SAS751-2017

Understanding SAS® Embedded Process with Hadoop® Security

David Ghazaleh, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® Embedded Process enables user-written DS2 code and scoring models to run inside Hadoop. It taps into the massively parallel processing (MPP) architecture of Hadoop for scalable performance. SAS Embedded Process explores and complies with many Hadoop components. This paper explains how SAS Embedded Process interacts with the existing Hadoop security technology, such as Apache Sentry and RecordService.

INTRODUCTION

Companies are substantially storing sensitive data in Hadoop. Securing access to the data has become imperative. Hadoop provides different levels of security that prevent unauthorized access to the cluster and the data stored in it. This essay concentrates on two main levels of security: authentication and authorization. Not covered within this work are other levels of security, such as data confidentiality and service-level authorization.

This paper gives an overview of Hadoop security components and provides examples of how to define certain levels of permission in Hive using Apache Sentry. Furthermore, this analysis does not cover details about the installation and configuration of Hadoop security and its components.

As of the second quarter of 2017, SAS Embedded Process-based products for Hadoop do not offer support for Apache Sentry RecordService. While there is no precise timetable of when this support will be added, this report offers a technology preview of how the SAS Embedded Process might work with Apache Sentry RecordService in a future release.

AUTHENTICATION WITH KERBEROS

Authentication is the process of authenticating the veracity of someone's identity. The authentication process is usually based on a user identification code and the password associated with it. Once the user identification code is verified against the password, the user is deemed authenticated.

Kerberos is a key component that provides authentication level security. Many Hadoop distributors use Kerberos on their security solution projects as part of their authentication process. Examples of such projects include Apache Sentry (adopted by Cloudera® and now a Top-Level Apache project) and Apache Ranger (adopted by Hortonworks®).

Kerberos is an authentication protocol created by the Massachusetts Institute of Technology (MIT) to solve network security problems. MIT defines Kerberos as follows: "*The Kerberos protocol uses strong cryptography so that a client can prove its identity to a server (and vice versa) across an insecure network connection. After a client and server has used Kerberos to prove their identity, they can also encrypt all of their communications to assure privacy and data integrity as they go about their business* (Kerberos: The Network Authentication Protocol 2016)."

KERBEROS COMPONENTS

The Key Distribution Center (KDC) is the authentication server in a Kerberos environment. KDC uses ticket distribution functions to provide access to services. KDC can be logically divided into three main components: Database, Authentication Server (AS) and Ticket Granting Server (TGS).

A Kerberos ticket is issued by the AS and contains information that confirms the user's identity. The Database stores entries associated with users and services. The name of an entry in the Database is called a principal. Entries in the Database contain information such as the principal name, encryption key, duration of a ticket, ticket renewal information, a flag characterizing the ticket's behavior, password expiration and principal expiration.

The AS authenticates a user based on its user name and password. Once authenticated, the AS issues a Ticket Granting Ticket (TGT) to the user. The user can use its TGT to obtain access to other services without having to enter its password again. The TGS distributes service tickets to users with a valid TGT.

Figure 1 illustrates the Kerberos main components and the steps to obtain a service ticket to access an application server.

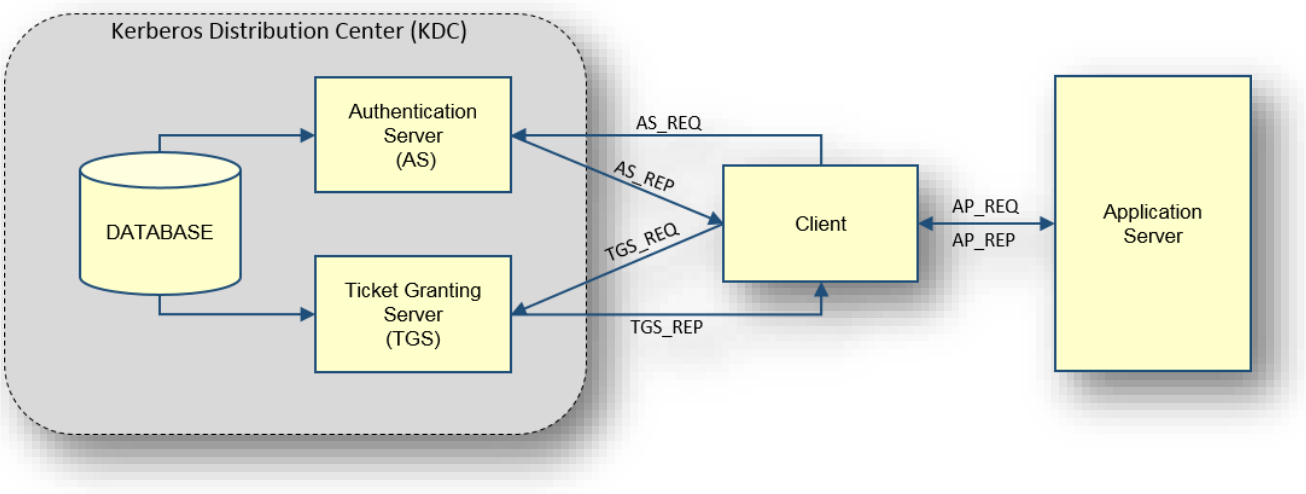

Figure 1 Kerberos Authentication Process (Source: kerberos.org)

The user initiates the authentication process by contacting the AS. Kerberos provides a tool called *kinit* to send a request to the AS identified as AS_REQ, as in Figure 1. Once the user has been validated, the AS provides a TGT in the response AS_REP. The user sends a TGS_REQ request with its TGT to the TGS requesting a service ticket. Subsequently, the TGS responds with TGS_REP, that contains the requested service ticket. The user then sends the service ticket in AP_REQ seeking access to the application server. When mutual authentication is required, the application server sends AP_REP, proving it is the server the client is expecting.

SAS does not directly interact with Kerberos. SAS relies on the underlying operating system and Application Programming Interfaces (APIs) to handle requesting tickets, manage ticket caches, and authenticate users. Servers hosting SAS components must be integrated into the Kerberos realm and configured for the secure Hadoop environment. This involves configuring the operating system's authentication processes to use either the same KDC as the secure Hadoop environment or a KDC with a trust relationship to the secure Hadoop environment.

Kerberos provides a robust authentication service that can be now used by the authentication layer.

AUTHORIZATION WITH APACHE SENTRY

Authentication is the process of allowing or denying someone's access to computer resources based on predefined permissions and rules. Definition of permissions and rules of access is also part of the authentication process.

Apache Sentry is a software layer that enforces fine-grained, role-based authorization to data, objects, and metadata stored on a Hadoop cluster. Apache Sentry is Remote Procedure Call (RPC) based server that stores authorization metadata. Its RPC interfaces are used to retrieve and manipulate privileges. Apache Sentry uses Kerberos authentication in order to securely access Hadoop services. Apache Sentry works with Apache Hive, Hive Metastore/HCatalog, Apache Solr, Impala, and Hive table data stored on Hadoop File System (HDFS).

Apache Sentry relies on seven key security concepts:

- **User**: Individual identified and validated by the authentication service.

- **Group**: Set of users with same privileges known and maintained by the authentication service. Existence and identification of users and groups are enforced by the authentication system.
- **Authentication**: Identification of a user based on the underlying authentication service, such as Kerberos or Lightweight Directory Access Protocol (LDAP).
- **Authorization**: Restriction of the user's access to resources.
- **Privilege**: Rule that allows the user's access to an entity protected by authorization rules. Examples of entities are objects such as servers, databases, tables, or HDFS Uniform Resource Identifier (URI).
- **Role**: Set of privileges or a combination of multiple access rules. Apache Sentry uses Role Based Access Control to manage authorization for a large number of users and data objects.
- **Authorization models**: Object subject to authorization rules and the definition of actions allowed on the object. For example, in the Structured Query Language (SQL), a table is an object and the actions allowed on the table are SELECT, INSERT, or CREATE – together these components make up the authorization model.

Figure 2 depicts Apache Sentry integration with the Hadoop Ecosystem.

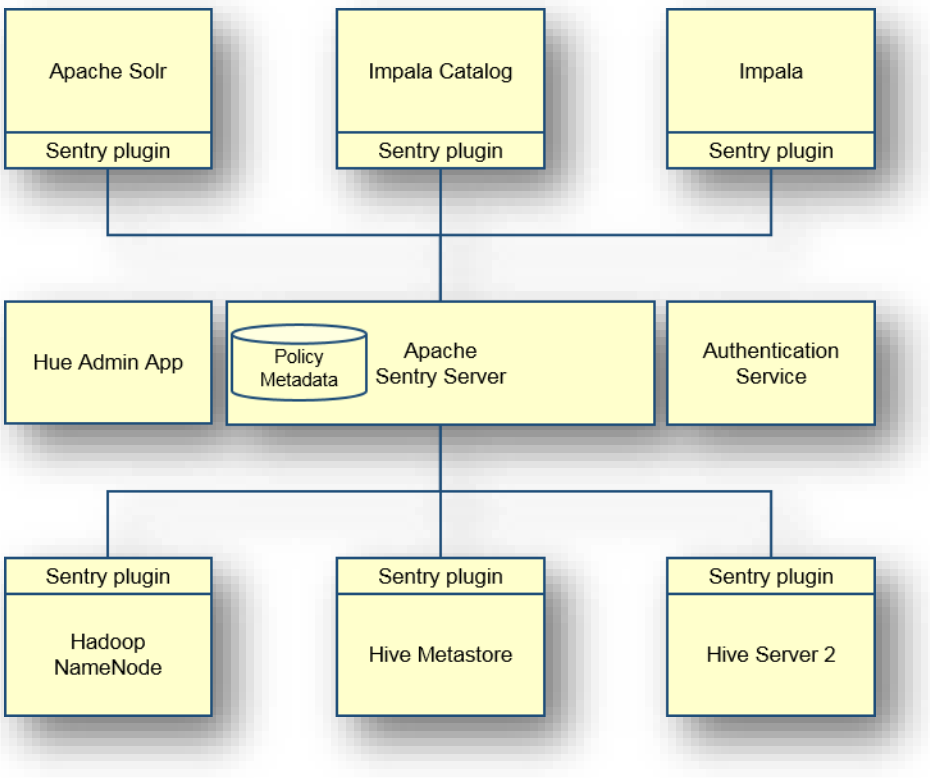

Figure 2 Apache Sentry Integration with the Hadoop Ecosystem

Hadoop services, when configured to use Apache Sentry, act as its client. Apache Sentry provides authorization metadata, and the client Hadoop service provides privilege enforcement. Apache Sentry applies authorization roles while the Hadoop service allows or denies access to its resources to a given user or application.

Data engines, such as Hive, Impala and Hadoop HDFS, provide access to the data. The Apache Sentry Plug-in runs inside each data engine. The plug-in interfaces allow the manipulation of authorization metadata stored in the Apache Sentry Server. The plug-in authorization policy engine uses the authorization metadata to validate access requests made by the data engines.

The authorization layer has now become a “must have” protection asset on Hadoop clusters.

ACCESSING DATA THROUGH APACHE SENTRY RECORDSERVICE

RecordService is part of Apache Sentry. It is a fine-grained, security enforcement layer that sits between the data engines and the compute services. RecordService provides a unified data access path to applications while imposing row- and column-level security. Applications running in Spark, MapReduce, Hive, or Impala gain access to the data through RecordService. RecordService promises better performance since it uses the Impala I/O subsystem to access data on the file system. RecordService allows applications to operate independently of the storage format. Figure 3 illustrates the RecordService architecture.

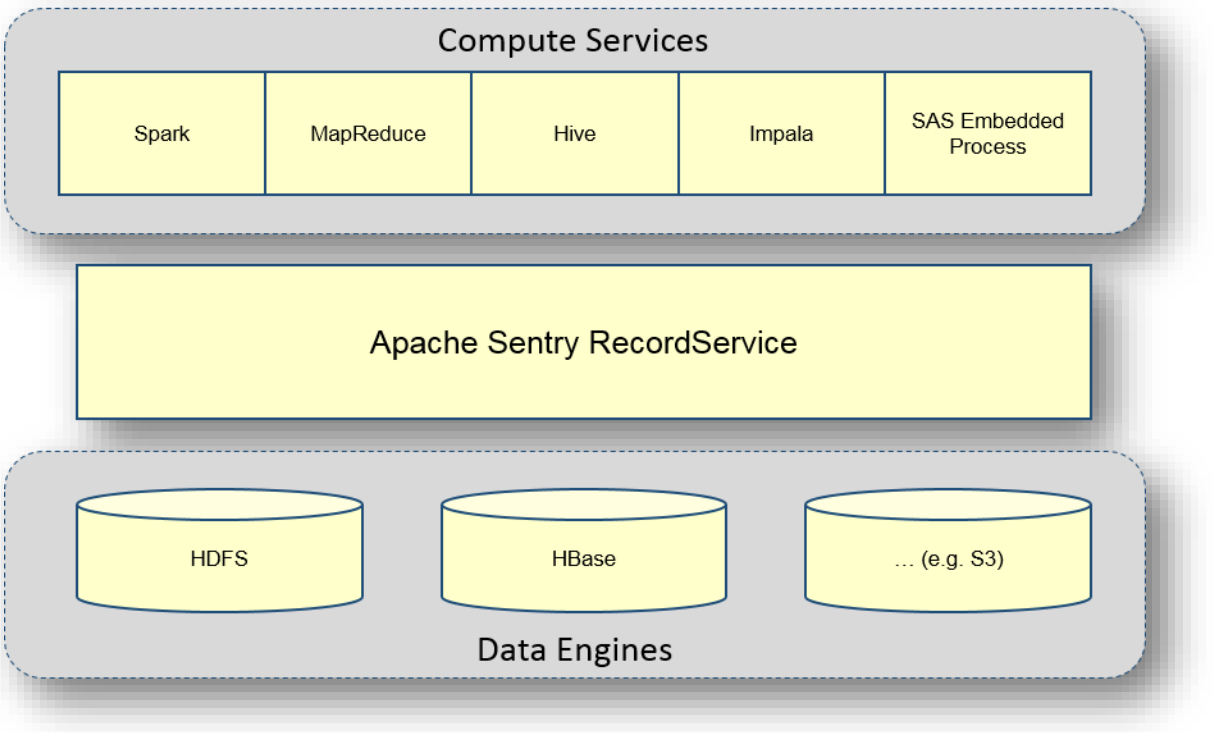

Figure 3 Apache Sentry RecordService Architecture

RecordService provides an API that implements the common Hadoop InputFormats. Existing applications might have to be modified in order to use RecordService as a unified data access path.

RecordService provides two main services: RecordService Planner and RecordService Worker.

The Planner is responsible for authorization checks, metadata access, and task creation. Tasks execute the user’s application. Task creation is very similar to MapReduce split generation. The number of tasks created is based on the number of file input splits. Each task is assigned a preferred node in the Hadoop cluster where it is executed. Task node assignment is based on data locality.

The Workers execute tasks and read/write to the storage. Workers return reconstructed, filtered records in a canonical format. The worker uses Impala I/O scheduler and file parsers, providing high-performance data access.

A good way to think about the system is that the Planners provide a layer of abstraction over the metadata services (Hadoop NameNode, Hive Metastore, Apache Sentry server), and the Workers provide a layer of abstraction of the data stores (Hadoop DataNode).

Figure 4 depicts RecordService components and execution environment.

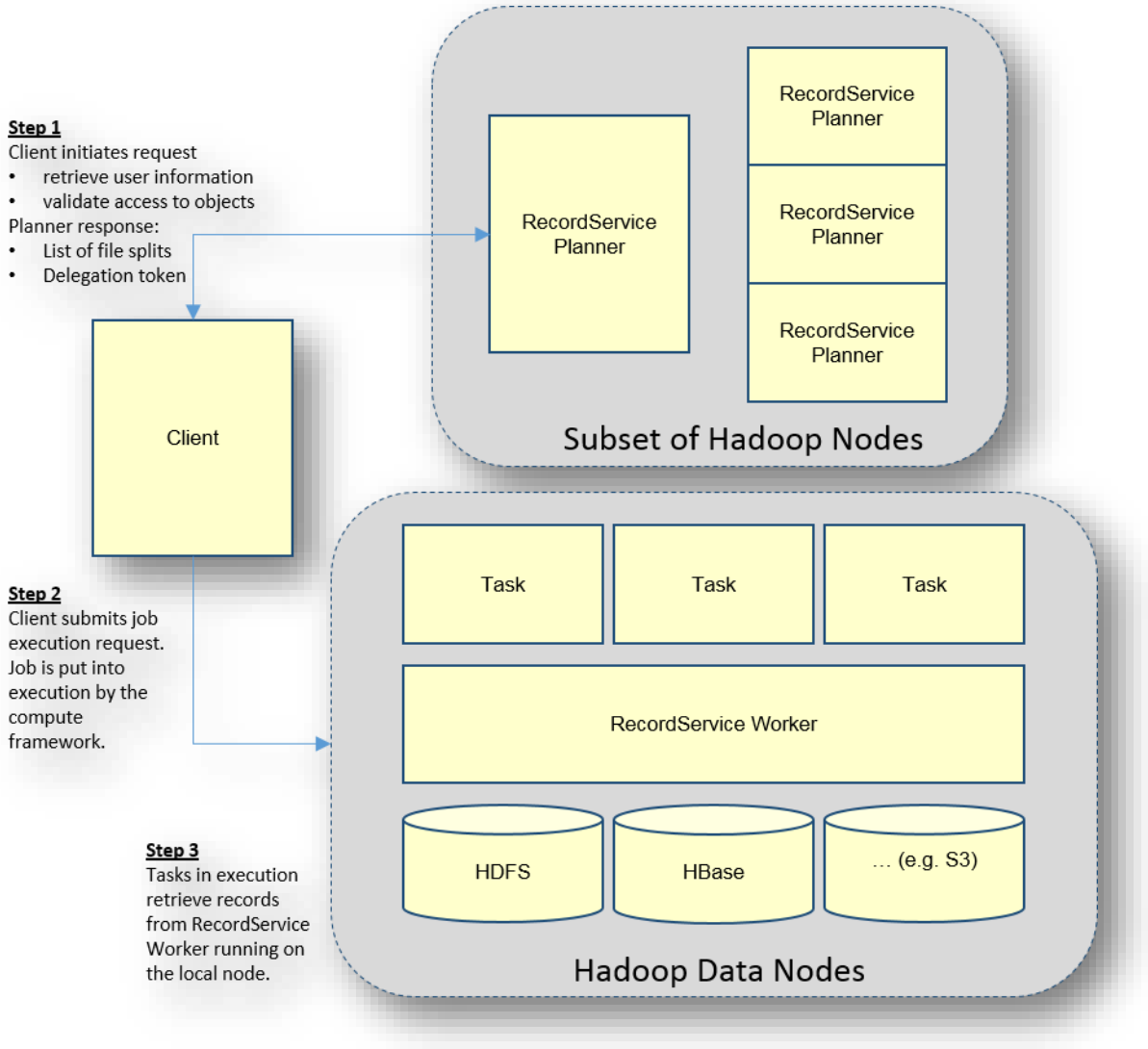

Figure 4 Apache Sentry RecordService Components and Execution Environment

During job submission, the client application calls into the RecordService planner in order to validate access to the data and to retrieve a list of file splits and create tasks. Tasks are then submitted for execution. An execution framework, such as YARN or MapReduce, controls task execution.

The task calls the RecordService Worker in order to retrieve records. On a typical Hadoop cluster, the Planner runs on a few, designated nodes. The Workers run on all nodes that contain data, in order to optimize data access based on data locality.

RecordService is designed to enforce fine-grained security while providing the main channel for high-performance data access. In addition, RecordService simplifies application development; using its API, application developers do not need to worry about the low-level details of file formats and underlying data engines-API.

SAS EMBEDDED PROCESS ON HADOOP

SAS Embedded Process is the core of SAS in-database products. It allows the parallel execution of SAS processes inside Hadoop and many other databases, such as Teradata®. SAS Embedded Process is a lightweight execution container for specialized DS2 code that makes SAS portable and deployable on a

variety of platforms. SAS Embedded Process runs inside a MapReduce task. Its job is orchestrated by Hadoop MapReduce framework while YARN manages load balancing and resource allocation.

SAS Embedded Process is a subset of Base SAS software that is sufficient to support 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 Hadoop.

Figure 5 depicts the SAS Embedded Process internal architectural components.

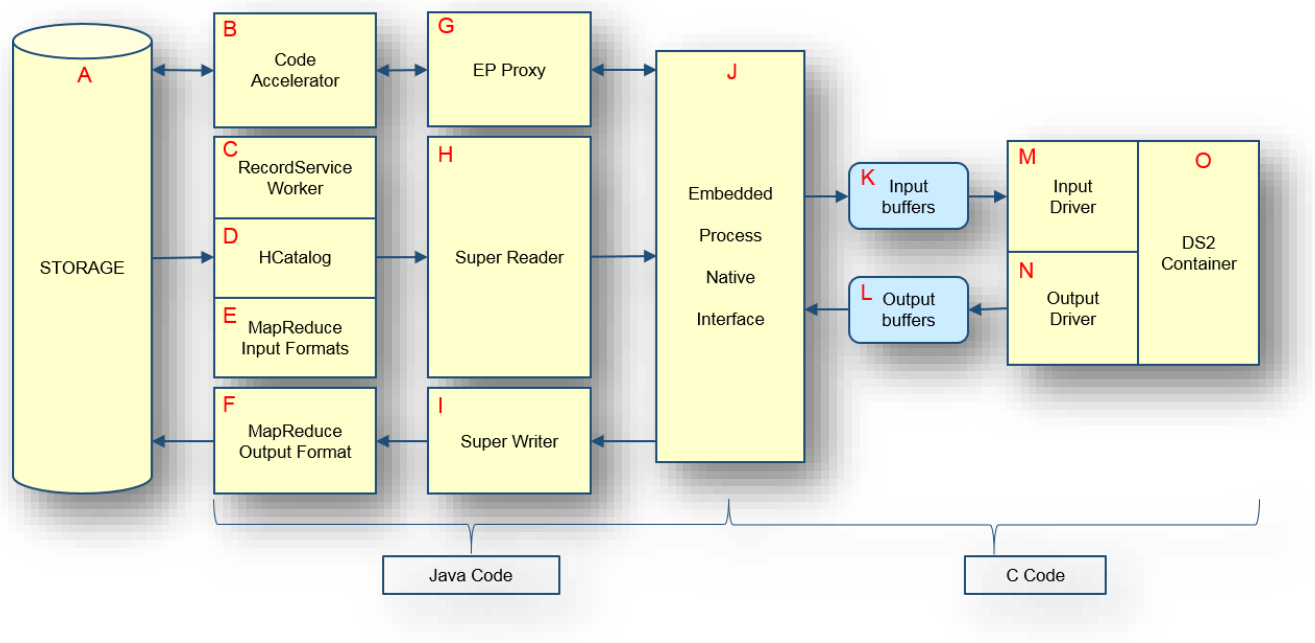

Figure 5 SAS Embedded Process Components

SAS In-Database Code Accelerator (**B**) is a full-fledged MapReduce application that engages SAS Embedded Process through a specialized interface called Embedded Process (EP) Proxy (**G**). The EP Proxy is responsible for pushing and pulling data to and from SAS Embedded Process Native Interface (**J**). SAS Embedded Process integrates seamlessly with Apache Sentry RecordService Worker (**C**) to retrieve record from files on storages (**A**) that are supported by RecordService. The HCatalog Reader (**D**) is capable of reading files that have a *SerDe* (Serializer / Deserializer) registered in Hive. HCatalog Reader enables one to read data from special format files, such as Avro, Parquet, ORC, and RCFile. The MapReduce Input Formats (**E**) retrieve data directly from files stored on HDFS using the SAS Embedded Process default readers, such as delimited text and fixed-record-length binary file.

Super Reader (**H**) initiates access to files via a multi-threaded and multi-split reader framework. The MapReduce Output Formats (**F**) are responsible for writing DS2 output data directly to HDFS. Super Writer (**I**) is a multi-threaded writer framework that writes data back to storage.

SAS Embedded Process Native Interface (**J**) is the communication channel between Java and C code. Data read from input files are stored directly into the native INPUT buffers (**K**). The DS2 program runs in the DS2 container (**O**) and retrieves input data through the input driver (**M**). The DS2 program operates like a User-Defined Table Function (UDTF). It takes a block of records at a time and processes them. At some point in time, DS2 creates an output result-set and makes it available for consumption. Output data generated by the DS2 program is stored in the native OUTPUT buffers (**L**) through the output driver (**N**).

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

- **Reader threads**: Java threads responsible for reading input splits and filling up input buffers.
- **Compute threads**: C threads responsible for the execution of the DS2 program.
- **Writer threads**: Java threads responsible for emptying output buffers and writing data to HDFS.

Together, these components work simultaneously with the SAS Embedded Process to make SAS portable and deployable on a variety of platforms in order to provide a secure and efficient parallel execution environment.

SCORING ON HADOOP USING SAS EMBEDDED PROCESS AND RECORDSERVICE

The technology that enables the execution of score code in a parallel manner inside Hadoop is the SAS Scoring Accelerator. Models are created in SAS® Enterprise Miner or SAS® Factory Miner. The scoring function is exported from SAS Enterprise Miner and the scoring model code (*score.sas*) is created along with its associated scoring files: a property file that contains model inputs and outputs (*score.xml*) or an analytic store file, and a catalog of user-defined formats. A publishing mechanism offered by the %INDHD_PUBLISH_MODEL macro publishes the model to Hadoop. Once published, the model is ready to be executed. The execution of scoring model in Hadoop is started by the %INDHD_RUN_MODEL macro.

The following example uses the %INDHD_RUN_MODEL macro to demonstrate how to score in Hadoop using SAS Embedded Process and Apache Sentry RecordService. Before running a model inside Hadoop, the scoring files must be published and the input table must be created in Hive. The model in the example reads rows from a Hive table called *DGCARS*. The rows in the table contain attributes describing a particular car. Make, model, cylinders, and engine size are examples of attributes of cars. A simple input/output score program is used to read rows from table *DGCARS* and generate an output file.

The example is executed from a Linux client machine that is already configured to use Kerberos authentication. The Hadoop cluster is also configured for Kerberos authentication and Apache Sentry authorization. All necessary installation and configuration steps to secure the cluster have been taken.

CONFIGURING APACHE SENTRY PERMISSIONS IN HIVE

The user connecting to the secured Hadoop cluster must be a valid user on all nodes of the cluster. The user must also belong to one or more operating-system groups. Apache Sentry is not an authorization system; therefore, it does not enforce the existence of the user and groups. In Apache Sentry, groups have multiple users, and roles have multiple privileges. Roles are assigned to groups, and privileges are assigned to roles. Apache Sentry does support privilege assignment directly to users. Figure 6 illustrates Apache Sentry relations.

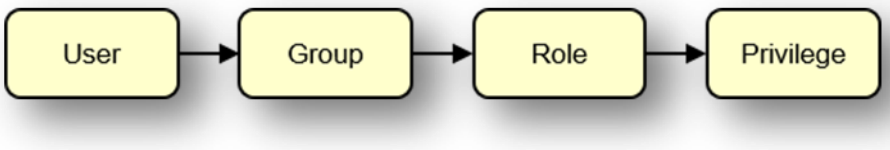

Figure 6 Apache Sentry Relations

Hive provides a command line interface called *beeline* that can be used to define the security roles. Permissions are configured from the *beeline* command line using GRANT and REVOKE SQL statements. The *beeline* command line interface is executed from one of the nodes of the Hadoop cluster. The user granting and revoking privileges must have administrative permissions in Apache Sentry.

The following is an example of steps that use the *beeline* interface to define security roles.

STEP 1. Before running the *beeline* command line interface, a Kerberos ticket must be acquired via *kinit*. The user *sasdemo* is an administrative user in Apache Sentry.

```
#> kinit sasdemo@KRB.SGF.COM
```

STEP 2. List Kerberos credentials. Kerberos provides a command to list the Kerberos principal and Kerberos ticket stored in the credential cache.

```
#> klist
Ticket cache: FILE:/tmp/krb5cc_1101
Default principal: sasdemo@KRB.SGF.COM

Valid    starting  Expires           Service principal
02/02/17 16:40:47  02/03/17 16:40:47 krbtgt/KRB.SGF.COM@KRB.SGF.COM
renew until 02/09/17 16:40:47
```

STEP 3. Start the *beeline* command line interface. At the *beeline* prompt, get a connection to the database *SASEP* by using the CONNECT command.

```
!connect jdbc:hive2://caesar:10000/sasep;principal=hive/_HOST@KRB.SGF.COM
```

STEP 4. List all the tables created in the database SASEP.

```
show tables;

+-------------+
|  tab_name   |
+-------------+
| dgcars      |
| dgcarsavro  |
| dgcarsorc   |
| dgcarsprqt  |
+-------------+
```

STEP 5. Define the authorization for group *sasep*. The group *sasep* is a valid operating system group on all nodes of the cluster. User *daghaz* is a valid user on all nodes of the cluster and it belongs to the *sasep* group. Create a role called *saseprole*.

```
create role saseprole;
```

STEP 6. Grant the role *saseprole* to group *sasep*:

```
grant role saseprole to group sasep;
```

STEP 7. Display the roles assigned to group *sasep*.

```
show role grant group sasep;

+------------+---------------+-------------+----------+
|    role    | grant_option  | grant_time  | grantor  |
+------------+---------------+-------------+----------+
| saseprole  | false         | NULL        | --       |
+------------+---------------+-------------+----------+
```

STEP 8. Grant SELECT on table *DGCARS* to role *saseprole*.

```
grant select on table dgcars to role saseprole;
```

STEP 9. Grant SELECT columns *make*, *model* and *enginesize* on table *DGCARSPRQT* to role *saseprole*.

```
grant select(make, model, enginesize) on table dgcarsprqt to role
saseprole;
```

STEP 10. Display the grants given to role *saseprole*.

```
show grant role saseprole;
```

```
+-----------+-------------+-------------+-----------------+-----------------+
| database  |    table    |   column    | principal_name  | principal_type  |
+-----------+-------------+-------------+-----------------+-----------------+
| sasep     | dgcarsprqt  | model       | saseprole       | select          |
| sasep     | dgcarsprqt  | enginesize  | saseprole       | select          |
| sasep     | dgcarsprqt  | make        | saseprole       | select          |
| sasep     | dgcars      |             | saseprole       | select          |
+-----------+-------------+-------------+-----------------+-----------------+
```

Note: the output has been truncated in order to fit the page.

TESTING PERMISSIONS USING BEELINE

The following is an example of steps that test the permissions.

STEP 1. On a separate terminal session, log on to one of the Hadoop nodes as user *daghaz* and create a Kerberos ticket by entering the *kinit* command followed by the *klist* command.

```
#> kinit daghaz@SAS.SGF.COM ; klist

Password for daghaz@SAS.SGF.COM:
Ticket cache: FILE:/tmp/krb5cc_45786
Default principal: daghaz@SAS.SGF.COM

Valid starting     Expires            Service principal
02/03/17 11:38:43  02/03/17 21:38:46  krbtgt/SAS.SGF.COM@SAS.SGF.COM
renew until 02/10/17 11:38:43
```

STEP 2. Start *beeline* and get a connection to *SASEP* database.

```
!connect jdbc:hive2://caesar:10000/sasep;principal=hive/_HOST@KRB.SGF.COM
```

STEP 3. Verify the authorization settings by listing all the tables user *daghaz* can access.

```
show tables;

+-------------+
|  tab_name   |
+-------------+
| dgcars      |
| dgcarsprqt  |
+-------------+
```

The SHOW TABLES command displays only the tables user *daghaz* can access. Any attempt to access a table other than *DGCARS* and *DGCARSPRQT* fails. Any attempt to access columns other than *make*, *model*, and *enginesize* on table *DGCARSPRQT* also fails. The following command demonstrates a SELECT on a table that user *daghaz* does not have access.

```
select * from dgcarsavro;

Error: Error while compiling statement: FAILED: SemanticException No valid
privileges
 User daghaz does not have privileges for QUERY
 The required privileges: Server=server1->Db=sasep->Table=dgcarsavro-
>Column=cylinders->action=select; (state=42000,code=40000)
```

As demonstrated below, any attempt to SELECT all columns on table *DGCARSPRQT* also fails.

```
select * from dgcarsprqt;
```

```
Error: Error while compiling statement: FAILED: SemanticException No valid
privileges
 User daghaz does not have privileges for QUERY
 The required privileges: Server=server1->Db=sasep->Table=dgcarsprqt-
>Column=cylinders->action=select; (state=42000,code=40000)
```

However, a SELECT on columns *make*, *model* and *enginesize* succeeds.

```
select make, model, enginesize from dgcarsprqt limit 1;

+--------+--------+-------------+
|  make  | model  | enginesize  |
+--------+--------+-------------+
| Acura  |  MDX   | 3.5         |
+--------+--------+-------------+
```

Permissions are now configured. User *daghaz* is now able to run scoring code on table *DGCARS*.

TESTING PERMISSIONS USING SAS/ACCESS INTERFACE TO HADOOP

SAS/ACCESS Interface to Hadoop provides enterprise data access and integration between SAS and Hadoop. With SAS/ACCESS Interface to Hadoop, users can connect to a Hadoop cluster to read and write data to and from Hadoop. One can analyze Hadoop data with any SAS procedures and the DATA step. SAS/ACCESS Interface to Hadoop works like other SAS engines. One executes a LIBNAME statement to assign a library reference and specify the engine. The LIBNAME statement associates a SAS library reference with Hadoop HDFS or Hive.

The following is an example of steps that test the permissions using the SAS/Access Interface to Hadoop. When connecting Base SAS to a Kerberos secured Hadoop cluster, a Kerberos ticket must be acquired.

STEP 1. Create a Kerberos ticket.

```
#> kinit daghaz@SAS.SGF.COM;
```

STEP 2. Start Base SAS. From within Base SAS, define a library reference to Hadoop using *HADOOP* as the engine name. Assign the LIBNAME.

```
LIBNAME HIVE HADOOP SERVER="caesar" SCHEMA=SASEP;
```

The library reference *HIVE* establishes a connection with Hive server on the node specified in SERVER= option. The Hive schema name is specified in SCHEMA= option.

STEP 3. Start PROC SQL using the connection provided by the library reference *HIVE*.

```
PROC SQL; CONNECT USING HIVE;
```

STEP 4. Select columns *make*, *model*, and *enginesize* from table *DGCARS*.

```
SELECT * FROM CONNECTION TO HIVE( SELECT MAKE, MODEL, ENGINESIZE FROM
DGCARS );
```

STEP 5. Select all columns from table *DGCARSAVRO*. The result is an error, since user *daghaz* does not have access to the table.

```
SELECT * FROM CONNECTION TO HIVE( SELECT * FROM DGCARSAVRO );

ERROR: Prepare error: Error while compiling statement: FAILED:
SemanticException No valid privileges
 User daghaz does not have privileges for QUERY
 The required privileges: Server=server1->Db=sasep->Table=dgcarsavro-
>Column=cylinders->action=select;
SQL statement: select * from dgcarsavro
```

STEP 6. Select column *cylinders* from table *DGCARSPRQT*. The result is an error, since user *daghaz* does not have access to that particular column on the table.

```
SELECT * FROM CONNECTION TO HIVE( SELECT CYLINDERS FROM DGCARSPRQT );

ERROR: Prepare error: Error while compiling statement: FAILED:
SemanticException No valid privileges
 User daghaz does not have privileges for QUERY
 The required privileges: Server=server1->Db=sasep->Table=dgcarsprqt-
>Column=cylinders->action=select;
SQL statement: select cylinders from dgcarsprqt
```

STEP 7. A SELECT on columns *make*, *model*, and *enginesize* succeeds.

```
SELECT * FROM CONNECTION TO HIVE( SELECT MAKE, MODEL, ENGINESIZE FROM
DGCARS LIMIT 1 );

make           model                                          enginesize
-------------------------------------------------------------------------
Acura           MDX                                                 3.5
```

STEP 8. End PROQ SQL by entering the procedure QUIT command.

```
QUIT;
```

RUNNING SCORING CODE FROM BASE SAS

The Base SAS environment needs to be prepared to connect to Hadoop. In Base SAS, the Hadoop connection attributes are specified using the INDCONN macro variable. Here is an example of how to set INDCONN macro variable.

```
%LET INDCONN=%str(USER= HIVE_SERVER=caesar SCHEMA=sasep);
```

The INDCONN macro variable holds the Hive server name and the Hive schema or database name. Since the connection is made to a secured cluster using Kerberos and the credentials are retrieved from the Kerberos ticket, the USER= option is left blank.

The following SAS statements set the Hadoop client JAR folder and the Hadoop configuration files folder. Sentry RecordService JAR and configuration files must be installed on the client machine. The example adds the RecordService JAR path to the SAS_HADOOP_JAR_PATH variable.

```
OPTIONS SET=SAS_HADOOP_JAR_PATH="/home/hadoop/jars:/home/hadoop/jars/
   recordserviceclient";
OPTIONS SET=SAS_HADOOP_CONFIG_PATH="/home/hadoop/conf";
```

The following code invokes the %INDHD_RUN_MODEL macro to run the model in Hadoop:

```
%INDHD_RUN_MODEL(  INPUTTABLE=dgcars
                 , OUTDATADIR=/user/daghaz/data/dgcarsoutput
                 , OUTMETADIR=/user/daghaz/meta/dgcarsoutput.sashdmd
                 , SCOREPGM=/user/daghaz/ds2/inout.ds2 );
```

The INPUTTABLE= option specifies the input table name in Hive. The OUTDATADIR= option specifies the directory on HDFS where the output files are stored. The OUTMETADIR= option specifies the directory on HDFS where the output file metadata is stored. The SCOREPGM= option specifies the name of the scoring model program file that is executed by the SAS Embedded Process. The execution of the model produces a delimited text output file.

The %INDHD_RUN_MODEL macro initiates the SAS Embedded Process MapReduce job. Figure 7 illustrates the scoring execution flow inside a SAS Embedded Process task. The shaded area represents

a MapReduce mapper task executing inside a Java Virtual Machine process on a particular node of the Hadoop cluster.

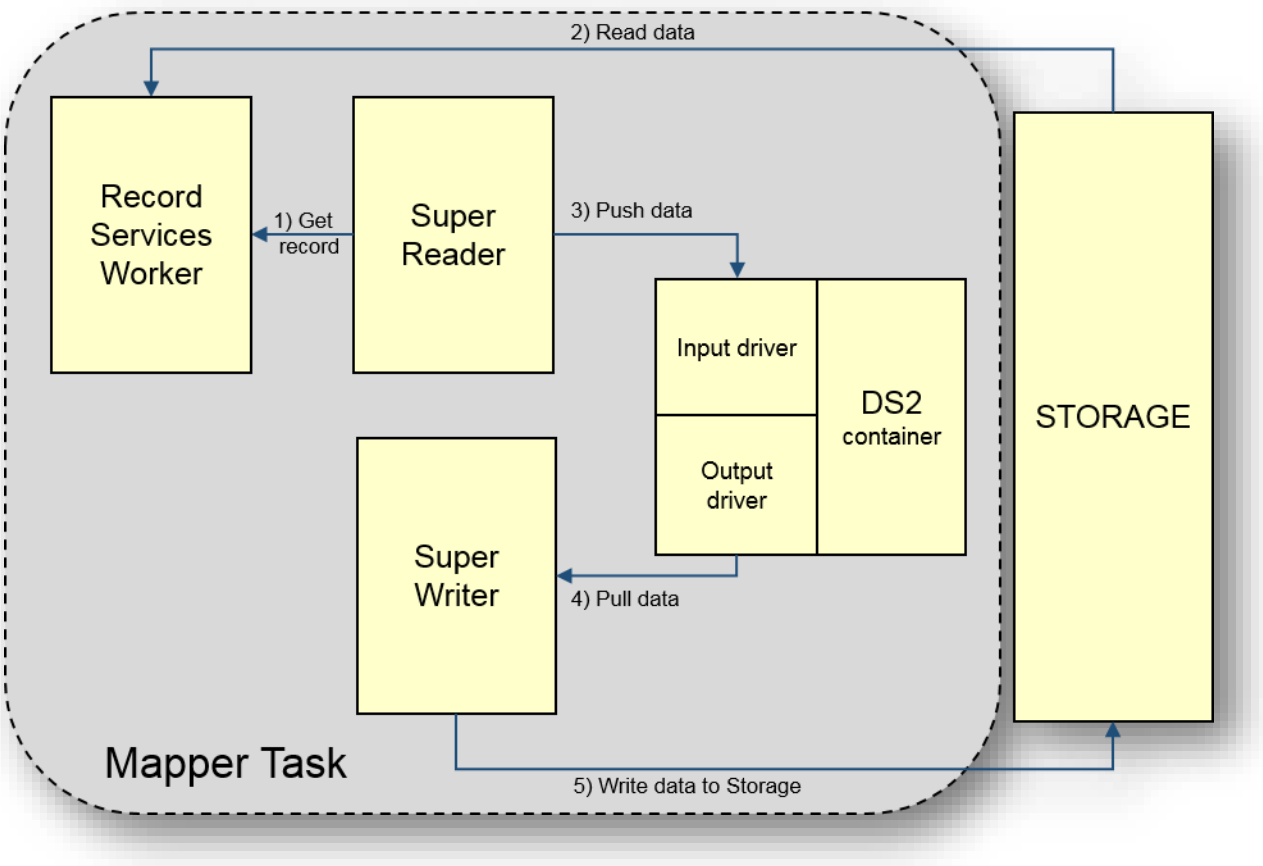

Figure 7 Scoring Execution Flow inside the SAS Embedded Process Mapper Task

A SAS Embedded Process mapper task is not a conventional MapReduce task. Super Reader is a multi-threaded, top-level input format capable of reading multiple file splits in parallel. Super Reader does not use the standard MapReduce split calculation. Instead of assigning one split per task, it assigns many. Super Reader calculates the splits, groups them, and distributes the groups to a configurable number of mapper tasks based on data locality. Super Writer is a multi-thread output channel capable of writing multiple parts of the output file in parallel.

Super Reader calls into RecordService (**1**) in order to retrieve records from the file splits (**2**). Records are serialized in a way the scoring code understands (**3**). The scoring code runs inside the DS2 container processing the records. At a given point in time, output data is flushed out to Super Writer (**4**), which writes the output data to HDFS (**5**). When all records are processed and all output is written to HDFS, the mapper task ends.

Super Reader improves task performance by reading records from many file splits at the same time and by processing the records in parallel. Super Writer improves performance by writing output data in parallel, producing multiple parts of the output file per mapper task. The output of the scoring code is stored on HDFS under the folder that was specified in the run model macro.

MONITORING THE SAS EMBEDDED PROCESS JOB EXECUTION

The SAS Embedded Process MapReduce job execution can be monitored through a web user interface provided by the YARN Resource manager at http://<*yarn-resource-manager-host-name*>:8088.

Figure 8 illustrates the YARN resource manager interface from where you can see the job in execution.

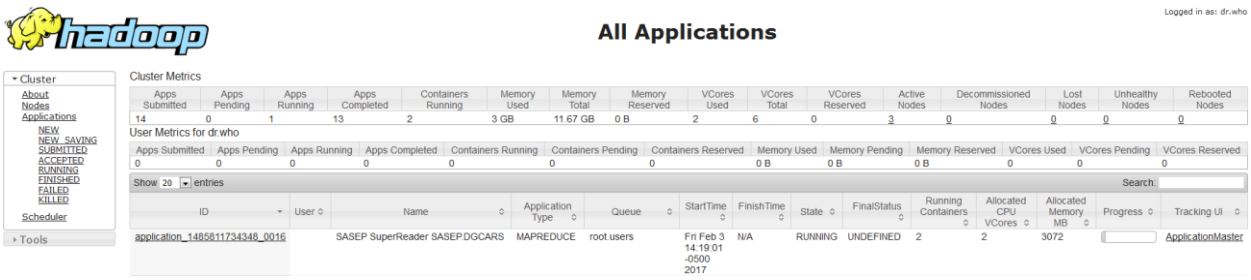

Figure 8 Scoring Accelerator MapReduce Job in Execution

The '*ApplicationMaster*' link under '*Tracking UI*' field provides another monitoring interface. After job completion, '*Tracking UI*' field provides a link to the job summary page as illustrated by Figure 9.

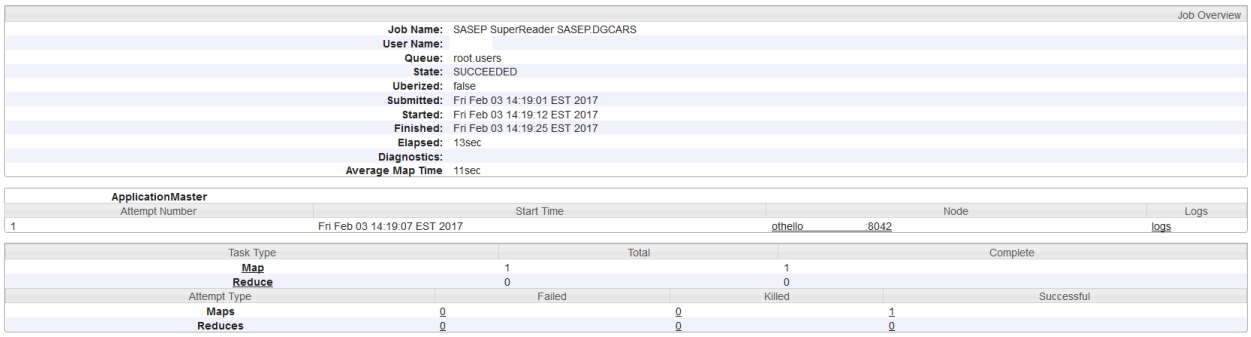

Figure 9 Scoring Accelerator MapReduce Job Summary Page

From the job-summary page, one can navigate to the *Maps* attempts page, where one can find a link to the attempt log, as illustrated by Figure 10.

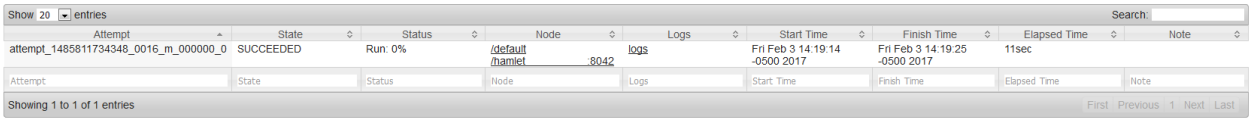

Figure 10 Scoring Accelerator Maps Attempts Page

The link '*logs*' provided under '*Logs*' field takes you to the attempt job log summary. There you can see parts of *stderr*, *stdout*, and *syslog* logs. SAS Embedded Process writes its log messages to *stdout* and *stderr* logs. The Java code of the SAS Embedded Process MapReduce task writes to the job *syslog*.

Output 1 shows a summary of the log messages generated during execution. The highlighted messages in the job log shows that RecordService was invoked to retrieve the records from the table. The output of the map task is written to the location specified in the OUTDATADIR= option of the %INDHD_RUN_MODEL macro. The output metadata is written to the location specified in OUTMETADIR= option.

```
INFO [DonAlvaro] Alvaro: Alvaro has been invoked to guide current task execution.
INFO [main] EPNativeInterface: Fetch size set to 4,002 records. Number of input buffers is 4.
Input buffers will consume 4,613,504 bytes.
INFO [main] EPNativeInterface: Number of output buffers is 3. Output buffers will consume
3,460,128 bytes.
INFO [SR-M00000-T0] custom.CustomFileUtils: Creating user input format class instance
superreader.in.RecordServiceInputFormat
INFO [SR-M00000-T0] superreader.in.RecordServiceInputFormat: SAS EP RecordService Input Format
created a SuperRecordServiceReader.
INFO [SR-M00000-T0] mr.WorkerUtil: Both data and RecordServiceWorker are available locally for
task 5b47f975dddfe2fd:ae00063256085bbe
INFO [SR-M00000-T0] core.ThriftUtils: Connecting to RecordServiceWorker at
hamlet.unx.sas.com:13050 using delegation token, with timeout: 10000ms
INFO [SR-M00000-T0] core.ThriftUtils: Connected to RecordServiceWorker at
hamlet.unx.sas.com:13050
INFO [SR-M00000-T0] core.RecordServiceWorkerClient: Got task handle: TUniqueId(hi:-
123404028071008522, lo:-818559185049730658)
INFO [SR-M00000-T0] superreader.in.CustomReader: Custom Reader will process records provided
by RecordService.
INFO [SR-M00000-T0] core.RecordServiceWorkerClient: Closing RecordServiceWorker task:
TUniqueId(hi:-123404028071008522, lo:-818559185049730658)
INFO [SR-M00000-T0] core.RecordServiceWorkerClient: Closing RecordServiceWorker connection.
INFO [main] superreader.SuperReader: Readers executed in 273 ms
INFO [main] EPNativeInterface: Task INPUT Summary:
INFO [main] EPNativeInterface:   Task status ..................: SUCCEEDED
INFO [main] EPNativeInterface:   Number of input threads ......: 1
INFO [main] EPNativeInterface:   Number of compute threads ....: 1
INFO [main] EPNativeInterface:   Number of splits .............: 1
INFO [main] EPNativeInterface:   Input records .................: 428
INFO [main] EPNativeInterface:   Input bytes ...................: 0
INFO [main] EPNativeInterface:   Transcode errors ..............: 0
INFO [main] EPNativeInterface:   Truncation errors .............: 0
INFO [main] EPNativeInterface:   Input buffer reallocations ...: 0
INFO [SWCH-M00000-T1] output.SuperWriterChannel: Received completion notification from thread
[1]. Thread status=[2].
INFO [SWCH-M00000-T0] output.SuperWriterChannel: Received completion notification from thread
[0]. Thread status=[2].
INFO [Thread-16] output.SuperWriterChannel: Writer threads executed in 303 ms
INFO [main] EPNativeInterface: Task OUTPUT Summary:
INFO [main] EPNativeInterface:   Writer status .................: SUCCEEDED
INFO [main] EPNativeInterface:   Number of output threads .....: 2
INFO [main] EPNativeInterface:   Output records ................: 428
INFO [main] EPNativeInterface:   Output bytes ..................: 34593
INFO [main] EPNativeInterface:   Output buffer reallocations ..: 0
INFO [main] EPNativeInterface: SAS Embedded Process Task Environment has been destroyed.
INFO [main] EPNativeInterface: SAS Embedded Process Core TK Environment has been destroyed.
INFO [main] TaskRunner: SAS Embedded Process Task Runner executed in 813 ms
```

Output 1 Output from Map Task Job Log Summary Page

In order to see the content of the output file from Base SAS, the following library reference definition is created to give access to HDFS. The library reference is not required to publish or to run a model. It is used by Base SAS to display the content of the output file. The following code assigns the library reference to HDFS:

```
LIBNAME HDFS HADOOP SERVER="caesar"
                    HDFS_DATADIR="/user/daghaz/data"
                    HDFS_METADIR="/user/daghaz/meta";
```

The library reference HDFS_DATADIR= option points to the folder where the output file is stored. The HDFS_METADIR= options points to the folder where the output file metadata is stored.

The output metadata can be displayed by running the PROC CONTENTS procedure on the output table. The following source code displays the output metadata:

```
PROC CONTENTS DATA=HDFS.DGCARSOUTPUT; RUN;
```

Output 2 displays the content of the output table metadata.

The CONTENTS Procedure
Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format	Informat	Label
9	cylinders	Num	8			cylinders
5	drivetrain	Char	5	$5.	$5.	drivetrain
8	enginesize	Num	8			enginesize
10	horsepower	Num	8			horsepower
7	invoice	Num	8			invoice
15	length	Num	8			length
1	make	Char	13	$13.	$13.	make
2	model	Char	40	$40.	$40.	model
11	mpg_city	Num	8			mpg_city
12	mpg_highway	Num	8			mpg_highway
6	msrp	Num	8			msrp
4	origin	Char	6	$6.	$6.	origin
3	type	Char	8	$8.	$8.	type
13	weight	Num	8			weight
14	wheelbase	Num	8			wheelbase

Output 2 Output Metadata

The output data can be displayed by running PROC PRINT procedure on the output table. Here is an example of how to print selected columns of the first 10 observations:

```
PROC PRINT DATA=HDFS.DGCARSOUTPUT(OBS=10); VAR MAKE MODEL ENGINESIZE; RUN;
```

Output 3 displays the first 10 observations of the output file.

Obs	make	model	enginesize
1	Acura	MDX	3.5
2	Acura	RSX Type S 2dr	2.0
3	Acura	TSX 4dr	2.4
4	Acura	TL 4dr	3.2
5	Acura	3.5 RL 4dr	3.5
6	Acura	3.5 RL w/Navigation 4dr	3.5
7	Acura	NSX coupe 2dr manual S	3.2
8	Audi	A4 1.8T 4dr	1.8
9	Audi	A41.8T convertible 2dr	1.8
10	Audi	A4 3.0 4dr	3.0

Output 3 Output Data for First 10 Observations

CONCLUSION

Securing the analytic infrastructure is a fundamental step that cannot be ignored. SAS Embedded Process integration with Apache Sentry Record services brings together the massively parallel processing power of Hadoop and a robust security system. SAS provides the platform you need to process data in an effective and efficient manner. Apache Sentry provides the level of security needed. Users are now ready to collect, store, and process data securely using the power of SAS and the protection of Apache Sentry RecordService.

REFERENCES

The Apache Software Foundation.2016. *Apache Sentry Tutorial*. Available https://cwiki.apache.org/confluence/display/SENTRY/Sentry+Tutorial. Accessed on February 21, 2017.

The Apache Software Foundation. 2016, *Apache Sentry Documentation*. Available https://cwiki.apache.org/confluence/display/SENTRY/documentation. Accessed on February 21, 2017.

The MIT Kerberos Consortium. 2017. *MIT Kerberos Documentation*. Available http://web.mit.edu/kerberos/krb5-current/doc. Accessed on February 21, 2017.

RecordService. 2016 *RecordService Documentation*. Available http://recordservice.io

Cloudera Engineering Blog. “RecordService: For Fine-Grained Security Enforcement across the Hadoop Ecosystem”. Available https://blog.cloudera.com/blog/2015/09/recordservice-for-fine-grained-security-enforcement-across-the-hadoop-ecosystem. Accessed on February 21, 2017.

RECOMMENDED READING

- 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.

CONTACT INFORMATION

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

David Ghazaleh
500 SAS Campus Drive
Cary, NC 27513
SAS Institute Inc.
David.Ghazaleh@sas.com
http://www.sas.com