** **4.3**
Null	Null expression	none	**NULL**
Parentheses	Operator and operands enclosed in parentheses for higher evaluation precedence	The resolved type of the expression enclosed in parentheses.	`(a + b) - c` *

* For the purpose of the example, the primary expression is contained in an expression or an assignment statement. The example expression is highlighted.

Complex Expression

A complex expression combines expressions and operators to create a more expansive expression, as in these expressions:

```
a + b * -c - 5
a = b = 5
x | y & a < c * d
x**y**z - 9 >= f
z >< c + e <> u**y * 10
x || c + 7
a in (1,2,3)
```

Evaluation of complex expressions is based on the operator order of precedence, as shown in Table 9.5 on page 69, and the data types of the primary expressions. Before any calculations can be done, operand data types must be the same general data type: numeric, character, binary, or date/time. If the data types are the same, processing can proceed. If they are not the same, the operand data types are converted based on the operator and the data type of the operands.

In the expression **a + b / -c - 5**, assume that **a** is 1.35 with a type of DOUBLE, **b** is 2 with a data type of INTEGER, and **c** is 3 with a data type of INTEGER. **b/-c** or **2/-3** evaluates to INTEGER 0. The INTEGER 0 is converted to a DOUBLE 0 before being added to **a**. Then **a + 0.0** evaluates to DOUBLE 1.35. The INTEGER 5 is converted to a DOUBLE 5 before the addition of the DOUBLE 1.35. The final result of the expression is DOUBLE –3.65.

For information about data type conversion, see Chapter 8, "DS2 Type Conversions," on page 49.

In the expression **a = b = 5**, if **b** is a value other than 5, then **b = 5** is evaluated to 0. Therefore, **a** is assigned a value of 0. The first equal sign (=) is an assignment operator and the second equal sign is a logical equality operator. For more information, see "Using an Expression with Multiple Equals Signs" in *SAS Viya: DS2 Language Reference*.

Note: DS2 supports using **eq** as well as the equal sign.

Array Expression

An array expression is a primary expression that represents a grouping of data items of the same data type. Although an array can have multiple dimensions, individual data item values are scalar values. Data items are accessed by specifying an index into the array.

The array expression consists of an array identifier followed by an array index expression for each dimension in the array, as in this syntax:

array-identifier \[*index-expression* < , …*index-expression* > \]

Note: Brackets in the syntax convention indicate optional syntax. The escape character (\ ) before a bracket indicates that the bracket is required for the syntax. Indexes in an array expression must be contained by brackets ([ ]).

The array identifier can be either a declared array variable or a variable used in a THIS expression. The index expression is a primary expression that resolves to an integer.

Here are some examples of array expressions:

```
a[i]
s[j * 2, k-3]
this.c[2, vwind, a[i]]
```

When an array is declared, the index values specify the boundaries for the array. If an index expression is beyond the boundaries of the array, DS2 issues an error.

The value of an array expression is the value of the indexed value in the array. For example, if the array values are a[1] = 12, a[2] = 15, and a[3] = 20, the value of the array expression a[2] is 15.

Note: Arrays are 1-based. The array index starts at 1.

Function Expression

A function expression invokes a function within a DS2 program. To invoke a function, use this syntax:

function-name (< *argument* < , …*argument* > >)

Functions might require arguments. If the function expression contains arguments, the argument data types are converted, if necessary, to the data types of the function

signature, which is the argument order and data type for a function. Parentheses in the function call are required, whether the function takes arguments or no arguments are required. For example, the TIME function does not require arguments:

```
t = time();
```

A function expression resolves to the value returned by the function. In the function expression above, **`time()`** resolves to the current time of day.

Methods and functions are similar. Functions have global scope. Methods are programming blocks and have local scope.

If the name of a function is identical to a method name, DS2 invokes the method. Functions with the same name as a method can be invoked only by using a SYSTEM expression.For more information, see "SYSTEM Expression" on page 63.

For a list of DS2 functions, see "DS2 Functions" in *SAS Viya: DS2 Language Reference*.

IN Expression

An IN expression determines whether an expression is contained in a constant list. See "Constant List" on page 32.

Here is the syntax of an IN expression:

expression < *not-operator* > **IN** *constant-list*

Note: Any valid data type for your data source can be used in *constant-list*. If any argument is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already). If any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already). Otherwise, all arguments are converted to a BIGINT. The result is always INTEGER, either 0 or 1.

Any of the NOT operators (~, ^, or NOT) are valid before the IN operator, which results in the logical negation of the expression value.

The following table shows the results of some IN expressions:

Table 9.2 *Examples of IN Expressions*

Input Values	IN Expression	Results
`a = 2`	`a in (5,34,2,67)`	`1`
`b = 3`	`b not in (3,22,43,65)`	`0`

LIKE Expression

Overview of the LIKE Expression

A LIKE expression determines whether a character string matches a pattern-matching specification.

Here is the syntax of a LIKE expression:

expression < NOT> **LIKE** *pattern-matching-expression* <ESCAPE *character-expression*>

The expressions can be any character string or binary string data type.

If *expression* matches the pattern specified by *pattern-matching-expression*, a value of 1 (true) is returned. Otherwise, a value of 0 (false) is returned.

NOT LIKE returns the inverse value of LIKE. For example, if **`x like y`** is true, then **`x not like y`** is false.

The ESCAPE argument is used to search for literal instances of the percent (%) and underscore (_) characters, which are usually used for pattern matching.

Patterns for Searching

Patterns consist of three classes of characters.

Table 9.3 *Pattern-matching Characters*

Character	Description
underscore (_)	matches any single character
percent sign (%)	matches any sequence of zero or more characters *Note:* Be aware of the effect of trailing blanks. To match values, you might have to use the TRIM function to remove trailing blanks.
any other character	matches that character

Searching for Literal % and _

Because the % and _ characters have special meaning in the context of the LIKE expression, you must use the ESCAPE argument to search for these character literals in the input character string.

These examples use the values **`app`**, **`a_%`**, **`a__`**, **`bbaa1`**, and **`ba_1`**.

- The condition **`like 'a_%'`** matches **`app`**, **`a_%`**, and **`a__`**, because the underscore (_) in the search pattern matches any single character (including the underscore), and the percent (%) in the search pattern matches zero or more characters, including '%' and '_'.
- The condition **`like 'a_^%' escape '^'`** matches only **`a_%`**, because the escape character (^) specifies that the pattern search for a literal '%'.
- The condition **`like 'a_%' escape '_'`** matches none of the values, because the escape character (_) specifies that the pattern search for an 'a' followed by a literal '%', which does not apply to any of these values.

Searching for Mixed-case Strings

The DS2 LIKE expression is case sensitive. To search for mixed-case strings, use the UPCASE function as the following example shows:

```
upcase(str) like 'SM%'
```

LIKE Expression Examples

The following table shows examples of the matches that would result when searching these strings: **`Smith, Smooth, Smothers, Smart, Smuggle`**.

Table 9.4 *Examples of LIKE Expressions*

LIKE Expression Example	Matching Results
`str like 'Sm%'`	`Smith, Smooth, Smothers, Smart, Smuggle`
`str like '%th'`	`Smith, Smooth`
`str LIKE 'S__gg%'`	`Smuggle`
`str like 'S_o'`	(no matches)
`str like 'S_o%'`	`Smooth, Smothers`
`str like 'S%th'`	`Smith, Smooth`
`str not like 'Z'`	`Smith, Smooth, Smothers, Smart, Smuggle`

Method Expression

A method expression invokes a method that has been defined by the METHOD statement.

To invoke a method, use this syntax:

method-name (< *expression* < , … *expression* > >)

Methods are invoked based on the method name and signature. DS2 first identifies the method name. If a method name is identical to a function name, DS2 invokes the method. A function with the same name as a method can be invoked by using a SYSTEM expression. For more information , see "SYSTEM Expression" on page 63.

Because DS2 allows overloaded methods, DS2 invokes the method whose arguments best match the number of arguments and the argument data types in the method signature, which is the argument order and data type for the method. The best match will be the one for which the number of method parameters is equal to the number of arguments, and such that no other method signature has as many exact parameter type matches for the given argument list. If a best match is not found, an error occurs.

Once the method to execute is identified, DS2 converts argument data types to the data type of the corresponding method parameter, if necessary. The method then executes.

A method expression resolves to the value returned by the method.

In the following example, methods CONCAT and ADD are defined and then invoked in the INIT() method. The highlighted expressions in the INIT() method are method expressions:

```
method concat(char(100) x, char(100) y) returns char(200);
   return trim(x)|| y;
end;

method add(double x, double y) returns double;
   return x + y;
end;
```

```
method init();
   dcl char(200) r;
   r = concat('abc', 'def');
   d = add(100,101);
end;
```

In this next example, the D method is an overloaded method. DS2 must compare the method expression arguments to the method signatures to find the best match:

```
method d(double x, double y) returns double;
   return x + y;
end;

method d(int x, int y) returns int;
   return x + y;
end;

method init();
   dcl double r;
   dcl int i;
   r = d(1.2345, 5.6789);
   i = d(99, 100);
end;
```

The first method calls the D method whose signature requires values with DOUBLE data types. The second method calls the D method whose signature requires values with INTEGER data types.

This final example shows that DS2 cannot determine whether the values in the method expression have a data type of INTEGER or DOUBLE. Because it is ambiguous, DS2 issues an error:

```
method d(int x, double y) returns double;
   return x + y;
end;

method d(double x, int y) returns double;
   return x + y;
end;

method run();
   d = d(100, 102);
end;
```

For more information, see the "METHOD Statement" in *SAS Viya: DS2 Language Reference*.

Package Method Expression

A package method expression instantiates a method that is defined in a package. To invoke a package method expression, use this syntax:

package-name.method-name (< *method-argument* < , … *method-argument* > >)

Package method expressions execute in a manner similar to method expressions. That is, once DS2 has determined that the package and the method exist, the best match of method signatures is determined, argument data types are converted if necessary, and the method executes.

In the following example, the highlighted expressions are package method expressions:

```
declare package myadd a1() a2();
a1.sale(3,4);
a1.add(1,2);
a2.bonus(5,12);
a2.add(10,20);
```

The first two package method expressions invoke the SALE and ADD methods in the A1 package, which was instantiated from the MYADD package. The last two package method expressions invoke the BONUS and ADD methods in the A2 package.

Note: You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement.

For information about packages, see the "PACKAGE Statement" in *SAS Viya: DS2 Language Reference* and Chapter 12, "DS2 Packages," on page 97.

SYSTEM Expression

When a method and a function have identical names, the method call takes precedence over the function call. The function can then be invoked only by using a SYSTEM expression.

To invoke a SYSTEM expression, use this syntax:

SYSTEM.*function-expression*

A SYSTEM expression prepends a function expression with the dot notation, **`system.`**. For example, if SUM is the name of a method as well as the name of a function, the SUM function only can be invoked by using the SYSTEM expression: **`system.sum(a,b,c)`**.

THIS Expression

A THIS expression provides an alternate method to simultaneously declare and use a global scalar variable from anywhere within a DS2 program. A THIS expression is used to circumvent the standard variable lookup. In a THIS expression, DS2 searches for a scalar variable declaration of the identifier in global scope. If there is no such declaration, DS2 declares the identifier in global scope with DOUBLE type. Global variables can be referenced by all programming blocks in a DS2 program.

To invoke a THIS expression, use this syntax:

THIS.*variable-name*

A THIS expression prepends a variable with the dot notation, **`THIS.`**.

Note: DS2 stores **`THIS.variable-name`** only as *variable-name*. If you have a local variable with the same name as the global scalar variable and DS2 issues a diagnostic message about the variable, you will not be able to distinguish which variable is a problem. For example, if DS2 issues a warning message that **`x`** is not declared, you would not know whether the message refers to the global variable, **`THIS.x`**, or the local variable, **`x`**.

In the following example, the variable **`s`** becomes a global variable by using a THIS expression:

```
method init();
  this.s = sum(a,b,c,d,e);
end;
method run();
   t = put(this.s 5.4);
```

```
end;
```

The THIS expression provides a method to access a global variable that is hidden by a local variable with the same name. Here is an example.

```
 data;
   declare double x;    /* declare global x */

   method run();
      declare double x;  /* declare local x */

      /* Two variables exist with same name, "x". */
      /* Identifier "x" refers to local x in      */
      /* scope of run method.  Global x is hidden */
      /* by local x.                              */

      this.x = 1.0;    /* assign 1.0 to global x   */
      x = 0.0;         /* assign 0.0 to local x    */

      output;          /* global x with 1.0 output */
    end;
enddata;
run;
```

IF Expression

Overview of the IF Expression

The conditional IF expression is used to select between two values based on whether a conditional expression evaluates to true (a nonzero value) or false (zero).

To invoke an IF expression, use this syntax:

IF *expression-1* **THEN** *expression-2* **ELSE** *expression-3*

If *expression-1*is a nonzero value, the result of the IF expression is the value of *expression-2*. Otherwise, the result of the IF expression is the value of *expression-3*. Here is an example.

```
 m=(if missing(u) then 0 else u);
```

The IF expression can be used wherever any other expression can be used. The precedence of an IF expression is lower than the arithmetic and logic operators, Therefore, parentheses are necessary in mixed expressions like this one.

```
r = 25.5 + (if sum < 15 then -a else b*2);
```

Without the parentheses, the plus (+) operator would be evaluated first resulting in a parse error from the subexpression **`25.5 + if`**.

Nested IF Expressions

IF expressions can be nested to select between many values for a multi-way decision.

IF *condition-expression-1***THEN** *result-expression-1*

ELSE IF *condition-expression-2* **THEN** *result-expression-2*

…

ELSE IF *condition-expression-n* **THEN** *result-expression-n*

ELSE *result-expression-default*

The condition expressions are evaluated in order. The result of the nested IF expression chain is the associated *result-expression* of the first *condition-expression* that evaluates to true (a nonzero value). If all the *condition-expressions* evaluate to false (zero), the result of the IF expression is the *result-expression-default*. Here is an example.

```
grade = if score >= 90 then 'A'
   else if score >= 80 then 'B'
   else if score >= 70 then 'C'
   else if score >= 60 then 'D'
   else if score >= 0  then 'F'
   else NULL;
```

IF Expression Data Type

The data type of an IF expression is determined by examining the type of the first result expression, *expression-2*.

```
IF expression-1 THEN expression-2 ELSE expression-3
```

If *expression-2* is not a numeric data type, then the IF expression is assigned the type of *expression-2*.

If *expression-2* is a numeric data type, then the IF expression is assigned the wider numeric data type of *expression-2* and *expression-3*. For example, if *expression-2* is an SMALLINT and *expression-3* is a DOUBLE, then the IF expression is assigned type DOUBLE. If *expression-2* is a numeric data type and *expression-3* is a not a numeric data type, then the IF expression is assigned the type of *expression-2*.

If the first result expression in a nested IF expression chain is a numeric data type, then all the result expressions are examined to find the widest numeric data type to assign as the type of the nested IF expression chain. In the following example, **t** is a TINYINT, **b** is a BIGNINT, and **d** is a DECIMAL(10,5). The 0 in the ELSE is assigned type BIGINT. Therefore, the nested IF expression chain is assigned the type decimal(10,5), the widest numeric type of TINYINT, BIGINT, and DECIMAL(10,5).

```
r = if n < 0 then t
   else if n = 0 then b
   else if n > 0 then d
   else 0;
```

Note: If 0.0 had been used for the ELSE value instead of 0, then the ELSE result would have been assigned type DOUBLE instead of type BIGINT. With the type DOUBLE ELSE expression, the widest numeric type of the result expressions would be type DOUBLE. Therefore, the nested IF expression chain would be assigned type DOUBLE instead of type decimal(10,5).

Lazy Evaluation of IF Expressions

The IF expression uses lazy evaluation for the result expressions.

```
IF expression-1 THEN expression-2 ELSE expression-3
```

For example, you could use this code to check for division by zero.

```
a = if c ne 0 then b/c else null;
```

Expression *expression-1* is always evaluated, but only one of *expression-2* or *expression-3* is evaluated. The expression that is not selected as the result of the IF expression is not evaluated. Thus, if *expression-1* is a nonzero value (true), then only *expression-2* is evaluated. If *expression-1* is zero (false), then only *expression-3* is evaluated.

Lazy evaluation also applies to the result expressions of nested if expression chains.

```
IF condition-expression-1 THEN result-expression-1
   ELSE IF condition-expression-2 THEN result-expression-2
   ...
   ELSE IF condition-expression-n THEN result-expression-2
   ELSE result-expression-default
```

The selected result expression is the only result expression evaluated. Lazy evaluation also applies to the condition expressions. Condition expressions are evaluated in order until a condition evaluates to true (nonzero) or all conditions are evaluated. If the IF expression has *n* condition expressions and the *i*th condition is the first nonzero condition, then only the first 1 to *i* conditions are evaluated. The *i*+1 to *n* conditions are not evaluated.

SELECT Expression

Overview of the SELECT Expression

A SELECT expression is used to select between multiple expressions based on the values of other expressions.

To invoke a SELECT expression, use this syntax:

SELECT <(*select-expression*)>

WHEN (*when-expression*) <…WHEN (*when-expression*)> *result-expression*

<…WHEN (*when-expression*) <…WHEN (*when-expression*)> *result-expression*>

OTHERWISE <*default-result-expression*>

END

The SELECT expression evaluates each WHEN expression in order until a matching expression is found. Then the associated *result-expression* is evaluated as the result of the SELECT expression.

The SELECT expression can be used wherever any other expression can be used. Here is an example.

```
r = 25.5 + select (t) when (1) -a when (3) b*2 end;
```

SELECT Expression with a Selection Expression

If a selection expression is present, then it is evaluated. Then the WHEN expressions are evaluated in order. The result of the SELECT expression is the result expression of the first WHEN expression that evaluates to the same value as the selection expression. If all the WHEN expressions evaluate to different values than the selection expression, the result of the SELECT expression is the default result expression if present. Otherwise, it is a missing or null value. Here is an example.

```
s = select (t)
   when (1) x*10
   when (3) x
   when (5) x*100
   when (0) 0
   otherwise .
end;
```

If `t` is 5, then the SELECT expression evaluates to 'x*100'.

SELECT Expression without a Selection Expression

If a selection expression is not present, then the WHEN expressions are evaluated in order. The result of the SELECT expression is the result expression of the first WHEN expression that evaluates to true (a nonzero value). If all the WHEN expressions evaluate to false (zero), the result of the select expression is the default result expression if present. Otherwise, it is a missing or null value. Here is an example.

```
grade = select
  when (score >= 90) 'A'
  when (score >= 80) 'B'
  when (score >= 70) 'C'
  when (score >= 60) 'D'
  when (score >= 0 ) 'F'
end;
```

If **score** is 76, then the first *when-expression* to evaluate to true is **score>=70**. The *select-expression* evaluates to 'C'.

Optional Otherwise Expression

If an otherwise default result expression is not supplied, then DS2 provides a default result value to select when none of the WHEN expressions are selected. If the SELECT expression has type DOUBLE or CHAR in SAS mode, the default result value is a missing value (.). For all other data types in either mode, the default result value is NULL.

Result Expression with Multiple When Expressions

Multiple WHEN expressions can be associated with a single result expression. The WHEN expressions are listed consecutively followed by the single result expression. If any of the WHEN expressions associated with a result expression is the first matching WHEN expression, then the result of the SELECT expression is the result expression.

For example, the following SELECT expression evaluates to 'airplane' if the value of variable **t** is either 'A', 'a', 'P', or 'p'.

```
s = select (t)
   when ('A')
   when ('a')
   when ('P')
   when ('p') 'airplane'
   when ('C')
   when ('c') 'car'
   when ('T')
   when ('t') 'train'
   otherwise  'walk'
end;
```

SELECT Expression Data Type

The type of a SELECT expression is determined by examining the type of the first result expression. If the first result expression is not a numeric data type, then the SELECT expression is assigned the type of the first result expression.

If the first result expression is a numeric data type, then all the result expressions are examined to find the widest numeric data type to assign as the type of the SELECT expression.

In the following example, **t** is a TINYINT, **b** is a BIGINT, **d** is a DECIMAL(10,5), and **s** is a CHAR(10). The select expression is assigned the type DECIMAL(10,5), the widest numeric type of TINYINT, BIGINT, DECIMAL(10,5), and CHAR(10).

```
r = select (t)
   when (1) t
   when (2) b
   when (3) d
   otherwise s
end;
```

Note: If **s** had been assigned to the first result expression, then the type of the SELECT expression would have been CHAR(10). If the first result expression has a non-numeric type, then the non-numeric type is assigned as the type of the SELECT expression.

Lazy Evaluation of the SELECT Expression

The SELECT expression uses lazy evaluation for the WHEN expressions. The WHEN expressions are evaluated in order until a matching WHEN expression is found or all when expressions are evaluated. If the SELECT expression has *n* WHEN expressions and the *i*th WHEN expression is selected, then only the first 1 to *i* WHEN expressions are evaluated. The *i*+1 to n when expressions are not evaluated.

Lazy evaluation also applies to the result expressions of the SELECT expression. The selected result expression is the only result expression evaluated.

In the following example, if **n[i]** equals 0, then only the first two WHEN expressions (**n[i] < 0** and **n[i] = 0**) are evaluated and only the second result expression **(y*10-r2)** is evaluated.

```
m(select
    when (n[i] < 0) y*100-r1
    when (n[i] = 0) y*10-r2
    when (n[i] > 0) y*10
 end);
```

Operators in Expressions

Operator Precedence

An operator symbolizes a type of operation that is to be performed on an operand, such as addition, comparison, and logical negation. When an expression contains multiple operators and operands, DS2 resolves the expression by using operator precedence. Operations are performed from the highest order of precedence to the lowest order of precedence.

The highest order of precedence is 1 and the lowest order of precedence is 9. Within a precedence level, with the exception of exponentiation, minimum, and maximum operators, operators associate from left to right. The exponentiation, minimum, and maximum operators associate from right to left.

By using the precedence order in Table 9.5 on page 69, in the expression **5+a**b*3**, **a**b** is calculated first and then multipled by **3**, and that result is added to **5**.

TIP In DS2, x < y < z is evaluated like x < y and y < z.

The following table lists the operators and their order of precedence:

Table 9.5 *Operators and Their Order of Precedence in DS2 Expressions*

Order of Precedence	Symbol	Associativity
1	( )	left to right
1	SELECT expression	left to right
2	+, –	right to left
2	^ or ~	left to right
2	**, <>, ><	left to right
3	*, /	left to right
4	+, –	left to right
5	|| or !!	left to right
5	..	left to right
6	IN, LIKE	left to right
7	=, ^= or ~=	right to left
7	>=, <=, >, < *	left to right
8	&	left to right
9	| or !	left to right
10	IF expression	right to left
none	:=	none
none	_NEW_	none

* In DS2, x < y < z is evaluated like x < y and y < z.

Expression Values by Operator

The following table shows the resolved value for expressions that are based on an operator:

Table 9.6 *Expression Values by Operator*

Operator	Value
Unary expressions	
Unary plus	Is the same as the expression operand
Unary minus	Is the arithmetic negation of the operand
NOT or ^ or ~	If the operand is nonzero, result is 0. If the operand is zero or missing, result is 1. If the operand is null, result is null.
Logical expressions	
OR or | or !	• the logical OR of the two operands • null when one operand is zero or missing and the other operand is null • null when both operands are null • 1 when either operand is nonzero (even if the other operand is null or missing) • 0 when both operands are zero or missing
AND or &	• the logical AND of the two operands • null when one operand is nonzero and the other operand is null • null when both operands are null • 1 when both operands are nonzero • 0 when either operand is zero or missing (even if the other operand is null)
Arithmetic expressions	
+	the arithmetic sum of the operands
–	the arithmetic difference of the operands
*	the arithmetic product of the operands
/	the arithmetic quotient of the operands
**	the left operand raised to the power of the right operand
><	the minimum of the left and right operands
<>	the maximum of the left and right operands
any arithmetic operator	null when either or both operators are null
Relational expressions	
<	1 when the left operand is less than the right operand; otherwise, 0

Operator	Value
>	1 when the left operand is greater than the right operand; otherwise, 0
<=	1 when the left operand is less than or equal to the right operand; otherwise, 0
>=	1 when the left operand is greater than or equal to the right operand; otherwise, 0
=	1 when the left operand is equal to the right operand; otherwise, 0
^=	1 when the left operand is not equal to the right operand; otherwise, 0
any logical operator	null when either or both operators are null
***Concatenation expression**	
|| or !!	the left operand merged with the right operand to form a new value
..	strips each argument before concatenating
IF expression	
if	1 when the comparison expression is contained in the constant list
IN expression	
in	1 when the comparison expression is contained in the constant list
LIKE expression	
LIKE	1 when the comparison expression is contained in the constant list
SELECT expression	
SELECT	1 when the comparison expression is contained in the constant list

***** The || concatenation operator does not remove any spaces. You can use the TRIM function to remove trailing spaces. However, the .. operator performs the same function as TRIM followed by concatenation. It is faster to use the .. operator (a .. b) than to use the TRIM function (TRIM(a) || TRIM(b)).

Chapter 10

Dates and Times in DS2

DS2 Dates, Times, and Timestamps

Overview of DS2 Dates, Times, and Timestamps

DS2 supports the SQL style date and time conventions that are used in other data sources. When your data source is a not a SAS data set, DS2 can process dates and times that have a data type of DATE, TIME, and TIMESTAMP.

Date and time values with a data type of DATE, TIME, and TIMESTAMP can be converted to a SAS date, time, or datetime value. SAS date, time, or datetime values can be converted to a value having a DATE, TIME, or TIMESTAMP data type.

DS2 provides date and time functions that convert any date or time value to SAS date, time, and datetime values, and back again to a recognizable date or time value. For more information, see “Date, Time, and Datetime Functions” on page 77.

The date and time intervals that are supported in ANSI SQL are not supported in DS2.

Declaring Date, Time, and Timestamp Variables

You declare a date, time, or timestamp variable by using the DATE, TIME, or TIMESTAMP data types in the DECLARE statement, as in this example:

```
dcl date dt;
dcl time tm;
dcl timestamp tmstmp;
```

Note: If you use a precision when you declare a time or timestamp variable, the time or timestamp values are not rounded to the specified precision until they are generated by the DATA statement. Internally, the time or timestamp constant values are simply copied to the time or timestamp variable.

Note: If you are working with TIME and TIMESTAMP values in a data source other than SAS and you do not specify a precision, the default precision is the DS2 default precision of 0 for TIME and 6 for TIMESTAMP.

For additional information about the DS2 date and time data types, see Chapter 5, "DS2 Data Types," on page 33.

DS2 Date, Time, and Timestamp Values

Once you declare a date, time, or timestamp variable, the value of the variable can be only a DS2 date, time, or timestamp constant that has the following syntax:

DATE'*yyyy-mm-dd*'

TIME'*hh*:*nn*:*ss*[.*fraction*]'

TIMESTAMP'*yyyy-mm-dd hh*:*nn*:*ss*[.*fraction*]'

where

- *yyyy* is a four-digit year
- *mm* is a two-digit month, 01–12
- *dd* is a two-digit day, 01–31
- *hh* is a two-digit military hour, 00–23
- *nn* is a two-digit minute, 00–59
- *ss* is a two-digit second, 00–60
- *fraction* can be one to nine digits, 0–9, is optional, and represent a fraction of a second

The string portion of the value after the DATE, TIME, or TIMESTAMP keyword must be enclosed in single quotation marks.

In the date constant, the hyphens are required and the length of the date string must be 10.

In the time constant, the colons are required. If the fraction of a second is not present, the time string must be 8 characters long and exclude the period. DS2 issues an error if the period is present without a fraction. If the fraction of second is present, the fraction can be up to 9 digits long and the time string can be up to 18 characters long (including the period).

In the timestamp constant, the hyphens in the date are required as well as the colons in the time. If the fraction of a second is not present, the timestamp string must be 19 characters long and exclude the period. If the fraction of a second is present, the fraction can be up to 9 digits long and the timestamp string can be up to 29 characters long.

Here are some examples of DS2 date, time, and timestamp constants:

```
date'2012-01-31'
time'20:44:59'
timestamp'2012-02-07 07:00:00.7569'
```

Operations on DS2 Dates and Times

The only operations that can be performed on DATE, TIME, and TIMESTAMP values are operations that use the relational operators <, >, <=, >=, =, ^=, and IN, such as in the following statement:

```
if tm in(time'10:22:31', time'12:55:01') then
   if tm < time'13:30:00' then put 'Early afternoon';
   else put 'Time not available';
```

DS2 does not calculate date and time intervals on values that have the data types of DATE, TIME, and TIMESTAMP.

SAS Date, Time, and Datetime Values

A SAS date value is the number of days between January 1, 1960 and a specified date. Dates before January 1, 1960 are negative numbers; dates after are positive numbers. For example, the SAS date value for January 1, 1960 is 0, -365 for January 1, 1959, and 17532 for January 1, 2008.

A SAS time value is the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.

A SAS datetime value is the number of seconds between January 1, 1960 and a specific hour, minute, and second of a specific date.

When a numeric column is read from a SAS data set and the numeric column has a SAS date, time, or datetime format associated with it, the column is converted to a DS2 type DATE, TIME, or TIMESTAMP. If the numeric column in a SAS data set does not have a format or has a format that is not a SAS date, time, or datetime format, the column is processed as type DOUBLE.

All calculations on dates and times are done as a SAS date value, a SAS time value, or a SAS datetime value. For more information, see "Date, Time, and Datetime Functions" on page 77.

After calculations are complete, there are other functions that can then format the SAS date, time, and datetime values to recognizable date and time formats.

Converting SAS Date, Time, and Datetime Values to a DS2 Date, Time, or Timestamp Value

SAS date, time, and datetime values can be converted to DS2 dates, time, and timestamp values by using the TO_DATE, TO_TIME, and TO_TIMESTAMP functions. The argument of these functions is any value or expression that represents a SAS date, time, or datetime value and has a type DOUBLE. You can then use either the PUT statement or a format in the DECLARE statement to format the date, time, or timestamp value.

Here is an example.

```
data _null_;
   dcl date ds2d having format YYMMDD10.;
   dcl time ds2t having format TIME18.9;
```

```
   dcl timestamp ds2dt having format DATETIME28.9;
   dcl double d t ts;
   method init();
      d = 19358;
      ds2d = to_date(d);
      ds2t= to_time(d);
      ds2dt= to_timestamp(d);
      put ds2d ds2t ds2dt;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
2012-12-31
 5:22:38.000000000
01JAN1960:05:22:38.000000000
```

For more information, see the "TO_DATE Function" in *SAS Viya: DS2 Language Reference*, the "TO_TIME Function" in *SAS Viya: DS2 Language Reference*, and the "TO_TIMESTAMP Function" in *SAS Viya: DS2 Language Reference*.

Converting DS2 Date, Time, and Timestamp Values to SAS Date, Time, or Datetime Values

DS2 date, time, and timestamp values can be converted to a SAS datetime value by using the TO_DOUBLE function. This function converts the date, time, or timestamp CHAR or NCHAR string to a SAS datetime value with a data type of DOUBLE. You can then use any DS2 format to display the value in a date, time, or datetime format.

The following DS2 program illustrates how you can convert a DS2 timestamp to a SAS date, time, and datetime values:

```
data _null_;
    method run();
       dcl timestamp DS2ts;
       dcl double sasdtval sasd sastm;
       dcl char(28) fmtdate fmttime fmtdt;
       DS2ts = timestamp '2012-06-04 10:54:34.012';
       put DS2ts;
       sasdtval = to_double(DS2ts);
       sasd = datepart(sasdtval);
       sastm = timepart(sasdtval);
       put sasdtval:best16.7;
       put sasd:best.;
       put sastm:best.;
       fmtdate = put(sasd, yymmdd10.);
       fmttime = put(sastm, time.);
       fmtdt = put(sasdtval, datetime21.7);
       put fmtdate=;
       put fmttime=;
         put fmtdt=;
    end;
enddata;
```

```
run;
```

The following output is written to the SAS log:

```
2012-06-04 10:54:34.012000000
1654426474.012
19148
39274.012
fmtdate=2012-06-04
fmttime=10:54:34
fmtdt=04JUN12:10:54:34.0120
```

In this example, a SAS date value is formatted to look like DS2 date value, but it has a data type of DOUBLE and not DATE. SAS date and time values cannot be assigned to DS2 date or time variables. Their data types are different. If you attempt to assign a SAS date or time value to a DS2 date or time variable, DS2 issues a data type invalid conversion error.

For more information, see the "PUT Function" in *SAS Viya: DS2 Language Reference*. For a complete list of formats, see "Date, Time, and Datetime Formats" on page 78.

Date, Time, and Datetime Functions

In order to perform date and time calculations, DS2 date and time functions do the following:

- convert a date or time into a SAS date, time, or datetime value
- convert a SAS date, time, or datetime value into a recognizable date or time
- extract a date or a time from a SAS datetime value

The following tables list the date and time functions and what they do. For specific information about any of these functions, see "DS2 Functions" in *SAS Viya: DS2 Language Reference*.

Table 10.1 *Functions That Convert Dates and Times into a SAS Date, Time, or Datetime Value*

Function	Description
DATE or TODAY	Returns the current date as a SAS date value
DATEJUL	Converts a Julian date to a SAS date value
DHMS	Returns a SAS datetime value from date, hour, minute, and second values
HMS	Returns a SAS time value from hour, minute, and second values
MDY	Returns a SAS date value from month, day, and year values
TIME	Returns the current time of day as a SAS time value.
YYQ	Returns a SAS date value from a year and quarter year values

Table 10.2 *Functions That Format a SAS Date, Time, or Datetime Value as a Recognizable Date or Time*

Function	Description
DAY	Returns the day of the month from a SAS date value
HOUR	Returns the hour from a SAS time or datetime value
JULDATE	Returns the Julian date from a SAS date value
JULDATE7	Returns a seven-digit Julian date from a SAS date value
MINUTE	Returns the minute from a SAS time or datetime value
MONTH	Returns a number that represents the month from a SAS date value
QTR	Returns the quarter of the year from a SAS date value
SECOND	Returns the second from a SAS time or datetime value
WEEKDAY	Returns an integer that corresponds to the day of the week, from a SAS date value
YEAR	Returns the year from a SAS date value

Table 10.3 *Functions That Extract Date and Times from SAS Date, Time, and Datetime Values*

Function	Description
DATEPART	Extracts the date from a SAS datetime value and returns the date as a SAS date value
TIMEPART	Extracts the time from a SAS datetime value and returns the time as a SAS datetime value

Date, Time, and Datetime Formats

DS2 formats write SAS date, time, and datetime values as recognizable dates and times. You use the PUT function to format a SAS date, time, or datetime value:

PUT(*sasDateOrTime*, *format*.);

The first argument to the PUT function is the SAS date, time, or datetime. The second argument is the format.

See “Converting DS2 Date, Time, and Timestamp Values to SAS Date, Time, or Datetime Values” on page 76 for an example of formatting dates and times in a DS2

program. The following table displays the results of formatting the date March 17, 2012 for each of the DS2 formats.

Table 10.4 *Examples of DS2 Date and Time Formats*

Type of Language Element	Language Element	Input	Result
Date formats	DATE.	19069	17MAR12
	DATE9.	19069	17MAR2012
	DAY.	19069	17
	DDMMYY.	19069	17/03/12
	DDMMYY10.	19069	17/03/2012
	DDMMYYB.	19069	17 03 12
	DDMMYYB10.	19069	17 03 2012
	DDMMYYC.	19069	17:03:12
	DDMMYYC10.	19069	17:03:2012
	DDMMYYD.	19069	17-03-12
	DDMMYYD10.	19069	17-03-2012
	DDMMYYN6.	19069	170312
	DDMMYYN8.	19069	17032012
	DDMMYYP.	19069	17.03.12
	DDMMYYP10.	19069	17.03.2012
	DDMMYYS.	19069	17/03/12
	DDMMYYS10.	19069	17/03/2012
	DOWNAME.	19069	Monday
	JULIAN.	19069	12077
	MMDDYY.	19069	03/17/12
	MMDDYY10.	19069	03/17/2012
	MMDDYYB.	19069	03 17 12

Type of Language Element	Language Element	Input	Result
	MMDDYYB10.	19069	03 17 2012
	MMDDYYC.	19069	03:17:12
	MMDDYYC10.	19069	03:17:2012
	MMDDYYD.	19069	03-17-12
	MMDDYYD10.	19069	03-17-2012
	MMDDYYN6.	19069	031712
	MMDDYYN8.	19069	03172012
	MMDDYYP.	19069	03.17.12
	MMDDYYP10.	19069	03.17.2012
	MMDDYYS.	19069	03/17/12
	MMDDYYS10.	19069	03/17/2012
	MMYY.	19069	03M2012
	MMYYC.	19069	03:2012
	MMYYD.	19069	03-2012
	MMYYN.	19069	032012
	MMYYP.	19069	03.2012
	MMYYS.	19069	03/2012
	MONNAME.	19069	March
	MONTH.	19069	3
	MONYY.	19069	MAR2012
	WEEKDATE.	19069	Monday, March 17, 2012
	WEEKDATX	19069	Monday, 17 March 2012
	WEEKDAY.	19069	2
Quarter formats	QTR.	19069	1
	QTRR.	19069	I

Type of Language Element	Language Element	Input	Result
Datetime formats	DATEAMPM.	19069	01JAN60:04:53:28 AM
	DATETIME.	19069	01JAN60:04:53:28
	DTDATE.	19069	01JAN60
	DTMONYY.	19069	JAN60
	DTWKDATX.	19069	Friday, 1 January 1960
	DTYEAR.	19069	1960
	DTYYQC.	19069	60:1
Time formats	HOUR.	19069	5
	TIME.	19069	4:53:28
	TIMEAMPM.	19069	4:53:28 AM
	TOD.	19069	04:53:28
Year formats	YEAR.	19069	2012
	YYMM.	19069	2012M03
	YYMMC.	19069	2012:03
	YYMMD.	19069	2012-03
	YYMMN.	19069	201203
	YYMMP.	19069	2012.03
	YYMMS.	19069	2012/03
	YYMMDD.	19069	12-03-17
	YYMMDDB.	19069	12 03 17
	YYMMDDC.	19069	12:03:17
	YYMMDDD.	19069	12–03–17
	YYMMDDN.	19069	20120317
	YYMMDDP.	19069	12.03.17
	YYMMDDS.	19069	12/03/17

Type of Language Element	Language Element	Input	Result
	YYMON.	19069	2012MAR
Year/Quarter formats	YYQ.	19069	2012Q1
	YYQC.	19069	2012:1
	YYQD.	19069	2012-1
	YYQP.	19069	2012.1
	YYQS.	19069	2012/1
	YYQN.	19069	20121
	YYQR.	19069	2012QI
	YYQRC.	19069	2012:I
	YYQRD.	19069	2012-I
	YYQRP.	19069	2012.I
	YYQRS.	19069	2012/I
	YYQRN.	19069	2012I
	YYQZ.	19069	1201

Chapter 11

DS2 Arrays

Overview of DS2 Arrays

In DS2, an array is a named aggregate collection of homogeneous data. DS2 has two types of arrays: temporary and variable. These arrays have the following characteristics.

- homogeneous by type
- multidimensional (the number of bounds can be >=1)
- indexed by signed integer values
- exists only for the duration of the DS2 program or DS2 procedure

- not a DS2 variable in the PDV, though the elements of a variable array can refer to variables in the PDV
- array elements do not appear in a result table, though variables referenced by elements of a variable array can appear in the results table

The following table shows some of the differences between temporary and variable arrays.

Temporary Array	Variable Array
set of temporary elements	set of references to variables in the PDV
created with a DECLARE statement	created with a VARARRAY statement
can be declared in local or global scope	must be declared in global scope
similar to arrays seen in other languages	similar to arrays of pointers or references seen in other languages
in DATA step, created with an ARRAY statement with _TEMPORARY_ argument	in DATA step, created with an ARRAY statement

Temporary Arrays

Overview of Temporary Arrays

The elements of a temporary array are temporary in that they are not located in the PDV and therefore do not appear in any result table. Temporary data element values are automatically retained across iterations rather than being reset to missing at the beginning of the next iteration. Temporary arrays exist only for the duration of the DS2 program.

You use the DECLARE statement to specify the name, data type, and number and size of the array bounds. You can also use a HAVING clause in the DECLARE statement to associate label, format, and informat attributes with a temporary array. For example, the following DECLARE statement specifies a three-element temporary array that stores three temporary double values outside the PDV.

```
declare double a[3]
```

Temporary Array Declaration

Temporary array declarations are similar to scalar declarations. In addition to the data type and name, you can also specify the number and size of the array bounds. Multiple bounds (or dimensions) are specified using comma separators.

The form of signed integer pairs specifies the lower and upper bounds for each dimension of the array [*l*:*h*], where *l* represents the lowest index for the given bound and *h* represents the highest index for the given bound. The lower bound specification, *l*, is optional. If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

An error is returned if the upper bound, *h*, is less than the lower bound. If you specify an array bound with only one integer, then that integer is interpreted as the upper bound. The default lowest bound is 1.

The upper bound of an array can also be sized based on the number of elements in a dimension of a previously declared array. You use a DIM function call for the upper bound. The DIM function is the only function that can be used to specify an upper array bounds. The DIM function cannot be used to specify the lower bound of a dimension.

The part of the DECLARE statement for temporary array declaration is as follows.

DECLARE *data-type* <variable-list> [*having-clause*];

<variable-list>::=<variable> […<variable>]
<variable>::=*identifier* <array-declaration>
<array-declaration>::=\[<array-bound>[, …<array-bound>]\]
<array-bound>::= {[*dim-lower*:]*dim-upper*} | {[*dim-lower*:] {DIM(*a*[, *n*]) | *}

For more information, see "DECLARE Statement" in *SAS Viya: DS2 Language Reference*.

Variable Arrays

Overview of Variable Arrays

Variable arrays are a way to simplify processing of a series of variables that have a similar name or purpose in the input data. The elements of a variable array refer to variables in the PDV. Variable arrays exist only for the duration of the DS2 program. However, the content of the referenced variables might be preserved in one or more result sets.

You use the VARARRAY statement to specify the name, data type, and number and size of the array bounds. For example, the following VARARRAY statement specifies a three-element variable array that refers to three double variables (a1, a2, a3) in the PDV.

```
vararray double a[3];
```

The VARARRAY statement in the previous example creates any of the double variables (a1, a2, a3) that have not previously been created. Variable array element a[1] references variable a1, a[2] references variable a2, and a[3] references variable a3.

After a variable array is created, the variable array elements act as a second set of identifiers that can be used to read or modify the data that is stored in the referenced variables in the PDV.

You can also use a HAVING clause in the VARARRAY statement to associate label, format, and informat attributes with a variable array.

For more information, see Chapter 11, "DS2 Arrays," on page 83 and the "VARARRAY Statement" in *SAS Viya: DS2 Language Reference*.

Variable Array Declaration

Variable array declarations are similar to scalar declarations. In addition to the data type and name, you can also specify the number and size of the array bounds. Multiple bounds (or dimensions) are specified using comma separators.

Array bounds have two forms.

signed integer pairs
: The form of signed integer pairs specifies the lower and upper bounds for each dimension of the array. [*l*:*h*], where *l* represents the lowest index for the given bound and *h* represents the highest index for the given bound. The lower bound specification, *l*, is optional. If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

  An error is returned if the upper bound, *h*, is less than 1. If you specify an array bound with only one integer, then that integer is interpreted as the upper bound. The default lowest bound is 1.

  `vararray double a[5];` declares an array a of type double, with five elements indexed from 1 to 5. **`vararray char b[5,10];`** declares a two dimensional character array b with 5 elements in the first dimension and 10 elements in the second dimension for a total of 50 elements in the array. **`vararray int c[3] x y z;`** declares an array c with three elements. The array is indexed with a lower bound of 1 and an upper bound of 3.

  The upper bound of an array can also be sized based on the number of elements in a dimension of a previously declared array. You use a DIM function call for the upper bound. The DIM function is the only function that can be used to specify an upper array bound. The DIM function cannot be used to specify the lower bound of a dimension.

* (asterisk)
: The * form specifies a one-dimensional array in which the lower bound is 1 and the upper bound is the number of variables in the variable list.

For more information, see "Variable Lists" on page 21. For more information about how to declare variable arrays and how to specify multiple bounds, see the "VARARRAY Statement" in *SAS Viya: DS2 Language Reference*.

Definition of Variables in a VARARRAY Statement

A VARARRAY statement defines any variable in its variable list that has not previously been defined. Some variable list types reference only existing variables and therefore do not result in the definition of new variables. The following table shows which variable list types can define new variables and which variable list types reference only existing variables.

Variable List	Example	Variable Expansion	Description
name	`x y z`	immediate	can define new variables
numbered range	`x1-x5`	immediate	can define new variables
name range	`sales_jan--sales_mar`	delayed	reference only existing variables
name prefix	`sales:`	delayed	reference only existing variables
type	`smallint`	delayed	reference only existing variables
special name	`_all_`	delayed	references only existing variables

The name and numbered range variable lists can be expanded without examining all the variables in the DS2 program. Therefore, these types of variable lists are expanded immediately when the VARARRAY statement is encountered in the program.

The other types of variable lists must examine all variables that are defined in the DS2 program. Expansion of these variable lists is delayed until after all statements in the DS2 program have been examined and all variables have been defined.

This delay can lead to some unexpected error conditions. For example, consider the following program.

```
data;
   1 vararray int x[1] x:;

   method run();
      2 x1 = 5.0;
   end;
enddata;
run;
```

1 The VARARRAY statement does not create any variables because the prefix variable list x: references only existing variables. The expansion of prefix variable list x: is delayed until all statements in the program have been examined.

2 In the assignment statement variable x1 is undefined. Therefore, the assignment statement assigns the type of the right hand side value (DOUBLE) to variable x1.

After all the program statements are examined, the prefix variable list x: is expanded to x1, the only existing variable with prefix x. The DS2 compiler then issues a compilation error because variable x1 of type DOUBLE is incompatible with variable array x of type INTEGER.

One way to remove the error condition is to change the VARARRAY statement, **`vararray int x[1] x:;`**, to **`vararry int x[1] x1;`** or **`vararray int x[1];`**. The revised statement defines the variable x1 as type INTEGER.

Delayed Variable Definition with DIM Variable Array Bounds

If a variable array has an upper dimension bound based on the dimension of another array, then the definition of variables for the array can be delayed until all statements in the program have been examined. Here is an example.

```
data;
   1 vararray double x[*] x:;
   2 vararray int out[dim(x)];

   method init();
   3 out1 = 0.0;
   end;

   method run();
      set in;
   end;
enddata;
run;
```

1 The expansion of prefix variable list x: is delayed until all statements in the program have been examined. Therefore, the size of variable array x is not known until all statements have been examined.

2 The out variable array has the default variable list out1-out*n*, where *n* is the number of elements specified for the variable array. The determination of the number of elements in out is delayed until the size of array x is known which was delayed until all statements have been examined.

3 In the assignment statement, variable out1 is undefined. Therefore, the assignment statement assigns the type of the right hand side value, DOUBLE, to variable out1.

After all statements in the above DS2 program are processed, the following occurs. Assume the table in has 3 double variables, x1 x2 x3.

- The prefix variable list x: is expanded to x1 x2 x3.
- The size of variable array x is determined to be 3.
- The size of variable array out is determined to be 3 (dim(x)).
- The default variable list out1-out3 is expanded to out1 out2 out3.
- The variables out2 and out3 are defined as type INTEGER because they were not previously defined. Note that out1 was defined as type DOUBLE by the assignment statement **`out1 = 0.0;`**.

The DS2 compiler issues a compilation error because variable out1 of type DOUBLE is incompatible with variable array out of type INTEGER.

One way to remove the error condition is to change the array assignment, **`out1 = 0.0;`**, to **`out[1]=0.0;`**. This change updates the out1 data value by means of an out array reference to prevent the assignment statement from assigning type DOUBLE to the variable out1.

Declaring Arrays with a HAVING Clause

The declaration statement for a temporary or variable array can contain a HAVING clause. The HAVING clause associates a label, format, and informat attribute with the array. If the array is a variable array, then the HAVING clause is also associated with the variables referenced by the variable array.

The decision about when to apply the HAVING clause to a variable that is referenced by the variable array depends on when the variable list that contains the variable reference is expanded. Name and numbered range variable lists are normally expanded when the VARARRAY statement is processed. Therefore, the HAVING clause is applied to all variables referenced by these lists at that time. Name range, name prefix, type, and special name variable lists are expanded after all statements in the program have been examined and all variables in the program have been defined. Consequently, the HAVING clause for all variables referenced by these variable lists is applied after all statements in the DS2 program have been examined.

Consider the following program.

```
data;
1  declare double x1 x2 having format 5.0;
2  vararray double x[3] having format 5.2;
3  declare double x3 having format ROMAN5.;
enddata;
```

1 The DECLARE statement is processed. Variables x1 and x2 are defined. The HAVING clause **`format 5.0`** is associated with variables x1 and x2.

Variable x1 x2

Format	5.0	5.0

2 The VARARRAY statement is processed. Default variable list x1-x3 is expanded to x1 x2 x3. Variable x3 is defined. The HAVING clause **`format 5.2`** is associated with variables x1 x2 x3.

Variable	x1	x2	x3
Format	5.2	5.2	5.2

3 The DECLARE statement is processed. The HAVING clause **`format ROMAN5.`** is associated with variable x3.

Variable	x1	x2	x3
Format	5.2	5.2	ROMAN5.

Now consider what happens if the variable list type is modified to a type that results in delayed processing of the variable list.

```
data;
1  declare double x1 x2 having format 5.0;
2 4  vararray double x[*] x: having format 5.2;
3  declare double x3 having format ROMAN5.;
enddata;
```

1 The DECLARE statement is processed. Variables x1 and x2 are defined. The HAVING clause **`format 5.0`** is associated with variables x1 and x2.

Variable	x1	x2
Format	5.0	5.0

2 The VARARRAY statement processing is delayed or begins processing. Prefix variable list x: cannot be expanded until all statements in the program have been examined and all variables are defined.

Variable	x1	x2
Format	5.0	5.0

3 The DECLARE statement is processed. Variable x3 is defined. The HAVING clause **`format ROMAN5.`** is associated with variable x3.

Variable	x1	x2	x3
Format	5.0	5.0	ROMAN5.

4 The VARARRAY statement completes processing. Prefix variable list x: is expanded to x1 x2 x3. The HAVING clause **`format 5.2`** is associated with variables x1 x2 x3.

Variable	x1	x2	x3
Format	5.2	5.2	5.2

Array Assignment

Overview of Array Assignment

DS2 supports array assignment with the := operator. The syntax for assigning an array or constant list is as follows:

array:=*array*;

array:=(*constant list*);

In an array assignment, *array* can be either a temporary or variable array.

Array Assignment from Another Array

When you assign one array to another array, the data types of the two arrays must be compatible (either the same or convertible). The number of dimensions and the total number of elements in each dimension do not have to be the same.

Consider the assignment from array y to array x as shown in this statement.

```
x:=y;
```

During the assignment, each element of array y is assigned to each element of array x, for example, **`x[1]=y[1];...x[n]=y[n];`**

The basic algorithm for evaluating x[*i*] = y[*i*] follows. First y[*i*] is examined to see whether it is missing (SAS mode) or null (ANSI mode).

- If y[*i*] is missing or null, then missing or null is assigned to x[*i*].[1]
- If y[*i*] is not missing or null, then the types of x[*i*] and y[*i*] are examined.
  - If the type of y[*i*] is different from the type of x[*i*], then y[*i*] is converted to the type of x[*i*].
    - If the conversion of y[*i*] succeeds, then the result of the conversion is assigned to x[*i*].
    - If the conversion of y[*i*] fails, then missing or null is assigned to x[*i*].
  - If the type of y[*i*] is the same as the type of x[*i*], then y[*i*] is assigned to x[*i*].

If array x and array y do not have the same number of elements, then as many elements as possible are assigned from array y to array x and null or missing is assigned to any remaining elements in array x. The length of array x is not modified by the assignment.

In the following example, array **`x`** has ten elements, array **`y`** has seven elements, and array **`y`** is assigned to array **`x`**. Therefore, the seven elements from array **`y`** are assigned to the first seven elements of array **`x`**, and missing is assigned to the last three elements of array **`x`**.

```
data _null_;
   method init();
      declare double x[10];
      declare double y[7];
```

1 The decision to assign missing or null to an array element depends on data type of the array and whether the program is running in SAS or ANSI mode. For more information, see Chapter 7, "How DS2 Processes Nulls and SAS Missing Values," on page 43.

```
      x := (0 0 0 0 0 0 0 0 0 0);
      y := (1 2 3 4 5 6 7);
      x := y;
      put x[*]=;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
x[1]=1 x[2]=2 x[3]=3 x[4]=4 x[5]=5 x[6]=6 x[7]=7 x[8]=. x[9]=. x[10]=.
```

Special Case for Double Missing Values

If y[*i*] is a DOUBLE with a SAS missing value, for example, .Z, then DS2 tries to preserve the SAS missing value during the assignment according to these rules:

- If x[*i*] is a DOUBLE, then the SAS missing value from y[*i*] is assigned to x[*i*].
- If x[*i*] is a character string, then the missing character representation of y[*i*] (for example, Z for .Z, is assigned to x[*i*]).

This special case processing occurs only when DS2 is in SAS mode and the type of y[*i*] is a DOUBLE.

For more information about null and missing values, see Chapter 7, "How DS2 Processes Nulls and SAS Missing Values," on page 43. For an example of an array assignment, see "Example: Arrays" in *SAS Viya: DS2 Language Reference*.

Array Assignment with Variable Arrays

The elements of a variable array reference variables in the PDV. In the array assignment statement x := y, the value of the elements of y are assigned elementwise to the elements of x. If either x or y is a variable array (vararray), then the assignment is always x[i]=y[[i].

Array assignment to a variable array does not modify the elements (this is, the references) in the variable array. Instead, the data in the variables referenced by the elements of the array are modified. Similarly, in an array assignment from a variable array, the data in the variables referenced by the elements of the variable array are used for the assignment.

Array Assignment from a Constant List

To assign from a constant list to an array, the constants in the constant list must be compatible (either the same or convertible) with the data type of the array. The constant list and the array do not have to have the same number of dimensions or the same number of elements in each dimension.

Assume this array assignment statement.

```
x := (c1 c2 c3 ... cn);
```

During array assignment from a constant list to array x, each element of the constant is assigned to each element of array x as shown in the following expanded form.

```
x[1] = c1;
x[2] = c2;
```

```
...
x[i] = ci;
...
x[n] = cn;
```

The basic algorithm for evaluating x[*i*] = c*i* follows. First c*i* is examined to see whether it is missing or null.[1]

- If c*i* is missing or null, then missing (SAS mode) or null (ANSI mode) is assigned to x[*i*].
- If c*i* is not missing or null, then the types of x[*i*] and c*i* are examined.
  - If the type of c*i* is different from the type of x[*i*], then ci is converted to the type of x[*i*].
    - If the conversion of c*i* succeeds, then the result of the conversion is assigned to x[*i*].
    - If the conversion of c*i* fails, then missing or null is assigned to x[*i*].
  - If the type of c*i* is the same as the type of x[*i*], then c*i* is assigned to x[*i*].

If the constant list and the array x do not have the same number of elements, then as many constants as possible are assigned from the constant list to array x and a null or missing value is assigned to any remaining elements in array x. The length of array x is not modified by the assignment.

In the following example, a constant list having five constants is assigned to array **x** having seven double elements. The five constants in the constant list are assigned to the first five elements of array **x**, and missing is assigned to the last two elements of array **x**.

```
data _null_;
   method init();
     declare double x[7];
     x := (1 '2' 3.3 '' .Z);
     put x[*]=;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
x[1]=1 x[2]=2 x[3]=3.3 x[4]=. x[5]=Z x[6]=. x[7]=.
```

Note: The types of the elements in the constant list can be heterogeneous as long as all the types of the elements are convertible to the type of the assigned to array.

Here is another example of an array assignment from a constant list.

```
declare char(2) a[2, 3];
   ...
a := (('aa' 'bb' 'cc')('dd' 'ee' ''));
```

The elements in array **a** after the above assignment statement would look like this.

'aa'	'bb'	'cc'
'dd'	'ee'	''

[1] The decision to assign missing or null to an array element depends on data type of the array and whether the program is running in SAS or ANSI mode. For more information, see Chapter 7, "How DS2 Processes Nulls and SAS Missing Values," on page 43.

Array Arguments

Overview of Array Arguments

DS2 arrays can be passed as arguments to DS2 methods. DS2 arrays are always passed by reference to methods. DS2 arrays cannot be passed by value, that is, a copy of the array cannot be supplied as an argument to a method. DS2 array arguments must have the same type as specified by the array parameter in order to match the array parameter. Array arguments do not support implicit type conversion to a different type. DS2 arrays are passed as either a bounded array parameter, for example, **`a[8]`** or an unbounded array parameter, for example, **`a[*]`**.

Defining Array Parameters

A DS2 method can be defined with array parameters. The type of array (temporary or variable) is specified in the parameter definition. The following table illustrates the syntax for defining different types of parameters.

Type of Parameter	Parameter Syntax	Example
scalar parameter	*data-type parameter-name*	`double x`
temporary array parameter	*data-type parameter-name* [*bounds*]	`double x[5]`
variable array parameter	**VARARRAY***data-type parameter-name* [*bounds*]	`vararray double x[2,4]`

The data type and type of an array argument must exactly match the type and kind specified in the array parameter definition. DS2 does not convert array arguments to a different kind or data type. For example, if a parameter is defined as a temporary array of doubles, then the argument must be a temporary array of doubles. If a variable array of doubles or a temporary array of integers is passed as an argument for the temporary array of doubles parameter, then an error occurs.

The following DS2 program illustrates the definition of a method that has array parameters and illustrates calls to the method using array arguments.

```
data _null_;
   declare  double x;
   declare  double y[4];
   vararray double z[4];

   method m(double u, double v[4], vararray double w[4]);
      do i = 1 to 4;
         v[i] = u;
         w[i] = u;
      end;
   end;

   method init();
      m(x, y, z);      /* call method m */
```

```
      put y[*]='
      put z[*]=;
   end;
 enddata;
run;
```

Bounded Array Parameters

A bounded array parameter supplies explicit bounds information for accessing elements of the array argument. These are examples.

```
method m(double a[4]);

method m(vararray double a[5:10,3:6]);
```

A bounded array parameter matches any DS2 array argument with the same number of elements regardless of the dimensionality of the array argument. For example, bounded array parameter a[2,4] would match array arguments a1[8], a2[11:12,11:14], and a3[2,2,2] because arrays a1, a2, and a3 each have 8 elements. If the dimensionality of the array argument differs from the array parameter, then an element of the array parameter is mapped to the corresponding element in the array argument. This mapping is based on the position of the element in the array, using row-major order. For example, array parameter a1[2,1] accesses the fifth element of an 8-element array and thus would map to a1[5], a2[12,11], and a3[2,2,1].

Unbounded Array Parameters

An unbounded array parameter does not supply any explicit bounds information for the corresponding array argument. Here is an example.

```
method m(double a[*]);
```

The asterisk (*) for the array bounds specifies that parameter a is an unbounded array parameter.

An unbounded array parameter matches any DS2 array argument regardless of the number of elements or dimensionality of the array argument. In a DS2 method, the array parameter is treated as a one-dimensional array even if the corresponding array argument is a multi-dimensional array. The unbound array parameter is mapped to the multi-dimensional array using row-major order. Consider the 2x3 array **`a[2, 3]`**.

1 2 3
4 5 6

If array a is passed to a method as an unbounded array parameter b, then b will be accessed as a one-dimensional array of six elements.

1 2 3 4 5 6

Note that accessing an element of array parameter b results in the access of an element of array a, because array a is passed by reference to the DS2 method. Here is an example.

```
method m(double b[*]);
   b[1] = 10;          /* assigns 10 to a[1, 1] */
   b[2] = 20;          /* assigns 20 to a[1, 2] */
   b[3] = 30;          /* assigns 30 to a[1, 3] */
   b[4] = 40;          /* assigns 40 to a[2, 1] */
   b[5] = 50;          /* assigns 50 to a[2, 2] */
```

```
   b[6] = 60;          /* assigns 60 to a[2, 3] */
end;

method init();
   declare double a[2, 3];
   m(a);
end;
```

In an array expression of the form a[i], where a is an unbound array parameter, bounds checking of the index, i, is performed at run time. If an index of an array parameter is beyond the boundaries of the array argument, DS2 issues an error and the array expression will evaluate to NULL or missing.

Note: An unbounded array parameter cannot be used as an argument for a bounded array parameter.

How to Query Array Dimensions

The following functions can be used to obtain dimension information about an array. For more information about each function, see "DS2 Functions" in *SAS Viya: DS2 Language Reference*.

DIM(*a*)
Returns the number of elements in the first dimension of array *a*

DIM(*a*, *n*)
Returns the number of elements in dimension *n* of array *a*

LBOUND(*a*)
Returns the lower bound of the first dimension of array *a*

LBOUND(*a*, *n*)
Returns the lower bound of dimension *n* of array *a*

HBOUND(*a*)
Returns the upper bound of the first dimension of array *a*

HBOUND(*a*, *n*)
Returns the upper bound of dimension *n* of array *a*

NDIMS(*a*)
Returns the number of dimensions of array *a*

For any of the query functions, the array argument **a** can be a temporary array or a variable array, and the dimension argument **n** should be an expression that evaluates to an integral value. The following example illustrates these query functions.

```
do i = 1 to dim(a1);
   put a1[i];
end;

numelems = 0;
do i = 1 to ndims(a2);
   numelems = numelems + dim(a2, i);
end;

do i = lbound(a2, 1) to hbound(a2, 1);
   do j = lbound(a2, 2) to hbound(a2, 2);
```

```
      do k = lbound(a2, 3) to hbound(a2, 3);
         sum = sum + a2[i,j,k];
      end;
   end;
end;
```

If an array function is called with a dimension value outside the dimensions of the array, then a run-time error will occur and the function will return a NULL integer value.

How to Write Array Content

The DS2 PUT statement can be used to write individual elements of an array or all elements of an array.

The syntax to write an individual array element is as follows.

PUT *array-name*[*element*]<=>;

The syntax to write all elements of an array is as follows.

PUT *array-name*[*]<=>;

The PUT statement can write elements of temporary arrays and variable arrays. When all elements of an array are written with the ***array-name*[*]** syntax, all of the elements of the array are written with the same format. In other words, different formats cannot be specified for different array elements with ***array-name*[*]**.

The following example illustrates the put statement output the contents of an array:

```
data _null_;
   vararray varchar(10) x[10];
   declare  double      y[2,2,2];
   method init();
     x[1] = 'a';
     do i = 2 to dim(x);
       x[i] = x[i-1] || x[1];
     end;
     put 'X:' x[*];

     y := (10 20 30 40 50 60 70 80);
     put 'Y:' y[*]=;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
X: a aa aaa aaaa aaaaa aaaaaa aaaaaaa aaaaaaaa aaaaaaaaa aaaaaaaaaa
Y: y[1,1,1]=10 y[1,1,2]=20 y[1,2,1]=30 y[1,2,2]=40 y[2,1,1]=50 y[2,1,2]=60
y[2,2,1]=70 y[2,2,2]=80
```

For more information, see the "PUT Statement" in *SAS Viya: DS2 Language Reference*.

Chapter 12

DS2 Packages

Introduction to DS2 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.

There are two types of packages:

User-defined packages
: These are packages that you can use to store methods for any purpose.

  For more information, see "User-Defined Packages" on page 106.

Predefined packages
: These packages are predefined in DS2.

  For more information, see "Predefined DS2 Packages" on page 107.

  FCMP
  : Supports calls to FCMP functions and subroutines from within the DS2 language.

    Note: The FCMP package is not supported in the CAS server.

  Hash and hash iterator
  : Enables you to quickly and efficiently store, search, and retrieve data based on unique lookup keys.

  HTTP
  : Constructs an HTTP client to access HTTP web services.

  JSON
  : Enables you to create and parse JSON text.

  Logger
  : Provides a basic interface (open, write, and level query) to the SAS logging facility.

  Matrix
  : Provides a powerful and flexible matrix programming capability.

  SQLSTMT
  : Provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS.

    Note: The SQLSTMT package is not supported in the CAS server.

  TZ
  : Provides a way to process local and international time and date values.

To use a package, a DS2 program, another package, or a thread instantiates the package and accesses its methods. For a comparison between packages, DS2 programs, and threads, see "Block Statements" in *SAS Viya: DS2 Language Reference*.

Note: You can invoke a DS2 package method as a function in a FedSQL SELECT statement. For more information, see "Package Method Expression" on page 62.

A package is used as a template to construct an instance of the package. A package variable is used to reference a particular instance of the package. Here is an example.

```
/* Create package animal */
package animal;
   declare varchar(100) s;
   method animal(varchar(100) s);
      this.s = s;
   end;

   method speak();
      put s;
   end;
endpackage;

data _null_;
   method init();
      /* Create variable a1 of type animal. */
      declare package animal a1;
```

```
      /* Create variable a2 of type animal.
         Construct an instance of type animal.
         Set variable a2 to reference the
         newly constructed animal instance. */
      declare package animal a2('meow');

      /* Set variable a1 to reference the same animal
         instance referenced by variable a2. */
      a1 = a2;

      a1.speak();
      a2.speak();

      /* Construct an instance of type animal.
         Set variable a1 to reference the
         newly constructed animal instance. */
      a1 = _new_ animal('woof');
      a1.speak();
      a2.speak();
   end;
enddata;
run;
```

Packages and Scope

Overview of Packages and Scope

The lifetime of a package instance is dependent on the scope in which the instance is created. A package instance is deleted automatically when execution exits the scope in which the instance was created.

By default, a package instance that is created in a method is created in the local scope of the method. As a result, these package instances are local to a method and are deleted when on return from the method.

If a method needs to return a package instance, then the package instance needs to be created in a scope outside the scope of the method. Otherwise, the package instance would be deleted prematurely when the method returns. With the _NEW_ operator, you can specify that the instance be created in a different scope than the default scope.

Specifying Scope

The scope in which to construct a package instance can be specified with the _NEW_ operator. The specified scope can be the scope of another package instance or the global scope of the package, thread, or data program containing the package instance.

Here is the syntax for the _NEW_ operator.

package-variable=**_NEW_** <\[THIS\] | \[*package-instance*\]>
package-name (<*constructor-arguments*>);

For more information, see the "_NEW_ Operator" in *SAS Viya: DS2 Language Reference*.

This example uses the _NEW_ operator to construct an instance of the package **mypkg** in the global scope of the data program. The package instance that is returned by the _NEW_ operator is assigned to the variable **p**.

```
package mypkg/overwrite=yes;
   method m(double x) returns double;
      return x+99;
   end;
endpackage;

data _null_;
   dcl package mypkg p;
   method m();
      p = _new_ [this] mypkg();
   end;
enddata;
```

This causes the instance assigned to the variable **p** to have global scope.

It is important to note the difference between the scope of the package variable and the scope of the package instance.

In the following example, the variable **p** is declared local to the method **m** and the instance is global to the entire program.

```
data _null_;
   method m();
      dcl package mypkg p;
      p = _new_ [this] mypkg();
   end;

   method init();
      m();
      /* instance of mypkg created in global */
      /* scope and therefore still exists. It is */
      /* a dead reference because the instance */
      /* is not referenced by any package variable */
      /* and therefore is inaccessible. */
   end;
enddata;
run;
```

If the instance is not explicitly deleted or reassigned to a variable with global scope, then the instance becomes a dead reference after the method **m** completes.

For more information, see the "_NEW_ Operator" in *SAS Viya: DS2 Language Reference*.

Global Scope

The THIS keyword is used as an argument in the _NEW_ operator to specify that a package instance be created in a global scope. DS2 supports three types of global scope:

- program
- package
- thread

A package instance created in a data program with the _NEW_ operator and the THIS scope keyword is created in program global scope. The lifetime of the package instance

is the entire program. Package instance **p** in the previous topic's example is in program global scope. Package instance **p** is not deleted until the program exits.

A package instance created in a package block with the _NEW_ operator and the THIS scope keyword is created in package global scope. Package global scope is limited to the lifetime of a specific instance of the package. When the package instance is deleted, all package instances in the instance's global scope are also deleted.

The following example illustrates a package instance with package global scope.

```
package pkgA;
   declare double i;
   method pkgA(double i);
      this.i = i;
   end;
endpackage;

package pkgB;
   declare package pkgA a;
   method pkgB(double i);
      a = _new_ [THIS] pkgA(i); 1
   end;
endpackage;

data _null_;
   method init();
      declare package pkgB b(5); 2
   end;
enddata;
run;
```

1 The package instance referenced by package variable **b** is created in the local scope of the data program's INIT method. Package instance **b** is deleted when the INIT method returns.

2 The package instance referenced by package variable **a** is created in the package global scope of package instance **b**. Package instance **a** is deleted when package instance **b** is deleted.

Package instance **a** is in the package global scope of package instance **b**. When package instance **b** is destroyed at the return of the INIT method, package instance **a** is also destroyed.

The third type of global scope is thread global scope. A package instance created with the THIS scope keyword in a thread program is in thread global scope. Thread global scope is limited to the lifetime of a specific thread. When the thread exits, all package instances in the thread's global scope are automatically deleted.

Package-Specific Scope

The _NEW_ operator can also be used to specify that a new package instance has the same lifetime as an existing package instance by specifying the existing package instance as the scope argument. The syntax is as follows.

```
package-variable = _NEW_ [existing-package-instance]package-name
   ([constructor-arguments]);
```

The new package instance is then created in the same scope in which the existing package instance was created. The existing package instance must be created prior to the

time that the _NEW_ operator executes. The existing package instance and the new package instance can be different package types.

In this example, two **pkgA** package instances are created with the same lifetime as a hash package instance.

```
package pkgA;
      declare double i;
      method pkgA(double i);
        this.i = i;
      end;
    endpackage;

    data _null_;
      declare package hash h(); 1
      declare int          i;
      declare package pkgA a;

      method init();
        h.definekey('i');
        h.definedata('a');
        h.definedone();

        i = 55;
        a = _new_ [h] pkgA(i); 2
        h.add(); 3

        i = 66;
        a = _new_ [h] pkgA(i); 4
        h.add(); 5

      end;
    enddata;
```

1 A hash package instance referenced by package variable **h** is created in program global scope.

2 A **pkgA** package instance is created in the same scope in which package instance **h** was created. The **pkgA** package instance is assigned to package variable **a**.

3 The **pkgA** package instance referenced by variable **a** is added to hash **h** with key **i (55)**.

4 A second **pkgA** package instance is created in the same scope in which package instance **h** was created. The second **pkgA** package instance is assigned to package variable **a**.

5 The second **pkgA** instance referenced by variable **a** is added to hash **h** with key **i (66)**.

Both instances of **pkgA** effectively have the same lifetime as the hash package instance **h** because the **pkgA** instances are created in the same scope (program global scope) as the hash package instance **h**. Thus, when the hash package instance **h** is destroyed at the end of the program, both instances of **pkgA** are also destroyed.

Returning Package Instances from Methods

You can use the RETURN statement to return package instances from methods. Here is an example.

```
data_null_;
   dcl package mypkg p;

   method r() returns package mypkg;
      return _new_ [this] mypkg();
   end;

   method init();
      p = r();
      x = p.m(100);
      put x=;
   end;

   method term();
      x = p.m(200);
      put x=;
    end;

enddata;
```

In this case, the method **r** returns a global instance of the package, **mypkg**, which is then used in the INIT method. Because the lifetime of **p** is global, it can be used again in other methods as shown here in the TERM method.

The variable **p** must be declared in global scope in order to do this. If it had been declared local to the INIT method, it would, of course, have not been available in the TERM method. Also, in that case, the instance of the package would have become a dead reference after the INIT method had finished, unless it had been explicitly deleted as in the following modified INIT method.

```
method init();
   p = r();
   x = p.m(100);
   put x=;
   p.delete();
end;
```

This effect could be more easily achieved by declaring **p** to be local to the INIT method.

Passing Package Arguments

In addition to returning packages from methods, DS2 allows package instances to be passed to methods. This example uses the packages **mypkg** and **mypkg2** that were created in "Package-Specific Scope" on page 101. The instance **p2** is passed to the method **tp**, which then instantiates the subpackage **mypkg** in **p2**.

```
data _null_;
   dcl package mypkg2 p2();

   method tp(package mypkg2 p2);
      p2.p = _new_ [p2] mypkg();
   end;

   method init();
      dcl package mypkg p;
      tp(p2);
      p = p2.p;
      x = p.m(100);
```

```
      put x=;
   end;
enddata;
```

p2 is declared in a global scope and **p** and **p2** are both automatically deleted at the end of the program. The following **tp** method is nearly equivalent to the **tp** method in the above program, but not tying the scope of **p** to **p2** is slightly different in that you can delete **p2** separately from **p** if necessary.

```
method tp(package mypkg2 p2);
   p2.p = _new_ [this] mypkg();
 end;
```

Note: If either THIS or **p2**'s package scope were not used, the instance of **p** would be deleted at the end of the method **tp**. This is the default behavior of packages.

Attributes and Methods

The PRIVATE access modifier can be used for attributes or methods that are intended for internal use within the package. This enables you to manage the complexity of your program by exposing only the attributes that you intended for the end user to directly get or set. Private attributes are useful for saving state information that results from calling a method which, if touched directly by the user, could invalidate further results. Here is an example of using private attributes where you do not want the attributes **min**, **max**, or **sum** modified directly in the **nextNumber** method.

```
proc ds2;
package stats / overwrite=yes;
   /*-- put scratch space in private attributes --*/
   dcl private double min max sum ini;
   /*-- put shared logic in a private method --*/
   private method p_update( double v );
      if missing(v) then return;
        if missing(ini) then do;
           min=v;
            max=v;
            sum=v;
            ini = 1;
        end;
      else do;
         if v < min then min = v;
         if v > max then max = v;
         sum = sum + v;
        end;
      return;
   end;

   method nextNumber( double v );
        put 'in the double method';
      p_update( v );
   end;

   method nextNumber( char(20) c );
      dcl double v;
        v = c;
        put 'in the char method';
      p_update( v );
```

```
   end;

   method nextNumber( int i );
      dcl double v;
        v = i;
        put 'in the int method';
      p_update( v );
   end;

   method getStats( in_out double min, in_out double max, in_out double sum );
      min = this.min;
      max = this.max;
      sum = this.sum;
   end;
endpackage;

data;
dcl package stats st();
dcl double min max sum;
method run();
   st.getStats( min, max, sum );
   output;
   st.nextNumber( 5 );
   st.nextNumber( '4.0' );
   st.nextNumber( 2.0 );
   st.getStats( min, max, sum );
   output;
end;
enddata;
run;
quit;
```

Dot Operator in Packages

In the DS2 language, standard dot notation is restricted to three-level names. If you have nested packages, the standard dot notation requires a series of operations to make the method call. Here is an example.

```
dcl package TOP top();
...
t1=top.middle;
t2=t1.bottom;
result=t2.calledmethod();
```

However, when referencing DS2 packages, you can use the dot(.) as a standard binary operator. This enables you to access methods of nested packages by using a four-level name.

The previous example can be simplified as follows.

```
dcl package TOP top();
...
result=top.middle.bottom.calledmethod();
```

Package Constructors and Destructors

Constructors and destructors are special package methods that are used during construction and destruction of package instances. The constructor method initializes a newly constructed package instance. The destructor method performs cleanup, releases resources held by the package instance before the package instance is destroyed, or both. A package's constructor method has the same name as the class, and a package's destructor is the DELETE method. Constructors and destructors do not have return types and do not return values.

DS2 automatically calls a package's constructor when an instance of the package is constructed. DS2 automatically calls a package's destructor when a package instance goes out of scope and is destroyed. Note that creating a package variable with a DECLARE PACKAGE statement does not result in DS2 calling the package's constructor. A package's constructor is called only when a package instance is constructed with either a DECLARE PACKAGE statement with constructor arguments or with a _NEW_ operator.

For more information, see the applicable DECLARE PACKAGE statements and _NEW_ operators in the language reference section of this document.

User-Defined Packages

You can store methods that you create in user-defined packages. These packages can be thought of as libraries of your methods. Any type of method can be saved in a package. Once you have stored methods in a package (using the PACKAGE statement), you 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, you can use the condensed constructor syntax:

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

For more information, see the "PACKAGE Statement" in *SAS Viya: DS2 Language Reference*, "DECLARE PACKAGE Statement" in *SAS Viya: DS2 Language Reference*, and the "_NEW_ Operator" in *SAS Viya: DS2 Language Reference*.

Note: You cannot use a user-defined package to hide a predefined DS2 package by overloading that package.

Here is an example of a very simple user-defined package called MATH. It contains a method that adds two numbers.

```
package math;
   method add(double x, double y) returns double;
      return x+y;
   end;
endpackage;
```

In the next example, two numbers are added by using the ADD method in the MATH package that was created in the previous example. First, the MATH package is declared and instantiated. Then the ADD method is called and the result is assigned to SUM.

```
data _null_;
   dcl double sum;
   method init();
      dcl package math f();
      sum = f.add(2,3);
         put 'sum= ' sum;
   end;
enddata;
```

Predefined DS2 Packages

Overview of Predefined DS2 Packages

SAS provides the following predefined packages for use in the DS2 language:

FCMP
: Supports calls to FCMP functions and subroutines from within the DS2 language.

  For more information, see "Using the FCMP Package" on page 108.

Hash and hash iterator
: Enables you 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 values can be directly assigned constant values, values from a table, or values can be computed in an expression.

  For more information, see "Using the Hash Package" on page 109 and "Using the Hash Iterator Package" on page 120.

HTTP
: Constructs an HTTP client to access HTTP web services.

  For more information, see "Using the HTTP Package" on page 121.

JSON
: Enables you to create and parse JSON text.

Logger
: Provides a basic interface (open, write, and level query) to the SAS logging facility.

  For more information, see "Using the Logger Package" on page 128.

Matrix
: Provides a powerful and flexible matrix programming capability. It provides a DS2-level implementation of SAS/IML functionality.

  For more information, see "Using the MATRIX Package" on page 130.

SQLSTMT
: Provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS.

  For more information, see "Using the SQLSTMT Package" on page 135.

TZ
: Provides a way to process local and international time and date values.

  For more information, see "Using the TZ Package" on page 139.

Using the FCMP Package

Overview of FCMP Packages

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

Note: The FCMP package is not supported in the CAS server.

You create an FCMP package by using the LANGUAGE=FCMP and TABLE= options in a PACKAGE statement. After the package is created, you declare an instance of the FCMP package. There are two ways to construct an instance of an FCMP package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

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

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package fcmp banking();
```

For more information, see "DECLARE PACKAGE Statement, FCMP Package" in *SAS Viya: DS2 Language Reference* and "PACKAGE Statement" in *SAS Viya: DS2 Language Reference*.

FCMP Package Capabilities

These are the capabilities of using the FCMP package in the DS2 language:

- Call an FCMP function or subroutine with scalar DOUBLE, CHAR, and NCHAR parameters, return parameters of both.

  The DS2 language does automatic type conversion so that almost any type is supported. An example is a conversion from TINYINT to DOUBLE. For more information, see "Overview of Type Conversions" on page 50.

- Call an FCMP function or subroutine with scalar DOUBLE OUTARGS parameters. The FCMP procedure's DOUBLE OUTARGS parameter is treated as an IN_OUT parameter in the METHOD statement. For more information about the IN_OUT parameter, see the "METHOD Statement" in *SAS Viya: DS2 Language Reference*.

  Note: DS2 must pass a DOUBLE variable, not an expression, through an OUTARGS parameter.

Considerations and Limitations When Using the FCMP Package

- The FCMP package does not support VARARGS functions calls and therefore cannot use the FCMP procedure's VARARGS interface.
- Errors caused when information is passed between the FCMP procedure and the FCMP package are not always reported correctly. For example, if you supply an incorrect table name in the PACKAGE statement, an error is written in the log file. However, there is no indication given that the operation fails.
- The FCMP package assumes the session encoding and currently has no mechanism that allows different encodings for different parameters within the same function call or for the same parameter across multiple function calls.
- You can access any FCMP library as long as the connection string defines the catalog in which the FCMP library is located.

Using the Hash Package

Overview of Hash Packages

The hash package provides an efficient, convenient mechanism for quick data storage and retrieval. The hash package stores and retrieves data based on unique lookup keys. Depending on the number of unique lookup keys and the size of the table, the hash package lookup can be significantly faster than a standard format lookup or an array.

Before you use a DS2 hash package, you must define and construct an instance of (**instantiate**) the hash package.

After you define and create a hash package instance, you can perform many tasks, including the following:

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

For example, suppose that you have a large table that contains numeric lab results that correspond to a unique patient number and weight and a small table that contains patient numbers (a subset of those in the large table). You can load the large table into a hash package using the unique patient number as the key and the weight values as the data. You can then iterate over the small table using the patient number to look up the current patient in the hash package whose weight is over a certain value and that data to a different table.

Defining and Creating a Hash Package Instance

To create an instance of a hash package, you provide keys, data, and optional initialization data about the hash instance to construct. A hash package instance can be defined either fully at construction or at construction and through a subsequent series of method calls.

In the following example, the hash instances, **h1** and **h2**, have the same instance definition. The hash instance **h1** is fully defined at construction while **h2** is defined at construction and through a series of method calls.

```
declare package hash h1([key], [data1 data2 data3],
   0, 'testdata', '', '', '', 'multidata');

declare package hash h2();
   method init();
      h2.keys([key]);
      h2.data([data1 data2 data3]);
      h2.dataset('testdata');
      h2.multidata();
      h2.defineDone();
   end;
```

For more information, see “Defining a Hash Instance by Using Constructors” on page 109 and “Defining a Hash Instance by Using Method Calls” on page 110.

Defining a Hash Instance by Using Constructors

A **constructor** is a method that you can use to instantiate a hash package and initialize the hash package data.

There are three different methods for creating a hash package instance with constructors.

- Create a partially defined hash instance.

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

  The key and data variables are defined by method calls. The optional parameters that provide the initialization data can be specified either in the DECLARE PACKAGE statement as shown above, in the _NEW_ operator, by method calls, or a combination of any of these. A single DEFINEDONE method call completes the definition.

- Create a completely defined hash instance with the 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.

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

  DECLARE PACKAGE HASH *instance*(\[*keys*\][, *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.

For more information about the optional parameters, see "Providing Initialization Data for a Hash Package" on page 111. For more information about defining the optional parameters using method calls, see "Defining a Hash Instance by Using Method Calls" on page 110.

Note: All variables that are passed to a hash instance must be global variables.

Defining a Hash Instance by Using Method Calls

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

KEYS
DEFINEKEY
DATA
DEFINEDATA
DATASET
DUPLICATE
HASHEXP
ORDERED
MULTIDATA
SUMINC
DEFINEDONE

For more information about these methods, see "DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

Note: After a hash instance specification is completed by a call to the DEFINEDONE method, a subsequent call to any of the above methods results in an error.

Here is an example of a hash instance, **h**, defined by 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

The hash package uses unique lookup keys to store and retrieve data. The keys and the data are variables that you use to initialize the hash package by using dot notation method calls.

You can define the key and data variables in one of three ways.

- Use the variable methods, DEFINEDATA and DEFINEKEY.
- Use the variable list methods, DATA and KEYS.
- Use key and data variable lists specified in the DECLARE PACKAGE statement.

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.

Here are examples.

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

/* Keys and data defined using the variable list methods */
declare package hash h();
h.keys([k]);
h.data([d]);
h.definedone();

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

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.

For more information, see "Implicit Variable and Variable List Methods" on page 113.

Providing Initialization Data for a Hash Package

In addition to the keys and data, you can provide the following optional parameters when you initialize a hash package:

- the internal table size (*hashexp*) where the size of the hash table is 2^n
- the name of the table to load (*datasource*) or a FedSQL query to select the data to load

  Note: Using a FedSQL query to select the data is not currently supported in the CAS server.
- whether or how the data is returned in key-value order (*ordered*)
- whether to ignore duplicate keys when loading a table (*duplicate*)
- the name of a variable that maintains a summary count of hash package keys (*suminc*)
- whether multiple data items are allowed for each key (*multidata*)

You can specify the initialization data in the DECLARE PACKAGE statement, the _NEW_ operator, by method calls, or a combination of these ways.

Note: When you initialize hash package data using a constructor in the DECLARE PACKAGE statement or the _NEW_ operator, you must provide the optional parameters in this order: *hashexp*, *datasource*, *ordered*, *duplicate*, *suminc*, and *multidata*. These positional constructor parameters must all be enclosed in a single set of parentheses, separated by commas, and, except for the *hashexp* parameter, wrapped by single quotation marks. Because the optional parameters are positional, you must provide a place holder for each parameter to the last parameter that you specify. The placeholder that must be used depends on the parameter. For more information about the placeholders, see the "DECLARE PACKAGE Statement, Hash Package" in *SAS Viya: DS2 Language Reference* or the "_NEW_ Operator, Hash Package" in *SAS Viya: DS2 Language Reference*. In the following example, to specify an ascending order and to replace duplicates, you must use –1 as a place holder for the *hashexp* parameter, empty single quotations marks (' ') as the place holder for the *datasource* parameter, **'a'** for *ordered* parameter, **'replace'** for *duplicate*. Because the *duplicate* parameter is the last one specified, no place holder is required for the *suminc* and *multidata* parameters. Here is an example.

```
declare package hash variable-name(8,'', 'a', 'replace');
```

For more information, see "Defining a Hash Instance by Using Constructors" on page 109 , "Defining a Hash Instance by Using Method Calls" on page 110, and "Using the _NEW_ Operator to Create a Hash Instance" on page 112.

Using the _NEW_ Operator to Create a Hash Instance

As an alternative to using the DECLARE PACKAGE statement to create a hash variable and a hash instance, you can use the DECLARE PACKAGE statement create the hash variable and the _NEW_ operator to create the hash instance. You declare a hash package variable using the DECLARE PACKAGE statement. Then you use the _NEW_ operator to instantiate an instance of the hash package and set the hash variable to reference the newly instantiated hash instance. With this scenario, initialization data cannot be provided using the DECLARE PACKAGE statement. You can provide initialization data for the hash instance with the _NEW_ operator and subsequent method calls if the hash instance was not constructed fully defined.

In the following example the DECLARE PACKAGE statement tells the compiler that the variable MYHASH is of type hash package. At this point, you have declared only the variable MYHASH. It has the potential to reference a hash instance, but it currently references nothing and therefore is a null package reference. You should declare the hash variable package only once. The _NEW_ operator creates an instance of the hash package and assigns it to the variable MYHASH.

```
declare package hash myhash();
myhash = _new_ hash(8, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

The above statement is equivalent to the following code:

```
declare package hash myhash(8, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

For more information, see the "_NEW_ Operator, Hash Package" in *SAS Viya: DS2 Language Reference*.

Implicit Variable and Variable List Methods

When you define a hash instance, you specify a series of key and data variables. After the hash instance is completely defined, the key and data variables can be implicitly or explicitly read and written during subsequent operations.

Note: All variables that are passed to a hash instance must be global variables.

There are two ways to pass keys and data variables to the hash instance: implicit variables methods and variable list methods.

The implicit variable method is similar to the DATA step hash object interface. Using the implicit variable methods, the key and data variables are defined through a series of DEFINEKEY and DEFINEDATA method calls and a single DEFINEDONE method call. Then the set of key and data variable definitions is used during execution of the other hash package methods as implicit arguments.

The following example defines a hash table with two key variables, **k1** and **k2**, and two data variables, **d1** and **d2**. The FIND method reads the values of the implicit key variables, looks up the key values in the hash table. If the key values are found, DS2 writes the corresponding data values to the implicit data variables defined for the hash instance.

```
declare package hash h();
   h.definekey('k1');
   h.definekey('k2');
   h.definedata('d1');
   h.definedata('d2');
   h.definedone();

   /* No explicit arguments specify what key values to find */
   /* or what to do with the data values if keys are found. */
   /* Implicitly uses key variables k1 and k2 and           */
   /* data variables d1 and d2.                             */
   h.find();
```

The variable list method involves specifying variables of interest in variable lists as explicit arguments when the hash methods are called.

This example uses the variable list methods and is the same as the one above that uses implicit variable methods.

```
declare package hash h();
rc=h.keys([k1 k2]);
rc=h.data([d1 d2]);
rc=h.definedone();
h.find([k1 k2], [d1 d2]);
```

The variable list method provides flexibility to use variables other than the implicit key and data variables. The following FIND method looks for the values that are stored in **x** and **y**. If the values are found, they are written to **u** and **v**.

```
 h.find([x y], [u v]);
```

For more information about variable lists, see "Variable Lists" on page 21.

All of the hash implicit variable methods work with hash instances that have both keys and data and with hash instances that are keys-only.

Some variable list methods work only with keys-only hash instances while others work only with hash instances that have both keys and data. A run-time error occurs if a keys-only hash instance invokes a variable list method that works only for a hash instance that has keys and data, and vice versa. For more information about which methods work with keys-only hash instances, see each method in "DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

Non-Unique Key and Data Pairs

By default, all of the keys in a hash package are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash package, that is, associate more than one set of data variables with a key.

For example, assume that the key is a patient ID and the data is a visit date. If the patient were to visit multiple times, multiple visit dates would be associated with the patient ID. When you create a hash package with the MULTIDATA parameter or method set to YES, multiple sets of the data variables are associated with the key.

If the table contains duplicate keys, by default, the first instance is stored in the hash package and subsequent instances are ignored. To store the last instance in the hash package, use the DUPLICATE parameter or method. The DUPLICATE parameter or method also writes an error to the SAS log if there is a duplicate key.

However, the hash package allows storage of multiple values for each key if you use the MULTIDATA parameter or method. The hash package keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.

To traverse a multiple data item list, you must know the current list item. Start by calling the FIND method for a given key. The FIND method sets the current list item. Then to determine whether the key has multiple data values, call the HAS_NEXT method. After you have determined that the key has another data value, you can retrieve that value with the FIND_NEXT method. The FIND_NEXT method sets the current list item to the next item in the list and sets the corresponding data variable or variables for that item.

In addition to moving forward through the list for a given key, you can loop backward through the list by using the HAS_PREV and FIND_PREV methods in a similar manner.

Note: The items in a multiple data item list are maintained in the order in which you insert them.

For more information about the MULTIDATA and DUPLICATE parameters, see the "DECLARE PACKAGE Statement, Hash Package" in *SAS Viya: DS2 Language Reference* or the "_NEW_ Operator, Hash Package" in *SAS Viya: DS2 Language Reference*. For more information about the MULTIDATA and DUPLICATE methods, see the "MULTIDATA Method" in *SAS Viya: DS2 Language Reference* and the "DUPLICATE Method" in *SAS Viya: DS2 Language Reference*.

Maintaining Key Summaries

You can maintain a summary count for a hash package key by using the SUMINC parameter or method. SUMINC instructs the hash package to allocate internal storage in each record to store a summary value in the record each time that the record is used by a FIND, CHECK, or REF method. The SUMINC value is also used to maintain a summary count of hash parameter keys after a FIND, CHECK, or REF method.

SUMINC is given a variable, which holds the sum increment, that is, how much to add to the key summary for each reference to the key. The SUMINC value can be greater than, less than, or equal to 0. The SUMINC value is also used to initialize the summary on an ADD method. Each time the ADD method occurs, the key to the SUMINC value is initialized.

The SUM method retrieves the summary value for a given key when only one data item exists per key.

If multiple items exist, the SUMDUP method retrieves the current value of the key summary.

You can use key summaries in conjunction with the DATASOURCE parameter or DATA method. As a table is read into the hash package using the DEFINEDONE method or a DECLARE PACKAGE statement, all key summaries are set to the SUMINC value and all subsequent FIND, CHECK, or ADD methods change the corresponding key summaries.

For more information about the SUMINC parameter, see the "DECLARE PACKAGE Statement, Hash Package" in *SAS Viya: DS2 Language Reference*. For more information about the SUMINC and SUMDUP methods, see the "SUMINC Method" in *SAS Viya: DS2 Language Reference* and the "SUMDUP Method" in *SAS Viya: DS2 Language Reference*.

Storing and Retrieving Data

After you initialize the hash package's key and data variables, you can store data in the hash package using the ADD method, or you can use the DATASOURCE parameter or DATASET method to load a table into the hash package. If you use the DATASOURCE parameter or DATASET method, and if the table contains more than one row with the same value of the key, by default, SAS keeps the first row in the hash table and ignores subsequent rows. To store the last instance in the hash package or to send an error to the log if there is a duplicate key, use the DUPLICATE parameter or method. To allow duplicate values for each key, use the MULTIDATA parameter or method.

You can then use the FIND method to search and retrieve data from the hash package. Use the FIND_NEXT and FIND_PREV methods to search and retrieve data if multiple data items exist for each key.

For more information, see the "ADD Method, Hash Package" in *SAS Viya: DS2 Language Reference*, "FIND Method" in *SAS Viya: DS2 Language Reference*, the "FIND_NEXT Method" in *SAS Viya: DS2 Language Reference*, and the "FIND_PREV Method" in *SAS Viya: DS2 Language Reference*.

You can consolidate a FIND method and ADD method using the REF method. In the following example, you can reduce the amount of code from this:

```
rc = h.find();
   if (rc != 0) then
      rc = h.add();
```

to a single method call:

```
rc = h.ref();
```

For more information, see the "REF Method" in *SAS Viya: DS2 Language Reference*.

Note: You can also use the hash iterator package to retrieve the hash package data, one data item at a time, in forward and reverse order. For more information about the hash iterator package, see "Using the Hash Iterator Package" on page 120.

Replacing and Removing Data

You can remove or replace data in the hash package using on of the following methods:

- Use the REMOVE method to remove the data items from the specified key.
- Use the REMOVEALL method to remove all the data items.
- Use the REMOVEDUP method to remove data items for keys that have multiple data items.
- Use the REPLACE method to replace all data items.
- Use the REPLACEDUP method to replace only the current data item.

Note: If an associated hash iterator is pointing to the key, the REMOVE method does not remove the key or data from the hash package. An error message is issued to the log.

For more information, see the "REMOVE Method" in *SAS Viya: DS2 Language Reference* , the "REMOVEALL Method" in *SAS Viya: DS2 Language Reference*, the "REMOVEDUP Method" in *SAS Viya: DS2 Language Reference*, the "REPLACE Method" in *SAS Viya: DS2 Language Reference*, and the "REPLACEDUP Method" in *SAS Viya: DS2 Language Reference*.

Saving Hash Package Data in a Table

You can create a table that contains the data in a specified hash package by using the OUTPUT method.

In the following example, the first table program generates the **data1** table. The second table program creates a hash package **h** with one key, **k**, and two data, **d1** and **d2**. Then each row from the **data1** table is read and the keys and data are added to hash package **h**. Finally, the data values stored in hash package **h** are written to the **out1** table. The third table program writes the contents of the **out1** table.

```
/* Generate and output 5 rows for table data1. */
data data1(overwrite=yes);
  declare double k d1 d2;
  method init();
    declare int i;
    do i = 1 to 5;
      k = i; d1 = i*10; d2 = i*2; output;
    end;
  end;
enddata;
run;

data _null_;
  declare double k d1 d2;
  declare package hash h(0, '', 'descending');

  /* Define key and data variables for hash h. */
  method init();
    h.defineKey('k');
    h.defineData('d1');
    h.defineData('d2');
    h.defineDone();
  end;

  /* Read rows from table data1.
   * Add key and data values from rows to hash h. */
```

```
  method run();
    set data1;
    h.add();
  end;

  /* Add additional key and data values to hash h.
   * Output hash h to table out1. */
  method term();
    k = 11; d1 = 110; d2 = 22; h.add();
    k = 12; d1 = 120; d2 = 24; h.add();
    k = 13; d1 = 130; d2 = 26; h.add();
    k = 14; d1 = 140; d2 = 28; h.add();
    h.output('out1');
  end;

enddata;
run;

/* Outputs rows from table out1. */
data;
  method run();
    set out1;
  end;
enddata;
run;
```

Output 12.1 *Output Rows from Table OUT1*

d1	d2
140	28
130	26
120	24
110	22
50	10
40	8
30	6
20	4
10	2

Note that the hash package keys are not stored as part of the output table. If you want to include the key in the output table, you must define the key as data in the DEFINEDATA method. In the previous example, the DEFINEDATA method would be written this way:

```
h.defineKey('k');
```

```
h.defineData('k');
h.defineData('d1');
h.defineData('d2');
```

With the above modification, the following lines are written.

Output 12.2 *Output Rows from Table OUT1 with Keys and Data*

k	d1	d2
14	140	28
13	130	26
12	120	24
11	110	22
5	50	10
4	40	8
3	30	6
2	20	4
1	10	2

Using a FedSQL Query with a Hash Instance to Get Rows Dynamically at Run Time

Note: Using a FedSQL query to select the data is not currently supported in the CAS server.

You can delay the decision of which rows to get from a table until run time by using a hash instance. At run time, the hash instance is created and loaded with the selected rows. The rows are selected based on a FedSQL query specified for the data source. You can use a hash iterator to loop over the rows and access the row data.

In the following example, the rows are selected by the **`SELECT * FROM test WHERE a=1`** FedSQL query. This query is passed to the **`execute`** method that loads them into the hash package. A hash iterator loops over the rows and writes the selected rows to the SAS log and the **`result`** table.

```
data test;
   a=1;  b=2;output;
   a=11; b=22;output;
   a=1;  b=3;output;
   a=22; b=44;output;
run;

proc ds2;
package pkg /overwrite=yes;
   dcl double a b;
```

```
   method execute(char(200) sql);
      dcl package hash  h();
      dcl package hiter hi(h);
      h.keys([a]);
      h.data([a b]);
      h.multidata('yes');
      h.dataset(sql);
      h.definedone();

      put 'SELECTED ROWS:';
      rc = hi.first();
      do while(rc = 0);
         put a= b=;
         rc = hi.next();
      end;

      h.output('result');
   end;
endpackage;
run; quit;

proc ds2;
data _null_;
   method init();
      declare package pkg p1();
      p1.execute('{SELECT * FROM test WHERE a=1}');
   end;

enddata;
run; quit;


proc print data=test;
   title2 'TEST TABLE';
run; quit;

proc print data=result;
   title2 'RESULT TABLE';
run; quit;
```

The following lines are written to the SAS log.

```
SELECTED ROWS:
a=1 b=2
a=1 b=3
```

The input and output tables are as follows.

Output 12.3 *Input Table*

TEST TABLE

Obs	a	b
1	1	2
2	11	22
3	1	3
4	22	44

Output 12.4 *Output Table*

RESULT TABLE

Obs	a	b
1	1	2
2	1	3

Using Hash Package Attributes

There are two attributes available to use with hash packages. NUM_ITEMS returns the number of items in a hash package and ITEM_SIZE returns the size (in bytes) of an item.

The following example retrieves the number of items in a hash package:

```
num_items = myhash.num_items;
```

The following example retrieves the size of an item in a hash package:

```
item_size = myhash.item_size;
```

You can obtain an idea of how much memory the hash package is using with the ITEM_SIZE and NUM_ITEMS attributes. The ITEM_SIZE attribute does not reflect the initial overhead that the hash package requires, nor does it take into account any necessary internal alignments. Therefore, the use of ITEM_SIZE does not provide exact memory usage, but it gives a good approximation. For more information, see the "ITEM_SIZE Attribute" in *SAS Viya: DS2 Language Reference* and "NUM_ITEMS Attribute" in *SAS Viya: DS2 Language Reference*.

Using the Hash Iterator Package

You use a hash iterator package to store and search data based on unique lookup keys. The hash iterator package enables you to retrieve the hash package data in forward or reverse key order.

You declare a hash iterator package by using the DECLARE PACKAGE statement. After you declare the new hash iterator package, use the _NEW_ operator to instantiate the package, using the hash package name as a parameter. For example:

```
declare package hiter myiter;
myiter = _new_ hiter('h');
```

The DECLARE PACKAGE statement tells the compiler that the variable MYITER is of type hash iterator. At this point, you have declared only the variable MYITER. It has the potential to reference a hash iterator instance, but it currently references nothing and thus is a null package reference. You should declare the hash iterator package variable only once. The _NEW_ operator constructs an instance of the hash iterator package and assigns it to the variable MYITER. The hash package, H, is passed as a constructor parameter.

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operators to declare and instantiate a hash iterator package, you can declare and instantiate a package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package hiter myiter('h');
```

For more information, see the "DECLARE PACKAGE Statement, Hash Iterator Package" in *SAS Viya: DS2 Language Reference*, and the "_NEW_ Operator, Hash Iterator Package" in *SAS Viya: DS2 Language Reference*.

Note: You must declare and instantiate a hash package before you create a hash iterator package.

Using the HTTP Package

Overview of the HTTP Package

Use the HTTP package to construct an HTTP client in order to access HTTP web servers.

Here are the general tasks:

1. Declare and instantiate an HTTP package.
2. Create an HTTP GET, HEAD, or POST method.

   You can use additional HTTP package methods to add header information and to send request data.
3. Execute the HTTP GET, HEAD, or POST method.
4. Retrieve the response information from the web server:
   - The response body as a complete entity or by streaming
   - The response content type
   - The response header

The HTTP package also enables you to perform these tasks:

- Retrieve status codes from HTTP responses.
- Set a socket time-out value.
- Log the HTTP traffic between the HTTP client and server using the SAS logging facility.

Declaring and Instantiating an HTTP Package

You must first declare and instantiate an HTTP package. There are two ways to construct an instance of an HTTP package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package http httpclt;
httpclt = _new_ http();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package http httpclt();
```

For more information, see "DECLARE PACKAGE Statement, HTTP Package" in *SAS Viya: DS2 Language Reference* and "_NEW_ Operator, HTTP Package" in *SAS Viya: DS2 Language Reference*.

TIP Web service applications might require only a single HTTP client to synchronously handle HTTP traffic. Or, if your application requires it, you can instantiate multiple HTTP clients to asynchronously request and process data.

Create an HTTP GET, HEAD, or POST Method

1. Use the CREATEGETMETHOD, CREATEHEADMETHOD, or CREATEPOSTMETHOD method to create the GET, HEAD, and POST method.

   For more information, see "CREATEGETMETHOD Method" in *SAS Viya: DS2 Language Reference*, "CREATEHEADMETHOD Method" in *SAS Viya: DS2 Language Reference*, and "CREATEPOSTMETHOD Method" in *SAS Viya: DS2 Language Reference*.

2. (Optional) To add a header to the HTTP GET method, use the ADDREQUESTHEADER method.

   For more information, see "ADDREQUESTHEADER Method" in *SAS Viya: DS2 Language Reference*.

3. (Optional) To add a request body to the HTTP method, use the SETREQUESTBODYASBINARY or SETREQUESTBODYASSTRING method.

   For more information, see "SETREQUESTBODYASBINARY Method" in *SAS Viya: DS2 Language Reference* and "SETREQUESTBODYASSTRING Method" in *SAS Viya: DS2 Language Reference*.

4. (Optional) To specify the request content type in the HTTP method, use the SETREQUESTCONTENTTYPE method.

   For more information, see "SETREQUESTCONTENTTYPE Method" in *SAS Viya: DS2 Language Reference*.

Executing an HTTP GET, HEAD, and POST Method

When you execute the HTTP GET, HEAD, and POST methods, you send the request to an HTTP web server.

Note: Before you execute the HTTP GET, HEAD, and POST methods, you must create the HTTP GET, HEAD, or POST method. For more information, see "Create an HTTP GET, HEAD, or POST Method" on page 122.

For most HTTP methods, use the EXECUTEMETHOD method to send the request to the web server. For more information, see "EXECUTEMETHOD Method" in *SAS Viya: DS2 Language Reference*.

The EXECUTEMETHOD method does not support streaming of the response body. If you want to stream the response body, a different execute method,

EXECUTEMETHODSTREAM, is required. For more information, see "Retrieving an HTTP Resource" on page 123.

To send another request, repeat the process starting with creating the GET, HEAD, or POST request and ending with the EXECUTEMETHOD or EXECUTEMETHODSTREAM method.

This example program instantiates an HTTP package (the client), creates an HTTP GET method, executes the GET method to send a request to an HTTP web service, and retrieves the body information from the response from the HTTP web service as a string.

```
data _null_;
  method run();
    /* instantiate the package */
    declare package http h();
    declare varchar(1024) character set utf8 body;
    declare int rc;

    /* create a GET */
    h.createGetMethod('http://api.worldbank.org/countries/fr/');
    /* execute the GET */
    h.executeMethod();
    /* retrieve the response body as a string */
    h.getResponseBodyAsString(body, rc);
    put body;
  end;
```

The following lines are written to the SAS log.

```
<?xml version="1.0" encoding="utf-8"?>
<wb:countries page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:country id="FRA">

<wb:iso2Code>FR</wb:iso2Code>
    <wb:name>France</wb:name>
    <wb:region id="ECS">Europe
&amp; Central Asia (all income levels)</wb:region>
    <wb:adminregion id="" />

<wb:incomeLevel id="OEC">High income: OECD</wb:incomeLevel>
    <wb:lendingType id="LNX">Not
classified</wb:lendingType>
    <wb:capitalCity>Paris</wb:capitalCity>

<wb:longitude>2.35097</wb:longitude>
    <wb:latitude>48.8566</wb:latitude>

</wb:country>
</wb:countries>
```

Retrieving an HTTP Resource

You can use the following HTTP package methods to retrieve headers, the response body, and the response body type from an HTTP resource. If you are retrieving the response body, you can retrieve it as one entity or stream the response body.

Task	HTTP package method
Retrieve header	"GETRESPONSEHEADERSASSTRING Method" in *SAS Viya: DS2 Language Reference*
Get response body as one entity*	"GETRESPONSEBODYASBINARY Method" in *SAS Viya: DS2 Language Reference* "GETRESPONSEBODYASSTRING Method" in *SAS Viya: DS2 Language Reference*
Stream the response body*	"STREAMRESPONSEBODYASBINARY Method" in *SAS Viya: DS2 Language Reference* "STREAMRESPONSEBODYASSTRING Method" in *SAS Viya: DS2 Language Reference*
Retrieve the response body content type	"GETRESPONSECONTENTTYPE Method" in *SAS Viya: DS2 Language Reference*

* If you get the response body as one entity, you must create the GET method first and execute that method with the EXECUTEMETHOD method. If you stream the response body, you must create the GET method first and execute that method with the EXECUTESTREAMMETHOD method.

Considerations When Using the HTTP Package

- The HTTP package supports only GET, HEAD, and POST HTTP methods.
- Each client sends requests and processes responses synchronously. The application can create multiple clients to asynchronously perform actions on HTTP resources.
- The HTTP package stores data for requests and from responses in memory as DS2 string values or binary values. If you want to store the data on disk as a file, consider using the HTTP procedure.
- The HTTP package can send requests to a secure HTTP endpoint that requires authentication. When an HTTP end-point requires client authentication, it responds to the client with its list of supported authentication mechanisms. The HTTP package currently supports two of the three most common authentication mechanisms: Basic and Negotiate. Since Basic authentication in itself does not provide any credential confidentiality, it should be used only when the data is being encrypted via Transport Layer Security (TLS). For complete information about how SAS validates TLS, see *Encryption in SAS Viya*. Negotiate authentication supports Kerberos, and, when on Windows, supports NT LAN Manager (NTLM).

Logging HTTP Traffic

The HTTP package supports logging through the SAS logging facility.

The App.TableServices.d2pkg.HTTP logger logs errors, headers, and data that are sent back and forth from the HTTP client and the web server.

For more information, see "HTTP Package Logger" on page 218.

Using the JSON Package

Overview of the JSON Package

Java Script Object Notation (JSON) is a text-based, open standard data format that is designed for human-readable data interchange. JSON is based on a subset of the JavaScript programming language and uses JavaScript syntax for describing data objects.

The JSON package provides an interface to create and parse JSON text. The JSON package Write methods accumulate the Write requests in memory, and the text can be retrieved. The JSON package parser enables you to read and parse text.

Declaring and Instantiating the JSON Package

There are two ways to construct an instance of a JSON package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package json myjsonpkg;
myjsonpkg = _new_ json();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package json myjsonpkg();
```

For more information, see the "DECLARE PACKAGE Statement, JSON Package" in *SAS Viya: DS2 Language Reference*, and the "_NEW_ Operator, JSON Package" in *SAS Viya: DS2 Language Reference*.

Writing JSON Text

To create JSON text, you create a JSON writer instance by using the CREATEWRITER method.

JSON output consists of two types of data structure containers:

JSON object container ({ })
: begins with a left brace ({) and ends with a right brace (}). An object container collects name-value pairs that are written as pairs of names and values. A value can be any of the supported JSON data types, an object, or an array. Each name is followed a colon and then the value. The name-value pairs are separated by a comma.

  Use the WRITEOBJOPEN and WRITECLOSE methods to create the object container.

JSON array container ([ ])
: begins with a left bracket ([) and ends with a right bracket (]). An array container collects a list of values that are written as a list of values without names. A value can be any of the supported JSON data types, an object, or an array. Values are separated by a comma.

  Use the WRITEARRAYOPEN and WRITECLOSE methods to create the array container.

The top-level container can include any number of containers. Containers, likewise, can nest containers to an arbitrary depth. When nesting containers, be careful to observe the data structure requirements of the current container.

- Objects require a list of name-value pairs, where the value can itself be an object or array.

- Arrays have no such structural requirement of name-value pairs and are merely a list of values, objects, or arrays.

With JSON package methods, you can write character and numeric values, null values, and Boolean true and false values.

Here is an example of JSON text output:

```
{"SASJSONExport":"1.0","SASTableData+CLASS":[{"Name":"Joyce","Sex":
"F","Age":11,"Height":51.3,"Weight":50.5},{"Name":"Thomas","Sex":"M",
"Age":11,"Height":57.5,"Weight":85}]}
```

You can get the JSON text that is produced by the writer by using the WRITERGETTEXT method.

When you finish writing the text, call the DESTROYWRITER method to remove the writer instance.

The following example creates some JSON text and writes it to the SAS log.

```
data _null_;
    method init();
        dcl package json j();
        dcl double dblVal;
        dcl int rc;
        dcl nvarchar(30) jsontxt;

        rc = j.createWriter();
             if rc=0 then rc = j.writeArrayOpen();
        if rc=0 then rc = j.writeString( '     Hello World!     ' );
        if rc=0 then rc = j.writeClose();
        j.writerGetText( rc, jsontxt );
        put rc= jsontxt=;
    end;
enddata;
run;
```

The following line is written to the SAS log.

```
rc=0 jsontxt=["     Hello World!     "]
```

For more information about the methods that enable you to write JSON text, see "DS2 JSON Package Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

Parsing JSON Text

To parse, or read, JSON text, you create a JSON writer instance by using the CREATEPARSER method. You can provide the input JSON text to the parser with either the CREATEPARSER method, the SETPARSERINPUT method, or both.

The GETNEXTTOKEN method is used to return the next validate JSON language element form the JSON text. The GETNEXTOKEN method can also return the token type, parse flags, the line number, and the column number. The JSON package IS* methods (for example, ISLEFTBRACE) enable you to query the following token types:

Boolean true

Boolean false

float

integer

label
left brace ({)
left bracket ([ )
null
numeric
partial
right brace (})
right bracket ( ])
string

When you finish parsing the text, call the DESTROYPARSER method to remove the parser instance.

The following example creates a parser and uses the CREATEPARSER method to provide the input JSON text.

```
data _null_;
    method init();
        dcl package json j();
        dcl int rc tokenType parseFlags;
        dcl bigint lineNum colNum;
        dcl nvarchar(128) token abc t1;
        abc = 'xyz';
        t1 = '{"abc" : 1 }';
        rc = j.createParser( t1 );
             if (rc ne 0) then goto TestError;

        * obj open;
        j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );
        if ( rc ne 0 ) then goto TestError;

        * obj label;
        j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );
        if ( rc ne 0 ) then goto TestError;

        * obj value;
        j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );
        if ( rc  ne 0) then goto TestError;

        * obj close;
        j.getNextToken( rc, token, tokenType, parseFlags, lineNum, colNum );
        if ( rc ne 0 ) then goto TestError;

     Exit:
        rc = j.destroyParser();
        return;

    TestError:
        put 'Test ended abnormally.';
        goto Exit;

    end;
enddata;
run;
```

For more information about the methods that enable you to parse JSON text, see "DS2 JSON Package Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

Using the Logger Package

Overview of the Logger Package

In the SAS logging facility, a logger is a named entity that identifies a message category. A logger's attributes consist of a level and one or more appenders that process the log events for the message category. The level indicates the threshold, or lowest event level, that will be processed for this message category.

You use a logger package to interface with the SAS logging facility. After you declare the new logger package, you can send messages to the logger at a specified logging level.

For more information about DS2 loggers, see Appendix 3, "DS2 Loggers," on page 217.

Declaring and Instantiating a Logger Package

There are two ways to construct an instance of a logger package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package logger logpkg;
logpkg = _new_ logger();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package logger logpkg();
```

For more information, see the "DECLARE PACKAGE Statement, Logger Package" in *SAS Viya: DS2 Language Reference*, and the "_NEW_ Operator, Logger Package" in *SAS Viya: DS2 Language Reference*.

Unformatted and Formatted Messages

You can specify a raw, or unformatted, message that is written to the SAS logging facility.

You can also use $s format markers to create a formatted message. Each $s format marker in the message format is replaced by the content of the corresponding argument that you specify.

For more information, see "LOG Method, Logger Package" in *SAS Viya: DS2 Language Reference*.

Log Messages to a Table

1. Create an empty table to store the messages.

   Here is an example:

```
libname logs 'c:\temp';
data logs.edmlog;
  length seqno  8;
  length date   8; format date DATETIME19.;
  length msg    $ 256;
  stop;
run;
```

2. Create a logger and appender with an XML logging configuration file.

Here is an example that creates an App.Program logger named **App.Program.EDM**. The appender that the logger writes to is **EDMAppender**. The appender is configured to write to a SAS data set name **edmlog** in **c:\temp**. Save this XML logging configuration file as **edm-l4s.xml**.

```
<?xml version="1.0"?>
<logging:configuration xmlns:logging="http://www.sas.com/xml/logging/1.0/">

   <appender name="EDMAppender" class="DBAppender">
      <param name="ConnectionString" value="
         DRIVER=base;CATALOG=base;
         schema=(name=base;primarypath='C:\temp')"/>
      <param name="MaxBufferedEvents" value="1000"/>
      <param name="TableName" value="edmlog"/>
      <param name="Column" value="sn"/>
      <param name="Column" value="d"/>
      <param name="Column" value="m"/>
   </appender>

   <appender name="null" class="ConsoleAppender">
      <filter class="DenyAllFilter"/>
   </appender>

   <logger name="App.Program.EDM">
      <appender-ref ref="EDMAppender"/>
   </logger>

   <root>
      <appender-ref ref="null"/>
   </root>
</logging:configuration>
```

3. Start SAS with the LOGCONFIGLOC system option set to the name of the XML configuration file that you created in Step 2.

   Here is an example.

   ```
   options logconfigloc edm-l4s.xml
   ```

4. Create a DS2 logger package instance that is associated with the logger that you created in the XML configuration file from Step 2.

   Here is an example.

   ```
   proc ds2;
   data _null_;
     dcl package logger l('App.Program.EDM');
     method init();
        dcl double i;
        i = l.islevelactive(4);  put i=;
        l.log('T', 'Hello World! Trace');
        l.log('D', 'Hello World! Debug');
        l.log('I', 'Hello World! Info');
        l.log('W', 'Hello World! Warning');
        l.log('E', 'Hello World! Error');
     end;
   enddata;
   run;
   quit;
   ```

You can print the contents of the **edmlog** data set.

```
libname logs 'c:\temp';
proc print data=logs.edmlog; run;
```

The data set contents looks like this.

```
Obs    seqno                  date            msg

  1      285      25APR2013:17:11:42    Hello World! Trace
  2      286      25APR2013:17:11:42    Hello World! Debug
  3      287      25APR2013:17:11:42    Hello World! Info
  4      288      25APR2013:17:11:42    Hello World! Warning
  5      289      25APR2013:17:11:42    Hello World! Error
```

Using the MATRIX Package

Overview of the MATRIX Package

A matrix is a two-dimensional array of numeric or character values. The dimensions of a matrix are defined by the number of rows and columns. The elements of an *n* x *p* matrix are arranged in *n* rows and *p* columns.

The matrix package provides a DS2-level implementation of SAS/IML functionality. You can use matrix package methods to perform complex tasks such as matrix inversion. You can perform arithmetic, relational, and logical operations. You can perform some operations on an elementwise basis and other operations on the entire data matrix.

You can load data in a matrix package using an array or external data. You can generate data by writing the entire matrix to an array at one time or row-by-row. The array can then be written to a result table.

For more information about these methods, see "DS2 Matrix Package Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

Declaring and Instantiating a MATRIX Package

A matrix package is created by declaring and instantiating the matrix package using the DS2 DECLARE PACKAGE statement. Here is an example:

```
declare package matrix m(2, 2);
```

This statement creates a 2 x 2 matrix and stores its instance in the variable **m**.

Note: The matrix is filled with zeros, not null or missing values.

For more information, see "DECLARE PACKAGE Statement, Matrix Package" in *SAS Viya: DS2 Language Reference*.

Matrix Data Input

There are two ways to initialize or load data into a matrix package.

- initialize with zero values
  - `declare package matrix instance-name(rows,columns);`
  - `instance-name=_new_ matrix(rows,columns);`
- initialize with array values
  - `/* loads an array using the array name */`

```
   instance-name=_new_ matrix(array-name, rows, columns);
```

- ```
  /* loads an array using the IN method */

  instance-name=_new_ matrix(rows,columns);
   instance-name.in(array-name);
  ```

- ```
  /* loads row-by-row using an input table*/
  /* a variable array, and the SET statement */

  instance-name=_new_ matrix(rows,columns);
  . . .
  set table-name;
  instance-name.in(variable-array-name, i);
  ```

These examples create a 2x2 matrix that is initialized with zero values.

```
declare package matrix m(2,2);

m=_new_ matrix(2, 2);
```

These examples load array **a** into the matrix **m**.

```
/* simple array */
method init();
   dcl double a[3, 3];
   dcl package matrix m;

   a :=(1, 2, -1, 2, 1, 0, -1, 1, 2);
   m=_new_ matrix(a, 3, 3);
end;

/* variable array */
vararray double a[3,3];
dcl package matrix m;

method init();
   a := (1, 2, -1, 2, 1, 0, -1, 1, 2);
   m = _new_ matrix(a, 3, 3);
end;
```

These examples load array **va** into the matrix **m** using the IN method.

```
/* simple array */
dcl double va[3,3];
dcl package matrix m;
m = _new matrix(3,3);
m.in(va);

/* variable array */
vararray double va[3,3];
dcl package matrix m;

m.in(va); */
```

This example reads table **x** using the SET statement into variable array **a**. That variable array is then used to load the matrix **m**.

```
vararray double a[4];
dcl package matrix m;
```

```
/* Create an empty matrix to hold the input values */
method init();
   m = _new_ [this] matrix(4, 4);
   i = 1;
end;

/* Read and initialize each row of matrix from vararray a */
method run();
    set x;
    m.in(a, i);
    i + 1;
end;
```

For more information, see the "_NEW_ Operator, Matrix Package" in *SAS Viya: DS2 Language Reference* and the "IN Method" in *SAS Viya: DS2 Language Reference*.

Note: The _NEW_ operator is part of the code stream. It is not used in the declarations.

Matrix Data Output

You can generate matrix data by writing the entire matrix to an array at one time or row-by-row.

Use the TOARRAY or TOVARARRAY methods to write the matrix to an array at one time. In this example, array **a** is loading into matrix **m**. The transpose of matrix **m**is then calculated and is written to array **c**.

```
dcl double a[3,3];
dcl double b[3,3];

method run();
   dcl package matrix m r;
     dcl double i j;

   a := (1,2,3,4,5,6,7,8,9);

   m = _new_ matrix(a, 3, 3);
   r = m.trans();
   r.toarray(b);

   do i = 1 to 3;
      do j = 1 to 3;
         put b[i,j];
      end;
   end;
end;
```

For more information, see the "TOARRAY Method" in *SAS Viya: DS2 Language Reference* and the "TOVARARRAY Method" in *SAS Viya: DS2 Language Reference*.

Variable arrays can be used to write data by using the OUT method and the OUTPUT statement. The OUT method writes the matrix row data to a variable array. The OUTPUT statement writes the data to a result table. The matrices are written one row at a time to the result table a row at a time by means of the variable array. For more information, see the "OUT Method" in *SAS Viya: DS2 Language Reference* and the "Loading and Writing Data" in *SAS Viya: DS2 Language Reference*.

Matrix Operations

The following table summarizes the operations that can be performed on matrices using typical DS2 dot syntax method calls.

Table 12.1 *Matrix Operations*

Type of Operation	Method Name	Operation performed	Notes
Binary Arithmetic	ADD MULT SUB	Addition Multiplication Subtraction	• The array dimensions for the matrices used in addition or subtraction operations do not have to be compatible. But the number of columns in the first matrix has to equal the number of rows in the second matrix or the second matrix has to be a 1x1 matrix. • The array dimensions for the matrices used in a multiplication operation have to be compatible. The number of columns in the first matrix must equal the number of rows in the second matrix.
Unary	ABS COPY DET EXP FLOOR INVERSE LOG SQRT TRANS	absolute value copy matrix determinant of a square matrix exponential value integer part of each matrix value inverse natural logarithm square root transposition	• Unary operations are performed on a single matrix. • Matrices must be square or singular. For inverse and determinant operations, the matrix must be square.
Binary relational	EQ GT GE LE LT NE	equal to greater than or equal to greater than less than or equal to less than not equal to	• The result of any binary relational operation is a matrix whose entries tell how the $[i, j]^{th}$ element of the first matrix compares to the $[i, j]^{th}$ element of the second matrix. The result values are either 0 if the comparison is false or 1 if the comparison is true. • The matrix sizes must match or you can use a scalar comparison.
Binary logical	AND OR	and comparison or comparison	• The result of any binary logical operation is a matrix whose entries tell how the $[i, j]^{th}$ element of the first matrix compares to the $[i, j]^{th}$ element of the second matrix. The result values are either 0 if the comparison is false or 1 if the comparison is true.

Type of Operation	Method Name	Operation performed	Notes
ALL relational	ALL_EQ ALL_GE ALL_GT ALL_LE ALL_LT ALL_NE	ALL equal ALL greater than or equal to ALL greater than ALL less than or equal to ALL less than ALL not equal to	• The ALL relational operations produce a scalar result that indicates whether the $[i,j]^{th}$ element of the first matrix satisfies the comparison with the $[i,j]^{th}$ element of the second matrix. The scalar result is 0 or 1. All of the $[i,j]$ element comparisons must be true in order for the result to be 1. Otherwise, the result is 0. • The matrix sizes must match or you can use a scalar comparison.
ANY relational	ANY_EQ ANY_GE ANY_GT ANY_LE ANY_LT ANY_NE	ANY equal ANY greater than or equal to ANY greater than ANY less than or equal to ANY less than ANY not equal to	• The ANY relational operations produce a scalar result that indicates whether the $[i,j]^{th}$ element of the first matrix satisfies the comparison with the $[i,j]^{th}$ element of the second matrix. The scalar result is 0 or 1. If any of the $[i,j]$ element comparisons is true, the result is 1. Otherwise, the result is 0. • The matrix sizes must match or you can use a scalar comparison.
ALL logical	ALL_AND ALL_OR	ALL AND ALL OR	• The ALL logical operations produce a scalar result that indicates whether the $[i,j]^{th}$ element of the first matrix satisfies the comparison with the $[i,j]^{th}$ element of the second matrix. The scalar result is 0 or 1. All of the $[i,j]$ element comparisons must be true in order for the result to be 1. Otherwise, the result is 0. • The matrix sizes must match or you can use a scalar comparison.
ANY logical	ANY_AND ANY_OR	ANY AND ANY OR	• The ANY relational operations produce a scalar result that indicates whether the $[i,j]^{th}$ element of the first matrix satisfies the comparison with the $[i,j]^{th}$ element of the second matrix. The scalar result is 0 or 1. If any of the $[i,j]$ element comparisons is true, the result is 1. Otherwise, the result is 0. • The matrix sizes must match or you can use a scalar comparison.
Elementwise	EDIV EMAX EMIN EMOD EMULT EPOW	elementwise division elementwise maximum elementwise minimum elementwise remainder of the division of elements elementwise multiplication elementwise raise to a power	• Elementwise operations enable you to apply an operation to a general matrix using a matrix with the same dimensions, a vector whose row dimension matches the row dimension of the general matrix, a vector whose column dimension matches the column dimension of the general matrix, or a 1x1 matrix effectively allowing a scalar operation on each $[i,j]$ element. • Elementwise operations produce a result matrix from the element-by-element operations on two argument matrices.

Considerations When Using a Matrix Package

- When a matrix is created using a constructor, for example, **`m= _new_ matrix(2,2);`** or **`declare package matrix m(2, 2);`**, the matrix is filled with zeros, not missing values.
- A matrix that contains null or missing values is not very useful. If a matrix does contain null or missing values, some operations on it might be considered anomalous. For example, if you multiply matrices with null or missing values, you receive a run-time error. But if you add or subtract matrices with null or missing values, you do not receive an error. If you divide a matrix element by zero, you do not get a floating-point exception. The result will be a missing value.
- If you use a loop to read or write matrix data, you want to avoid operations on the matrix while the loop is being processed. The reason is that the matrix is only partially filled until the loop is complete.
- It is not always easy to keep track of what size matrix you have. Make sure that the dimensions for your matrix operations are consistent.
- When a matrix is declared in a thread program, each thread program has its own, individual instance of a matrix. The DS2 matrix package does not support data partitioning between nodes or threads to perform parallel matrix operations. Instead, each thread performs the matrix operations on its own instance of the matrix.

Moving Values from a Matrix into a DS2 Array

The following example shows how to use the TOARRAY method. Using this method is the only way that you can move values from a matrix into a DS2 array for use in a DS2 program.

```
dcl double a[3, 3];
dcl double c[3, 3];
declare package matrix m;

a := (1, 2, 3, 4, 5, 6, 7, 8, 9);
m=_new_ matrix(a, 3, 3);
m.toarray(c);

do i=1 to 3;
   do j=1 to 3;
      put c[i, j];
   end;
end;
```

Using the SQLSTMT Package

Overview of the SQLSTMT Package

Note: The SQLSTMT package is not supported in the CAS server.

The SQLSTMT package provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS. The FedSQL statements can create, modify, or delete tables. If the FedSQL statements selects rows from a table, the SQLSTMT package provides methods for interrogating the rows returned in a result set.

When an 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 dynamically during execution of the DS2 program.

Hadoop Distribution
If you are using a Hadoop distribution, the use of the SQLSTMT package requires Hive 0.13 or later.

Here is a simple example of using the SQLSTMT to insert values into a Teradata table.

```
dcl package sqlstmt s('insert into td.testdata (x, y, z) values (?, ?, ?)',
    [x y z]);

   do i=1 to 5;
      x=i;
      y=i*1.1;
      z=i*10.01;
      s.execute();
   end;
end;
```

The following rows are inserted into the table:

```
1 1.1 10.01
2 2.2 20.02
3 3.3 30.03
4 4.4 40.04
5 5.5 50.05
```

For more examples, see "DS2 Example Programs" in *SAS Viya: DS2 Language Reference*.

Declaring and Instantiating an SQLSTMT Package

You use the DECLARE PACKAGE statement to declare the SQLSTMT package. When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance.

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

- Use the DECLARE PACKAGE statement along with its constructor syntax. There are two syntax forms:

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

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

- Use the DECLARE PACKAGE statement along with the_NEW_ operator. There are two syntax forms:

  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.

- Use the DECLARE PACKAGE statement without 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 FedSQL Statement Parameter Values

If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values are one of the following:

- the current values of the variables specified in the constructor DECLARE PACKAGE statement or the _NEW_ operator

  For more information, see "DECLARE PACKAGE Statement, SQLSTMT Package" in *SAS Viya: DS2 Language Reference* and the "_NEW_ Operator, SQLSTMT Package" in *SAS Viya: DS2 Language Reference*.

- the current values of the variables specified in the BINDPARAMETERS method

  If you use the BINDPARAMETERS method and execute a FedSQL statement, the values of bound variables are read when the statement is executed and used as the values of the statement's parameters. If the type of a bound variable differs from the corresponding parameter's type, the bound variable's value is converted to the parameter's type. For more information, see the "BINDPARAMETERS Method" in *SAS Viya: DS2 Language Reference*.

- the values specified in the SET*type* methods

  For more information about these methods, see "DS2 SQLSTMT Package Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

Parameter values must be specified exclusively with bound variables or exclusively with the SET*type* methods.

Note: The rules for identifiers for the FedSQL language apply to variables used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL parses the string containing the SQL statement rather than DS2.

Specifying a Connection String

A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

You can specify a connection string when you declare and instantiate an SQLSTMT package.

If a 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 the 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 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.

The FedSQL statement executes dynamically at run time. 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

The FETCH method returns the next row of data from the result set. A status indicator is returned. 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 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 bound to the result set columns, the fetched data can be returned by the GET*type* methods.

Note: For character data, you can call the GET*type* method repeatedly until all of the result set column data is retrieved. For numeric data, you can call the GET*type* method only once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator. For more information, see the GET *type* methods in "DS2 SQLSTMT Package Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*.

An 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 the SQLSTMT Package and the SQLEXEC Function

Note: The SQLSTMT package and the SQLEXEC function are not supported in the CAS server.

The following table compares the SQLSTMT package and the 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

SQLSTMT Package	SQLEXEC Function
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

Using the TZ Package

Overview of the TZ Package

DS2 supports the SQL style date and time conventions that are used in other data sources. When your data source is a not a SAS data set, DS2 can process dates and times that have a data type of DATE, TIME, and TIMESTAMP. SAS date, time, and datetime values can be converted to DS2 dates, time, and timestamp values by using the TO_DATE, TO_TIME, and TO_TIMESTAMP functions. However, these functions do not incorporate a time zone.

The TZ package enables you to process local and international time and date values.

Declaring and Instantiating a TZ Package

There are two ways to construct an instance of a TZ package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package tz tzpkg;
tzpkg = _new_ tz();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package tz tzpkg();
```

For more information, see the "DECLARE PACKAGE Statement, TZ Package" in *SAS Viya: DS2 Language Reference*, and the "_NEW_ Operator, TZ Package" in *SAS Viya: DS2 Language Reference*.

Returning Time and Time Zone Information

You can use the TZ package to return the following values:

- current local time
- current Coordinated Universal Time (UTC) time
- current time zone ID
- current time zone name
- the time zone offset of the time zone from UTC at the specified local time
- the time zone offset of the time zone from UTC at the specified UTC time

Here is an example of how to use the TZ package to get the world clock.

```
data _null_ ;
   method init();

     declare package tz tzone() ;
     dcl double tokyo_time london_time new_york_time utc_time ;
```

```
    tokyo_time = tzone.getLocalTime('Asia/Tokyo') ;
    london_time = tzone.getLocalTime('Europe/London') ;
    new_york_time = tzone.getLocalTime('America/New_York') ;

    utc_time = tzone.getUTCTime();

    put utc_time = datetime. ;
    put tokyo_time = datetime. ;
    put london_time = datetime. ;
    put new_york_time = datetime. ;
  end ;
enddata ;
run;
```

The following lines are written to the SAS log:

```
utc_time=18NOV14:13:53:11
tokyo_time=18NOV14:22:53:11
london_time=18NOV14:13:53:11
new_york_time=18NOV14:08:53:11
```

For more information about the methods to perform these actions, see "DS2 TZ Package Methods, Operators, and Statements" in *SAS Viya: DS2 Language Reference*. For more information about time zone ID and names, see "Time Zone IDs and Time Zone Names" in *SAS Viya National Language Support: Reference Guide*.

Returning Time Zone Offset

You can use the TZ package to return the time zone offset from UTC at either the specified local time or at the specified UTC time.

The time zone offset specifies the number of hours and minutes that a time zone is off from the UTC in the form +|–*hhmm* or +|–*hh*:*mm*

Here is an example that returns the time zone offset from 'asia/tokyo' and from 'America/New_York'.

```
data _null_ ;
  method init();

  declare package tz tzone('asia/tokyo') ;
  dcl double new_york local_time;
  dcl char(40) cstr ;

  local_time = tzone.getOffset() ;
  put local_time time.;

  new_york = tzone.getOffset('America/New_York') ;
  put new_york time.;


  end;
  enddata ;
run;
```

The following lines are written to the SAS log:

```
9:00:00
-4:00:00
```

Converting Local or UTC Time

You can use the TZ package to convert local to one of the following time formats:

- ISO8601 with or without a time zone offset
- a TIMESTAMP string with a time zone ID
- UTC time

In addition, you can convert UTC time to local time.

In this example,

```
data _null_ ;
   method init();

      declare package tz tzone('asia/tokyo') ;
      dcl double local_time ;
      dcl char(35) local_time_iso local_time_utc local_time_tz;


      local_time = tzone.tolocaltime(15550) ;
      put local_time time.;

      local_time_iso = tzone.toiso8601(15500) ;
      put local_time_iso ;

      local_time_utc = tzone.toutctime(15500) ;
      put local_time_utc time.;

      local_time_tz =tzone.totimestampz(15500);
      put local_time_tz;

   end;
enddata ;
run;
```

The following lines are written to the SAS log.

```
13:19:10
1960-01-01T04:18:20.00+09:00
-4:41:40
1960-01-01 04:18:20.00 asia/tokyo
```

Chapter 13

Threaded Processing

Overview of Threaded Processing

In threaded processing, each concurrently executing section of code is said to be running in a **thread**. DS2 threading works well both on a machine with multiple cores and within a massively parallel processing (MPP) database.

A DS2 program processes input data and produces output data. A DS2 program can run in two different ways: as a program and as a thread. When a DS2 program runs as a program, here are the results:

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be either database tables or rows that are returned to the client application.

When a DS2 program runs as a thread, here are the results:

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.

To enable DS2 code to run in threads:

1. Create the thread by enclosing your DS2 code between THREAD...ENDTHREAD statements.
2. Create one or more instances of the thread in a DS2 program by using a DECLARE THREAD statement.
3. Execute the thread or threads by using a SET FROM statement.

In this example, a very simple thread, T, is created by using the THREAD statement.

```
thread t;
   dcl int x;
   method init();
      dcl int i;
      do i = 1 to 3;
         x = i;
         output;
      end;
   end;
endthread;
```

In this DS2 program, an instance of T is declared, and two threads are executed, using the SET FROM statement in the RUN method. Each of the two threads generates three rows for x for a total of six rows in the output table.

```
data;
  dcl thread t t_instance;
  method run();
    set from t_instance threads=2;
    put 'x= ' x ;
  end;
enddata;
```

When you run the DS2 program, the SAS log might display the following output. Because of how threads are processed, the order of the output could be different.

```
x=  1
x=  2
x=  3
x=  1
x=  2
x=  3
```

Note: If one computation thread can keep up with the I/O thread, then that single thread is used for all computation.

Note: A single reader feeds all threads. A SET statement in a thread program shares a single reader for that SET statement. Each row in the input table is sent to exactly one thread.

For more information, see the "THREAD Statement" in *SAS Viya: DS2 Language Reference*.

Threading and DS2 Programs

Overview of Serial and Parallel Programs

A DS2 program can perform manipulations on multiple data observations, thus concurrently reducing the time required to process big data sets. Based on the structure of the DS2 program, the DS2 compiler determines which operations can be performed on multiple observations concurrently and which operations must be applied to each observation sequentially. A DS2 program is classified as either a serial program, parallel program, or parallel-serial program.

serial program
: contains operations with data dependencies across observations. Thus, observations must be processed in serial. One thread processes the complete data set and generates the complete result set.

parallel program
: contains no operations with data dependencies across observations. Thus, multiple data observations can be processed in parallel. Each thread processes a subset of the data set and generates a subset of the result set.

parallel-serial program
: contains some operations with data dependencies across observations and some operations without data dependencies. The processing of the operations is divided into two stages, a parallel stage and a serial stage. During the parallel stage, each thread processes a subset of the input data set and generates a subset of an intermediate data set. During the serial stage, one thread processes the complete intermediate data set and generates the complete result set.

Note: For information about how DS2 threads are run in CAS, see "How DS2 Runs in CAS" on page 193.

Data Manipulation Operations

A DS2 data manipulation operation is classified as either a serial operation or a parallel operation.

serial operation
: operation having data dependencies across observations.

- Serial operations must be applied sequentially to each data observation.
- Serial operations are implemented by statements in a DS2 data program.

parallel operation
: operation having no data dependencies across observations.

- Parallel operations can be applied to multiple data observations in parallel.
- Parallel operations are implemented by statements in a DS2 thread program.

The DS2 compiler categorizes statements based on the placement of the statements within the DS2 program. The DS2 compiler assumes that all statements in a thread program implement parallel operations and all statements in a data program implement serial operations. The only exceptions are data input and output statements. The DS2 compiler does not consider SET, SET FROM, or OUTPUT as data manipulation statements.

DS2 Program Summary

A DS2 program is classified as either a serial program, parallel program, or a parallel-serial program based on the type of programs and type of data manipulation operations that it contains.

DS2 serial program
: program that contains only serial operations.

- Has only a data program (no thread program).
- Data program contains serial data manipulation operations.

DS2 parallel program
: program that contains only parallel operations.

- Has a thread program and a data program.
- Thread program contains parallel data manipulation operations.
- Data program contains no data manipulation operations. That is, the data program does not contain any statements besides SET FROM and OUTPUT.

DS2 Parallel-Serial Program
: program contains both parallel and serial operations.

- Has a thread program (parallel stage) and a data program (serial stage).
- Thread program contains parallel data manipulation operations.
- Data program contains serial data manipulation operations. That is, the data program contains at least one statement besides SET FROM and OUTPUT.

Automatic Variables That Are Useful in DS2 Threading

There are several automatic variables that are used for subsetting a problem across DS2 threads. These automatic variables are also useful for providing context when you are debugging with PUT statements.

Variable	Description
HOSTNAME	Returns the name of the worker node or host on which the DS2 program is running.
NTHREADS	Total number of DS2 threads running in the program. In a parallel environment, _NTHREADS_ is the total number of DS2 threads across all nodes on which the DS2 program is running.
THREADID	One-based thread number for this DS2 thread in the running program. In a parallel environment, _THREADID_ is unique across all DS2 threads on all nodes on which the DS2 program is running. In an executing thread stage, _THREADID_ = 1 ... _NTHREADS_. In an executing data stage, _THREADID_ = 0.

Chapter 14

Using DS2 and FedSQL

Dynamically Executing FedSQL Statements from DS2

Note: Using a FedSQL query to select the data is not currently supported in the CAS server.

You can embed and execute FedSQL statements from within your DS2 programs. You can use FedSQL with DS2 in the following instances:

- You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement.
- You can use the SQLSTMT package to generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table at run time.

  The SQLSTMT package is intended for use with FedSQL statements that are executed multiple times, statements with parameters, or statements that generate a result set. For more information, see "Using the SQLSTMT Package" on page 135.
- You can also use the SQLEXEC function to generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table at run time.

  The SQLEXEC function is intended for use with FedSQL statements that are executed only one time, do not have parameters, and do not produce a result set.

  For more information, see the "SQLEXEC Function" in *SAS Viya: DS2 Language Reference*.
- You can load data into a hash instance at run time by using a FedSQL SELECT statement in the DECLARE PACKAGE statement or the DATASET method.

  For more information, see "Using a FedSQL Query with a Hash Instance to Get Rows Dynamically at Run Time" on page 118 the "DATASET Method" in *SAS Viya: DS2 Language Reference* and the "DECLARE PACKAGE Statement, Hash Package" in *SAS Viya: DS2 Language Reference*.
- You can use the SET statement to read in data by using a FedSQL SELECT statement.

  For more information, see "SET Statement with Embedded FedSQL" on page 152 and the "SET Statement" in *SAS Viya: DS2 Language Reference*.

Chapter 15

DS2 Input and Output

Overview of DS2 Input and Output

You can use these methods to read DS2 data:

- You can generate data using variable or array assignment within packages and methods in your DS2 program.
- You can read an existing table by using the SET statement.

  For more information, see "Reading Data Using the SET Statement" on page 150 and the "SET Statement" in *SAS Viya: DS2 Language Reference*.
- You can read data using a hash package.

  For more information, see "Reading Data Using the Hash Package" on page 153.
- You can retrieve data using an SQLSTMT package by executing a SELECT statement and accessing the result set.

  Note: The SQLSTMT package is not supported in the CAS server.

  For more information, see "Reading and Writing Data Using the SQLSTMT Package and the SQLEXEC Function" on page 153.

You can use these methods to write DS2 data:

- You write DS2 data by using the OUTPUT statement. The OUTPUT statement writes rows to a result table.

  For more information, see "Writing Data Using the OUTPUT Statement" on page 154.

- You can use the hash package OUTPUT method.

  For more information, see "OUTPUT Method" in *SAS Viya: DS2 Language Reference* and "Saving Hash Package Data in a Table" on page 116.

- You can use FedSQL INSERT and UPDATE statements in the SQLEXEC function or the SQLSTMT package.

  Note: The SQLEXEC function is not supported in the CAS server.

  "Reading and Writing Data Using the SQLSTMT Package and the SQLEXEC Function" on page 153, "SQLEXEC Function" in *SAS Viya: DS2 Language Reference*, and "Accessing Result Set Data" on page 138.

Reading Data Using the SET Statement

Overview of the SET Statement

The SET statement is flexible and has a variety of uses in DS2 programming. These uses are determined by the options and statements that you use with the SET statement:

- reading rows and columns from existing tables for further processing in a DS2 program
- concatenating and interleaving tables, and performing one-to-one reading of tables

  For more information, see Chapter 16, "Combining Tables," on page 157.

Each time the SET statement executes, one row is read into the program data vector. SET reads all columns and all rows from the input tables unless you specify otherwise. A SET statement can contain multiple tables; a DS2 program can contain multiple SET statements.

For more information, see "SET Statement" in *SAS Viya: DS2 Language Reference*.

Note: A SET statement in a thread program shares a single reader for that SET statement. Each row in the input table is sent to exactly one thread.

Note: The SET statement is best used in the RUN method to take advantage of the RUN method's implicit looping capability.

SET Statement Compilation

When the DS2 compiler evaluates a SET statement, it reads the column information for each *table-reference*. For each column in each table, it creates a global variable in the DS2 program with the same type attributes as those of the column. In this way, the SET statement creates a set of associated column variables. An error occurs if a global variable already exists and the type does not match the type of the table column. This means that if two tables have a column with the same name, those columns' types must be compatible.

SET Statement Execution

In a data program, when a SET statement executes, it reads the first row from the first table. Successive executions continue to read rows until the current table has been completely read. Then the first row from the next table is read. Each time a row is read, the row values are assigned to the corresponding column variables. Column variables that are created by the SET statement are retained. SET statement execution ends when all rows from all tables have been completely read.

For a thread program, the order of rows entering any given thread is undefined unless you use a BY statement. Even then, there is no way in a thread program for a thread to assume that all the rows are received before any rows from the other table or tables.

Any column variable that does not appear in the table being read is set to a SAS missing value or a null value depending on whether you are in SAS or ANSI mode. Variables that are declared by a SET statement are initialized as follows:

- In SAS mode:
  - DOUBLE data types are initialized to the SAS numeric missing value (.)
  - Fixed-width CHAR and NCHAR data types are initialized to the SAS character missing value (a blank filled string).
  - All other data types, including VARCHAR and NVARCHAR are initialized to ANSI null.
- In ANSI mode all types are initialized to ANSI null.

Note: For more information, see Chapter 7, "How DS2 Processes Nulls and SAS Missing Values," on page 43.

Here is an example. In this program, the column variable *modifiers* is not specified in the second and fourth rows. The OUTPUT method uses the values of all defined variables whether they have been assigned a value after the last OUTPUT statement.

```
proc ds2;
data test (overwrite=yes);
   dcl char string1 string2 modifiers having informat $char8. format $char8.;
   method init();
      string1='aBc'; string2='AbC'; modifiers='i'; output;
      string1='   abc'; string2='abc'; output;
      string1='   abc'; string2='abc';  modifiers='l'; output;
      string1=' abc'; string2='   abx'; output;
      string1=' abc'; string2='   abx'; modifiers='l';  output;
   end;
enddata;
run;

data test_out (overwrite=yes);
   dcl double result;
   method run();
      set test;
      result=compare(string1, string2, modifiers);
      put 'String 1= ' string1 'String 2= ' string2 'Modifier= ' modifiers
         'Result= ' result;
   end;
enddata;
run;
```

```
quit;
```

The following lines are written to the SAS log.

```
String 1=  aBc      String 2=  AbC      Modifier=  i        Result=  0
String 1=     abc   String 2=  abc      Modifier=  i        Result=  -1
String 1=     abc   String 2=  abc      Modifier=  l        Result=  0
String 1=   abc     String 2=     abx   Modifier=  l        Result=  -3
String 1=   abc     String 2=     abx   Modifier=  l        Result=  -3
```

To ensure that the second and fourth rows do not have a value specified for the *modifiers* column variable, you must set the variable to a missing or null value.

```
proc ds2;
data test (overwrite=yes);
   dcl char string1 string2 modifiers having informat $char8. format $char8.;
      method init();
         string1='aBc'; string2='AbC'; modifiers='i'; output;
         string1='   abc'; string2='abc'; modifiers=' '; output;
         string1='   abc'; string2='abc';  modifiers='l'; output;
         string1=' abc'; string2='   abx'; modifiers=' '; output;
         string1=' abc'; string2='   abx'; modifiers='l';  output;
      end;
enddata;
run;

data test_out (overwrite=yes);
   method run();
      set test;
      result=compare(string1, string2, modifiers);
      put 'String 1= ' string1 'String 2= ' string2 'Modifier= ' modifiers
         'Result= ' result;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
String 1=  aBc      String 2=  AbC      Modifier=  i        Result=  0
String 1=     abc   String 2=  abc      Modifier=          Result=  -1
String 1=     abc   String 2=  abc      Modifier=  l        Result=  0
String 1=   abc     String 2=     abx   Modifier=          Result=  2
String 1=   abc     String 2=     abx   Modifier=  l        Result=  -3
```

SET Statement with Embedded FedSQL

A SET statement can use FedSQL code to read a table, as in this example:

```
set {select * from catalog_base.investment};
```

Note: Using a FedSQL query to select the data is not supported in the CAS server.

It is possible to interleave table names and embedded FedSQL in a SET statement. In this example, the SET statement reads the same table twice:

```
set {select * from catalog_base.investment} catalog_base.investment;
```

Embedded FedSQL used in a SET statement must be valid FedSQL code, and it must resolve to a set of table rows. Otherwise, an error occurs.

Note: There is no guarantee on the order or rows that is surfaced from embedded SQL, regardless of what that embedded SQL contains:

```
set {select * from sql13a order by "X", "Y"};
```

Some environments might preserve the order imposed by the ORDER BY clause, but others do not. DS2 wraps the embedded SQL with additional code. Also, the ORDER BY clause is not honored because DS2 cannot detect and produce FIRST or LAST information. A better way is to write the program with the BY outside of the embedded SQL text:

```
set {select * from sql13a}; by "X" "Y";
```

Reading Data Using the Hash Package

You can use a hash package to read data from a table.

The DECLARE PACKAGE statement, the _NEW_ operator, and the DATASET method accept a name that identifies a table as a data source. either of these methods to identify a data source:

- a name that identifies a table
- a valid FedSQL SELECT statement that resolves to a set of table rows

  Note: Using a FedSQL query to select the data is not currently supported in the CAS server.

For more information, see "Using the Hash Package" on page 109.

Reading and Writing Data Using the SQLSTMT Package and the SQLEXEC Function

The SQLSTMT package and the SQLEXEC function enable DS2 programs to dynamically generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table. With an instance of the SQLSTMT package or the SQLEXEXC function, the FedSQL statement allocate, prepare, execute, and free occurs at run time.

In addition, the SQLSTMT package can interrogate the result set that is produced by the executed FedSQL statement.

Note: The SQLSTMT package and the SQLEXEC function are not supported in the CAS server.

For more information, see "Using the SQLSTMT Package" on page 135 and the "SQLEXEC Function" in *SAS Viya: DS2 Language Reference*.

TIP By using a parameterized FedSQL query in the SQLSTMT package, you can achieve what the DATA step SET statement with KEY= does.

Writing Data Using the OUTPUT Statement

The OUTPUT statement creates an output row, using values for the row that are contained in the global variables when the output statement executes. The OUTPUT statement writes the current row to a table immediately, not at the end of the DS2 program. If no table name is specified in the OUTPUT statement, the row is written to the table or tables that are listed in the DATA statement.

DS2 keeps track of the values in the order in which the compiler encounters them within a DS2 program, whether they are read from existing tables or created in the program.

If you do not supply an OUTPUT statement, DS2 adds one implicitly at the end of the RUN method that writes rows to the table or tables that are being created.

Placing an explicit OUTPUT statement in a DS2 program overrides the automatic output, and adds a row to a table only when an explicit OUTPUT statement is executed. Once you use an OUTPUT statement to write a row to any one table, however, there is no implicit OUTPUT statement at the end of the RUN method. In this situation, a DS2 program writes a row to a table only when an explicit OUTPUT executes. You can use the OUTPUT statement alone or as part of an IF-THEN/ELSE or SELECT statement or in DO loop processing.

Note: OUTPUT statements in thread programs cannot contain any table names. Each output row is returned to the data program that started the thread.

Column Order in Output Tables When Using Data Sources Outside SAS

Some data sources choose their one preferred order for columns in the output table from DS2. For example, on Hive, the BYPARTITION columns are always moved to the end of the table. This is common as various data sources try to optimize their performance.

The order of declaration in a DS2 program might not be used as the order of columns in the data source. For example, if you use **`keep K1- - K4;`**, you might not get the columns as you expect or you might get an error because **`K1`** appears after **`K4`** in the CREATE TABLE statement.

NLS Transcoding Failures

Transcoding is the process of converting character data from one encoding to another encoding. An NLS transcoding failure can occur during row input or output operations, or during string assignment. By default, this run-time error causes row processing to halt. You can change the default behavior by using one of the following options:

- SAS Federation Server: specify the DEFAULT_ATTR= connection option with the XCODE_WARN=n statement handle option.
- PROC FEDSQL and PROC DS2: set the XCODE= option.

Using the options, you can choose to ignore the errors and continue processing of the row.

For more information, see the SAS Federation Server and SAS procedure documentation.

Chapter 16

Combining Tables

Definitions for Combining Data

In the context of D2 processing, combining data has these meanings:

- concatenating
- interleaving
- one-to-one reading
- match-merging

The two statements that are used for combining tables are MERGE and SET.

Combining Tables: Basic Concepts

What You Need to Know Before Combining Information Stored in Multiple Tables

Many applications require input data to be in a specific format before the data can be processed to produce meaningful results. The data typically comes from multiple sources and might be in different formats. Therefore, you often, if not always, have to take

intermediate steps to logically relate and process data before you can analyze it or create reports from it.

Application requirements vary, but there are common factors for all applications that access, combine, and process data. Once you have determined what you want the output to look like, you must perform the following tasks:

- Determine how the input data is related.
- Ensure that the data is properly sorted, if necessary.
- Select the appropriate access method to process the input data.
- Select the appropriate tools to complete the task.

The Four Ways That Data Can Be Related

Data Relationship Categories

Relationships among multiple sources of input data exist when each of the sources contains common data, either at the physical or logical level. For example, employee data and department data could be related through an employee ID column that shares common values. Another table could contain numeric sequence numbers whose partial values logically relate it to a separate table by row number.

You must be able to identify the existing relationships in your data. This knowledge is crucial for understanding how to process input data in order to produce desired results. All related data falls into one of these four categories, characterized by how rows relate among the tables:

- one-to-one
- one-to-many
- many-to-one
- many-to-many

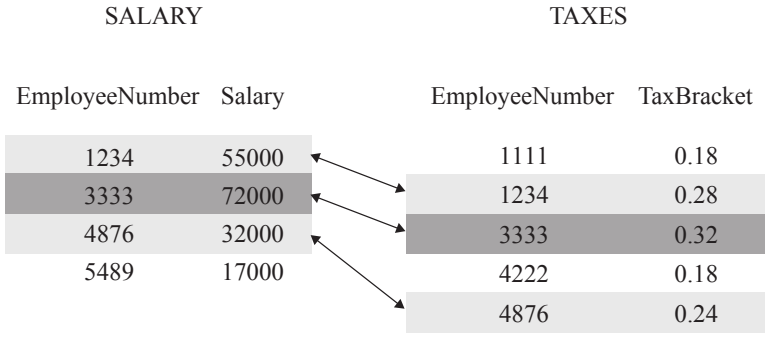
To obtain the results that you want, you should understand how each of these methods combines rows, how each method treats duplicate values of common columns, and how each method treats missing values or nonmatched values of common columns. Some of the methods also require that you preprocess your tables by sorting them. See the description of each method in “Methods for Combining Tables” on page 160.

One-to-One Relationship

In a one-to-one relationship, typically a single row in one table is related to a single row from another based on the values of one or more selected columns. A one-to-one relationship implies that each value of the selected column occurs no more than once in each table. When you work with multiple selected columns, this relationship implies that each combination of values occurs no more than once in each table.

In the following example, rows in tables Salary and Taxes are related by common values for EmployeeNumber.

Figure 16.1 *One-to-One Relationship*

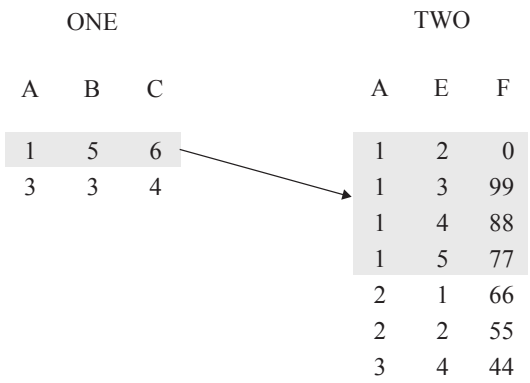


One-to-Many and Many-to-One Relationships

A one-to-many or many-to-one relationship between input tables implies that one table has at most one row with a specific value of the selected column, but the other input table can have more than one occurrence of each value. When you work with multiple selected columns, this relationship implies that each combination of values occurs no more than once in one table. However, the combination can occur more than once in the other table. The order in which the input tables are processed determines whether the relationship is one-to-many or many-to-one.

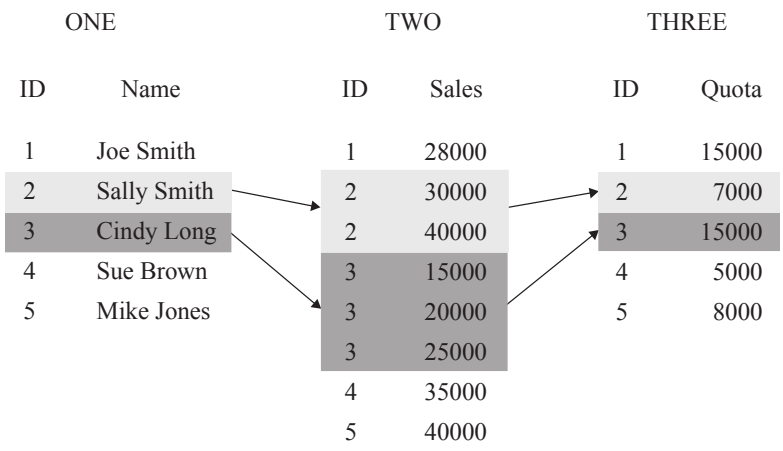
In the following example, rows in tables One and Two are related by common values for column A. Values of A are unique in table One but not in table Two.

Figure 16.2 *One-to-Many Relationship*



In the following example, rows in tables One, Two, and Three are related by common values for column ID. Values of ID are unique in tables One and Three but not in Two. For values 2 and 3 of ID, a one-to-many relationship exists between rows in tables One and Two, and a many-to-one relationship exists between rows in tables Two and Three.

Figure 16.3 *One-to-Many and Many-to-One Relationships*

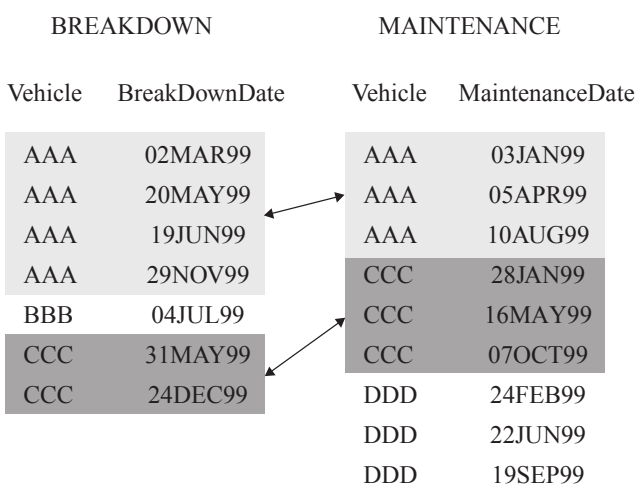


Many-to-Many Relationships

The many-to-many category implies that multiple rows from each input table can be related based on values of one or more common columns.

In the following example, rows in tables BreakDown and Maintenance are related by common values for column Vehicle. Values of Vehicle are not unique in either table. A many-to-many relationship exists between rows in these tables for values AAA and CCC of Vehicle.

Figure 16.4 *Many-to-Many Relationship*



Overview of Methods for Combining Tables

Methods for Combining Tables

You can use these methods to combine tables:

- concatenating
- interleaving
- one-to-one reading
- match-merging

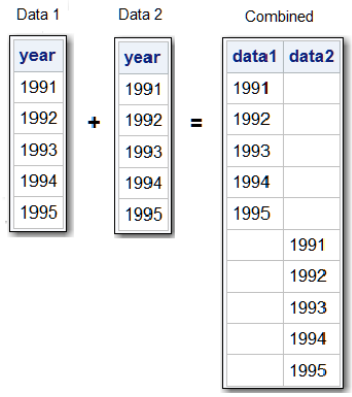
Concatenating

The following figure shows the results of concatenating two tables. Concatenating the tables appends the rows from one table to another table. The data program reads Data1 sequentially until all rows have been processed, and then reads Data2. Table Combined contains the results of the concatenation. Note that the tables are processed in the order in which they are listed in the SET statement.

```
data concatenate;
   method run();
      set data1 data2;
   end;
enddata;
run;
```

Note: For a thread program, the order of rows entering any given thread is undefined unless you use a BY statement. Even then, there is no way in a thread program for a thread to assume that all the rows are received before any rows from the other table or tables.

Figure 16.5 *Concatenating Two Tables*

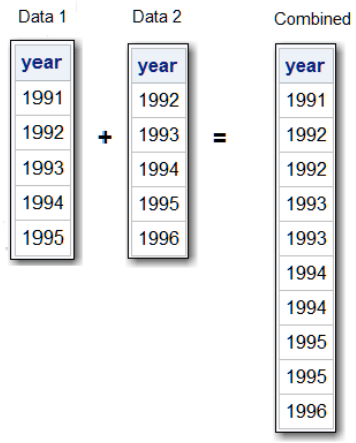


Interleaving

The following data program interleaves two tables. Interleaving intersperses rows from two or more tables, based on one or more common columns. Table Combined shows the results.

```
data interleave;
   method run();
      set data1 data2;
      by year;
   end;
enddata;
run;
```

Figure 16.6 *Interleaving Two Tables*



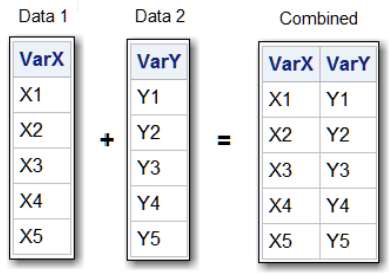
One-to-One Reading

The following figure shows the results of one-to-one reading. One-to-one reading combines rows from two or more tables by creating rows that contain all of the columns from each contributing table. Rows are combined based on their relative position in each table, that is, the first row in one table with the first in the other, and so on. The data program stops after it has read the last row from the smallest table. Table Combined shows the results.

```
data one2one;
   method run();
      set data1;
      set data2;
   end;
enddata;
run;
```

The following data program results in one-to-one reading of two tables.

Figure 16.7 *One-to-One Reading*



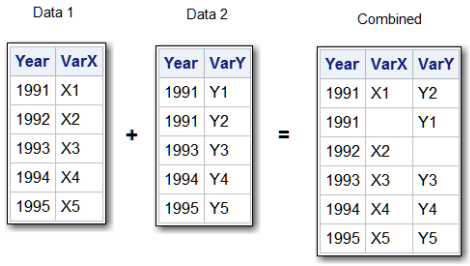
Match-Merging

The following figure shows the results of match-merging. Match-merging combines rows from two or more tables into a single row in a new table based on the values of one or more common columns. The following data program results in one-to-one reading of two tables. Table Combined shows the results.

```
data one2one;
   method run();
      merge data1 data2;
```

```
      by year;
   end;
enddata;
run;
```

Figure 16.8 *Match-Merging Two Tables*



How to Prepare Your Tables

Guidelines to Prepare Your Tables

Before combining tables, follow these guidelines to produce the results that you want:

- Know the structure and the contents of the tables.
- Look at sources of common problems.
- Ensure that rows are in the correct order, or that they can be retrieved in the correct order (for example, by presorting them).
- Test your program.

Knowing the Structure and Contents of the Tables

To help determine how your data is related, look at the structure of the tables. To see the table structure, execute the DATASETS procedure, the CONTENTS procedure, or access the SAS Explorer window in your windowing environment to display the descriptor information. Descriptor information includes the number of rows in each table, the name and attributes of each column, and an alphabetic list of extended attributes (including table and column extended attributes). To print a sample of the rows, use the PRINT procedure or the REPORT procedure.

You can also use functions such as VTYPE and VLENGTH to show specific descriptor information. For more information, see "DS2 Functions" in *SAS Viya: DS2 Language Reference*.

Looking at Sources of Common Problems

If your program does not execute correctly, review your input data for the following errors:

- columns that have the same name but that represent different data

  DS2 includes only one column of a given name in the new table. If you are merging two tables that have columns with the same names but different data, the values from the last table that was read are written over the values from other tables.

  To correct the error, you can rename columns before you combine the tables by using the RENAME= table option in the SET or MERGE statement. Or you can use the DATASETS procedure.

- common columns with the same data but different attributes

  The way DS2 handles these differences depends on which attributes are different:

  - type attribute

    If the type attributes are incompatible, DS2 stops processing the data program and issues an error message stating that the columns are incompatible.

    To correct this error, you must use a separate DS2 data program to change the types as necessary. The DS2 statements that you use depend on the nature of the column. The following table contains the result attribute when combining columns.

Table 16.1 *Type Promotion*

Left-hand side	Right-hand side	Result
BIGINT	BIGINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	SMALLINT	BIGINT
BIGINT	TINYINT	BIGINT
INTEGER	BIGINT	BIGINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
INTEGER	TINYINT	INTEGER
SMALLINT	BIGINT	BIGINT
SMALLINT	INTEGER	INTEGER
SMALLINT	SMALLINT	SMALLINT
SMALLINT	TINYINT	INTEGER
TINYINT	BIGINT	BIGINT
TINYINT	INTEGER	INTEGER
TINYINT	SMALLINT	INTEGER
TINYINT	TINYINT	TINYINT
DOUBLE	DOUBLE	DOUBLE
DOUBLE	FLOAT	DOUBLE
INTEGER	DECIMAL	DOUBLE
REAL	REAL	DOUBLE
BIGINT	DOUBLE	DOUBLE
DECIMAL (X,Y)	DECIMAL (X,Z)	DOUBLE
TIME*	TIME*	TIME*
TIMESTAMP	TIMESTAMP	TIMESTAMP
TIMESTAMP	DATE	ERROR
DATE	DATE	DATE

Left-hand side	Right-hand side	Result
VARBINARY*	VARBINARY	VARBINARY
BINARY	BINARY	BINARY
BINARY	VARBINARY	VARBINARY
CHAR(N)	CHAR(N)	CHAR(N)
CHAR(N) **	CHAR(N) **	VARCHAR(N) **
VARCHAR(N)	VARCHAR(N)	VARCHAR(N)
VARCHAR(N)	VARCHAR(M)	VARCHAR(MAX (N, M))
CHAR(N)	CHAR(M)	VARCHAR (MAX (N, M))
VARCHAR(N)	CHAR(M)	VARCHAR (MAX (N, M))
NCHAR(M)	VARCHAR(M)	VARCHAR(M)
NCHAR(M)	NCHAR(M)	NCHAR(M)
INTEGER	CHARACTER	ERROR
DATE	DOUBLE	ERROR

* No Hive support

** For DB2, Impala, ODBC, and Oracle

Note: A best practice is to declare every variable in a thread program. By doing so, you can avoid type mismatches among data sources.

- length attribute

  If the length attribute is different, DS2 takes the length from the table that contains the column with the maximum length. In the following example, all tables that are listed in the MERGE statement contain the column Mileage. In Quarter1, the length of the column Mileage is four bytes; in Quarter2, it is eight bytes and in Quarter3 and Quarter4, it is six bytes. In the output table Yearly, the length of the column Mileage is eight bytes, which is the length derived from Quarter2.

```
data yearly;
   dcl char(4) quarter1 char(8) quarter2 char(6) quarter3 quarter4;
   method run();
      merge quarter1 quarter2 quarter3 quarter4;
      by Account;
   end;
enddata;
run;
```

- label, format, and informat attributes

  If any of these attributes are different, DS2 takes the attribute from the first table that contains the column with that attribute. However, any label, format, or informat that you explicitly specify overrides a default. If all tables contain explicitly specified attributes, the one specified in the first table overrides the others.

You can also use functions such as VLABEL to show specific descriptor information. For more information, see "DS2 Functions" in *SAS Viya: DS2 Language Reference*.

Testing Your Program

As a final step in preparing your tables, you should test your program. Create small temporary tables that contain a sample of rows that test all of your program's logic. If your logic is faulty and you get unexpected output, you can debug your program.

Combining DS2 Tables: Methods

Concatenating

Definition

Concatenating tables is the combining of two or more tables, one after the other, into a single table. The number of rows in the new table is the sum of the number of rows in the original tables. The order of row is sequential. All rows from the first table are followed by all rows from the second table, and so on.

In the simplest case, all input tables contain the same columns. If the input tables contain different columns, rows from one table have missing values for columns defined only in other tables. In either case, the columns in the new table are the same as the columns in the old tables.

Syntax

Use this form of the SET statement to concatenate tables:

SET *table(s)*;

Arguments

table(s)
specifies any valid table name.

For more information, see "SET Statement" in *SAS Viya: DS2 Language Reference*.

DS2 Processing during Concatenation

Compilation phase
DS2 reads the descriptor information of each table that is named in the SET statement and then creates a program data vector that contains all the columns from all tables as well as columns created by the data program.

Execution — Step 1
DS2 reads the first row from the first table into the program data vector. It processes the first row and executes other statements in the data program. It then writes the contents of the program data vector to the new table.

The SET statement does not reset the values in the program data vector to missing, except for columns whose value is calculated or assigned during the data program. Columns that are created by the data program are set to missing at the beginning of each iteration of the data program unless they are retained. Variables that are read from a table are not.

Execution — Step 2

In the data program, DS2 continues to read one row at a time from the first table until it finds an end-of-file indicator. The values of the columns in the program data vector are then set to missing, and the data program begins reading rows from the second table, and so on, until it reads all rows from all tables. For a thread program, the order of rows entering any given thread is undefined unless you use a BY statement. Even then, there is no way in a thread program for a thread to assume that all the rows are received before any rows from the other table or tables.

Example 1: Concatenation Using the Data Program

In this example, each table contains the columns Common and Number, and the rows are arranged in the order of the values of Common. Generally, you concatenate tables that have the same columns. In this case, each table also contains a unique column to show the effects of combining tables more clearly. The following program uses a SET statement to concatenate the tables and then prints the results:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal number;
   method init();
      common='a'; animal='Ant'; number='5'; output;
      common='b'; animal='Bird'; number=''; output;
      common='c'; animal='Cat'; number='17'; output;
      common='d'; animal='Dog'; number='9';  output;
      common='e'; animal='Eagle'; number=''; output;
      common='f'; animal='Frog';  number='76'; output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant number;
   method init();
      common='g'; plant='Grape'; number='69'; output;
      common='h'; plant='Hazelnut'; number='55'; output;
      common='i'; plant='Indigo'; number=''; output;
      common='j'; plant='Jicama';  number='14'; output;
      common='k'; plant='Kale'; number='5'; output;
      common='l'; plant='Lentil';  number='77'; output;
   end;
enddata;
run;


/*  set concatenates */
data concatenate (overwrite=yes);
   method run();
      set animal plant;
   end;
enddata;
run;

proc print data=concatenate;
run;
quit;
```

Output 16.1 *Concatenated Table (SET Statement)*

Animal

common	animal	number
a	Ant	5
b	Bird	
c	Cat	17
d	Dog	9
e	Eagle	
f	Frog	76

+

Plant

common	plant	number
g	Grape	69
h	Hazelnut	55
i	Indigo	
j	Jicama	14
k	Kale	5
l	Lentil	77

=

Concatenate

common	animal	number	plant
a	Ant	5	
b	Bird		
c	Cat	17	
d	Dog	9	
e	Eagle		
f	Frog	76	
g		69	Grape
h		55	Hazelnut
i			Indigo
j		14	Jicama
k		5	Kale
l		77	Lentil

The resulting table, Concatenate, has 12 rows, which is the sum of the rows from the combined tables. The program data vector contains all columns from all tables. The values of columns found in one table but not in another are set to missing.

Example 2: Concatenation Using SQL

You can also use the SQL language to concatenate tables. In this example, SQL reads each row in both tables and creates a new table named Combined. The following shows the YEAR1 and YEAR2 input tables:

The following SQL code creates and prints the table Combined.

```
proc sql;
   create table combined as
      select * from animal
            union all
         select * from plant;
         select * from combined;
quit;

proc print data=combined;
run;
quit;
```

The output is exactly the same as the output when using the SET statement.

Interleaving

Definition

Interleaving uses a SET statement and a BY statement to combine multiple tables into one new table. This is also known as BY-group processing. BY-group processing is a method of combining rows from one or more tables that are grouped or ordered by

values of one or more common columns. When a BY statement is specified immediately after a SET statement, the SET statement interleaves the rows of the input tables in sorted order. The sort order or sort key is specified by the column names in the BY statement. The number of rows in the new table is the sum of the number of rows from the original tables.

The keyword DESCENDING can be used before the name of the column in the BY statement in order to sort that column in descending instead of ascending order.

Syntax

Use this form of the SET statement to interleave tables when you use a BY variable:

SET *table(s)*;

BY <DESCENDING>*column* <…<DESCENDING> *column*>;

Arguments

table
specifies a table name.

DESCENDING
specifies that the tables are sorted in descending order by the column that is specified. DESCENDING means largest to smallest for numeric columns, or reverse alphabetical for character columns.

column
specifies each column by which the table is sorted. These columns are referred to as BY variables for the current data program.

Sort Requirements

When a BY statement is used, internally DS2 requests the rows in sorted order. If the rows are already sorted, "re-sorting" of the data might be necessary.

Note: When the SAS In-Database Code Accelerator executes the DS2 programs, BY groups might not necessarily be in sorted order depending on whether the host environment supports sorting as opposed to simply hashing. The SAS In-Database Code Accelerator chooses the most efficient technique for clustering like values together without causing local re-sorting of potentially large data volumes.

DS2 Processing during Interleaving

Compilation phase

- DS2 reads the descriptor information of each table that is named in the SET statement and then creates a program data vector that contains all the columns from all tables as well as columns created by the data program.
- In the DS2 program, SAS identifies the beginning and end of each BY group by creating two temporary variables for each BY column: FIRST.*variable* and LAST.*variable*. Their values indicate whether a row has the following characteristics:
  - the first one in a BY group
  - the last one in a BY group
  - neither the first nor the last one in a BY group
  - both first and last, as is the case when there is only one row in a BY group.

  When a row is the first in a BY group, SAS sets the value of FIRST.*variable* to 1 for the column whose value changed, as well as for all of the columns that follow in the BY statement. For all other rows in the BY group, the value of

FIRST.variable is 0. Likewise, if the row is the last in a BY group, SAS sets the value of LAST.variable to 1 for the column whose value changes on the next row, as well as for all of the columns that follow in the BY statement. For all other rows in the BY group, the value of LAST.variable is 0. For the last row in a table, the values of all LAST.variable variables are set to 1. These temporary variables are available for DS2 programming but are not added to the output table.

You can take actions conditionally, based on whether you are processing the first or the last row in a BY group.

Note: See "Interleaving Tables" in *SAS Viya: DS2 Language Reference* for an example that illustrates BY-group processing.

Note: For an SPD Engine data set, utility files are used for certain operations that need extra space. The BY statement requires a utility file and the SAS UTILLOC= system option allocates space for that utility file. For more information, see the SAS UTILLOC= system option in *SAS Viya System Options: Reference*.

Execution — Step 1

DS2 compares the first row from each table that is named in the SET statement to determine which BY group should appear first in the new table. It reads all rows from the first BY group from the selected table. If this BY group appears in more than one table, it reads from the tables in the order in which they appear in the SET statement. The values of the columns in the program data vector are set to missing each time DS2 starts to read a new table and when the BY group changes.

Execution — Step 2

DS2 compares the next rows from each table to determine the next BY group and then starts reading rows from the selected table in the SET statement that contains rows for this BY group. DS2 continues until it has read all rows from all tables.

Example 1: Interleaving in the Simplest Case

In this example, each table contains the BY variable Common, and the rows are arranged in order of the values of the BY variable. The following example creates the Animal and the Plant input tables.

```
data animal(overwrite=yes);
   dcl varchar(10) common animal number;
   method init();
      common='a'; animal='Ant'; number='5'; output;
      common='b'; animal='Bird'; number=''; output;
      common='c'; animal='Cat'; number='17'; output;
      common='d'; animal='Dog'; number='9';  output;
      common='e'; animal='Eagle'; number=''; output;
      common='f'; animal='Frog';  number='76'; output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant number;
   method init();
      common='a'; plant=''; number='69'; output;
      common='b'; plant='Bamboo'; number='55'; output;
      common='c'; plant='Cabbage'; number=''; output;
      common='d'; plant='Daffodil';  number='14'; output;
```

```
      common='e'; plant='Eucalyptus; number='5'; output;
      common='f'; plant='Fig';  number='77'; output;
   end;
enddata;
run;
```

The following program uses SET and BY statements to interleave the tables, and prints the results:

```
/*  set with by interleaves */
data interleave (overwrite=yes);
   method run();
      set animal plant; by common;
   end;
enddata;
run;

proc print data=interleave;
run;
quit;
```

Output 16.2 *Interleaved Table (SET Statement)*

Animal

common	animal	number
a	Ant	5
b	Bird	
c	Cat	17
d	Dog	9
e	Eagle	
f	Frog	76

+

Plant

common	plant	number
a	Azalea	69
b	Bamboo	55
c	Cabbage	
d	Daffodil	14
e	Eucalyptus	5
f	Fig	77

=

Interleave

common	animal	number	plant
a	Ant	5	
a		69	Azalea
b	Bird		
b		55	Bamboo
c	Cat	17	
c			Cabbage
d	Dog	9	
d		14	Daffodil
e	Eagle		
e		5	Eucalyptus
f	Frog	76	
f		77	Fig

The resulting table Interleave has 12 rows, which is the sum of the rows from the combined tables. The new table contains all columns from both tables. The value of columns found in one table but not in the other are set to missing, and the rows are arranged by the values of the BY variable.

Example 2: Interleaving with Duplicate Values of the BY Variable

If the tables contain duplicate values of the BY variables, the rows are written to the new table in the order in which they occur in the original tables. This example contains duplicate values of the BY variable Common. The following program creates the Animal and Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal number;
```

```
   method init();
      common='a'; animal='Ant'; number='5'; output;
      common='a'; animal='Bird'; number=''; output;
      common='b'; animal='Cat'; number='17'; output;
      common='c'; animal='Dog'; number='9';  output;
      common='d'; animal='Eagle'; number=''; output;
      common='e'; animal='Frog';  number='76'; output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant number;
   method init();
      common='a'; plant='Grape'; number='69'; output;
      common='b'; plant='Hazelnut'; number='55'; output;
      common='c'; plant='Indigo'; number=''; output;
      common='c'; plant='Jicama';  number='14'; output;
      common='d'; plant='Kale'; number='5'; output;
      common='e'; plant='Lentil';  number='77'; output;
   end;
enddata;
run;
```

The following program uses SET and BY statements to interleave the tables, and prints the results:

```
   /*  set with by interleaves */
   data interleave (overwrite=yes);
      method run();
         set animal plant; by common;
      end;
   enddata;
   run;

   proc print data=interleave;
   run;
   quit;
```

Output 16.3 *Interleaved Table with Multiple BY Variables (SET Statement)*

Obs	common	animal	number	plant
1	a	Bird		
2	a	Ant	5	
3	a		69	Grape
4	b	Cat	17	
5	b		55	Hazelnut
6	c	Dog	9	
7	c		14	Jicama
8	c			Indigo
9	d	Eagle		
10	d		5	Kale
11	e	Frog	76	
12	e		77	Lentil

The number of rows in the new table is the sum of the rows in all the tables. The rows are written to the new table in the order in which they occur in the original tables.

Example 3: Interleaving with Different BY Values in Each Table

The tables Animal and Plant both contain values that are present in one table but not in the other. The following program creates the Animal and the Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal number;
   method init();
      common='a'; animal='Ant'; number='5'; output;
      common='c'; animal='Bird'; number=''; output;
      common='d'; animal='Cat'; number='17'; output;
      common='e'; animal='Dog'; number='9';  output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant number;
   method init();
      common='a'; plant='Grape'; number='69'; output;
      common='b'; plant='Hazelnut'; number='55'; output;
      common='c'; plant='Indigo'; number=''; output;
      common='d'; plant='Jicama';  number='14'; output;
      common='e'; plant='Kale'; number='5'; output;
      common='f'; plant='Lentil';  number='77'; output;
   end;
enddata;
run;
```

This program uses SET and BY statements to interleave these tables, and prints the results:

```
data interleave (overwrite=yes);
   method run();
      set animal plant; by common;
   end;
enddata;
run;

proc print data=interleave;
run;
quit;
```

Output 16.4 *Interleaved Table with Different BY Variables (SET Statement)*

Obs	common	animal	number	plant
1	a	Ant	5	
2	a		69	Grape
3	b		55	Hazelnut
4	c	Bird		
5	c			Indigo
6	d	Cat	17	
7	d		14	Jicama
8	e	Dog	9	
9	e		5	Kale
10	f		77	Lentil

The resulting table has ten rows arranged by the values of the BY variable.

Comments and Comparisons

- In other languages, the term merge is often used to mean interleave. DS2 reserves the term merge for the operation in which rows from two or more tables are combined into one row. The rows in interleaved tables are not combined; they are copied from the original tables in the order of the values of the BY variable.
- If one table has multiple rows with the same BY value, the DATA step preserves the order of those rows in the result.
- To use the data program, the input tables must be appropriately sorted. SQL does not require the input tables to be in order.

One-to-One Reading

Definition

One-to-one reading combines rows from two or more tables into one row by using two or more SET statements to read rows independently from each table. This process is also

called one-to-one matching. The new table contains all the columns from all the input tables. The number of rows in the new table is the number of rows in the smallest original table. If the tables contain common columns, the values that are read in from the last table replace the values that were read in from earlier tables.

Syntax

Use this form of the SET statement for one-to-one reading:

SET *table-1*;

SET *table-2*;

Arguments

table-1
specifies a table name. *table-1* is the first table that the data program reads.

table-2
specifies a table name. *table-2* is the second table that the data program reads.

CAUTION:
Use care when you combine tables with multiple SET statements. Using multiple SET statements to combine rows can produce undesirable results. Test your program on representative samples of the tables before using this method to combine them.

For more information, see "SET Statement" in *SAS Viya: DS2 Language Reference*.

DS2 Processing during a One-to-One Reading

Compilation phase
DS2 reads the descriptor information of each table named in the SET statement and then creates a program data vector that contains all the columns from all tables as well as columns created by the data program.

Execution — Step 1
When DS2 executes the first SET statement, DS2 reads the first row from the first table into the program data vector. The second SET statement reads the first row from the second table into the program data vector. If both tables contain the same columns, the values from the second table replace the values from the first table, even if the value is missing. After reading the first row from the last table and executing any other statements in the data program, DS2 writes the contents of the program data vector to the new table. The SET statement does not reset the values in the program data vector to missing, except for those columns that were created or assigned values during the data program.

Execution — Step 2
DS2 continues reading from one table and then the other until it detects an end-of-file indicator in one of the tables. DS2 stops processing with the last row of the shortest table and does not read the remaining rows from the longer table.

Example 1: One-to-One Reading: Processing an Equal Number of Rows

The tables Animal and Plant both contain the column Common, and are arranged by the values of that column. The following program creates the Animal and the Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal number;
   method init();
      common='a'; animal='Ant';  output;
```

```
      common='b'; animal='Bird';  output;
      common='c'; animal='Cat';  output;
      common='d'; animal='Dog';   output;
      common='e'; animal='Eagle';  output;
      common='f'; animal='Frog';   output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant number;
   method init();
      common='a'; plant='Grape'; output;
      common='b'; plant='Hazelnut'; output;
      common='c'; plant='Indigo';  output;
      common='d'; plant='Jicama';  output;
      common='e'; plant='Kale';  output;
      common='g'; plant='Lentil';  output;
   end;
enddata;
run;
```

The following program uses two SET statements to combine rows from Animal and Plant, and prints the results:

```
data one2one (overwrite=yes);
   method run();
      set animal;
      set plant;
   end;
enddata;
run;

proc print data=one2one;
run;
quit;
```

Output 16.5 *One-to-One Reading with an Equal Number of Rows (SET Statement)*

Obs	common	animal	plant
1	a	Ant	Grape
2	b	Bird	Hazelnut
3	c	Cat	Indigo
4	d	Dog	Jicama
5	e	Eagle	Kale
6	g	Frog	Lentil

Each row in the new table contains all the columns from all the tables. Note, however, that the Common column value in row 6 contains a "g." The value of Common in row 6 of the Animal table was overwritten by the value in Plant, which was the table that DS2 read last.

One-to-One Reading with an Uneven Number of Rows

The tables Animal and Plant both contain the column Common, and are arranged by the values of that column. The tables have different number of rows. The following program creates the Animal and the Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
      common='a'; animal='Bird';  output;
      common='b'; animal='Cat';  output;
      common='c'; animal='Dog';  output;
      common='d'; animal='Eagle';  output;
      common='e'; animal='Frog';  output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Grape';output;
      common='b'; plant='Hazelnut';  output;
      common='c'; plant='Indigo'; output;
      common='d'; plant='Jicama';  output;
   end;
enddata;
run;
```

The following program uses two SET statements to combine rows from Animal and Plant, and prints the results:

```
data one2onerowsnotequal (overwrite=yes);
   method run();
      set animal;
      set plant;
   end;
enddata;
run;
quit;

proc print data=one2onerowsnotequal;
run;
```

Output 16.6 *One-to-One Reading with an Uneven Number of Rows (SET Statement)*

Obs	common	animal	plant
1	a	Ant	Grape
2	b	Bird	Hazelnut
3	c	Cat	Indigo
4	d	Dog	Jicama

The result table contains only four rows because DS2 stops processing with the last row from the shortest table.

Comments and Comparisons

- Using multiple SET statements with other DS2 statements makes the following applications possible:
  - merging one row with many
  - conditionally merging rows
  - reading from the same table twice

Match-Merging

Definition

Match-merging combines rows from two or more tables into a single row in a new table according to the values of a common column. The number of rows in the new table is the sum of the largest number of rows in each BY group in all tables. To perform a match-merge, use the MERGE statement with the required BY statement. When you perform a match-merge, all tables are sorted by the columns that you specify in the BY statement.

Syntax

Use this form of the MERGE statement to match-merge tables:

MERGE *table(s)*;

BY *column(s)*;

Arguments

table
names at least two existing tables from which rows are read.

column
names each column by which the table is sorted. These columns are referred to as BY variables.

For more information, see the "MERGE Statement" in *SAS Viya: DS2 Language Reference* and the "BY Statement" in *SAS Viya: DS2 Language Reference*.

DS2 Processing during Match-Merging

Compilation phase

DS2 reads the descriptor information of each table that is named in the MERGE statement and then creates a program data vector that contains all the rows from all tables as well as rows created by the data program.

Execution – Step 1

DS2 looks at the first BY group in each table that is named in the MERGE statement to determine which BY group should appear first in the new table. The data program reads into the program data vector the first row in that BY group from each table, reading the tables in the order in which they appear in the MERGE statement. If a table does not have rows in that BY group, the program data vector contains missing values for the rows that are unique to that table.

Execution – Step 2

Each row in any of the input tables is used exactly once in the output table. Columns that are unique to a table are filled with missing or null values if that table is exhausted while producing a BY group.

Execution – Step 3

DS2 repeats these steps until it reads all rows from all BY groups in all tables.

CAUTION:

BY variables in a DS2 merge that have a DECIMAL or NUMERIC data type are converted to a DOUBLE data type. If matching DECIMAL columns are not BY variables, the DECIMAL columns remain as a DECIMAL data type.

CAUTION:

If there is a type, scale, or precision mismatch between columns with a DECIMAL or NUMERIC data type between tables, the column is converted to a DOUBLE data type.

Example 1: Merging Rows

The tables Animal and Plant each contain the BY variable Common, and the rows are arranged in order of the values of the BY variable. The following program creates the Animal and the Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant';  output;
      common='b'; animal='Bird';  output;
      common='c'; animal='Cat';  output;
      common='d'; animal='Dog';   output;
      common='e'; animal='Eagle';  output;
      common='f'; animal='Frog';   output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Grape'; output;
      common='b'; plant='Hazelnut'; output;
      common='c'; plant='Indigo';  output;
      common='d'; plant='Jicama';  output;
```

```
      common='e'; plant='Kale';  output;
      common='f'; plant='Lentil';  output;
   end;
enddata;
run;
```

The following program merges the tables according to the values of the BY variable Common, and prints the results:

```
data mmerge (overwrite=yes);
   method run();
      merge animal plant;
      by common;
   end;
enddata;
run;
quit;

proc print data=mmerge;
run;
quit;
```

Output 16.7 *Simple Match Merge (MERGE Statement)*

Obs	common	animal	plant
1	a	Ant	Grape
2	b	Bird	Hazelnut
3	c	Cat	Indigo
4	d	Dog	Jicama
5	e	Eagle	Kale
6	f	Frog	Lentil

Each row in the new table contains all the columns from all the tables.

Example 2: Match-Merge with Duplicate Values of the BY Variable

In the following example, the tables Animal and Plant contain duplicate values of the BY variable Common. The following program creates the Animal and the Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
      common='a'; animal='Ape';  output;
      common='b'; animal='Bird';  output;
      common='c'; animal='Cat';  output;
      common='d'; animal='Dog';  output;
      common='e'; animal='Eagle';  output;
   end;
enddata;
run;
```

```
data plant(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Apple';output;
      common='b'; plant='Banana';  output;
      common='c'; plant='Coconut'; output;
      common='c'; plant='Celery'; output;
      common='d'; plant='Dewberry';  output;
      common='e'; plant='Eggplant'; output;
   end;
enddata;
run;
```

The following program produces the merged table MATCH1, and prints the results:

```
data mmdiffby (overwrite=yes);
   method run();
      merge animal plant;
      by common;
   end;
enddata;
run;
quit;

proc print data=mmdiffby;
run;
quit;
```

Output 16.8 *Match-Merge with Duplicate BY Variables (MERGE Statement)*

Obs	common	animal	plant
1	a	Ape	Apple
2	a	Ant	
3	b	Bird	Banana
4	c	Cat	Coconut
5	c		Celery
6	d	Dog	Dewberry
7	e	Eagle	Eggplant

In row 2 of the output, the value of the column Plant is not retained. Match-merging also did not duplicate values in Animal for row 5.

Note: The MERGE statement does not produce a Cartesian product on a many-to-many match-merge. Instead, it performs a one-to-one merge while there are rows in the BY group in at least one table. When all rows in the BY group have been read from one table and there are still more rows in another table, DS2 fills the columns with missing or null values.

Example 3: Match-Merge with Non-matched Rows

When DS2 performs a match-merge with nonmatched rows in the input tables, DS2 retains the values of all columns in the program data vector even if the value is missing. The tables Animal and Plant do not contain all values of the BY variable Common. The following program creates the Animal and the Plant input tables:

```
data animal(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
      common='c'; animal='Cat';  output;
      common='d'; animal='Dog';  output;
      common='e'; animal='Eagle';  output;
   end;
enddata;
run;


data plant(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Apple';output;
      common='b'; plant='Banana';  output;
      common='c'; plant='Coconut'; output;
      common='e'; plant='Eggplant'; output;
      common='f'; plant='Fig';  output;end;
enddata;
run;
```

The following program produces the merged table Mmnomrow, and prints the results:

```
data mmnomrow (overwrite=yes);
   method run();
      merge animal plant;
      by common;
   end;
enddata;
run;
quit;

proc print data=mmnomrow;
run;
quit;
```

Output 16.9 *Match-Merge with Non-Matched Rows (MERGE Statement)*

Obs	common	animal	plant
1	a	Ant	Apple
2	b		Banana
3	c	Cat	Coconut
4	d	Dog	
5	e	Eagle	Eggplant
6	f		Fig

As the output shows, all values of the column Common are represented in the new table, including missing values for the columns that are in one table but not in the other.

Chapter 17

Reserved Words

Reserved Words in the DS2 Language

The following words are reserved as DS2 language keywords and cannot be used as variable names or in any other way that differs from their intended use.

Note: You can use a reserved word as a variable name if the word is enclosed in double quotation marks. For more information and an example, see "Delimited Identifiers" on page 40.

Table 17.1 *DS2 Reserved Words*

Special Characters	**A**	**B**	**C**	**D**
___KPLIST	ABORT	BIGINT	CALL	DATA
ALL	AND	BINARY	CATALOG	DATE
NEW	AS	BY	CHAR	DCL
NULL			CHARACTER	DECIMAL
_RC_			COMMIT	DECLARE
ROWSET			CONTINUE	DELETE
TEMPORARY				DESCENDING
THREADID				DIM
				DO
				DOUBLE
				DROP
				DS2_OPTIONS

E	**F**	**G**	**H**	**I**
ELIF	FILE	GE	HAVING	IDENTITY
ELSE	FILENAME	GLOBAL		IF
ENCRYPT	FLOAT	GOTO		IN
END	FORMAT	GT		INDSNAME
ENDDATA	FORWARD			INFILE
ENDPACKAGE	FROM			INFORMAT
ENDTABLE	FUNCTION			INPUT
ENDTHREAD				INT
EQ				INTEGER
ERROR				IN_OUT
ESCAPE				

K	**L**	**M**	**N**	**O**
KEEP	LABEL	MERGE	NATIONAL	ODS
	LE	METHOD	NCHAR	OR
	LEAVE	MISSING	NE	OTHER
	LIBNAME	MODIFY	NG	OTHERWISE
	LIKE		NL	OUTPUT
	LIST		NOT	OVERWRITE
	LT		NULL	
			NUMERIC	
			NVARCHAR	

P	**R**	**S**	**T**	**U**
PACKAGE	REAL	SELECT	TABLE	UNTIL
PRECISION	REMOVE	SET	THEN	UPDATE
PRIVATE	RENAME	SMALLINT	THIS	
PROGRAM	REPLACE	SQLSUB	THREAD	
PUT	REQUIRE	STOP	THREADS	
	RETAIN	STORED	TIME	
	RETURN	SUBSTR	TIMESTAMP	
	RETURNS	SYSTEM	TINYINT	
	ROLLBACK		TO	
			TRANSACTION	
			T_UDF	
			TSPL_OPTIONS	

V VARARRAY VARBINARY VARCHAR VARLIST VARYING	**W** WHEN WHERE WHILE			

Part 3

DS2 and CAS

Chapter 18

DS2 in CAS

Running DS2 Programs in CAS

There are two ways to run a DS2 program in CAS:

- PROC DS2

```
cas casauto;
caslib _all_ assign;

proc cas;
  session casauto;
/* this will set the active caslib to casdata*/
table.addCaslib /
     caslib="casdata"
     dataSource={srcType="path"}
     path="path-to-your-data";

 /* Load source data (cars) into a table.*/
   table.loadTable /
      caslib="casdata"
      path="cars.sashdat"
      casOut={name="cars", replace=true};
run;
quit;
```

```
proc ds2 sessref=casauto;
thread cars_thd / overwrite=yes;
   method run();
      set casdata.cars;
      if (msrp > 100000) then do;
         put make= model= msrp=;
         output;
      end;
   end;
endthread;

data cars_luxury / overwrite=yes;
   dcl thread cars_thd t;
   method run();
      set from t threads=4;
   end;
enddata;
run;
quit;
```

For an explanation of this program, see "DS2 Program Walk-Through" on page 195.

For more information about PROC DS2, see "DS2" in *SAS Viya Data Management and Utility Procedures Guide*.

- DS2 runDS2 action

```
cas casauto;
caslib _all_ assign;

proc cas;
  session casauto;
/* This will set the active caslib to casdata*/
   table.addCaslib /
      caslib="casdata"
      dataSource={srcType="path"}
      path="path-to-your-data";

/* Load source data (cars) into a table.*/
   table.loadTable /
      caslib="casdata"
      path="cars.sashdat"
      casOut={name="cars", replace=true};
run;

/*DS2 program to search for cars over $100K */
   ds2.runDS2 program="thread cars_thd / overwrite=yes;
   method run();
      set casdata.cars;
      if (msrp > 100000) then do;
         put make= model= msrp=;
         output;
      end;
   end;
endthread;

data cars_luxury / overwrite=yes;
   dcl thread cars_thd t;
```

```
   method run();
      set from t threads=4;
   end;
enddata;";
run;
quit;
```

For more information about the runDS2 action, seeFor more information, see "Run program" in *SAS Cloud Analytic Services: System Programming Guide*.

TIP Unless you are using Lua or Python, it is recommended that you use PROC DS2 to run your DS2 program. There are advantages to using PROC DS2. For more information, see "DS2 Action Set Details" in *SAS Cloud Analytic Services: System Programming Guide*.

DS2 Program Classification

A DS2 program can perform manipulations on multiple data observations, thus concurrently reducing the time required to process big data sets. Based on the structure of the DS2 program, the DS2 compiler determines which operations can be performed on multiple observations concurrently and which operations must be applied to each observation sequentially. A DS2 program is classified as either a serial program, parallel program, or parallel-serial program.

serial program
: contains operations with data dependencies across observations. Thus, observations must be processed in serial. One thread processes the complete data set and generates the complete result set.

parallel program
: contains no operations with data dependencies across observations. Thus, multiple data observations can be processed in parallel. Each thread processes a subset of the data set and generates a subset of the result set.

parallel-serial program
: contains some operations with data dependencies across observations and some operations without data dependencies. The processing of the operations is divided into two stages, a parallel stage and a serial stage. During the parallel stage, each thread processes a subset of the input data set and generates a subset of an intermediate data set. During the serial stage, one thread processes the complete intermediate data set and generates the complete result set.

For more information, see Chapter 13, "Threaded Processing," on page 143.

How DS2 Runs in CAS

Active and Passive Nodes

During execution of a DS2 program on a CAS server, nodes might be active or passive. An active node processes observations. Active nodes read, manipulate, and write data. A passive node does not process observations.

When serial operations are processed, one and only one node processes each observation sequentially. This special node is the principal worker. The controller cannot be the principal because the controller is unable to read or write data. Therefore, a worker is selected as the principal worker.

Type of DS2 Program	Program	Operations	Controller	Principal Worker	Other Workers
Serial	Data	serial	passive	active	passive
Parallel	Thread	parallel	passive	active	active
	Data	no-op	passive	active	active
Parallel-Serial stage one	Thread	parallel	passive	active	active
	Data	no-op	passive	active	active
Parallel-Serial stage two	Data	serial	passive	active	passive

What occurs when a DS2 program executes in CAS depends on the classification of the DS2 program

DS2 Serial Program on a CAS Server

A DS2 serial program contains operations with data dependencies across observations. Therefore, observations must be processed in serial. One CAS worker processes the complete data set and generates the complete result set.

Here is how the DS2 serial program is executed:

- The principal worker is the only active node. The other workers and the controller are passive nodes.
- On the principal worker, the DS2 data program accomplishes the following tasks:
  - reads the data set.
  - performs the data manipulations on the read data.
  - writes the manipulated data to the result set.

DS2 Parallel Program on a CAS Server

A DS2 parallel program contains no operations with data dependencies across observations. Thus, multiple data observations can be processed in parallel. Each CAS worker processes a subset of the data set and generates a subset of the result set. CAS data management supports each worker reading a subset of the data set in parallel and writing a subset of the result set in parallel.

Here is how the DS2 parallel program is executed:

- All workers are active nodes. The controller is a passive node.
- On each worker, the DS2 thread program accomplishes the following tasks:
  - reads a subset of the data set.

    - performs the parallel data manipulations on the read data.
    - passes the manipulated data to the DS2 data program.
- On each worker, the DS2 data program writes the manipulated data to the result set.

DS2 Parallel-Serial Program on a CAS Server

A DS2 parallel-serial program contains some operations with data dependencies across observations and some operations without data dependencies:

parallel stage
: executes parallel data manipulation operations from thread program.

serial stage
: executes serial data manipulation operations from data program.

During the parallel stage, each CAS worker processes a subset of the input data set and generates a subset of an intermediate data set. During the serial stage, the principal CAS worker processes the complete intermediate data set and generates the complete result set.

When a DS2 parallel-serial program is run in CAS, the DS2 compiler refactors the DS2 parallel-serial program into two DS2 programs:

- Stage one program - parallel stage
    - All workers are active nodes. The controller is a passive node.
    - On each worker, the program accomplishes the following tasks:
        - reads a subset of the input data set.
        - performs the parallel data operations from the thread program on the read data.
        - writes the subset of the manipulated data to an intermediate data set.
- Stage two program - serial stage
    - The principal worker is the only active node. The other workers and the controller are passive nodes.
    - On the principal worker, the program accomplishes the following tasks:
        - reads the complete intermediate data set.
        - performs the serial data operations from the data program on the read data.
        - writes the complete result data set.

DS2 Program Walk-Through

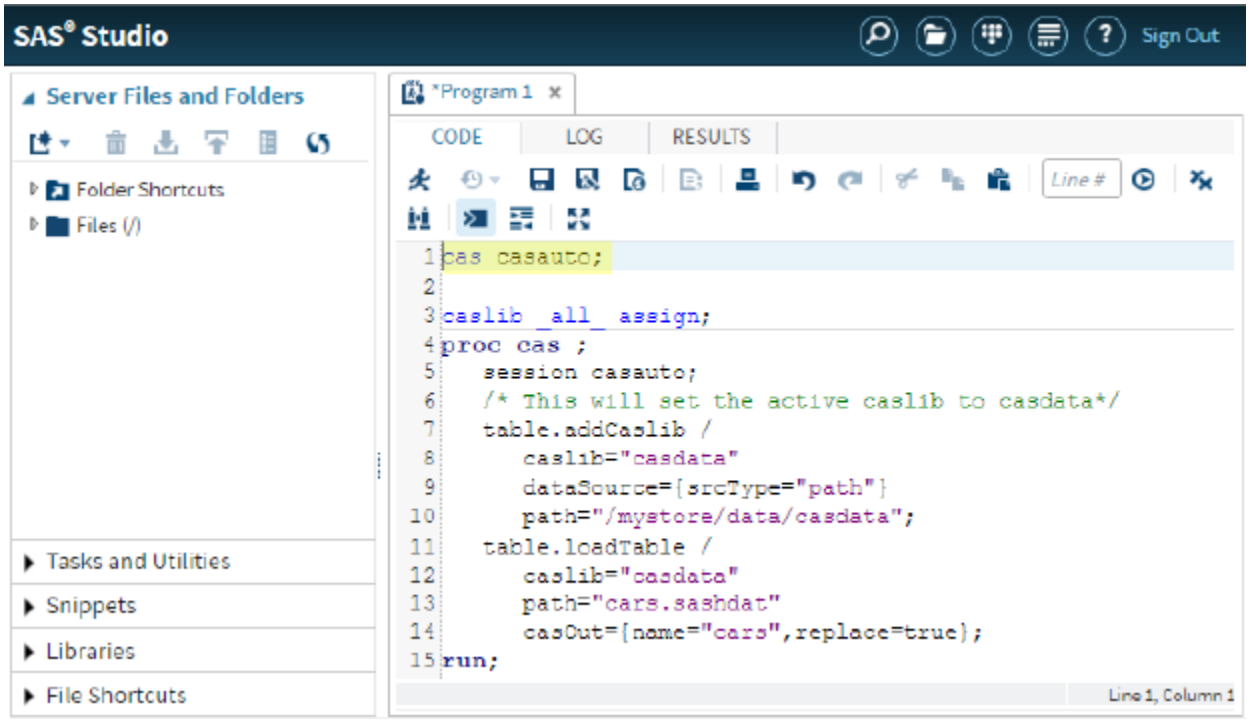
The following example shows the steps that you need to perform to run a DS2 program in CAS.

1. Start a CAS session.

   To access CAS, start and activate the CAS session. Specify the name of a CAS session in the CAS statement.

   ```
   cas casauto;
   ```

Here is a view of the **SAS Studio Code** tab:



The SAS log contains notes similar to the following:

```
  NOTE: The session CASAUTO connected successfully to Cloud Analytic Services
cloud.example.com using port 5570. The UUID is
       session-UUID. The user is casdemo and the active caslib is
CASUSERHDFS(casdemo).
 NOTE: The SAS option SESSREF was updated with the value CASAUTO.
 NOTE: The SAS macro _SESSREF_ was updated with the value CASAUTO.
 NOTE: The session is using nnn workers.
 59         caslib _all_ assign;
 NOTE: CASLIB CASUSER(casdemo) will be mapped to SAS Library CASUSER.
 NOTE: CASLIB CASUSERHDFS(casdemo) will be mapped to SAS Library CASUSERH.
```

2. Associate a caslib.

You can use the caslib statement to associate the default CASUSER caslib with all SAS librefs that you create.

```
caslib _all_ assign;
```

Here is a partial listing of the notes that are displayed in the SAS log:

```
NOTE: CASLIB CASUSER(casdemo) will be mapped to SAS Library CASUSER.
NOTE: CASLIB CASUSERHDFS(casdemo) will be mapped to SAS Library CASUSERH.
```

3. Create a caslib and load data.

   To create a caslib that provides access to files on your file system, use the table.addCaslib action along with the dataSource option set to **`path`**. In this example, the CASDATA caslib is the interface between the CASAUTO session and the source tables found on /mystore/data/casdata.

```
proc cas ;
   session casauto;
```

```
/* This will set the active caslib to casdata*/
table.addCaslib /
   caslib="casdata"
   dataSource={srcType="path"}
   path="/mystore/data/casdata";
```

For more information about caslibs, see "Working with Caslibs" in *SAS Cloud Analytic Services: System Programming Guide*.

To load your source data (cars) into an in-memory table on CAS, you can use the table.loadTable action.

```
table.loadTable /
   caslib="casdata"
   path="cars.sashdat"
   casOut={name="cars",replace=true};
run;
```

Results describing the CASDATA caslib are displayed on the **Results** tab in SAS Studio.

Results from table.addCaslib

CAS Library Information

Library	Type	Path	Sub-directories included	Session local	Active
casdata	PATH	/mystore/data/casdata/	No	Yes	Yes

4. View your CAS table.

   You can view the rows of data from the in-memory table (cars) using the table.fetch action.

   ```
   table.fetch / table="cars" to=10;
   run;
   ```

   Here are the results of fetching the first 10 rows from the in-memory table.

Selected Rows from Table CARS

Index	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	Engine Size (L)	Cylinders	Horsepower	MPG (City)	MPG (Highway)	Weight (LBS)	Wheelbase (IN)	Length (IN)
1	BMW	X3 3.0i	SUV	Europe	All	$37,000	$33,873	3	6	225	18	23	4023	110	180
2	Honda	S2000 convertible 2dr	Sports	Asia	Rear	$33,260	$29,965	2.2	4	240	20	25	2835	95	182
3	Nissan	Altima S 4dr	Sedan	Asia	Front	$19,240	$18,030	2.5	4	175	21	26	3039	110	192
4	Audi	A6 4.2 Quattro 4dr	Sedan	Europe	All	$49,690	$44,936	4.2	8	300	17	24	4024	109	193
5	Honda	Civic LX 4dr	Sedan	Asia	Front	$15,950	$14,531	1.7	4	115	32	39	2513	103	175
6	Mitsubishi	Eclipse GTS 2dr	Sports	Asia	Front	$25,092	$23,456	3	6	210	21	28	3241	101	177
7	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100	$41,100	3.5	6	225	18	24	3893	115	197
8	GMC	Safari SLE	Sedan	USA	Rear	$25,640	$23,215	4.3	6	190	16	20	4309	111	190
9	Mercury	Marauder 4dr	Sedan	USA	Rear	$34,495	$31,558	4.6	8	302	17	23	4195	115	212
10	Volvo	S80 2.5T 4dr	Sedan	Europe	All	$37,895	$35,688	2.5	5	194	20	27	3601	110	190

5. Run a DS2 program

   When used with the sessref= option, PROC DS2 enables you to run your program in CAS using your in-memory CAS table.

   ```
   proc ds2 sessref=casauto;          /*1*/

   thread cars_thd / overwrite=yes;   /*2*/
      method run();
   ```

```
      set casdata.cars;             /* 3 */
      if (msrp > 100000) then do;
         put make= model= msrp=;
         output;
      end;
   end;
endthread;

data cars_luxury / overwrite=yes;    /* 4 */
   dcl thread cars_thd t;
    method run();
      set from t threads=4;
    end;
enddata;

run;
quit;
```

a Add the sessref= option in the PROC DS2 statement to run your program in CAS.

b Operations in the parallel stage of the program are applied to multiple data rows in parallel.

c Read a row from the in-memory table.

d The thread instances are created and executed.

Here are the results that are displayed in the SAS log:

```
 NOTE: Running THREAD program on all nodes
Make=Porsche       Model= 911 GT2 2dr                              MSRP=$192,465
Make=Mercedes-Benz Model= CL600 2dr                                MSRP=$128,420
Make=Mercedes-Benz Model= SL55 AMG 2dr                             MSRP=$121,770
Make=Mercedes-Benz Model= SL600 convertible 2dr                    MSRP=$126,670
NOTE: Created thread cars_thd in data set "casdata".cars_thd.
NOTE: Running THREAD program on all nodes
NOTE: Running DATA program on all nodes
NOTE: Execution succeeded. 4 rows affected.
104        quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           4.89 seconds
      cpu time            0.02 seconds
```

BY-Group Processing in CAS

On the SAS client, DS2 BY-group processing groups the rows from input tables and orders the rows by values of one or more columns in the BY statement.

In the CAS server, there is no guarantee of global ordering between BY groups. In CAS, data is distributed on different data partitions. Each DS2 thread running in CAS has access to one data partition. Each DS2 thread can group and order only the rows in the same data partition. Consequently, the data partition might have only part of the entire group of data. You must run a final aggregation in the main data program.

DS2 Logging in the CAS Server

DS2 and CAS support the SAS logging facility. For more information about logging in SAS Viya, see “Logging” in *SAS Viya Administration*.

Part 4

Appendixes

Appendix 1

Data Type Reference

Data Types for SAS Data Sets

The following table lists the data type support for a SAS data set.

The BINARY and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a SAS character data type, or DOUBLE, which is a SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see "DATA Step Basics" in *SAS Cloud Analytic Services: Accessing and Manipulating Data*.

Table A1.1 *Data Types for SAS Data Sets*

Data Type Definition Keyword*	SAS Data Set Data Type	Description	Data Type Returned
BIGINT**	DOUBLE	64-bit double precision, floating-point number. *Note:* There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	Fixed-length character string. *Note:* Cannot contain ANSI SQL null values.	CHAR(*n*)

Data Type Definition Keyword*	SAS Data Set Data Type	Description	Data Type Returned
DATE ***	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.	DOUBLE
DECIMAL| NUMERIC(*p*,*s*)**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
DOUBLE**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
INTEGER**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
NCHAR(*n*)	CHAR(*n*)	Fixed-length character string. By default, sets the encoding to Unicode UTF-8. †	CHAR(*n*)
NVARCHAR(*n*)	CHAR(*n*)	Fixed-length character string. By default, sets the encoding to Unicode UTF-8. †	CHAR(*n*)
REAL**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
SMALLINT**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)***	DOUBLE	64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)***	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	Fixed-length character string. *Note:* Cannot contain ANSI SQL null values.	CHAR(*n*)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE

can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** Do not apply date and time SAS formats to a numeric data type. For date and time values, use the DATE, TIME, or TIMESTAMP data types.

*** Because the values are stored as a double precision, floating-point number, you can use the values in arithmetic expressions.

† UTF-8 is an MBCS encoding. Depending on the operating environment, UTF-8 characters are of varying width, from one to four bytes. The value for *n*, which is the maximum number of multibyte characters to store, is multiplied by the maximum length for the operating environment. Note that when you are transcoding, such as from UTF-8 to Wlatin2, the variable lengths (in bytes) might not be sufficient to hold the values, and the result is character data truncation.

Data Types for Hive

The following table lists the data type support for Hive. Hive versions 0.10 and later are supported.

The NCHAR and NVARCHAR data types are not available for data definition. Nor are the Hive complex types ARRAY, MAP, STRUCT, and UNION.

For data-source specific information about Hive data types, see the Hive database documentation.

Table A1.2 *Data Types for Hive*

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
*	ARRAY<*data-type*>	An array of integers (indexable lists).	STRING ‡
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
BINARY(*n*)	BINARY**	A varying length binary string up to 32K.	BINARY
*	BOOLEAN	A textual true or false value.	TINYINT
CHAR(*n*)	CHAR(n)**	A character string up to 255 characters. ***	CHAR
DATE	DATE†††	An ANSI SQL date type.	DATE
DECIMAL/NUMERIC(*p*,*s*)	DECIMAL**†††	A fixed-point decimal number, with 38 digits precision.	DOUBLE
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
INTEGER	INTEGER	A signed four-byte integer.	INTEGER

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
*	MAP<*primitive-type*, *data-type*>	An associative array of key-value pairs.	STRING‡
REAL	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,768 to 32,767.	SMALLINT
*	STRING	A variable-length character string.	VARCHAR(n)‡‡‡
*	STRUCT<*col-name*: *data-type*>	A structure with established column elements and data types. Column elements and data types are mapped using a double-dot (:) notation.	STRING ‡
TIME(*p*)	‡‡‡	A time value.	STRING
TIMESTAMP(*p*)	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A signed one-byte integer, from -128 to 127.	TINYINT
*	UNION<*data-type*, *data-type-n*>	A type that can hold one of several specified data types.	STRING‡
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)
VARBINARY	BINARY	A varying length binary string up to 32K.	BINARY**

* The Hive data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

** Full support for this data type is available in Hive 0.13 and later. In Hadoop environments that use earlier Hive versions (which do not support the CHAR and DECIMAL types), columns defined as CHAR are mapped to VARCHAR. Columns that are defined as DECIMAL are not supported. In Hadoop environments that use Hive versions earlier than Hive 0.13, BINARY columns can be created but not retrieved.

*** If you specify CHAR with a value greater than 255 characters, the column is created as VARCHAR(n) instead.

† Full support for this data type is available in Hive 0.12 and later. In Hadoop environments that use earlier Hive versions (which do not support the DATE type), any SASFMT TableProperties that are defined on STRING columns are applied when reading Hive, effectively allowing the STRING columns to be treated as DATE columns. When the DATE data type is used for data definition in earlier Hive versions, the DATE type is mapped to a STRING column with SASFMT TableProperties. For more information about SASFMT TableProperties, see "SAS Table Properties for Hive and HADOOP" in *SAS/ACCESS for Relational Databases: Reference*.

†† The supported date values are between October 15, 1582 and December 31, 9999. Date values containing years earlier than 1582 will return an error. Date values later than 9999 will be read back as null values.

††† Decimals processed by SAS are processed using a DOUBLE, which can alter the precision.

‡ The maximum length of VARCHAR(*n*) and the Hive complex types is determined by the DBMAX_TEXT= data source connection option.

‡‡ SASFMT Table Properties are applied when reading STRING columns.

‡‡‡ Hive does not support the TIME(p) data type. When data is read from Hive, STRING columns that have SASFMT TableProperties defined that specify the SAS TIME8. format are converted to the TIME(*p*) data type. When the TIME type is used for data definition, it

is mapped to a STRING column with SASFMT TableProperties. Fractional seconds are not preserved. For more information about SASFMT TableProperties, see "SAS Table Properties for Hive and HADOOP" in *SAS/ACCESS for Relational Databases: Reference*.

The complex types ARRAY, MAP, STRUCT, and UNION are read as their STRING representation of the underlying complex type. ARRAY values are read back within brackets, for example: [1, 2, 4]. STRUCT and MAP values are read back within braces, for example: {"firstname":"robert","nickname":"bob"}.

Data Types for Impala

The following table lists the data type support for Impala. Impala version 2.0 and later are supported, running on CDH 5.1 and later.

The BINARY, DECIMAL(*p*,*s*)/NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not available for data definition.

For data-source specific information about Impala data types, see the vendor documentation.

Data Type Definition Keyword	Impala Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
CHAR(*n*)	CHAR*	A fixed-length character string up to 255 characters.	CHAR
DATE	**	An ANSI SQL date type.	TIMESTAMP
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
INTEGER	INT	A signed four-byte integer, from -2,147,483,648 to 2,147,483,647.	INTEGER
REAL	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,768 to 32,767.	SMALLINT
TIME	**	A time value.	TIMESTAMP
TIMESTAMP	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP
TINYINT	TINYINT	A signed one-byte integer, from -128 to 127.	TINYINT
VARCHAR(*n*)	VARCHAR*	A varying-length character string.	VARCHAR

* Support for this data type is available in CDH 5.2 and later. In environments that use earlier CDH versions, which do not support the CHAR and VARCHAR types, columns defined as CHAR or VARCHAR are mapped to STRING.

** Impala does not support this data type. When a DATE or TIME column is defined, it is created as a column of type TIMESTAMP.

Data Types for ODBC

The following table lists the data type support for an ODBC-compliant data source.

For data source specific information about ODBC SQL data types, see the specific ODBC data source documentation.

Table A1.3 *Data Types for ODBC*

Data Type Definition Keyword*	ODBC SQL Identifier	Description	Data Type Returned
BIGINT	SQL_BIGINT	Large signed, exact whole number.	BIGINT
BINARY(*n*)	SQL_BINARY	Fixed-length binary string.	BINARY(*n*)
**	SQL_BIT	Single bit binary data.	**
CHAR(*n*)***	SQL_CHAR	Fixed-length character string.	CHAR(*n*)
DATE	SQL_TYPE_DATE	Date values.	DATE
DECIMAL|NUMERIC(*p*,*s*)	SQL_DECIMAL| SQL_NUMERIC	Signed, fixed-point decimal number.	DECIMAL|NUMERIC(*p*,*s*)
DOUBLE	SQL_DOUBLE	Signed, double precision, floating-point number.	DOUBLE
**	SQL_GUID	Globally unique identifier.	**
INTEGER	SQL_INTEGER	Regular signed, exact whole number.	INTEGER
**	SQL_INTERVAL	Intervals between two years, months, days, dates or times.	**
**	SQL_LONGVARBINARY	Varying-length binary string.	**
**	SQL_LONGVARCHAR	Varying-length Unicode character string.	**
NCHAR(*n*)	SQL_WCHAR	Fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	SQL_WVARCHAR	Varying-length Unicode character string.	NVARCHAR(*n*)

Data Type Definition Keyword*	ODBC SQL Identifier	Description	Data Type Returned
REAL	SQL_REAL	Signed, single precision, floating-point number.	REAL
SMALLINT	SQL_SMALLINT	Small signed, exact whole number.	SMALLINT
TIME(*p*)	SQL_TYPE_TIME	Time value.	TIME(*p*)
TIMESTAMP(*p*)	SQL_TYPE_TIMESTAMP	Date and time value.	TIMESTAMP(*p*)
TINYINT	SQL_TINYINT	Very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	SQL_VARBINARY	Varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)***	SQL_VARCHAR	Varying-length character string.	VARCHAR(*n*)
**	SQL_WLONGVARCHAR	Varying-length Unicode character string.	**

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The ODBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** When you use the CHAR(*n*) or VARCHAR(*n*) data type to store multibyte data in a DB2, Greenplum, or Oracle database, you must specify the encoding in the CLIENT_ENCODING= connection option. Or, for Oracle only, to avoid having to set the encoding, use the NCHAR or NVARCHAR data types for multibyte data instead.

Data Types for Oracle

The following table lists the data type support for an Oracle database.

For data source specific information about Oracle data types, see the Oracle database documentation.

Table A1.4 *Data Types for Oracle*

Data Type Definition Keyword*	Oracle Data Type	Description	Data Type Returned
BIGINT	NUMBER	Large signed, exact whole number.	BIGINT
BINARY(*n*)	RAW(*n*)	Fixed or varying length binary string.	BINARY(*n*)

Data Type Definition Keyword*	Oracle Data Type	Description	Data Type Returned
CHAR(*n*)	CHAR(*n*)	Fixed-length character string.	CHAR(*n*)
DATE	DATE	Date values.	TIMESTAMP(*p*)***
DECIMAL|NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)	Signed, fixed-point decimal number.	DOUBLE†
DOUBLE	BINARY_DOUBLE	Signed, double precision floating-point number.	DOUBLE
INTEGER	NUMBER	Regular signed, exact whole number.	INTEGER
**	LONG	Varying-length character string data.	**
NCHAR(*n*)	NCHAR(*n*)	Fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	Varying-length Unicode character string.	NVARCHAR(*n*)
REAL	FLOAT	Signed, single precision floating-point number.	REAL
SMALLINT	NUMBER	Small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	Time value.	TIMESTAMP(*p*)***
TIMESTAMP(*p*)	TIMESTAMP(*p*)	Date and time value.	TIMESTAMP(*p*)
TINYINT	NUMBER	Very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	LONG RAW(*n*)	Varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR2(*n*)††	Varying-length character string.	VARCHAR(*n*)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Oracle data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** The timestamp returned by the DATE and TIME data types can be changed to date and time values by using the DATEPART function with the PUT function.

† The ORNUMERIC= connection argument and table option determine how numbers read from or inserted into the Oracle NUMBER column are treated. ORNUMERIC=YES, which is the default, indicates that non-integer values with explicit precision are treated as NUMERIC values.

†† The VARCHAR2(*n*) type is supported for up to 32,767 bytes if the Oracle version is 12c and the Oracle MAX_STRING_SIZE= parameter is set to EXTENDED.

Data Types for PostgreSQL

The following table lists the data type support for a PostgreSQL database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source specific information about PostgreSQL data types, see the PostgreSQL database documentation.

Table A1.5 *Data Types for PostgreSQL*

Data Type Definition Keyword*	PostgreSQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number. OR Signed eight-byte integer.	BIGINT
**	BIGSERIAL	Autoincrementing eight-byte integer.	**
**	BIT(*n*)	Fixed-length bit string.	**
**	BIT VARYING(*n*)	Variable-length bit string.	**
**	BOOLEAN	Logical Boolean (true/ false).	**
**	BOX	Rectangular box on a plane.	**
**	BYTEA	Binary data (byte array).	**
CHAR(*n*)	CHAR(*n*)	Fixed-length character string.	CHAR(*n*)
**	CIDR	IPv4 or IPv6 network address.	**
**	CIRCLE	Circle on a plane.	**
DATE	DATE	Date value.	DATE
DECIMAL|NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	Signed, fixed-point decimal number.	DECIMAL|NUMERIC(*p*,*s*)
DOUBLE	DOUBLE PRECISION	Signed, double precision, floating-point number.	DOUBLE
**	INET	IPv4 or IPv6 host address.	**

Data Type Definition Keyword*	PostgreSQL Data Type	Description	Data Type Returned
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
**	INTERVAL	Time span.	**
**	LINE	Infinite line on a plane.	**
**	LSEG	Line segment on a plane.	**
**	MACADDR	Media Access Control address.	**
**	MONEY	Currency amount.	**
**	PATH	Geometric path on a plane.	**
**	POINT	Geometric point on a plane.	**
**	POLYGON	Closed geometric path on a plane.	**
REAL	REAL	Signed, single precision floating-point number.	REAL
**	SERIAL	Autoincrementing four-byte integer.	**
SMALLINT	SMALLINT	Small signed, exact whole number.	SMALLINT
**	SMALL SERIAL	Autoincrementing two-byte integer.	**
**	TEXT	Variable-length character string.	**
TIME(*p*)	TIME(*p*)	Time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	Date and time value.	TIMESTAMP(*p*)
**	TSQUERY	Text search query.	**
**	TSVECTOR	Text search document.	**
**	TXID_SNAPSHOT	User-level transaction ID snapshot.	**
**	UUID	Universally unique identifier.	**

Data Type Definition Keyword*	PostgreSQL Data Type	Description	Data Type Returned
VARCHAR(*n*)	CHARACTER VARYING(*n*)	Varying-length character string.	VARCHAR(*n*)
**	XML	XML data.	**

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The PostgreSQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for Teradata

The following table lists the data type support for a Teradata database.

The NCHAR, NVARCHAR, NUMBER, and REAL data types are not supported for data type definition.

For data source specific information about the Teradata data types, see the Teradata database documentation.

Table A1.6 *Data Types for Teradata*

Data Type Definition Keyword*	Teradata Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
BINARY(*n*)	BYTE(*n*)	Fixed-length binary string	BINARY(*n*)
**	BLOB	Large Binary Object.	**
CHAR(*n*)	CHAR(*n*)	Fixed-length character string.	CHAR(*n*)
**	CLOB	Large Character Object.	**
DATE	DATE	Date values.	DATE
DECIMAL|NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	Signed, fixed-point decimal number.	DECIMAL(*p*,*s*)
DOUBLE	FLOAT	Signed, double precision, floating-point number.	DOUBLE
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER

Data Type Definition Keyword*	Teradata Data Type	Description	Data Type Returned
**	LONG VARCHAR	Varying-length character string.	**
***	NUMBER	Represents a fixed or floating point decimal.	***
SMALLINT	SMALLINT	Small signed, exact whole number	SMALLINT
TIME(*p*)	TIME(*p*)	Time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	Date and time value.	TIMESTAMP(*p*)
TINYINT	BYTEINT	Very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	VARBYTE(*n*)	Varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	Varying-length character string.	VARCHAR(*n*)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Teradata data type cannot be defined and, when data is retrieved, the native data type is mapped to a similar data type.

*** The Teradata data type is not yet supported. Columns of type NUMBER are ignored.

Appendix 2

DS2 Type Conversions for Expression Operands

The following table lists the automatic type conversions for expression operands. The first column is the “from” type. The first row is the “to” type.

Table A2.1 *DS2 Automatic Type Conversions for Assignment Statement*

From/To	TinyInt	SmallInt	Integer	BigInt	Decimal/ Numeric	Real	Double	Date	Time	Timestamp	Char	Varchar	NChar	NVarchar	Binary	Varbinary
TinyInt	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
SmallInt	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Integer	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
BigInt	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Decimal/ Numeric	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Real	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Double	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Date	N	N	N	N	N	N	N	Y	N	N	Y	Y	Y	Y	N	N
Time	N	N	N	N	N	N	N	N	Y	N	Y	Y	Y	Y	N	N
Timestamp	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	N	N
Char	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Varchar	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
NChar	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
NVarchar	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	N	N
Binary	N	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
Varbinary	N	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y

Appendix 3

DS2 Loggers

Overview of DS2 Loggers

The SAS logging facility is a framework that categorizes and filters log messages in SAS server and SAS programming environments, and writes log messages to various output devices. In the server environment, the logging facility logs messages based on predefined message categories, such as Admin for administrative messages, App for application messages, and Perf for performance messages. Messages for a category can be written to files, consoles, and other system destinations simultaneously. The logging facility also enables messages to be filtered based on the following thresholds: TRACE, DEBUG, INFO, WARN, ERROR, and FATAL.

DS2 provides several loggers to report both configuration and run-time information. In addition, DS2 provides a logger for the HTTP package. In general, INFO provides the minimum amount of information and DEBUG provides most (perhaps all) of the information that is needed to debug a field problem. The TRACE level provides anything and everything that might be of interest.

Configuration Loggers

Configuration loggers track information as the DS2 compiler starts up and provide context for the actual execution of the user's code.

The following configuration loggers are available.

App.TableServices.DS2.Config.Options
shows the options supplied to the DS2 compiler.

App.TableServices.DS2.Config.Source
shows the DS2 source code processed by the DS2 compiler.

App.TableServices.DS2.Config.Version
shows version information for all threaded kernel extensions loaded by the DS2 compiler.

Run-Time Loggers

Run-time loggers track actual execution. Some of the tracked information is generated for each row processed. Therefore, a large input table can produce very large amounts of data.

The following run-time loggers are available.

App.TableServices.DS2.Runtime.Calls
shows a trace of all method calls during execution.

App.TableServices.DS2.Runtime.SQL
shows all SQL statements either prepared by the DS2 compiler, executed by the DS2 compiler, or both.

App.TableServices.DS2.Runtime.Timing
shows the time that is spent during code compilation and execution. Depending on the level of information that is requested (INFO, DEBUG, TRACE), timing information is provided at various points throughout DS2 execution. Examples are parse time, compilation time, various audit and transformation pass times, as well as the INIT, RUN, and TERM method execution times.

App.TableServices.DS2.Runtime.Put
records all PUT statement output.

App.TableServices.DS2.Runtime.Log
records all messages that are sent to the SAS log in a SAS session. On SAS Federation Server, these messages are also appended to the ODBC statement handle as diagnostic records by default.

HTTP Package Logger

The HTTP client supports logging of HTTP traffic through the SAS logging facility.

The name of the logger is App.TableServices.d2pkg.HTTP. The logger supports these logging levels:

INFO
shows general traffic information such as connections and disconnections from the web server, request information, and status information.

DEBUG
shows the headers from all requests and responses plus the first 64 bytes of body data. This enables you to see what the client and server are doing without having to see all of the data that is transmitted to the server.

TRACE
shows all of the data sent to and received from the web server.

Example: Logging All SQL Operations

The following example creates a logging configuration file to log all SQL commands. After the configuration file is created, the LOGCONFIGLOC= system option is set to specify the name of the configuration file that is used to initialize the SAS logging facility.

The following code creates the logging configuration file.

```
<?xml version="1.0"?>
   <logging:configuration xmlns:logging="http://www.sas.com/xml/logging/1.0/">
   <logger name="App.TableServices.DS2.Runtime.SQL">
       <level value="trace"/>
   </logger>
   <logger name="App.TableServices.DS2" additivity="false">
       <appender-ref ref="DetailedOutput"/>
   </logger>
   <root>
       <appender-ref ref="RootLogger"/>
       <level value="info"/>
   </root>
   <appender name="DetailedOutput" class="FileAppender">
       <param name="append" value="false"/>
       <param name="FileNamePattern" value="sql.%S{App.Log}"/>
       <layout>
           <param name="ConversionPattern" value="%-5p:%sn:[%c{3}]:%m"/>
       </layout>
   </appender>
   <appender name="RootLogger" class="FileAppender">
       <param name="Append" value="false"/>
       <param name="ImmediateFlush" value="true"/>
       <param name="FileNamePattern" value="%S{App.Log}"/>
       <layout>
           <param name="ConversionPattern" value="%m"/>
       </layout>
   </appender>
   </logging:configuration>
```

The LOGCONFIGLOC= system option is set to reference the configuration file as **`config.14s`**.

```
sas -log test.log -logconfigloc config.14s
```

This produces a file, **`sql.test.log`**, which contains a series of messages similar to these.

```
DEBUG:00000084:[DS2.Runtime.SQL]:Found 0 NOCHANGE columns
   INFO :00000085:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:CREATE TABLE WORK.outp
      ("i" DOUBLE, "j" DOUBLE )
   DEBUG:00000086:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:passed:rc=0x00000000
   DEBUG:00000087:[DS2.Runtime.SQL]:Found 0 NOCHANGE columns
   INFO :00000088:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:SELECT * FROM WORK.outp
   DEBUG:00000089:[DS2.Runtime.SQL]:0x0afce4b0:exec-direct:passed:rc=0x00000000
   INFO :00000095:[DS2.Runtime.SQL]:stmt=0x0afae060:prepare:SELECT * FROM WORK.outp
   DEBUG:00000096:[DS2.Runtime.SQL]:stmt=0x0afae060:prepare:passed:rc=0x00000000
```

```
INFO :00000097:[DS2.Runtime.SQL]:stmt=0x0afae060:execute
DEBUG:00000098:[DS2.Runtime.SQL]:stmt=0x0afae060:execute:passed:rc=0x00000000
```

The exact content of the output file is defined by the **`ConversionPattern`** parameter of the layout within the DetailedOutput appender. The DetailedOutput appender is associated with the definition of the App.TableServices.DS2 logger. This causes every logger in the hierarchy that is rooted at App.TableServices.DS2 to be logged to the same file. The **`additivity="false"`** modifier prevents the log messages from moving upward.

Recommended Reading

- *Encryption in SAS Viya*
- *Mastering the SAS DS2 Procedure: Advanced Data Wrangling Techniques*
- *SAS Cloud Analytic Services: Accessing and Manipulating Data*
- *SAS Viya: DS2 Language Reference*
- *SAS Viya Data Management and Utility Procedures Guide*
- *SAS FedSQL Language Reference for SAS Cloud Analytic Services*
- *SAS Viya Formats and Informats: Reference*
- *SAS Viya National Language Support: Reference Guide*
- *SAS Viya System Options: Reference*
- *The DS2 Procedure: SAS Programming Methods at Work*
- SAS offers instructor-led training and self-paced e-learning courses to help you get started with the DS2 programming language, land earn how the DS2 language works with the other SAS products. For more information about the courses available, see sas.com/training.

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Index

D

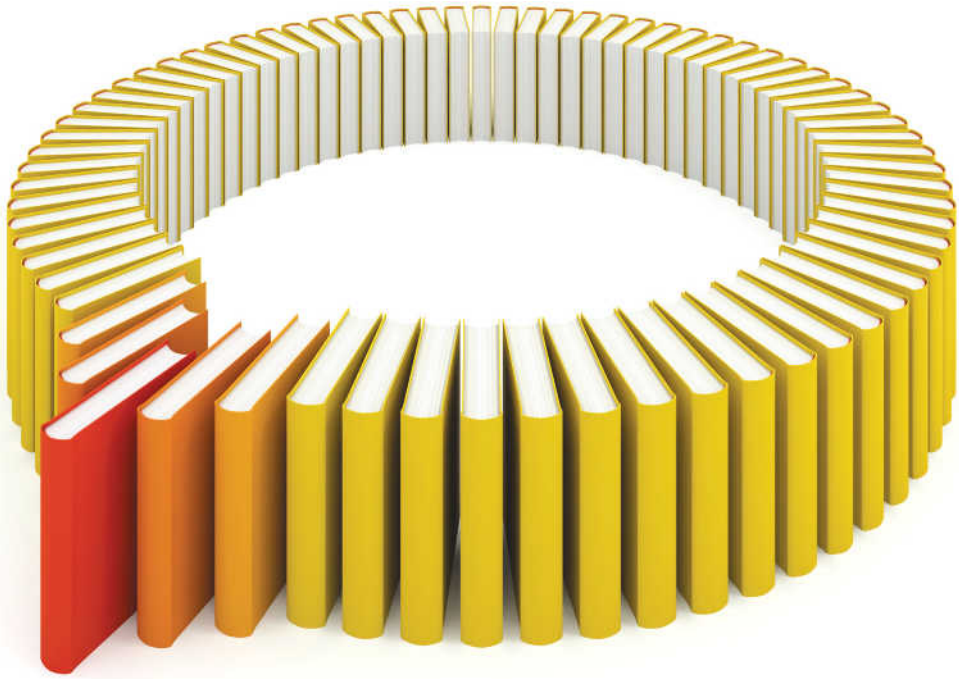
E

F

R

S

T



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.