Paper SAS309-2017

The Architecture of the SAS® Cloud Analytic Services in SAS® Viya™

Jerry Pendergrass, SAS Institute Inc.

ABSTRACT

SAS® Cloud Analytic Services (CAS) is the cloud-based run-time environment for data management and analytics in SAS®. By run-time environment, we refer to the combination of hardware and software where data management and analytics take place. CAS is a platform for high-performance analytics and distributed computing. CAS provides data management and an analytics framework that can run in the cloud, that can act as a cloud, and that provides the best-in-class analytics that SAS is known for. This new architecture functions as a public API, allowing access from many different clients such as Lua, Python, Java, REST, and yes, even SAS®. CAS provides user-level sessions, which provides security, allows the user to share data between sessions, and provides fault tolerance. Fault tolerance allows a worker node to crash without losing data and allows the user request to continue running to completion. The isolation provided to each session allows one session to fail without affecting other sessions. The concept of "always in memory" in CAS means a request to the server is not aware of what the server does to allow the action to access the data. The entire file might be in memory or just pieces of the file might be mapped into memory, just in time for the action to access the data. This allows CAS tables to be loaded that are larger than the memory available across the grid.

INTRODUCTION

SAS® Viya™ is a new direction for SAS. No longer are the interfaces to analytics private to SAS, they are public and accessed through third-party clients, open-source clients, Java and SAS®. SAS® Viya™ chose a cloud/grid model to implement its newest version of the SAS architecture .The Cloud Analytic Services (CAS) is the server designed to operate in the cloud, on a single host or in a public or private cluster. When looking at the design goals for CAS,SAS looked toward the cloud as inspiration.

There are major characteristics that we believe are important in a cloud server environment.

- Broad network access with a public API that supports all types of clients.
- Customer data is held in a secure cloud allowing all computing to be performed close to the data.
- Elasticity. A server can add and remove nodes as the demands on the server changes.
- Fault tolerance. Clusters can contain hundreds of nodes. One or more of the nodes is going to fail at some point. The server has to be able to recognize the failure, isolate it, remove it from the cluster and continue on as if nothing ever happened. Data replication allows other nodes to take over blocks of data as needed.

The cloud is a new way of thinking about computing, and it requires new models and new technologies to take full advantage of it. CAS has been built from the ground up to address each of these requirements. CAS is not just a server but an integral part of the SAS® Viya™ architecture. SAS® Viya™ is designed to meet the needs of the analytic community with a fully operational answer for using analytics to your advantage. This architecture is backed by the best technical support you have come to expect from SAS®. SAS® Viya™ is a robust architecture, not something you have to piece together from one source repository to another and expect it to work. This paper focuses on CAS, its concepts and how the normal user will interface with CAS, and how its design helps achieve the goals of being a real cloud/cluster analytic solution.

MAIN CONCEPTS FOR THE CLOUD ANALYTIC SERVICES

The best way to understand CAS is to understand the terminology and concepts used in the server and how it all fits together to achieve the goals mentioned above.

THE SERVER CONTROLLER

CAS consists of from 1 to N hosts. These hosts can be a part of a cluster, a single system, in the Amazon cloud, or OpenStack. One host is designated the "controller". The controller executes one process as the "server controller" and one process for each "server worker". The controller is the repository for the global state of the server.

The client connects to the server controller using either the binary API port or the REST API port. Once the user has been authenticated, the server controller creates a session process on the controller and each worker that is to be included as part of this session. The client connection is then transferred to the session controller. At this point, the client can start submitting work to CAS.

THE WORKERS

The purpose of the worker nodes is to execute the same request as the other workers and controller, and process the data that is local to that worker. This allows the user or admin to spread data across the cluster to allow for local parallel processing of data that will come back together to produce a result. Just like the controller nodes, the worker nodes support one process as the server worker and one process for each session.

TWO DIFFERENT EXECUTION MODES

CAS can be deployed in one of two modes:

- SMP: Symmetric Multi-Processing
- MPP: Massive Parallel Processing

In SMP mode, there is just one node running CAS. This allows a single node to take advantage of CAS without having to invest in a large cluster or cloud environment. You get the advantage of running the analytics on multiple processors and the same functionality of an MPP system. Hadoop is not available in SMP mode.

In MPP mode, CAS supports workers to help offload the analytics and spread the data out to allow for parallel processing on the multiple workers. MPP allows the user to create a session with as many worker nodes as they need up to the current number of workers available. In this environment Hadoop is available if desired. With Hadoop, the controller is the name node and the workers run the analytics on the CAS data. CAS data is known as “in-memory tables”.

These two modes allow an administrator a lot of freedom to configure CAS in the way that benefits the users. A user might want to bring up a CAS in SMP mode on a laptop to work on an application that will eventually be deployed onto a large MPP cluster.

SESSIONS

CAS uses sessions to track users and offers a full Security interface to protect data at the file level, as well as the column level. The sessions provide isolation for the user, which protects the integrity of the server.

The purpose of connecting to CAS is to execute server requests. A user must create a session to submit a request. The user has two ways to connect to the server:

1. REST interface (HTTP-based)
2. Binary interface (ProtoBuf-based)

The user must be authenticated by CAS in order to a create session.

The information needed to connect to CAS:

- port #
- host name
- locale (optional)
- number of nodes for this session (zero means all workers)
- Authentication information. The supported authentication methods include.
    - Kerberos
    - username/password
    - LDAP/PAM/OAUTH

Once the user has been authenticated, the server controller creates a session controller process for the user and a session worker process for each worker in the session. As you can see, the server is not just one process, but a series of interconnected processes across many hosts. Maybe a picture will help you visualize what CAS looks like in MPP mode.

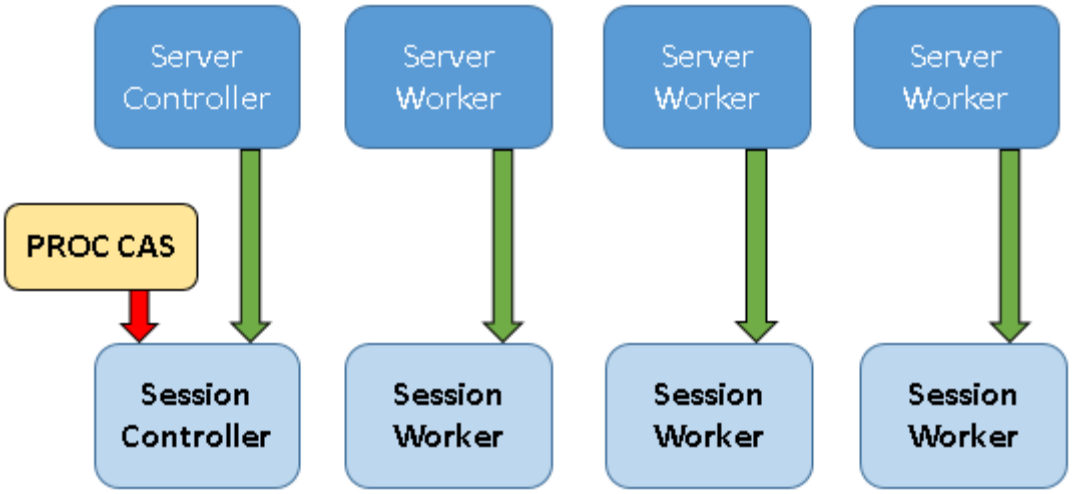

Figure 1. Example of a Server Using Multiple Workers and a Session Active from Running the PROC CAS Procedure

CAS TABLES

The data in CAS is represented as a "table". Tables must be loaded into memory to be accessed. Once loaded, a table can be saved as a file to a specific name on disk. Tables are the data containers for CAS. They consist of rows and columns. Columns consist of:

- Variable name
- Label name
- Data type
- Attributes
- Display format
- Access control

Loaded tables do not have to fit into memory. CAS manages the tables to allow rows to be traversed by workers. The data for the table is normally spread out across multiple workers to allow for parallel processing of the data. For maximum multi-lingual capability, processing is almost entirely in UTF-8 (legacy SASHDAT/LASR data being the sole exception).A table that has been “loaded” is called an in-memory table.

SASHDAT FILE FORMAT

The format of an in-memory table is called SASHDAT format. A disk file that is in SASHDAT format can easily be loaded as a table without having to do much more than read the header. Other table file formats can be loaded or streamed into CAS. These formats include:

- CSV
- Base (for example, sas7bdat)
- Database access (Oracle, Teradata, and so on)
- SPSS
- Excel
- JMP
- Client-specific format
- Hive

The table is loaded as a SASHDAT in-memory table. This is the native format that all analytics actions use to process table rows. Once an in-memory table has been loaded, it is advantageous to save that file for future use as a SASHDAT file on disk. This allows the `loadtable` action to optimize the access to that table. CAS supports many different filesystems that allow for easy access to SASHDAT and other file formats, including DNFS, NFS, and Hadoop. If the user is finished with a table and chooses not to save it, the table can be “dropped” and it will no longer be available as an in-memory table.

WHERE DO TABLES COME FROM AND WHERE ARE THEY STORED

One of the important concepts in CAS is where a table comes from, how it gets loaded, and what metadata is needed to load the table. Tables can be loaded from disk, or can be streamed into the server from a database, ESP stream, or any other client that knows the format of that data. CAS provides a concept to organize these data sources. We call this concept “CAS libraries” or caslibs.

CASLIB: A LOCATION TO STORE DATA AND METADATA.

CAS libraries are referred to as caslibs and are an important concept in accessing data in CAS. They differ enough from SAS libraries that they have a different name. A caslib is a container instance that can contain zero or more CAS tables, either in-memory or files on a disk. A caslib is associated with a data source from which the server can access data. A caslib can provide connection information to the data source. The connection information can be as simple as a directory path or more complicated such as the host, port, and credentials for connecting to a database.

A caslib is associated with access controls that define which groups and individual users are authorized to access the contents of the caslib.

The `addcaslib` action can be used to create a new caslib.

Below are two examples of a caslib creation. The first is a base directory to /disk/lax/smith. All disk references for this caslib are in reference to the directory. CAS does not allow absolute paths when identifying a file on disk. The reference is always in relation to the hub directory for the caslib. The second example defines access to a data stream from ORACLE.

Example of a simple CASLIB (casl syntax)

```
addcaslib /
     caslib="mylib"
     datasource={srctype="path"}
       path="/disk/lax/smith;
```

Example of a more complex CASLIB to ORACLE: (LUA syntax)

```
r = s:addCaslib{lib="orlib",
             datasource={srctype="oracle",
                         username="SCOTT",
                         password="tiger",
                         path="ORA11G"
                        }
```

ACTIONS

The purpose of CAS is to allow users to run analytic requests against their in-memory tables and get back results. CAS calls requests “actions”. Actions are combined into logical action sets. There are hundreds of actions sets available to the user. An action set must be loaded before the actions in that action set can be run. There are management action sets that control loading tables, access control, options processing and other system support. But, of course, the most important action sets are for the many varieties of analytics.

An action is a task performed by CAS at the request of the user and returns a response containing results and status. These are the building blocks for analytic exploration. The action is executed on all nodes in the cluster. Each node processes the data that exists on its node and synchronizes the response back to the controller. You will see many examples of actions as we continue describing the concepts of CAS. The interface to an action is the action name plus an array of directories as parameters. One important concept for CAS is that all clients use the same interface to the action and all clients get the same response back. It is up to the client to position the response to best suit the client interface and the expectation of the user. The concepts discussed in this paper are crucial to knowing how to use actions effectively in the client.

RESULTS SENT BACK TO THE CLIENT

As you can see, the CAS in-memory table is very important to CAS. So it should also be important to the results returned to the client. CAS returns results in many different formats. The most information-rich results usually take the form of a "result table". These are similar in structure to the ODS output tables in SAS programming.

Normally the size of the result table is not very large, but there can be many of them. The BY-group processing creates one result table for each group combination. This can lead to hundreds of result tables.

These result tables have rows and columns just like in-memory tables. They have variables, labels, data type, formats, and attributes. The client takes the result table and presents it to the user in the native form of the client. In Java, this is a Java object; in Python, it is a data frame; in PROC CAS, it is a result table. PROC CAS presents the table as it was sent from the server and allows the user to access and process the result. Result tables have attributes to describe contents of the table. Attributes are used to display a result table. BY group information is stored as attributes. PROC CAS is integrated into ODS to allow for a nice display of result tables using the BY group attributes.

The results are sent back as an array of dictionaries. The expected result depends on the action. It is up to the client to allow useful access to these results. Later in this paper we will look at all of the results types that an action can return.

A CLOSER LOOK AT SESSIONS IN CAS

So why does CAS use sessions and why is each session a new process? The new process provides many benefits to CAS.

LOCAL TABLES

The session can create local tables that are only accessible by this session. This allows the session to be protected from other users who could access the memory if the sessions were in the same process. The session can "promote" a table to a global scope to allow other sessions to see the data. The user can set the access rights to the table to control which users can see the global table. Access is extended to the column level. Promoting a table does not copy the table, it just copies the information about the table to the server controller. The sessions can then request access to the table and the information about the table will be given to the session. Data is only copied when necessary. The expectation is that all sessions have access to the files they are allowed to see without the overhead of copying data for each session. There is a lot of communication from the session to the server and from the controller to workers. This provides the consistency needed across the entire server. If a session has not promoted a table when the session exits, the table is dropped.

ISOLATION

Since the session is in a separate process, if it crashes or corrupts memory, this will only affect this session. The rest of the server will not be affected other than to lose this session. The data in memory is protected from other users. The user can be running under the user's ID or as a proxy ID provided by the admin. In either case, the real identity of the user is used for access control.

SUBSETTING THE GRID

A user can choose how many workers the session will use, which allows the user to determine how much power the session requires. Smaller sessions might be a good choice if your data is small and connection time is important. The overhead of a large number of workers might not be an advantage in this case. A subset session still has full access to all tables.

SESSION IDENTIFICATION

The session is identified by a 36-character UUID. This `sessionid` is used to allow sessions with the same user identity to run actions that affect each other. An example is a session requesting that another session cancel the request they are executing. Sessions with different identities cannot see each other, but can share data. The UUID can be used to allow multiple users to connect to the same session. This UUID allows a user to close the connection to the session, and the reconnect at a later time.

PARALLEL EXECUTION OF ACTIONS IN MULTIPLE SESSIONS

Sessions can only execute one CAS action at a time. If a client submits a new action while one is already running, the additional action is queued. If your application requires parallel execution, you can simply start additional sessions as you need them.

Figure 2 is an example of a client running multiple actions to different sessions.

These actions can run in parallel.

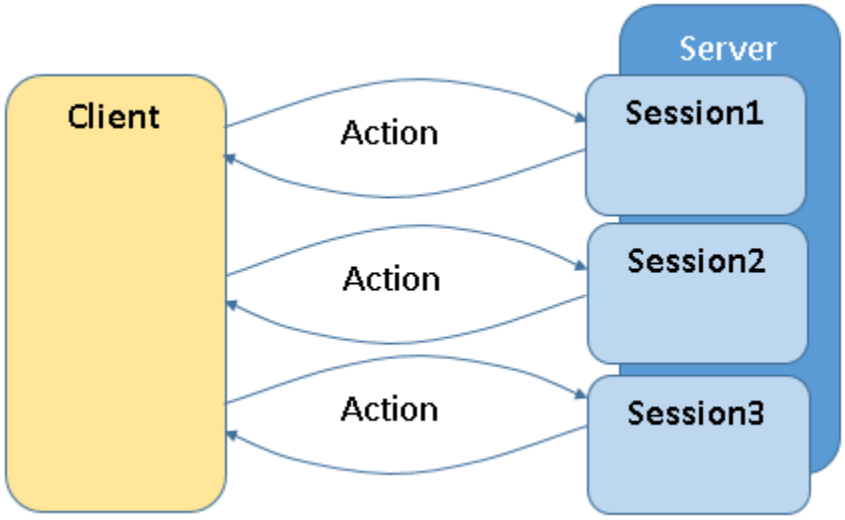

Figure 2. Multiple Session Interactions

Figure 3 shows synchronous action execution to one session.

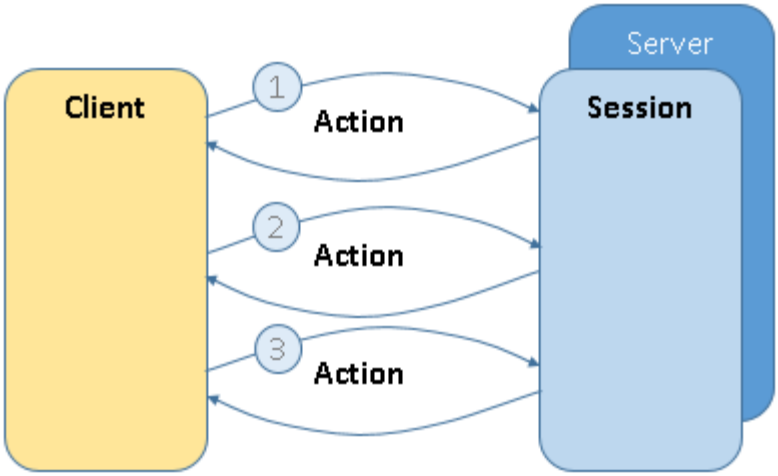

Figure 3. Single Session Interactions

DNFS AS A DATA REPOSITORY

DNFS, which stands for "Distributed Network File System" is CAS's support for distributed data access to NFS directories. DNFS gives customers a good alternative for deployments where CAS is not co-located with Hadoop and yet must provide similar capabilities via a distributed NFS interface.

The fundamental design principle behind DNFS is that NFS-mounted directories are accessed concurrently by each controller or worker node in a CAS grid

DATA REDUNDANCY

The fault tolerance in CAS depends on data redundancy. Tables stored on disk are a collection of blocks across all nodes. This allows each node to process the rows on that node in parallel with other nodes.

Each node might contain copies of blocks from other nodes to allow that node to take over responsibility of those blocks in case of a worker failure.

FAULT TOLERANCE

Fault tolerance is an important feature in CAS. In order to provide robust recovery, the communication between controllers and workers is a new implementation of "cluster communications" we call GCCOMM.

When implementing a system that can contain hundreds of nodes, one or more of them is going to fail at some point. CAS can handle a failure of one or more workers. The new GCCOMM subsystem is designed specifically to solve this issue. GCCOMM can detect a failure of one or more workers. This causes all workers and the controller to reconfigure the system without the failed workers and restart the action allowing the remaining workers to recover the data from the lost worker from redundant blocks.

In order for an action to recover data, blocks must be replicated across the cluster so that the workers can take over responsibility for a given block if a worker fails. CAS supports different types of data source access to allow for data replication.

The action is in charge of how much has to be re-executed to complete the action. It might be a full execution of the action or the action could have checkpoints to recover work completed and continue after the last checkpoint. The client makes this failure seamless to the user. In some cases data might have already been delivered to the user, and that data has to be refreshed after a failure. As stated, the success of recovery depends on redundancy, which is provided by replication or access to the original SASHDAT file in the case of DNFS. Hadoop can be used to provide this replication or the user can direct CAS to replicate tables as needed to give the needed fault tolerance.

ADD NODE AND REMOVE NODE

CAS initially starts out with a given controller and set of worker nodes. Once the server is operational, the admin can add nodes to the system through the `addnode` action. The new node will synchronize with the rest of the nodes into memory and become available as a new worker. Sessions that did not set their worker node size will be able to use this new node at the next action boundary.

The admin can also remove a node from the system. This is a little more work since the data that is active on this node is relocated to another active node. Once all data has been relocated, the node can be taken out of the system. This is not the same as a failure, as the node just disappears from the list of nodes for a session at the next action boundary.

MORE ABOUT CAS TABLES AND CASLIBS

We can distinguish CAS tables according to the following criteria:

- Global and session tables.

  A table has a session scope after it is loaded or created in a CAS session. Only the session has access to the table. If you want to make the table available to other sessions, then you need to promote the table into a global caslib. The `promote` action sets the table to a global scope. All tables start out as session tables and then may be promoted. Unless a more specific authorization rule is added for the table, then anyone who has access to the caslib will then be able to access the promoted table.

- Temporary and permanent tables.

  All tables in CAS are permanent unless declared temporary. A temporary table is created during an action and is removed from the server automatically when the action completes.

- In-memory tables and on-disk files.

  CAS manages memory associated with tables behind the curtains. The user typically does not know whether the data for a table is on disk, or has been allocated with pool-based memory, or

exists in a mixture of the two. However, all data that hits the disk is memory mapped to give CAS the same advantages in memory management.

- Distributed and repeated tables.

  A table is distributed if worker nodes hold portions of the table, but no worker has access to all the rows of the table. Only the active rows of the table are counted here. Other nodes can hold replicated copies of rows, to be activated in the case of a node failure. A repeated table, on the other hand, is one where each worker node has access to all the rows of the table. Duplicated tables are useful in some operations, such as table joins, where the rows of a dimension table need to be matched against the keys in the fact table; if all rows of the dimension table are repeated on each worker node, then the join can be completed without exchanging rows between the worker nodes. Repeated tables are not managed for failover, since each node has all the rows anyway.

- Table access control

  You do not need to worry about permissions to access a table. If the user requesting your action is not authorized to access the table, the action will not be invoked. If the user does not have access to certain columns of the table, these variables do not appear in the variable list for your table.

SO WHY DID WE CREATE CASLIBS?

- All operations are governed by access controls. A permission file determines what groups and users can and cannot do. Access control is integrated into caslibs.
- Permission-based column filters and row filters can now be in effect.
- Every table is associated with a caslib.
- The concept is that you load a table from a caslib and the result is a CAS table in a caslib. The two caslibs do not have to be the same. You might be loading from an ORACLE caslib and placing the CAS table in a private caslib.

The key concept here is that a table is accessed through a caslib. A loaded table exists in a caslib; the caslib describes everything needed to load or access a table.

DATA LIFE CYCLE

The following graphic in Figure 4 summarizes fundamental concepts of the data life cycle for CAS.

- CAS operates on in-memory tables. Clients can add or upload data. The server can load data from server-side data sources.
- The server can save and load data from server-side data sources.
- When an in-memory table is dropped, it does not affect a persisted file. Persisted files are removed from the data source by deleting them.

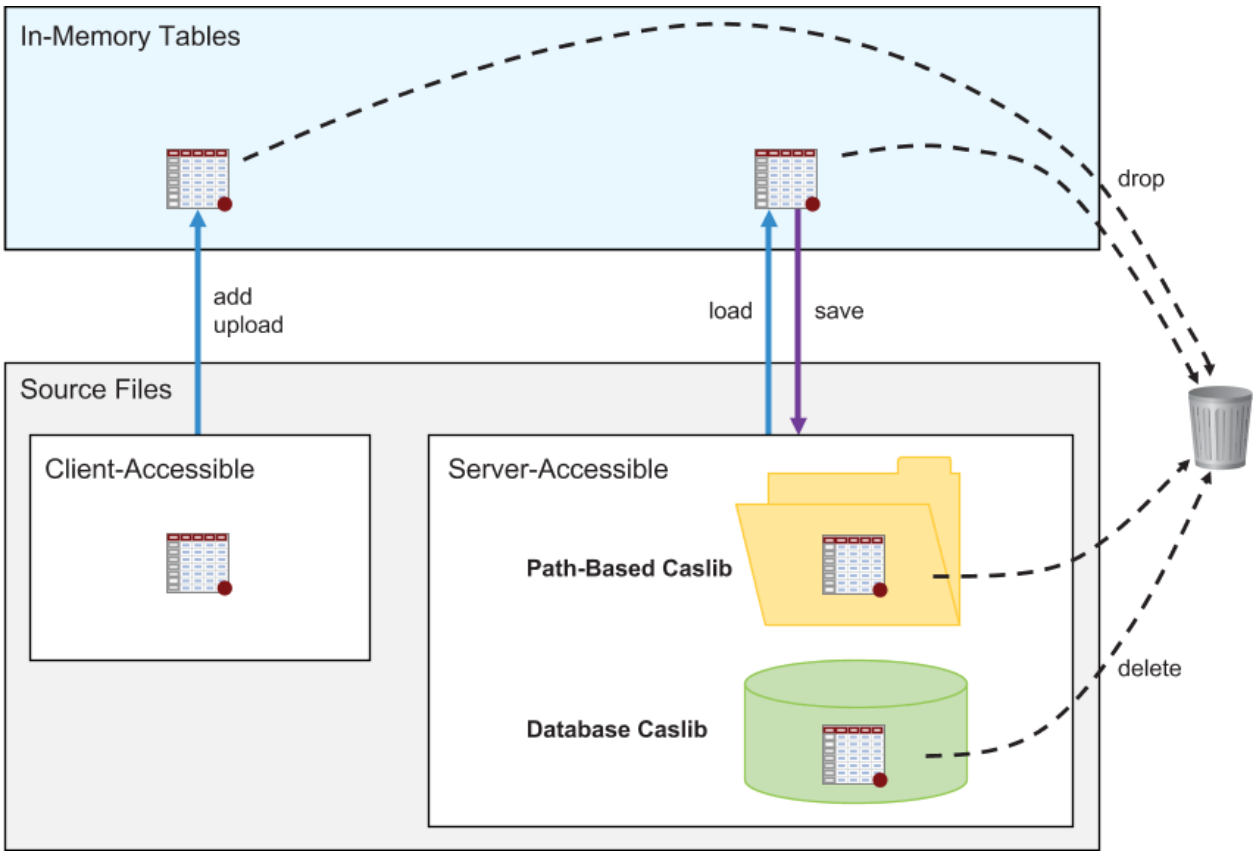

Figure 4. Fundamental Concepts of the Data Life Cycle for CAS

EXAMPLE OF AN ACTION

An example of an action is the `glm` action. In order to run the `glm` action, a table must be loaded. You can use the DATA step to load a table into CAS. In fact, the DATA step itself will be an action running within CAS. That's right! That DATA step is now multi-threaded and multi-processor. This allows those DATA steps that take hours to now take minutes. There is not a one for one replacement of the code going from SAS to CAS, but the changes are minor and the speed increase is worth the effort.

In this example, a session named `myserver` is created. The LIBNAME `sascas1` is then defined using the CAS LIBNAME engine that will use the newly created session.

The DATA step defines its output destination to be in CAS; the DATA step will run as an action in CAS. SAS will parse the syntax and then send the resulting compiled program to CAS to be run.

```
cas myserver;     /*   Create a session to CAS */

libname sascas1 cas sessref=myserver ;    /* Define the libname to this
                                             session. The DATA step will use
                                             this as the session to use*/


%let nObs    = 1000;
%let nCont   =    5;
%let nClass  =    4;
```

```
data sascas1.glmdata;
    drop i j xbeta;
    array x{&nCont};
    array c{&nClass};
    call streaminit(1);
    do by=1 to 2;
    do name='joe','jane';
    do mood='good','bad';
    do i=1 to &nObs;
       xbeta=0;
       do j=1 to dim(x);
          x{j}=rand('uniform');
          if j < 3 then do;
            xbeta=xbeta+j*x{j};
          end;
       end;
       do j=1 to dim(c);
          c{j}=int(rand('uniform')*3);
          if j < 3 then do;
            xbeta=xbeta+j*c{j};
          end;
       end;
       y = xbeta+10*rand('normal');
       yb=int(rand('uniform')*2);
       output;
     end;
     end;
     end;
run;
```

This creates a CAS table called glmdata in the server in the default caslib. You can then run any client to execute the `glm` action on this CAS table. Note that this action uses a GROUPBY, so multiple result tables are returned as results. The attributes on the result tables define the definition of the GROUPBY. These attributes are used in SAS to tell ODS how to display the results.

The arguments to the action are an array of named dictionaries. PROC CAS is designed to make the creation of the parameters simple. See below for an example. Note the two-level action name. Actions are contained within action sets. The regression action set is loaded as part of the action. The results of the action are placed into the PROC CAS variable glmResult when the action is complete. As you can see, the parameters might contain other arrays and named directories.

Using PROC CAS the syntax is:

```
glm result=glmResult
    table={name='glmdata', groupBy={'name', 'mood', 'by'}}
    model={depvars={{name='y'}}
    effects={'x1', 'x2', 'x3'}};

print "Number of Result Tables : " dim(glmResult);
print glmResult[2]; /* jan bad 1 */
```

Using Python the syntax is:

```
tbl = conn.CASTable('glmdata');
result = tbl.groupby(['name','mood','by']).
             glm(model=[depvars=dict(name='y')]],
             effects=['x1','x2','x3']);

print('number of Result tables:',len(result));
  print( result.get_group(name='jan', mood='bad', by=1));
```

Note the difference in how the client prepares the parameters for the action. PROC CAS creates a dictionary with the necessary parameters. Python creates a CASTable object that represents the table in the CAS server and then executes the groupby method and then executes the glm method which invokes the action.

The input to the server is identical, but these clients took different paths to create the parameters. The results sent back to the client will also be identical. The client will present the results in way representative of that client.

This action, because it's a GROUPBY, creates 428 result tables. A partial output of this action is displayed in figure 5. It is the second result table, which is the GROUPBY with name=jan, mood=bad and by=1. This output is printed by PROC CAS and uses ODS to render thus result. Note that you can use the BY group descriptions on the labels and the table of contents to traverse through the results.

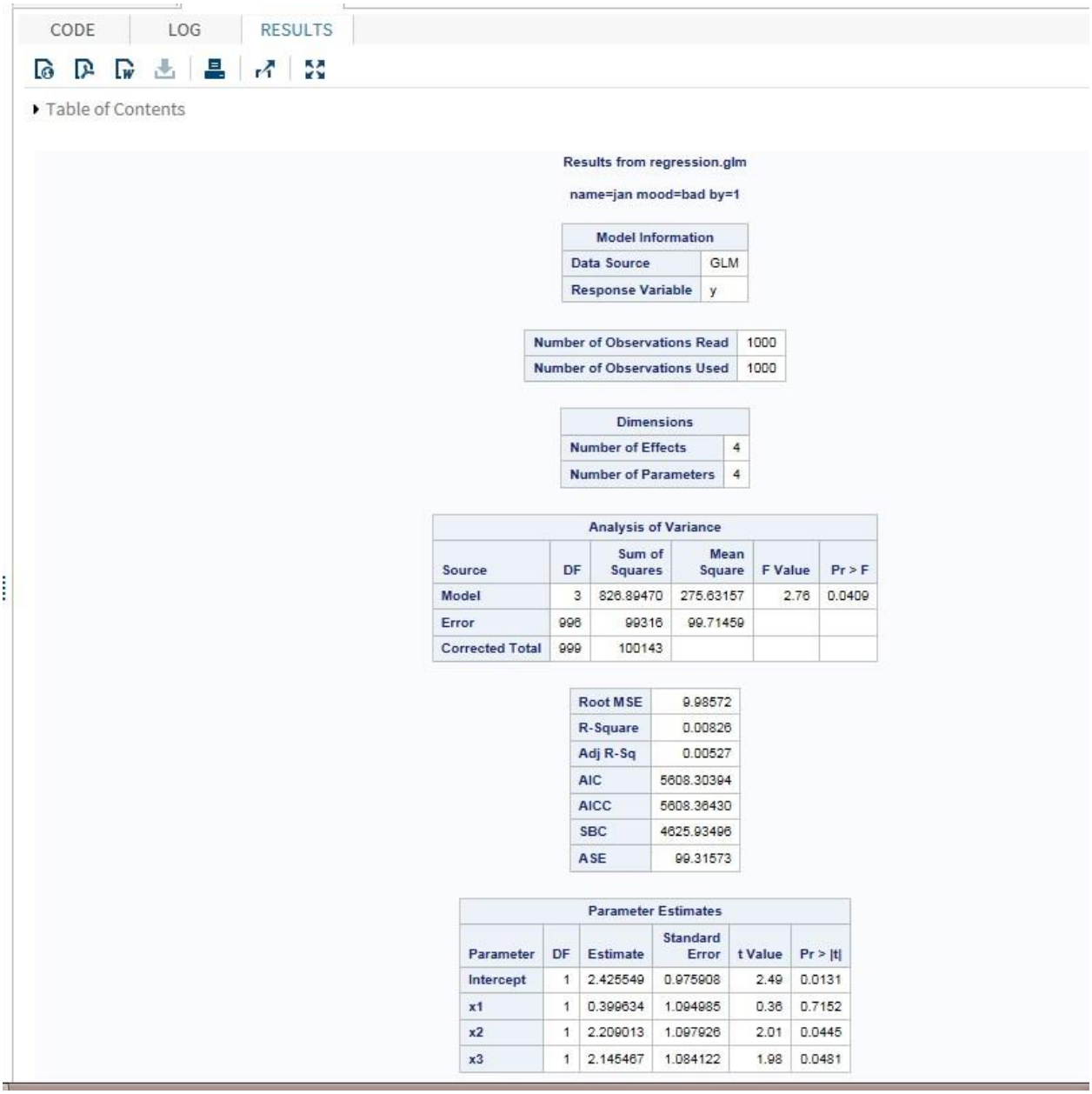

Figure 5. Output of the First BY Group Result Table Displayed by ODS

The interface to actions can get complex. To help with complexity, the system provides a “reflection” interface to describe the parameter interface of any action. This reflection can be used to generate class or data screens to allow the user to know what the parameters are and to know their default values and ranges. In the case of data sources, the parameters change based on the type of data source, and this is reflected in the data returned from the reflect action.

The REST interface also has access to the `reflect` action, so a user using REST can ask what is the interface to a given action.

SERVER DATA TYPES

The results from an action are an array of dictionaries. Each item in the dictionary can be one of the following types:

- integer (64 bit or 32 bit)
- double
- string
- result table (rows and column)
- item store (organized binary data)
- blob (binary data)
- Time (cas)
- Date (cas)
- DateTime (cas)
- list of values (array or dictionary)

Note that a result table might not support all of these types, but these types can be returned in the list of results from an action.

SUPPORTED CLIENTS

So what would a server be without clients? Below are the binary clients supported by SAS® Viya™.

Each client has a different internal syntax to talk to CAS. CAS always sees the same request syntax for the same request. Below you will find an example showing the syntax to request the loading of a .csv file into CAS and promote the in-memory table for several of the supported clients. Note that the sum.csv is found in the user's personal caslib (CASUSER(jerry)) and the in-memory table will be loaded into the CASTestTmp caslib.

- Lua

```
s:table_loadTable{casOut={caslib="CASTestTmp"},caslib="CASUSER(jerry)",
importOptions={allowTruncation=true,delimiter=",",encoding="utf8",
fileType="csv",getNames=true,guessRows=20.0,nThreads=0.0,stripBlanks=false,
varChars=true},path="sum.csv",promote=true}
```

- Python and IPython using Jupyter Notebook

```
s.table.loadtable(casout={'caslib':'CASTestTmp'},caslib='CASUSER(jerry)',
importoptions={'allowtruncation':True,'delimiter':',',
'encoding':'utf8','filetype':'csv','getnames':True,'guessrows':20,'nthread
s':0,'stripblanks':False,'varchars':True},path='sum.csv',promote=True)
```

- Java CAS client (SAS® Visual Analytics, SAS® Visual Investigator)

```
// Setup the options
LoadTableOptions options = new LoadTableOptions();

// Setup the casOut options
Casouttablebasic casOutOptions = new Casouttablebasic();
casOutOptions.setCaslib("CASTestTmp");
options.setCasOut(casOutOptions);
```

```
  options.setCaslib("CASUSER(jerry)");

  // Setup the importOptions options
  Xlsopts importOptionsOptions = new Xlsopts();
  importOptionsOptions.setParameter("allowTruncation", true);
  importOptionsOptions.setParameter("delimiter", ",");
  importOptionsOptions.setParameter("encoding", "utf-8");
  importOptionsOptions.setParameter("fileType", "csv");
  importOptionsOptions.setGetNames(true);
  importOptionsOptions.setParameter("guessRows", 20.0);
  importOptionsOptions.setParameter("nThreads", 0.0);
  importOptionsOptions.setParameter("stripBlanks", false);
  importOptionsOptions.setParameter("varChars", true);
  options.setImportOptions(importOptionsOptions);

  options.setPath("sum.csv");
  options.setPromote(true);

  // Invoke the action
CASActionResults<CASValue> results
```

- REST interface using JSON

```
          {
 "casOut":
    {
     "caslib": "CASTestTmp"
    },
   "caslib": "CASUSER(jerry)",
   "importOptions":
    {
     "allowTruncation": true,
     "delimiter": ",",
     "encoding": "utf-8",
     "fileType": "csv",
     "getNames": true,
     "guessRows": 20.0,
     "nThreads": 0.0,
     "stripBlanks": false,
     "varChars": true
    },
   "path": "sum.csv",
   "promote": true
  }
```

- SAS® Studio (access to traditional SAS)
  - SAS procedures
  - DATA step
  - PROC CAS integrated with ODS using the CAS Language (CASL)

```
table.loadTable /
   casOut={caslib="CASTestTmp"}
   caslib="CASUSER(jerry)"
   importOptions={allowTruncation=true,delimiter=",",
encoding="utf8",fileType="csv",getNames=true,guessRows=20.0,nThreads=0.0,
stripBlanks=false,varChars=true} path="sum.csv" promote=true;
```

DATA STEP IN CAS

The DATA step is now fully integrated into CAS. You have two choices.

1. You can run the DATA step from SAS and it will run in CAS if the data is resident in CAS.
2. You can submit DATA step code directly to CAS from any client.

Since the DATA step can run in CAS, it is multi-threaded and multi-processor. The DATA step will process data in the given node with possibly many threads per node. This makes a significant impact on reducing the time for data preparation.

The DATA step is one way to upload a table into a CAS. This is useful when you want to subset or modify the resulting table from the given input data stream.

Here is a simple code example to show running the DATA step to load a table and then accessing the table using PROC CAS. If I wanted to run this fetch in Python after the DATA step, I could connect to this session using the UUID of the session and run the same fetch action.

```
Cas session1;
Libname mycas cas sessref=session1
data mycas.carssashelp;
    set sashelp.cars;
    run;

proc cas;
   session session1;
   fetch / table={name="carssashelp"};
run;
```

Here is the output using ODS:

CODE | LOG | RESULTS

Table of Contents

table1: Results from table.fetch

Selected Rows from Table CARSSASHELP

Index	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	Engine Size (L)	Cylinders	Horsepower	MPG (City)	MPG (Highway)	Weight (LBS)	Wheelbase (IN)	Length (IN)
1	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6	265	17	23	4451	106	189
2	Buick	Rendezvous CX	SUV	USA	Front	$26,545	$24,085	3.4	6	185	19	26	4024	112	187
3	Chrysler	300M 4dr	Sedan	USA	Front	$29,865	$27,797	3.5	6	250	18	27	3581	113	198
4	GMC	Envoy XUV SLE	SUV	USA	Front	$31,890	$28,922	4.2	6	275	15	19	4945	129	208
5	Isuzu	Rodeo S	SUV	Asia	Front	$20,449	$19,261	3.2	6	193	17	21	3836	106	178
6	Lincoln	Town Car Signature 4dr	Sedan	USA	Rear	$41,815	$38,418	4.6	8	239	17	25	4369	118	215
7	Mercury	Grand Marquis LS Ultimate 4dr	Sedan	USA	Rear	$30,895	$28,318	4.6	8	224	17	25	4052	115	212
8	Pontiac	Vibe	Wagon	USA	Rear	$17,045	$15,973	1.8	4	130	29	36	2701	102	172
9	Toyota	Highlander V6	SUV	Asia	All	$27,930	$24,915	3.3	6	230	18	24	3935	107	185
10	Volvo	C70 LPT convertible 2dr	Sedan	Europe	Front	$40,565	$38,203	2.4	5	197	21	28	3450	105	186
11	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2	4	200	24	31	2778	101	172
12	Buick	Century Custom 4dr	Sedan	USA	Front	$22,180	$20,351	3.1	6	175	20	30	3353	109	195

Figure 6: ODS output from running the DATA step to load a table.

MEMORY MANAGEMENT IN CAS

Memory is a major issue with any server. The memory assumptions in CAS are:

- Memory is expensive; the old fixed length strings are expensive. We now have a string type called VarChar. VarChar is a new type that is a pointer, length. It can be used instead of fixed length strings.
- Memory takes different forms to provide rows to actions.
- Memory is local and global and can move between processes.
- Memory is compressible. At a cost.
- Memory is treated as a read-only resource that can be appended to.
- Look at memory as a set of large blocks containing small blocks. Blocks are either active or inactive. They are active when you need to access the data.
- Memory mapping enables the server to provide best performance when memory use exceeds physical capacity.
- SASHDAT files enable memory mapping that speeds up access to the data.
- Memory mapping does not mean that the server runs at the speed of disk. It means that the server takes advantage of the best efficiencies available from the operating system.

CONCLUSION

In this paper you have learned important concepts about the architecture of Cloud Analytics Services in SAS® Viya™. The stated goals are to provide an analytics service with a public API accessible by many clients supported by SAS or open-source clients using plug-in modules from SAS. The availability of the DATA step in the server environment addresses one of the primary goals to get the preparation of the data into the cloud to reduce overall elapsed time. The definition of caslibs allows the user and administrator a well-defined interface to manage the data tables needed to process the workload, along with providing the metadata to access streaming data and provide access control onto those tables. The server in SAS® Viya™ is capable of providing a well-designed deployable solution for any analytic need. The flexibility and fault tolerance of CAS make it well suited for large analytic problems. CAS is also useful for small requests that require a small number of nodes. This allows for better throughput for these small requests. CAS can live in the cloud, on a private cluster, or on a single server. This brings all of the potential client interfaces into one server that can provide a central interface to your analytic needs.

REFERENCES

ACKNOWLEDGMENT

This content of this paper is a collaboration of information from all of the developers and technical writers working in Cloud Analytic Services.

RECOMMENDED READING

- *Base SAS® Procedures Guide*

CONTACT INFORMATION

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

Jerry Pendergrass
Distinguished Software Developer
Advanced Server Division
SAS Institute Inc.
Jerry.pendergrass@sas.com