High-Performance Data Processing with CASL in SAS Viya®

CHEAT SHEET

High-Performance Data Processing with CASL in SAS Viya® - CHEAT SHEET

Prepared date 30Mar2023

Print

Review of CAS Language (CASL) Fundamentals

General CAS Concepts

- Use the CAS statement to make a connection to CAS.

```
cas <sessionName> <options>;
```

Example:

```
cas conn sessopts=(timeout=1800 metrics=TRUE);
```

CASL Programming Components

- The CAS language is very different from the traditional SAS programming language. CASL which is more like Python, includes 4 major components:
  - CAS actions
  - variable data types
  - statements
  - functions

- Use the CAS Procedure to execute CASL code:

```
proc cas;
    …;
quit;
```

- **CAS Actions**

```
proc cas;
    action-set.action-name < / parameter=value, parameter=value, ...>;
quit;
```

- **CAS Variable Data Types**

	Data Types
Numeric	double, INT32 and INT64
String	varchar and string
Boolean	stores the values of TRUE or FALSE
Other	array, dictionary, and table

- **Statements**

Statement Syntax	What it does
DESCRIBE *variable-name | expression;*	writes the structure and data type of CASL variables and expressions to the log
PRINT *value-1 <value-2>...<value-n>;*	writes the values of constants, variables, and expressions to the current output location

- **Arrays**
  - A CASL array is one of the two list data types.
  - Arrays are most useful with CASL programming for grouping a series of strings or numbers in a variable and then using the variable as a parameter to a CAS action.

Syntax	What it does
array-name={*value-1 <,value-2 ...>***};**	defines an array
array-name[*position***]** **array-name[***lower-bound***:***upper-bound***]** **array-name[{***position-n, …, position-z***}]**	access array elements

  - Loop over an array:

```
DO <variable> OVER <array-name>;
    ... repetitive CASL code ...
END;
```

  - Array Operators

Syntax	What it does
array-1 **=** *array-1* **||** *value;* *array-1* **=** *array-1* **||** *array-2;*	appends to an array
variable **=** *array-1* **/** *array-2;*	finds unique values in arrays
variable **=** *array-1* **&** *array-2;*	finds common values in arrays
variable **=** *array-1* **==** *array-2;*	compares two arrays

variable **=** *value* **==** *array*;	checks for a single value in an array

- Array Functions

Function	What it does
DIM(*array***);**	returns the number of elements in an array
SORT(*array***);**	return an array in ascending order
SORT_REV(*array***);**	returns an array in descending order

- **Dictionaries**
  - Creating dictionaries

```
dictionary-name = {key-1=value-1 <, key-n=value-n,...>};

dictionary-name.key-1 = value-1;
<dictionary-name.key-n = value-n;>

dictionary-name["key-1"] = value-1;
<dictionary-name["key-n"] = value-n;>
```

  - Accessing dictionary values

```
dictionary-name["key"]

dictionary-name.key
```

  - Deleting a dictionary key

```
DELETE dictionary-name["key"]
DELETE dictionary-name.key
```

  - Loop over an dictionary

```
DO <key> ,<value> OVER <dictionary>;
    ... repetitive CASL code ...
```

```
END;
```

- **CAS Actions Overview**
  - CAS actions return a dictionary back to the client.
  - There are no rules about how many keys are contained in the dictionary, or what data types are returned.
  - You can store the results of a CAS action in a variable:

```
action-set.action-name <result = results-variable> / ...;
```


Close

Print

Review of Connecting to CAS and Accessing Data

Exploring the CAS Connection with CAS Actions

- Exploring a CAS Connection with Actions

Action	What it does
session.listSessions	Displays a list of the sessions on the server.
session.sessionStatus;	Displays the status of the current session.
session.metrics / on=*TRUE	FALSE*;	Displays the metrics for each action after it executes.
sessionProp.listSessOpts;	Displays the session options and session values.
sessionProp.getSessOpt / name="*option to display*";	Displays the value of a session option.
sessionProp.setSessOpt / timeout=*seconds*, **metrics=***TRUE	FALSE <, ...>*;	Sets session option(s).
builtins.serverStatus;	Shows the status of the server.
builtins.getLicensedProductInfo;	Shows the information for licensed SAS products.
builtins.getLicenseInfo;	Shows the license information for a SAS product.
builtins.actionSetInfo / all=*TRUE	FALSE*;	Shows the build information from loaded action sets.
builtins.help / actionSet="*action-set*", **action="***action*";	Shows the parameters for an action or lists all available actions.

Exploring and Accessing Data Sources

- **Caslibs**

- A caslib is an in-memory space to hold tables, access control lists, and data source information. All data is available to CAS through caslibs and all operations in CAS that use data are performed with a caslib in place.

- A caslib can have session scope or global scope.
  - Session-scope caslibs are accessible only from the session that adds the caslib. This enables server-side data access to a programmer and does not interfere with other sessions. Use a session-scope caslibs when you need to access but not share tables.
  - Global-scope caslibs can be accessible to any session on the server. Depending on access controls, users can share access to in-memory tables. Personal caslibs and pre-defined caslibs are global caslibs. Global-scope caslibs are useful when you want other users to have access to the table, subject to access controls.

- Exploring Caslibs with Actions

Action	What it does
table.caslibInfo;	Shows all available caslibs and caslib information.
table.fileInfo / caslib="*caslib-name*";	Lists the available data source files in a caslib.
table.tableInfo / caslib="*caslib-name*";	Shows all available in-memory CAS tables in a caslib.
table.addCaslib / **name**="*caslib-name*", **path**="*folder-path*", <,*additional parameters*>;	Displays the value of a session option.

- **LIBNAME Statement**

  - Create a SAS Compute server library reference name, or libref, that points to a caslib.

LIBNAME *libref* **CAS** <**SESSREF**="*cas-session-name*" **CASLIB="***caslib-name*">**;**

- **Managing In-Memory Tables**

  - Server-side files are files that are associated with a caslib.

  - Client-side are files that are not associated with a caslib. They can be either:
    - Path-based files that are are available to the CAS server, but are not associated with a caslib.
    - SAS data sets in SAS libraries that are available only to the compute server. These files need the CASUTIL procedure to load into CAS.

  - Load a **server-side** file into memory:

```
table.loadTable /
    path="file-name.ext", caslib="caslib-name",
    casOut={name="table-name",
            caslib="caslib-name",
            replace= TRUE |FALSE...},
    importOptions={fileType="TYPE", ...},
    <additional parameters>;
```

- Load a **client-side path-based** file into memory:

```
table.upload /
    path="file-path",
    casOut={name="table-name"
            caslib="caslib-name"...};
```

- Load a **client-side SAS data set** from a SAS library into memory:

```
proc casutil;;
    load data=SAS-data-set
        <casout="table-name">
        <outcaslib="caslib">
        <promote> <replace> <option(s)>;
quit;
```

- **Saving and Dropping CAS Tables**
  - Save a CAS in-memory table to a caslib's data source:

```
table.save /
   table={caslib="caslib-name",
          name="table-name"},
    caslib="target-caslib", name="file-name.ext";
```

  - Drop a CAS in-memory table:

```
table.dropTable /
    caslib="caslib-name",
    name="table-name"
    <quiet=TRUE|FALSE>
...;
```

- **CAS Table Scope**
  - CAS tables have either session scope or global scope.
  - Considerations for Session-scope tables
    - Best used for general purpose programming.
    - Typically provides better performance than global-scope tables because concurrency locks are not used.
  - Considerations for Global-scope tables

- Best used for tables that are accessed by a large number of users, especially the visual interfaces.
- A global-scope table cannot be replaced. You must drop it and load the replacement data.

- Promoting Tables

  - Promote a table when loading into memory:

```
table.loadTable /...
    casOut={...promote=TRUE};
```

```
table.upload /...
    casOut={...promote=TRUE};
```

  - Promote an existing session scope table to global scope:

```
table.promote /
    name="table-name",
    caslib="caslib-name",
    <target="output-table-name">,
    <targetLib="output-casib">;
```

Type and save a note for this page.

Submit

Close

Print

Review of Exploring and Validating Data

Exploring Data Using CAS Actions

- Return rows from a table

```
table.fetch /
    table={caslib="caslib-name",
           name="table-name",
           where="where-expression",
           vars={column-names}
           ...},
    fetchVars={"column-name-1",..., "column-name-n"},
    from=first-row-number,
    to=last-row-number,
    index=TRUE | FALSE,
    sortBy={{name="column-name"<, order="DESCENDING">}}
;
```

- Show column information:

```
table.columninfo /
    table={castable},
    <, additional parameters>;
```

- Show the number of rows in a Cloud Analytic Services table:

```
simple.numrows /
    table={castable};
```

- Compute the distinct and missing values in a CAS table:

```
simple.distinct /
    table={castable},
    inputs={column-names},
    casout={casouttable}
    <, additional parameters>;
```

- Generates a frequency distribution for one or more columns:

```
simple.freq /
    table={castable},
    inputs={column-names/em>},
    casout={casouttable}
    <, additional parameters>;
```

- Store the result of a CAS action:

```
    action-set.action <result=variable> / ...;
```

Working with Result Tables

- Result table operators

Action	What it does
table-variable-name[*row-selection,column-selection*]	Selects rows and columns from a result table object.
result-table[{*row-1, row-2, ...*}]	Selects specific rows.
result-table[*row-lower-bound*:*row-upper-bound*]	Selects rows based on lower and upper bounds, inclusive.
result-table[,*column-positions*]	Selects columns by position.
result-table[*row-positions*,{"*column-1*","*column-2*",...}]	Selects specific rows and columns.
result-table[,{"*column-name-1*", "*column-name-2*", ...}]	Selects all rows and specific columns.
result-table.where(*filter-expression*)	Subsets the rows in a result table according to the expression and returns a new result table.
result-table.compute <{'*variable-name*', '*label*', '*format*'},> *expression*)	Adds a temporary computed column to a result table and returns a new result table. The values are computed according to an expression.

- Result Table Properties

Action	What it does
***result-table*.nrows;**	Contains the number of rows in the table.
***result-table*.ncols;**	Contains the number of columns in the table.
***result-table*.attrs;**	Contains action-specific attributes for the table. This property is not added by all actions.
***result-table*.name;**	Contains the name of the table. In most cases, it is the same as the dictionary key that is used to access the table from the results.
***result-table*.title;**	Contains the title of the table.

- Save a Result Table

Action	What it does

SAVERESULT *variable-name* <**DATAOUT**=<libref.>*data-set-name* | **LIB**=*libref*> <**REPLACE** | **NOREPLACE**>;	Save as a SAS data set.
SAVERESULT *variable-name* <**CSV**="*file-name*" | **FILE**=*output-file*> <**REPLACE** | **NOREPLACE**>;	Save as a CSV file.
SAVERESULT *variable-name* <**CASLIB**=*caslib-reference-name* <**CASOUT**="*table-name*"> <**REPLACE** | **NOREPLACE**>;	Save as a CAS table.

Validating Data with CAS Actions

- Eliminate rows that have duplicate or unique group-by variable values:

```
deduplication.deduplicate /
    table={caslib="caslib-name",
           name="table-name",
           groupBy={"column-name" <, ...>},
    }
    casOut={caslib="caslib-name", name="table-name"},
     <duplicateOut={caslib="caslib-name", name="table-name"},>
    noDuplicateKeys=TRUE | FALSE;
```

- Copy one table to another:

```
table.copytable /
   table={caslib="caslib-name",
          name="table-name",
          computedVars={
               {name="variable-name", format="string", label="string"...},
               <{column-metadata-n}>
           computedVarsProgram="statement;
                                <more statements;>",
    },
    casout={caslib="caslib", name="table-name"};
```

- Embed text in the program and assign it to a given variable:

```
SOURCE variable;
      <text>
ENDSOURCE;
```


Close

Print

Review of Preparing Data

CAS Action Overview

When you program using CAS, there are multiple ways to interact with the CAS server to process data. You can use:

- CAS-enabled PROCS,
- traditional DATA step programs,
- and open-source languages.

All of these techniques leverage the CAS API to convert native language elements to CAS actions.

Modifying Tables

- Update rows in a table:

```
table.update /
    table={castable},
    set={
          var="column-name", value="expression"},
          <{var-n={"column-name-n", value-n="expression"},>
     } ...;
```

- Copy one table to another:

```
table.copyTable /
    table={castable},
    casout={casout-table};
```

Preparing Data

- Add a format to a computed column:

```
...
table = {...
          computedVars={
                              name="variable-name",
                              format="string",
                              label="string"...},
                              { ... }
} ...
```

- Create a calculated column:

```
table={...
      computedVarsProgram="expression(s)"
```

```
};
```

- Convert a character value to numeric:

```
INPUTN(source, informat)
```

- Run DATA step code in CAS using the CAS action:

```
dataStep.runCode /
    code="string";
```

- Alter CAS table metadata:

```
table.alterTable /
      caslib="string", name="table-name",
      rename="string",
      label="string",
      drop={column-names},
      keep={column-names},
      columns={
                  {AlterTableColumn-1}
                  <,{AlterTableColumn-n}, ...>
     };
```

- Impute missing values:

```
dataPreprocess.impute /
      table={castable},
      casout={casouttable},
      inputs={
                {column-to-impute-1}
               <,{column-to-impute-n}, ...>
      }
      <copyAllVars=FALSE | TRUE,>
      <methodInterval="impute-method",>
      <methodNominal="impute-method">;
```

- Transpose a CAS table:

```
transpose.transpose /
      table={ name="table-name", caslib="caslib", groupBy="column"},
      casOut={casouttable},
      label="string",
      transpose={"column-name-1" <,"column-name-n" ...>},
      ID={"column-name-1" <,"column-name-n" ...>};
```

- Calculate quantiles, high and low whiskers, and outliers:

```
percentile.boxPlot /
      table={castable},
```

inputs={column-names},
casOut={*casouttable*},
<, additional parameters},

- Execute SQL in CAS:

fedSQL.execDirect / query="*sql-query*";

Type and save a note for this page.

Submit

Close

Print

Review of Analyzing and Summarizing Data

Introduction

Typically, you want to do all your analyzing, summarizing, and modeling of large data in CAS. Remember, CAS stores entire tables in memory for multi-threaded parallel processing, resulting in extremely fast results. When the processing is complete, CAS can return a subset of the data or summarized results to the compute server for additional processing, visualization, and reporting.

Analyzing Data with CAS Actions

- Generate simple descriptive statistics of numeric variables including the sample mean, sample variance, sample size, and sum of squares:

```
simple.summary /
      table={castable},
      inputs={column-names},
      subset={summary-statistics},
      casout={casouttable},
      <, additional parameters>;
```

- Perform aggregation on selected variables with additional flexibility and control:

```
aggregation.aggregate /
      table={castable},
      varSpecs={
                  {name="column-name", agg="summary-statistic" | subset={summary-
statistics},
                  {colNames={column-names},
                  <,{additional columns}>
      };
      saveVariableSpecification=TRUE | FALSE,
      casout={casouttable}
      <,{additional columns}>;
```

- Generate a frequency distribution for one or more variables:

```
simple.freq /
      table={castable},
      inputs={column-names},
      casout={casouttable}
      <,additional parameters>;
```

- Construct the frequency and crosstabulation tables:

```
freqTab.freqTab /
      table={castable},
      tabulate={
                  {vars={column-names},<, cross={column-names}>}
```

```
            <, {freqTab_tabulate-n}, ...>
    }
    tabdisplay="CROSSLIST" | "LIST",
    includeMissing=TRUE | FALSE,
    order="FORMATTED" | "FREQ" | "INTERNAL"
    <,additional parameters>;
```

- Performs one-way or two-way tabulations:

```
simple.crossTab /
    table={castable},
    row="column-name",
    col="column-name",
    weight="column-name",
    aggregator="aggregator-value",
    <, additional parameters>;
```

Visualizing and Reporting

- There are two programming options to create visualizations.
- The first and easiest option is to reference a CAS table in a SAS ODS graphics procedure. This works well on data that is not extremely large.
- If a CAS table is referenced in an ODS Graphics procedure, CAS will send the ENTIRE table to the compute server for processing.
  - If the CAS table is a manageable size, the compute server will process the data and create a data visualization
  - If the table exceeds the transfer size limit, an error is returned. This protects you from transferring extremely large tables.
  - The table size limit that CAS can send to the compute server is typically set by the administrator

Close