Paper SAS4497-2020

Achieving Optimal Performance with the SAS® Data Connector Accelerator for Hadoop®

David Ghazaleh, SAS Institute Inc.

ABSTRACT

The SAS® Data Connect Accelerator for Hadoop uses the SAS® Embedded Process to improve performance when moving data between Apache™ Hadoop® sources and SAS® Cloud Analytic Services. Achieving optimal performance during data movement can be challenging. Some of the variables to consider include cluster size, number of cores available, size of the data, and number of splits in a file. This paper explains how to optimize the SAS Embedded Process and how to take advantage of the new Apache™ Spark™ Continuous Processing mode available in the SAS Embedded Process.

INTRODUCTION

SAS Embedded Process contains a subset of Base SAS software that supports the multithreaded SAS DS2 language. Running DS2 code directly inside Hadoop effectively leverages massive parallel processing and native resources. Strategic deployment such as scoring code, data transformation code, or data quality code can be applied in this manner. The parallel syntax of the DS2 language, coupled with SAS Embedded Process, allows traditional SAS developers to create portable algorithms that are implicitly executed inside Map Reduce or Spark. SAS Embedded Process is also used for parallel data transfer between SAS Cloud Analytic Services and Apache Hadoop Distributed File System (HDFS) or Apache Hive™. SAS Embedded Process for Hadoop is orchestrated by Map Reduce framework or Apache Spark while YARN manages load balancing and resource allocation.

SAS Cloud Analytic Services (CAS) is an in-memory analytics server deployable on cloud-based environments as well as on-premises. The CAS server provides a fast, scalable, and fault-tolerant platform for complex analytics and data management. CAS is the ideal run-time environment for faster processing of huge amounts of data.

While the CAS server can be deployed to a single machine, the use of a distributed server deployed across multiple machines unleashes the potential of the massively parallel processing (MPP) architecture.

In a single machine environment, also known as symmetric multiprocessing (SMP) mode, the CAS server is comprised of a single controller node where all of the analytics are executed. In an MPP architecture, the distributed CAS server is comprised of a controller node, an optional backup controller, and one or more worker nodes. All of the analytics are processed in parallel across multiple worker nodes.

A CAS client connects to the controller node. Upon connection, a session process is started on the session controller node and on all session worker nodes. The client submits work requests to the session controller, which parses out work to each session worker so that the computation can be executed in parallel.

Analytics and data management operations are executed on tables that are already loaded into the CAS server. There are three different ways to move data in and out of CAS:

1. Serial data transfer: data is moved sequentially from the data source into the CAS

controller node and then to the CAS worker nodes.

2. Multinode data transfer: this is an extension of the serial data transfer where CAS workers make simultaneous connections to the data source to read and write data. This mode is an attempt to parallelize data transfer, since each CAS worker uses one single thread connection and it is subject to environment limitations imposed by the data source, such as maximum number of concurrent connections.
3. Parallel data transfer: this mode offers true parallel load and best throughput where data is moved directly into the CAS worker nodes using multiple concurrent channels.

This paper concentrates on the parallel data transfer mode from an Apache Hadoop Distributed File System or Apache Hive.

SAS DATA CONNECTOR ACCELERATOR TO HADOOP

SAS Data Connector Accelerator for Hadoop allows parallel data transfer between CAS server and Hadoop Distributed File System or Hive. In order to use parallel data transfer, SAS Embedded Process needs to be installed on every node of the cluster that is capable of running a Map Reduce or Spark task. Figure 1 depicts the components involved in the parallel data transfer between CAS and Hadoop.

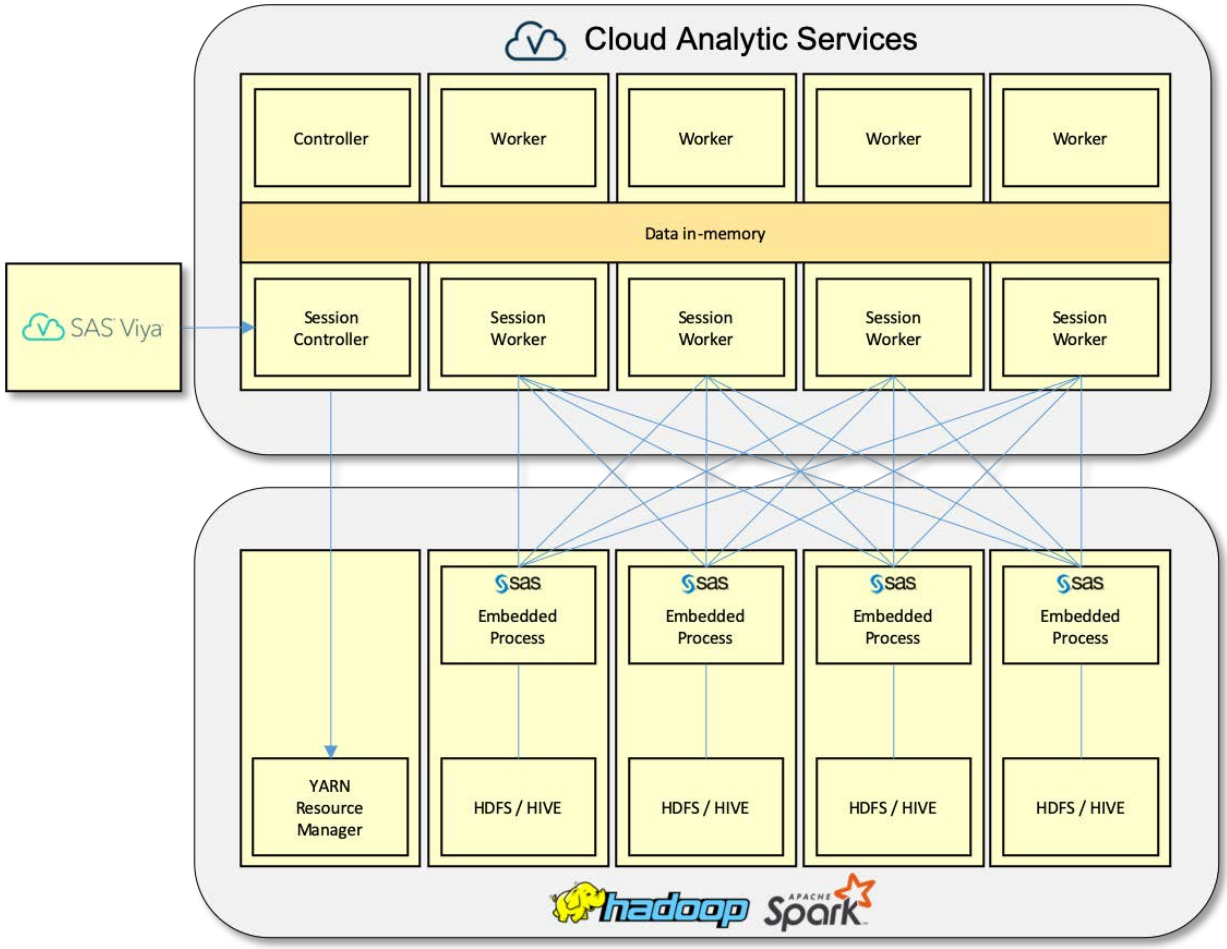

Figure 1. Parallel Data Transfer Using SAS Embedded Process.

A client connects to the CAS controller in order to obtain a controller session. An example of client is the SAS® Studio V, a web-based application that allows submission of SAS code and is integrated with the SAS® Viya® components.

In order to use the SAS Data Connector Accelerator, the client needs to use the addCasLib action to define a CAS library connection to Hadoop as shown in this code:

```
caslib hdplib
       datasource=(
           srctype="hadoop",
           server=”hiveservername”,
           hadoopJarPath=”/opt/sas/viya/config/data/hadoop/lib”,
           hadoopConfigDir=”/opt/sas/viya/config/data/hadoop/conf”,
           platform="mapred", /* Either mapred or spark. Default: mapred */
           dataTransferMode="parallel"
          );
```

The addCasLib action defines a CAS library to source type Hadoop and connects to the Hive server specified in the server parameter. The hadoopJarPath parameter is the folder where the Hadoop JAR files are stored on the CAS controller node. The hadoopConfigDir parameter is the folder where the Hadoop configuration files are stored on the CAS controller node. The platform parameter specifies the platform on which to run the SAS Embedded Process as a Map Reduce job or a Spark application. The default platform is Map Reduce. The dataTransferMode parameter specifies the mode of data transfer. Possible data transfer mode values are:

- auto: specifies to first try to load or save the data in parallel using embedded processing. If it fails, a note is written to the SAS log and serial processing is attempted.
- parallel: specifies to load or save the data in parallel by using the SAS Data Connector Accelerator for Hadoop.
- serial: specifies to load or save the data serially by using the SAS Data Connector to Hadoop.

When data needs to be transferred from the Hadoop file system to the CAS server, the client submits a CAS loadTable action to the session controller. The session controller starts the embedded process job to read the input data from the Hadoop file system and evenly distribute the data among the CAS worker nodes. This type of SAS Embedded Process job is called CAS input mode. The following code sample submits a CAS action to load a table into the CAS server memory:

```
proc cas;
  loadTable
  caslib="hdplib"
  path="numericdata400"
run; quit;
```

When data needs to be transferred from the CAS server to the Hadoop file system, the client submits a CAS save action to the session controller. The session controller starts the embedded process job to receive the data stored in the CAS workers’ memory and evenly store the data in the Hadoop file system. This type of SAS Embedded Process job is called CAS output mode. The following code sample submits a CAS save action to save a table to Hive:

```
proc cas;
   save
   caslib="hdplib"
   name="numericdata400out"
   replace=true
```

```
   table={name="numericdata400"};
run; quit;
```

Additional information about CAS loadTable and save table actions can be found in the SAS Cloud Analytic Services User's Guide.

RUNNING SAS EMBEDDED PROCESS ON MAP REDUCE

A Map Reduce job is a collection of map tasks and reduce tasks that are executed on nodes in the Hadoop cluster. The map tasks read splits of the input file, process them and send the resulting records to the reduce tasks. A file input split is a fixed-size slice of the input file that is assigned to one map task. An input split is associated with one file path, a starting offset in the file, and the length of the split. The minimum and maximum split size is configurable and may not translate into a physical block. Input splits are calculated by a file input format implementation, which is also responsible for the instantiation of a record reader. For example, if a job is reading data from a Parquet file, the Parquet input format implementation is responsible for calculating the file input splits and for instantiating the Parquet record reader. Input split calculation happens before the Map Reduce job is submitted for execution in the cluster.

In a conventional Map Reduce job, each map task reads records from the single input split that is assigned to it. A map task runs in a dedicated Java Virtual Machine (JVM) process in one node of the Hadoop cluster.

SAS Embedded Process is a Map Reduce application. However, it does not assign only one input split to a task. It assigns many. That means, the SAS Embedded Process map task can read multiple input splits in parallel. All SAS Embedded Process processing happens at the record reader level before the map function is called. Assigning multiple file splits to one single task avoids excessive start and stop of Map Reduce tasks (JVMs) reducing unnecessary overhead. This technology is called super reader.

When saving data to Hadoop file system or Hive table, the SAS Embedded Process job writes multiple parts of the output file in parallel from within the same Map Reduce task. This technology is called super writer.

The super reader initiates access to files via a multi-threaded and multi-split reader framework. It retrieves the input splits from the underlying file input format implementation and distributes them among the SAS Embedded Process tasks based on data locality. When reading Hive tables, the super reader calls into the Hive/HCatalog input format and record reader to retrieve records. The Hive table may be stored in any file format, for example, Avro, Parquet, and ORC. When reading records from a file stored on HDFS, the super reader calls into the standard SAS Embedded Process input format and record readers. SAS Embedded Process also offers a mechanism for user-written custom input format and record reader.

SAS Embedded Process attempts to schedule tasks on nodes where physical blocks of the input file can be found. For example, if file blocks are found on nodes N1, N2, N3, and N4, the embedded process schedules tasks to run on those four nodes. However, depending on resource utilization, the YARN Resource Manager may decide to schedule the tasks on different nodes.

The SAS Embedded Process may take a path representing a directory on HDFS or a Hive table name as its input to the job. A Hive table ultimately points to a directory under the Hive warehouse location. A path representing a directory includes all files under it as input to the job. Multiple file paths as input may lead to unbalanced amount of data per task. This is due to the fact an input split is associated with one file path. A well distribution of data among the physical files may help avoid unbalanced amount of data per task.

There are three independent sets of threads controlling the SAS Embedded Process execution:

- Reader threads: these are Java threads responsible for reading records from input splits and filling up input buffers that are given to the compute threads. Input buffers are stored outside of the JVM heap space.
- Compute threads: these are native language threads responsible for the transmission of records between CAS and SAS Embedded Process. Compute threads run inside the SAS CAS driver container that is allocated inside the embedded process task. When loading data into CAS, the compute threads transmit the records stored in the input buffers to the CAS worker nodes. When saving data to Hadoop, the compute threads receive data from CAS workers and stores it in the output buffers. Output buffers are given to the writer threads.
- Writer threads: these are Java threads responsible for writing records stored in the output buffers to an HDFS file or Hive table. Output buffers are stored outside if the JVM head space.

Figure 2 illustrates the main SAS Embedded Process components on Map Reduce.

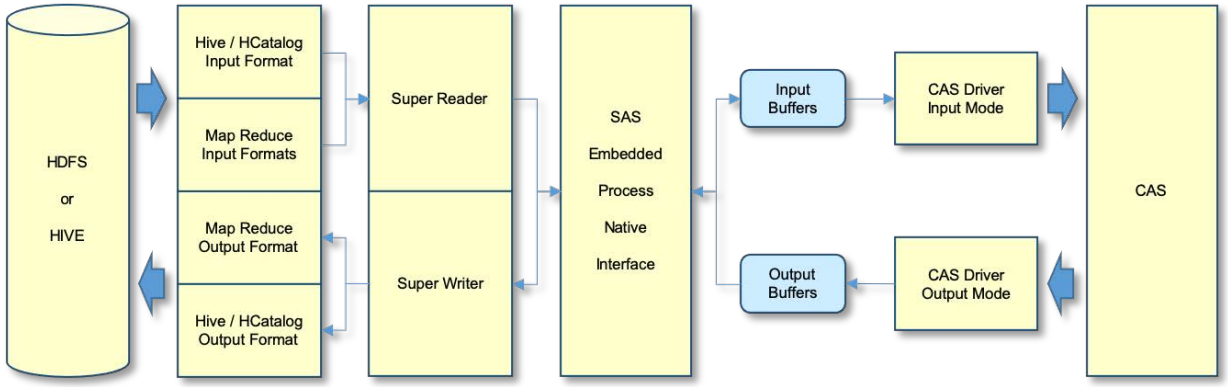

Figure 2. SAS Embedded Process Components on Map Reduce.

SAS EMBEDDED PROCESS CONFIGURATION PROPERTIES

The SAS Embedded Process installation process on the Hadoop cluster creates two identical configuration files, ep-config.xml and sasep-site.xml, and stores them under /opt/sas/ep/home/conf folder. The ep-config.xml is the global configuration file that gets copied to the default HDFS location /sas/ep/config. The sasep-site.xml is the client-side configuration file that can be stored under the same folder where all other client-side Hadoop configuration files reside. For example, the sasep-site.xml may be stored on the folder that is specified in the hadoopConfigDir parameter of the addCasLib action. When the sasep-site.xml file is found under the Hadoop configuration folder, the ep-config.xml file is not utilized. The default set of configuration properties defined in sasep-site.xml and ep-config.xml should not be changed. However, new properties may be added to either configuration files. When adding properties to the SAS Embedded Process configuration file, it is recommended to do so using the client-side sasep-site.xml file. Changes to the global ep-config.xml configuration file located on HDFS affects all SAS Embedded Process jobs. SAS Embedded Process configuration properties can also be added to the client-side mapred-site.xml file. Properties in the configuration file must be in the following format:

```
<property>
   <name>PROPERTY_NAME</name>
   <value>PROPERTY_VALUE</value>
```

```
</property>
```

There are a number of SAS Embedded Process properties that affect job execution behavior and performance:

- *sas.ep.superreader.tasks.per.node*: specifies the number of SAS Embedded Process Map Reduce tasks per node. During job submission, the SAS Embedded Process super reader technology calls into the underlying input format implementation to calculate the file input splits. In a standard Map Reduce job submission, each split is assigned to a map task. However, instead of assigning one split per task, the super reader groups the input splits and distributes them among the SAS Embedded Process tasks based on data locality. The default number of tasks per node is 6. When changing this property, the following should be taken into consideration:
    - the super reader does not control the number of nodes used to run a job. Tasks are assigned to all nodes where a physical file block can be found. For example: if a cluster contains 100 nodes, but file blocks can only be found on 50 of them, the super reader assigns tasks to 50 nodes.
    - decreasing the number of tasks per node increases the number of input splits per task, which reduces parallelism.
    - fewer tasks means better chance of having fewer tasks queued for execution.
    - increasing the number of tasks per node decreases the number of splits per task, which increase parallelism.
    - more tasks per node increase the chance of having more tasks queued for execution.
- *sas.ep.input.threads*: specifies the number of concurrent super reader threads per task. Input splits are queue for processing within a task. Each reader thread takes an input split from the input splits queue, opens the file, positions itself at the beginning of the split, and starts reading the records. Records are stored on a native buffer that is shared with the CAS driver container compute threads. When the native buffer is full, it is pushed to the CAS driver container for processing. When a reader thread finishes reading an input split, it takes another one from the input splits queue. The default number of input threads is 3.
- *sas.ep.compute.threads*: specifies the number of CAS driver container compute threads. Each compute thread runs one instance of the CAS driver. The CAS driver can operate in input or output mode. Input mode transmits data from the SAS Embedded Process to the CAS worker nodes. Output mode receives data from the CAS worker nodes and stores it in the output buffers. Data is transmitted in and out of the SAS Embedded Process by blocks of records. The number of records in a block depends on the length of a record and the size of the input and output buffers. The default number of compute threads is 1.
- *sas.ep.output.threads*: specifies the number of super writer threads writing data to the output file on HDFS or to a table in Hive. Super writer improves performance by writing output data in parallel, producing multiple parts of the output file per task. Each writer thread is responsible for writing one part of the output file. The default number of output threads is 2.
- *sas.ep.input.buffers*: specifies the number of native buffers that are used to cache input data. Input buffers are populated by the super reader input threads and consumed by the compute threads. The number of input buffers should not be less than sas.ep.compute.threads plus sas.ep.input.threads. The default number of input buffers is

4.

- *sas.ep.output.buffers*: specifies the number of native buffers that are used to cache output data. Output buffers are populated by the compute threads and consumed by the super writer output threads. The number of output buffers should not be less than sas.ep.compute.threads plus sas.ep.output.threads. The default number of output buffers is 3.
- *sas.ep.optimal.input.buffer.size.mb*: specifies the optimal size of one input buffer in mega-bytes (MB). The optimal number of records in a buffer is calculated based on the optimal buffer size and the maximum length of a record. The default value is 1.
- *sas.ep.hpa.output.concurrent.nodes*: specifies the number of concurrent nodes that are allowed to run CAS output tasks. If this property is set to zero, the SAS Embedded Process allocates tasks on all nodes capable of running a YARN container. If this property is set to –1, the number of concurrent nodes equates to the number of CAS worker nodes. If the number of concurrent nodes exceeds the number of available nodes, the property value is adjusted to the number of available nodes. The default value is 0.
- *sas.ep.hpa.output.tasks.per.node*: specifies the number of CAS output tasks per node. The default number of tasks per node is 1. If the total number of tasks for the entire job is less than the number of CAS worker nodes, the SAS Embedded Process allocates more tasks up to the number of CAS worker nodes.
- *sas.ep.max.memory*: specifies the maximum amount of native memory, in bytes, that the SAS Embedded Process native code is allowed to use. The amount of memory specified in this property does not supersede the YARN maximum memory per task. The default value is 0. Adjust the YARN container limit to change the amount of memory that the SAS Embedded Process is allowed to use.

LOADING AND SAVING DATA USING MAP REDUCE

In order to demonstrate the effects of the SAS Embedded Process configuration properties, let's run some jobs that load and save data into CAS using a few different configuration settings and analyze the differences between the executions. The test cases are executed on a CAS cluster with the following configuration:

- 1 CAS controller node.
- 4 CAS worker nodes.
- 256 GB of RAM per node.
- 32 Cores per node.

The Hadoop cluster is configured as follows:

- 1 HDFS Name Node/YARN Resource Manager/Hive Server.
- 4 YARN Node Manager nodes.
- 48 GB of RAM per node.
- 4 Cores per node.

The Hadoop and CAS clusters are set up for demonstration purpose only. Results may vary depending on cluster configuration and available resources. The comparison tables below show the results between the best and worst configuration settings. The test cases are executed using the following CAS library definition:

```
caslib hive
       datasource=(
```

```
 srctype="hadoop",
 server="hiveserver1.sas.com",
 hadoopJarPath="/opt/sas/viya/config/data/hadoop/lib",
 hadoopConfigDir="/opt/sas/viya/config/data/hadoop/conf",
 dataTransferMode="parallel"
);
```

Since Map Reduce is being used as the execution platform, the platform parameter may be omitted.

The input data is a Hive table stored using Parquet file input format. The table contains 400 numeric columns, 1,690,116 records, and 120 file blocks. Each file input split is exactly one file physical block. The following code is used to load the table into CAS memory using the SAS Embedded Process on Map Reduce:

```
proc cas;
  loadtable
  caslib="hive"
  path="numericdata400"
run; quit;
```

Table 1 shows the results of each job execution using different configuration settings.

MapRed	Configuration Properties			Execution Time in Seconds			Job Numbers		
Test Case	sas.ep.superreader.tasks.per.node	sas.ep.input.threads	sas.ep.compute.threads	CAS Action Time	Map Red Job Time	Avrg Task Time	Number of Tasks	Concurrent Tasks	Splits Per Task
Worst	140	1	1	120	110	14	120	19	1
Best	4	3	3	39	32	21	16	16	8

Table 1. Load Table into CAS Using Map Reduce.

On the first job, the number of tasks per node is set to 140. Since the input table contains 120 splits, the job runs with 120 tasks. Setting the number of tasks per node to a value greater than or equals to the total number of input split always results in one input split per task. Therefore, the number of input and compute threads defaults to 1. This would be the same as running a standard Map Reduce application where one task processes one input split. With this configuration setting, not all tasks run in parallel at the same time and there is a good amount of overhead imposed by the start and stop of multiple map tasks. The job finishes in 120 seconds. The time to get the Map Reduce job in running state is added to the CAS action execution time. That is the reason it is greater than the job execution time. Given the cluster configuration, the number of concurrent tasks observed is 19.

On the second jobs, the number of tasks per node is set to 4, which is the same number of cores per node. The job runs with 16 concurrent tasks. Super reader technology assigns 8 input splits per task, avoiding unnecessary start and stop of tasks every time a new input split is processed. There are 3 input reader threads per task, which allows each task to process 3 input splits at the same time. The number of compute threads is set to 3. The overall job execution is much better than the first job.

The following code saves the table that was just loaded into CAS to a Hive table in Hadoop using the SAS Embedded Process on Map Reduce:

```
proc cas;
   save
   caslib="hive"
   name="numericdata400out"
   replace=true
   table={name="numericdata400"};
run; quit;
```

Table 2 below shows the results of each job execution using different configuration settings.

MapRed	Configuration Properties		Execution Time in seconds			Job Numbers	
Test Case	sas.ep.hpa.output.concurrent.nodes	sas.ep.hpa.output.tasks.per.nodes	CAS Action Time	Job Time	Avrg Task Time	Number of Tasks	Concurrent Tasks
Worst	4	1	96	83	68	4	4
Best	4	4	66	49	38	16	16

Table 2. Save Table to Hive Using Map Reduce.

Since CAS and Hadoop clusters have the same number of nodes, the *sas.ep.hpa.concurrent.nodes* property is set to 4 on both test cases. There is a substantial difference between job execution times when comparing the first and second jobs. The second job uses 16 concurrent tasks instead of 4, which provides a much better throughput.

RUNNING SAS EMBEDDED PROCESS ON SPARK

Apache Spark has become one of the most popular platforms for distributed in-memory parallel processing. Apache Spark provides a combination of libraries that allows SQL, event streaming, machine learning, and graph processing operate seamlessly in the same application.

Apache Spark processes massive amounts of data stored on a cluster of commodity hardware providing an open-source parallel processing framework. Spark is developed for low cost, fast, and efficient massively parallelized data manipulation.

Apache Spark provides the ability to read HDFS files and query structured data from within a Spark application. With Spark SQL, data can be retrieved from a table stored in Hive using an SQL statement and the Spark Dataset API. Spark SQL provides ways to retrieve information about columns and their data types and supports the HiveQL syntax as well as Hive SerDe (Serializer and Deserializer).

SAS Embedded Process allows the parallel execution of SAS processes inside Spark. SAS Embedded Process on Spark is supported when Spark is running on YARN. In order to run SAS code in parallel inside Spark, SAS Embedded Process needs to be installed on every node of the cluster that can run a Spark task. All the computing resources used by SAS Embedded Process on Spark are completely manageable by YARN.

SAS Embedded Process on Spark consists of a Spark driver program that runs in the Spark application master container and a set of specialized functions that run in the Spark executor containers.

SAS Embedded Process is written mainly in C language where all the interactions with the CAS driver container happen. It is also written in Java and Scala where all the interactions with the Spark framework happen. The Java code is responsible for extracting data from Hive tables or Hadoop Distributed File System files and passing them to the CAS driver container. Scala code drives the whole SAS Embedded Process Spark application.

Both the C, Java, and Scala code run on the same Java Virtual Machine (JVM) process that is allocated by the Spark application master and executor containers. In order to eliminate multiple copies of the input data, Java and C code access shared memory buffers allocated by native code. In order to minimize Java garbage collections, the shared native buffers are allocated outside of the JVM heap space. Shared native memory allocations and CPU consumption are seen by the YARN resource management. Journaling messages generated by the SAS Embedded Process are written to the Spark standard output (*stdout*), standard error (*stderr*) and application log (*syslog*).

SAS Viya 3.5 introduces the new SAS Embedded Process for Spark continuous session (EPCS). EPCS is an instantiation of a long-lived SAS Embedded Process session on a cluster that can serve one CAS session. EPCS provides a tight integration between CAS and Spark by processing multiple execution requests without having to start and stop the SAS Embedded Process for Spark every time an execution request is made. Users can improve system performance by using the EPCS and the SAS Data Connector to Hadoop to perform multiple actions within the same CAS session.

A Spark application is a user-written program, called a driver program, that contains a set of actions and transformations applied to the data. The driver program, along with the SparkContext, coordinates the entire job execution. In a cluster configuration, the driver program runs on its own container.

Data in Spark is stored in a resilient and distributed collection of records or objects spread over one or more partitions called a Resilient Distributed Dataset (RDD). An RDD is resilient because it is capable of re-computing missing or damaged partitions. It is distributed because data resides on multiple nodes of a cluster. It is also a dataset, a collection of partitioned data with primitive values, tuples, or other objects. In summary, RDD is Spark's primary data abstraction.

A Spark application consists of one or more jobs, for example, actions or transformations applied to RDDs. A transformation is a function applied to an RDD that produces another RDD. Examples of transformations are map, filter, group, and sort functions. An action is a function that produces a non-RDD value. For example, a function that returns a Boolean or Integer value, or even a void function.

A Spark job is a parallel computation consisting of one or multiple stages. A stage is a computational boundary that consists of a set of tasks that depend on each other. Tasks are executed in response to actions. A task is the single unit of work, scheduled by the driver, that runs in the Spark executor container and operates on a partition of an RDD.

Spark executors are distributed processes running on many nodes of a cluster. Executors are responsible for executing tasks and providing in-memory storage for RDDs. Tasks can be executed concurrently or sequentially inside an executor. An executor process communicates directly with the driver program container in order to make itself available to execute tasks.

When EPCS is launched, a driver container and executor containers are allocated on the cluster. The driver container is the process where the EPCS driver program runs.

The EPCS driver receives execution requests from the CAS controller node. An execution request is a generated Scala program capable of performing specialized SAS Embedded Process Spark functions, such as running scoring code, or loading and saving a table. The Scala program is interpreted and compiled on-the-fly by the Scala Interpreter and Compiler component and put into execution.

The SAS Embedded Process functions run in the executor container. Each task is assigned an RDD partition, which might come from a Hive table, an HDFS file, or data transmitted from a CAS worker node. Figure 3 depicts the SAS Embedded Process components running in the executor container.

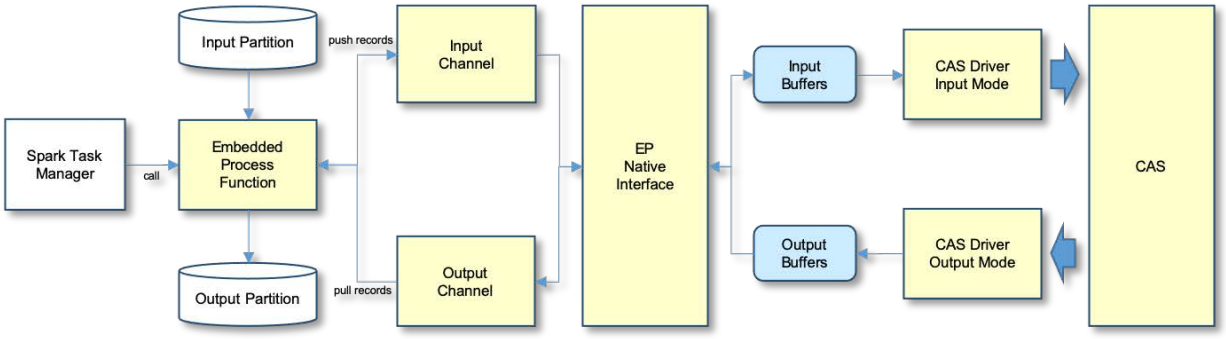

Figure 3. SAS Embedded Process Executor Container Components

Depending on the execution request made by the CAS action (save or loadTable), the embedded process function running in the executor container creates an input or output channel. When loading a file or table into CAS, the function retrieves records from the input partition and pushes them to the CAS driver for data transmission to the CAS worker nodes. When saving a file or table, the function receives records from the CAS driver and writes them to a file on HDFS or to a Hive table.

SAS EMBEDDED PROCESS FOR SPARK ACTION SET

The SAS Embedded Process for Spark action set (sparkEmbeddedProcess) enables the start and stop of the SAS Embedded Process for Spark continuous session. If the EPCS is not started before a CAS action is submitted for execution, a SAS Embedded Process for Spark application is automatically started. However, the Spark application is terminated right after the CAS action is executed.

The SAS Embedded Process for Spark action set consists of the following actions:

- startSparkEP: starts the SAS Embedded Process for Spark continuous session.
- stopSparkEP: stops the SAS Embedded Process for Spark continuous session.

Detailed information about the action set and action parameters can be found in the SAS Visual Analytics: Programming Guide.

STARTSPARKEP ACTION

Starts the SAS Spark Embedded Process for Spark continuous session on a particular cluster. The action syntax is as follows:

```
sparkEmbeddedProcess.startSparkEP
    caslib="string"
    classpath="string"
    configpath="string"
    customJar="string"
    executorCores=integer
    executorInstances=integer
    executorMemory=integer
    password="string"
    properties={"string-1" <, "string-2", ...>}
    taskCores=integer
    timeout=integer
    trace=TRUE | FALSE
    username="string"
    ;
```

The startSparkEP action may work in conjunction with a CAS library definition. When the caslib parameter is specified, the following parameters may be suppressed: classpath, configpath, username, and password. The parameters executorCores, executorInstances, executorMemory, and taskCores are optional. If not specified, the defaults specified in the Spark configuration file (*spark-defaults.conf* that resides under the Hadoop configuration folder on the CAS controller node) are taken.

The following example shows how to start the SAS Embedded Process for Spark continuous session using a CAS library definition:

```
caslib hive
       datasource=(
           srctype="hadoop",
           server=&hiveservername.,
           schema="default",
           hadoopJarPath=&HADDOPJARPARTH.,
           hadoopConfigDir=&HADOOPCONFIGPATH.,
           platform="spark",
           dtm="parallel"
         );

proc cas;
  sparkEmbeddedProcess.startsparkep caslib="hive",
          executorInstances=4, executorCores=4;
run; quit;
```

STOPSPARKEP ACTION

Stops the SAS Spark Embedded Process for Spark continuous session on a particular cluster. EPCS is automatically terminated when the CAS session associated with it is terminated. The action syntax is as follows:

```
sparkEmbeddedProcess.stopSparkEP
    caslib="string"
    classpath="string"
    configpath="string"
    password="string"
    username="string"
    ;
```

When the caslib parameter is specified, the following parameters may be suppressed: classpath, configpath, username, and password. The following example shows how to stop the SAS Embedded Process for Spark continuous session:

```
proc cas;
  sparkEmbeddedProcess.stopsparkep caslib="hive";
run; quit;
```

LOADING AND SAVING DATA USING EPCS

In this example, the same set of test cases executed using the SAS Embedded Process on Map Reduce is now executed using the EPCS. The input is a Hive table stored using Parquet file input format, with 400 numeric columns and 1,690,116 records. The Hadoop cluster is configured as follows:

- 1 HDFS Name Node/YARN Resource Manager/Hive Server.
- 4 YARN Node Manager nodes.
- 48 GB of RAM per node.

- 4 Cores per node.
- YARN is configured to allow 4 virtual cores and 32GB of RAM per container.

The number of cores and the amount of memory per executor container are the key resources Spark and YARN take into consideration when running applications. There are Spark resource management properties that can be used to control the application execution. Some of the properties can be set when EPCS is started. Additional Spark configuration properties can be set in the *spark-default.conf* file that is stored under the Hadoop configuration folder on the CAS controller node. Here are some of the key Spark resource management properties:

- *spark.executor.instances*: specifies the number of executors allocated per application.
- *spark.executor.memory*: specifies the amount of memory used by an executor process.
- *spark.executor.cores*: specifies the number of cores used per executor process.
- *spark.task.cpus*: specifies number of cores allocated for each task. The default value is 1. This property does not need to be changed when running EPCS.

All executors run with the same number of cores and amount of memory. The degree of parallelism depends on the number of executors, the number of cores per executor and the number of cores per task. The number of tasks depends on the number of partitions in the input RDD. Each partition is processed by one task. The number of concurrent tasks in an executor is the number of cores per executor divided by the number of cores per task, as follows:

```
executorConcurrentTasks = spark.executor.cores / spark.task.cpus
```

Therefore, the total number of concurrent tasks is the number of executors multiplied by the number of concurrent tasks per executor, as follows:

```
totalConcurrentTasks = spark.executor.instances * executorConcurrentTasks
```

The Spark application resource allocation must be within the limits imposed by YARN. For example, if YARN only allows 4 cores per container, asking for more than that may result in EPCS start-up failure due to resource constraints.

The resources consumed by the Spark driver container also takes up resources from YARN. This should be taken into account when sizing the application resources. Two important Spark driver properties to take into consideration are:

- *spark.driver.cores*: specifies the number of cores used by the driver container. Default value is 1. This property does not need to be changed when running EPCS.
- *spark.driver.memory*: specifies the amount of memory used by the driver container. Default value is 1GB. The recommended driver container memory with EPCS is at least 2GB.

The test cases are executed using the following CAS Library definition:

```
caslib hive
       datasource=(
           srctype="hadoop",
           server=”hiveserver1.sas.com”,
           hadoopJarPath=”/opt/sas/viya/config/data/hadoop/lib”,
           hadoopConfigDir=”/opt/sas/viya/config/data/hadoop/conf”,
           platform=”spark”,
           dataTransferMode="parallel"
          );
```

The platform parameter is used in the CAS library definition to indicate the jobs are executed using Spark. The next step is to start the SAS Embedded Process for Spark continuous session. If EPCS is not started before a loadTable or save table action is executed, the SAS Embedded Process Spark application is started and stopped for each action.

The following example starts EPCS using configuration settings that yield better performance without allocating 100% of resources to YARN containers. Based on the cluster configuration used to run the test cases, the number of executor instances is set to 11. Since the cluster has 4 Node Manager nodes, 3 of them runs with 3 executor containers, while 1 runs with 2 executor containers, leaving 1 slot to the driver container. This configuration allows 3 concurrent tasks per executor, which allows a total of 33 concurrent tasks per stage:

```
proc cas;
  sparkEmbeddedProcess.startsparkep
  caslib="hive",
  executorInstances=11,
  executorCores=3,
  properties={"spark.driver.memory=2g"};
run; quit;
```

The EPCS instance is associated with the current CAS session. When the CAS session is terminated, EPCS is terminated. EPCS can also be terminated at any moment by running the stopSparkEP action, as shown in the code below:

```
proc cas;
  sparkEmbeddedProcess.stopsparkep caslib="hive";
run; quit;
```

EPCS can also timeout if the timeout parameter is used when EPCS is started. For example:

```
proc cas;
  sparkEmbeddedProcess.startsparkep
  caslib="hive",
  timeout=60, /* number of seconds */
  executorInstances=11,
  executorCores=3,
  properties={"spark.driver.memory=2g"};
run; quit;
```

EPCS must be stopped and started when experimenting with different configuration options.

The following code loads the table into CAS memory. Optionally, the platform can be specified using the dataSourceOptions parameter in the loadTable action, as shown here:

```
proc cas;
  loadtable
  caslib="hive"
  path="numericdata400"
  dataSourceOptions={platform="spark"};
run; quit;
```

Table 3 shows the results of each Spark job execution using different resource allocations.

SPARK	Configuration Properties			Execution Time in Seconds			Job Numbers	
Test Case	Executors	Executor Cores	Task Cores	CAS Action Time	Spark Job Time	Avrg Task Time	Number of Tasks	Concurrent Tasks
Worst	4	1	1	43	40	38	4	4
Best	11	3	1	17	15	12	30	30

Table 3. Load Table into CAS Using EPCS.

The first job uses the basic minimal resources, which allocates 4 executor containers with 1 core each, allowing only 1 task per executor. As seen on the second job, allowing more executors and more cores per executor unleashes the parallel processing power offered by the platform, which yields better throughput.

The following code saves the CAS table to Hive using the Parquet file format. The platform parameter in the data source options is optional, since it was already defined in the CAS library.

```
proc cas;
  save
  caslib="hive"
  name="numericdata400out"
  replace=true
  table={name="numericdata400"}
  dataSourceOptions={platform="spark",
                     dbCreateTableOpts="stored as parquet"};
run; quit;
```

By default, the number of tasks allocated to the output job is the number of cores assigned to an executor. The default value may be overwritten by starting EPCS with the following property:

spark.sas.ep.hpa.output.tasks.per.executor: specifies the number of tasks that are assigned to an executor. For example:

```
proc cas;
  sparkEmbeddedProcess.startsparkep caslib="hive",
     executorInstances=11,
     executorCores=3,
     properties={"spark.driver.memory=2g",
                 "spark.sas.ep.hpa.output.tasks.per.executor=4"};
run; quit;
```

Table 4 shows the results of each Spark job execution using different resource allocations.

SPARK	Configuration Properties			Execution Time in Seconds			Job Numbers	
Test Case	Executors	Executor Cores	Task Cores	CAS Action Time	Spark Job Time	Avrg Task Time	Number of Tasks	Concurrent Tasks
Worst	4	1	1	90	87	78	4	4
Best	11	3	1	42	38	29	33	33

Table 4. Save Table to Hive Using EPCS.

As expected, more processing power yields better response time. One observation regarding CAS output jobs is that the number of tasks cannot be less than the number of CAS worker nodes. In order to avoid running into this situation, the number of tasks is automatically adjusted to match the number of CAS worker nodes.

CONCLUSION

Data is of the essence. Data is everywhere. Data is moving. Data needs storage. Data needs processing. Data needs SAS.

SAS Cloud Analytic Server and SAS Embedded Process were built to handle massive amounts of data. Not only in terms of parallel processing power, but also in terms of parallel data transfer.

Whether you are moving the data to the process or the process to the data, finding the perfect configuration settings for best performance is not an easy task, especially when combining different execution platforms, such as CAS, Spark and Map Reduce. A clear understanding of both worlds is a crucial step towards achieving the best results. You need to take into consideration the number of nodes on the CAS and Hadoop clusters, maximum number of concurrent containers allowed by YARN, size of the data, number of file splits or table partitions, and number of cores and memory available on each node of the cluster.

This paper has prepared you to exercise the different configuration settings offered by the SAS Embedded Process and achieve the best parallel data transfer result. It has explained the internal components of the SAS Embedded Process running on Map Reduce and Spark. It introduced the SAS Embedded Process for Spark continuous session in SAS Viya 3.5, which brings modernization and performance improvements to SAS Viya and SAS In-Database Technologies.

REFERENCES

Kumar, Uttam. “Multi Node Data Transfer to CAS”. Available https://blogs.sas.com/content/sgf/2018/03/29/multi-node-data-transfer-to-cas/.

Collum, Rob. 2019. “Seriously Serial or Perfectly Parallel Data Transfer with SAS Viya” *Proceedings of the SAS Global Forum 2019 Conference*, Cary, NC: SAS Institute Inc. Available https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3479-2019.pdf.

Ghazaleh, David. 2016. “Exploring SAS Embedded Process Technologies on Hadoop” *Proceedings of the SAS Global Forum 2016 Conference*, Cary, NC: SAS Institute Inc. Available http://support.sas.com/resources/papers/proceedings16/SAS5060-2016.pdf.

Ghazaleh, David. 2019. “Execution of User-Written DS2 programs inside Apache Spark using SAS In-Database Code Accelerator” *Proceedings of the SAS Global Forum 2019 Conference*, Cary, NC: SAS Institute Inc. Available https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3116-2019.pdf.

SAS Institute Inc. 2019. “SAS Embedded Process for Spark Action Set.” In *SAS Visual Analytics 8.5: Programming Guide.* Cary, NC: SAS Institute Inc. Available http://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=casanpg&docsetTarget=titlepage.htm&locale=en.

CONTACT INFORMATION

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

David Ghazaleh
SAS Institute, Inc.
(919) 531-7416
david.ghazaleh@sas.com