Paper SAS3116-2019

Execution of User-Written DS2 programs inside Apache Spark using SAS® In-Database Code Accelerator

David Ghazaleh, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® In-Database Technologies offers a flexible, efficient way to leverage increasing amounts of data by injecting the processing power of SAS wherever the data lives. SAS® In-Database Technologies can tap into the massively parallel processing (MPP) architecture of Apache Hadoop and Apache Spark for scalable performance. SAS® In-Database Code Accelerator for Hadoop allows the parallel execution of user-written DS2 programs using Apache Spark.

This paper explains how the SAS In-Database Code Accelerator exploits Scala and the parallel processing power of Apache Spark and prepares you to get started with SAS In-Database Technologies.

INTRODUCTION

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 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.

SAS Embedded Process is a portable, lightweight execution container that allows the parallel execution of SAS processes inside Hadoop, Spark, Teradata, and many other MPP databases.

DS2 is a procedural programming language influenced by the SAS DATA step. The DS2 language excels at achieving parallel execution. Most DATA step functions can be called from a DS2 program. DS2 programs can run in the Base SAS language interface using PROC DS2, SAS High-Performance Analytics, SAS In-Database Scoring Accelerator, SAS In-Database Code Accelerator, and SAS® Viya® Cloud Analytic Services.

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 MapReduce and Spark.

There are many benefits in bringing together big data, Hadoop and Spark processing power, and the intelligence offered by SAS, including:

- Greater storage capabilities. Store all the data you can collect. Experience the maximum accuracy of larger data.
- Greater parallel processing capabilities. Write more complex algorithms to obtain more precise results.
- Faster data growth and processing time. Maximizing and expanding the value of Hadoop and Spark across the enterprise is essential and desired. SAS Embedded Process is as scalable as your Hadoop and Spark cluster.
- Data management and integration in order to promote broad reuse while being compliant with Information Technology policies and procedures.

- Boost the value of analytics infrastructure while reducing the cost to maintain it.

SAS EMBEDDED PROCESS ON SPARK

SAS Embedded Process is the core of SAS In-Database Technology that is sufficient to support the multi-threaded DS2 language. Running DS2 code directly inside Spark effectively leverages the massive parallel processing and native resources. Applying the process to the data eliminates data movement and decreases overall processing time. Information becomes more secure because the data never leaves the cluster.

SAS Embedded Process on Spark is supported when Spark is running on YARN. In order to run DS2 code in parallel inside Spark, SAS Embedded Process needs to be installed on every node of the cluster that is capable of running a Spark task. All the computing resources used by SAS Embedded Process 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 executors' containers.

SAS Embedded Process is written mainly in C language where all the interactions with DS2 execution container happen. It is also written in Java and Scala where all the interactions with the Spark environment happen. The Java code is responsible for extracting data from Hive tables or Hadoop File System (HDFS) files and passing them to the DS2 execution container. The Scala code drives the whole 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 executors' 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. Therefore, the SAS Embedded Process complies with the resource constraints imposed by YARN. Journaling messages generated by the C and DS2 code are written to the Spark standard output (*stdout*) or standard error (*stderr*) logs. Messages generated by the Java and Scala code are written to the Spark application log (*syslog*).

CLIENT-SIDE COMPONENTS

Figure 1 illustrates the SAS Embedded Process client-side architectural components.

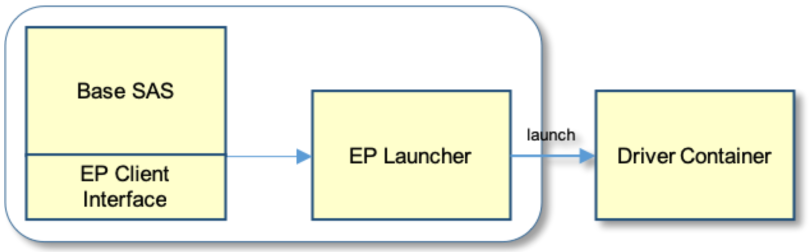

Figure 1. Client-Side Architectural Components

SAS In-Database Code Accelerator DS2 programs are started from *Base SAS* using PROC DS2. When the execution platform is set to SPARK, the *EP Client Interface* generates the Scala program that is the SAS Embedded Process Spark *Main Driver* program, which runs in the *Driver Container* (see Figure 2). The *EP Launcher* deploys the user-written DS2 and the generated Scala programs to the cluster and submits the SAS Embedded Process Spark application for execution.

DRIVER CONTAINER COMPONENTS

Figure 2 illustrates the SAS Embedded Process driver container architectural components.

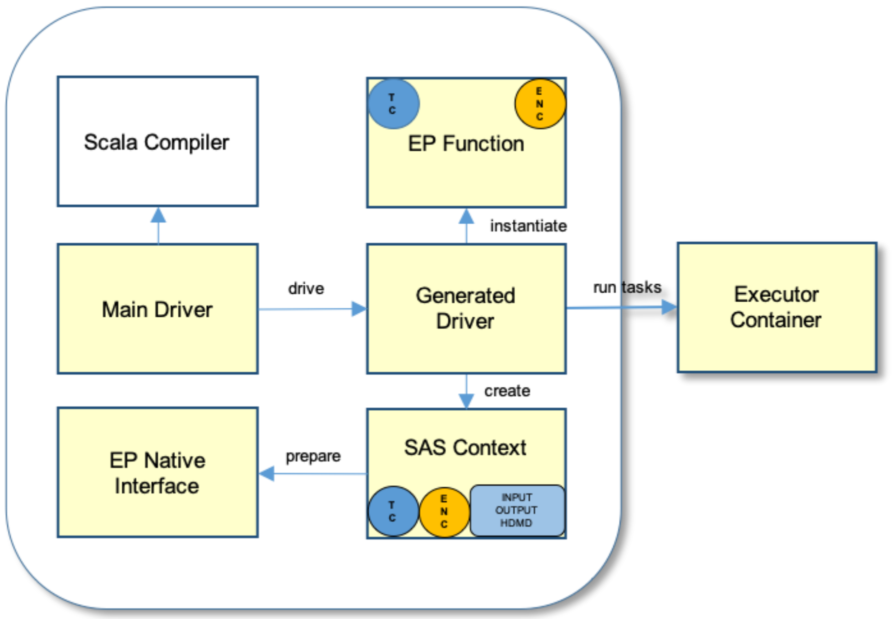

Figure 2. Driver Container Architectural Components

Upon Spark application submission, a *Driver Container* and multiple *Executor Containers* (see Figure 3) are allocated on the cluster. The *Driver Container* is the Spark YARN Application Master process where the *Main Driver* program runs.

The *Main Driver* receives an execution request containing the generated Scala program. The Scala program is compiled on the fly by the *Scala Compiler* component. The *Generated Driver* component is the product of the generated Scala program compilation into a class that is stored in the Java Virtual Machine class loader. The *Generated Driver* drives the execution of the SAS Embedded Process Spark application.

The *Main Driver* component puts the *Generated Driver* into execution by calling its *drive()* method. Inside the *drive()* method, the *Generated Driver* creates a *SAS Context*, which is responsible for an early compilation of the user-written DS2 program and for collecting and storing all necessary information to run the DS2 program when it is dispatched to the *Executor Containers*. *SAS Context* holds the SAS Embedded Process *Task Context* (*TC*), the input and output file or table metadata (INPUT OUTPUT HDMD), and the output encoder object (*ENC*).

When reading data from an HDFS file, the input metadata is passed by PROC DS2 during application submission. When reading data from a Hive table, the input metadata is retrieved from the Spark Dataset schema that is returned from the Spark SQL statement execution request. *SAS Context* creates the output metadata (output HDMD) based on the early compilation of the user-written DS2 program inside the *EP Native Interface* component. The output metadata is used to create the *ENC*. The ENC is used to create the schema of the output Dataset or RDD (Resilient Distributed Datasets) that comes out of the *EP Function*.

A specialized *EP Function* is instantiated by the *Generated Driver* and applied to a Spark Dataset or RDD. There are many types of specialized EP functions. Their instantiations depend on the many different ways to run SAS In-Database Code Accelerator inside Spark.

EXECUTOR CONTAINER COMPONENTS

Figure 3 illustrates the SAS Embedded Process executor container architectural components.

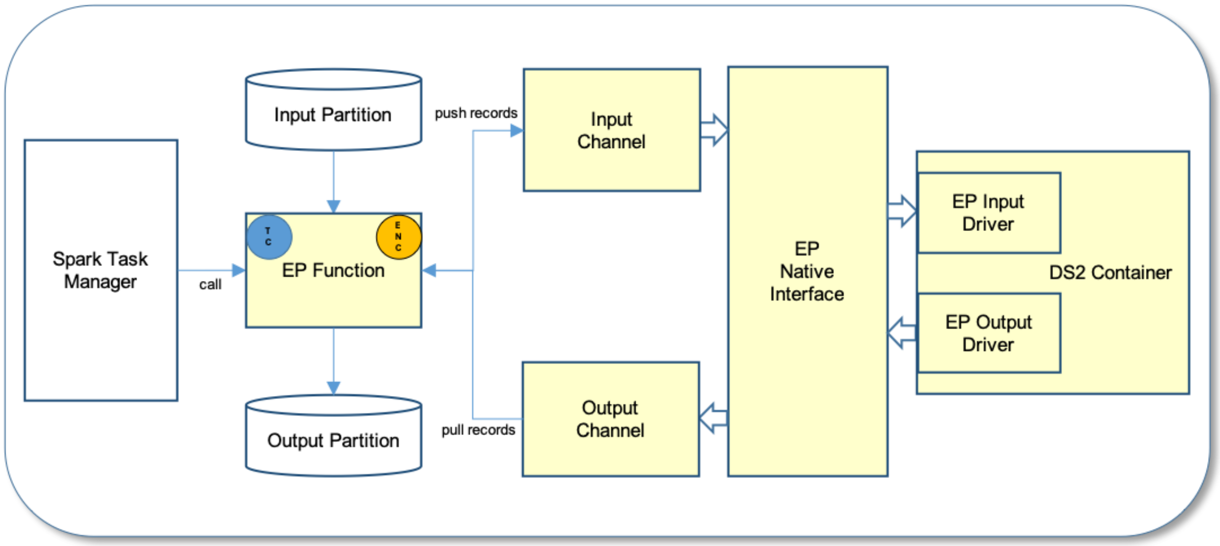

Figure 3. Executor Container Architectural Components

Spark tasks run in the executor container process. Each task is assigned an input partition, which might come from a Hive table or an HDFS file. The specialized *EP Function* that was instantiated in the *Driver Container* and deployed to the executors' containers is put into execution by the *Spark Task Manager* to process records retrieved from the input partition. Here is a sequence of steps executed in the *Executor Container*:

1. *EP Function* creates the input and output channels. The *EP Function* is also responsible for controlling the allocation and de-allocation of *DS2 Containers*.
2. *EP Function* retrieves records from the *Input Partition* and pushes them into the *DS2 Container* through the *Input Channel* and *EP Input Driver*. Input data are serialized in a format that is understood by the DS2 program and stored in native input buffers. The *EP Native Interface* is the frontier between the Java and C code.
3. *DS2 Container* obtains the serialized input records from the native input buffers through the *EP Input Driver* and processes them.
4. *DS2 Container* outputs and stores one or more records in output native buffers through the *EP Output Driver*.
5. *Output Channel* retrieves output records from native output buffers. Using the output encoder object, the *Output Channel* serializes the records in a format understood by Spark. Serialized output records are stored in the *Output Partition*.

Each task produces an output partition. When all tasks are finished, the *Generated Driver* sees all output partitions as a single abstract unit called output Dataset or RDD. The output Dataset or RDD is then persisted to a Hive table or to an HDFS file.

SAS IN-DATABASE CODE ACCELERATOR

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 SerDes (Serializers and De-serializers).

SAS In-Database Code Accelerator on Spark is a combination of generated Scala programs, Spark SQL statements, HDFS files access, and DS2 programs.

User-written DS2 programs can be complex. When running inside a database, SAS In-Database Code Accelerator execution plan might require multiple phases. For example: when running inside Hadoop MapReduce, SAS In-Database Code Accelerator exploits the map and reduce phases in order to get to the final result; in some cases, two MapReduce jobs are required. By generating Scala programs that integrates with the SAS Embedded Process program interface to Spark, the many phases of a SAS In-Database Code Accelerator job can be comprised of one single Spark job.

SAS In-Database Code Accelerator for Hadoop enables the publishing of user-written DS2 thread or data programs to Spark, where they can be executed in parallel exploiting Spark's massively parallel processing power. Examples of DS2 thread programs include large transpositions, computationally complex programs, scoring models, and BY-group processing. For more information about DS2 BY-group processing, consult the SAS In-Database product documentation.

To use Spark as the execution platform, the DS2ACCEL option in the PROC DS2 statement must be set to YES or the DS2ACCEL system option must be set to ANY; the HADOOPPLATFORM system option must be set to SPARK; the Hive table or HDFS file used as input must reside on the cluster; and SAS Embedded Process must be installed on all the nodes of the Hadoop cluster that are capable of running a Spark Executor. In addition, the following products must be licensed:

- Base SAS
- SAS In-Database Code Accelerator for Hadoop
- SAS/ACCESS Interface to Hadoop

Data is distributed on different nodes of the Hadoop cluster. Spark breaks down the input data into Dataset (input from table) or RDD (input from file) partitions. Data partitions are also known as file blocks or file splits. Each partition is assigned to a Spark task, where the DS2 program runs. Each DS2 program has access to its own data partition.

Parallel execution of SAS In-Database Code Accelerator inside the SAS Embedded Process on Spark consists of one Spark application. A Spark application consists of one or more jobs. Jobs consist of one or more stages. A stage is a set of tasks that depend on each other. A task is a unit of work that runs in the *Executors Containers*.

Figure 4 illustrates the Spark application key execution components.

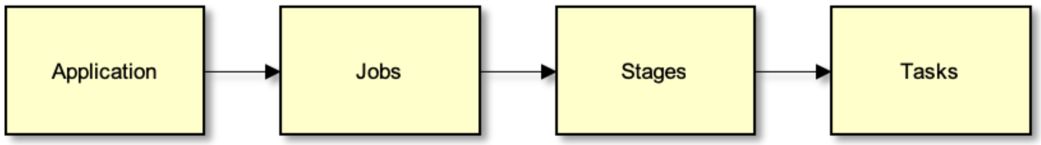

Figure 4. Spark Application Key Components

The degree of parallelism depends on how the data is partitioned. Therefore, the number of tasks depends on the number of input data partitions. Spark assigns one partition per task. The number of parallel tasks depends on the number of available executors, the number of cores per executor and the number of cores per task. There are many performance tuning properties that can be used to control the application execution. Spark properties are set in the *spark-default.conf* configuration file that is stored under the folder that is defined in the SAS global option SAS_HADOOP_CONFIG_PATH. Examples of such Spark configuration properties are:

- *spark.executor.instances*: specifies the number of executors allocated per application.
- *spark.executor.cores*: specifies the number of cores allocated per executor.
- *spark.task.cpus*: specifies the number of cores to allocate for each task.

As shown in Figure 3 above, the task runs a specialized *EP Functions* that retrieves records from the input partition, serializes and pushes them into the DS2 container. Output records that are generated by the DS2 program are pulled by the *EP Function* and stored in the output partition.

SAS Embedded Process on Spark provides two sets of specialized functions that are capable of reading data from an input partition and applying them to the DS2 program:

1. File functions: applied when the input data is read from a file stored in HDFS.
2. Dataset functions: applied when the input data is read from a table stored in Hive.

The functions are categorized as transformations or actions. Transformation functions consume data from a Dataset or RDD and produce another Dataset or RDD. Action functions consume data from a Dataset or RDD and write the output data directly to a file stored in HDFS.

HOW IS SAS IN-DATABASE CODE ACCELERATOR EXECUTED INSIDE SPARK?

There are six different ways to run SAS In-Database Code Accelerator inside Spark. They are called cases. The generation of the Scala program by the *EP Client Interface* depends on how the DS2 program is written. The main factors deciding the Case to be applied are:

- Is there a thread program?
- Is there a thread program with a BY statement?
- Is there a data program with logic worth accelerating?
- Is there a data program with a BY statement?

A simple SET statement or OUTPUT statement in the data program does not trigger acceleration. When the data program does not contain logic worth accelerating, it means that there is no data program to be executed in accelerated mode inside the SAS Embedded Process container.

SAS In-Database Code Accelerator cases depend on how the DS2 program is written. The six SAS In-Database Code Accelerator cases are described below:

- There is a thread program with no BY statement; there is no data program.
- There is a thread and a data program; none of them with a BY statement.
- There is a thread program with no BY statement and a data program with a BY statement.
- There is a thread program with a BY statement and no data program.
- There is a thread program with a BY statement and a data program with no BY statement.
- There is a thread and data program; both with a BY statement.

SAS IN-DATABASE CODE ACCELERATOR CASE 1

Table 1 illustrates the deciding factors for Case 1:

Thread Program	Thread BY Statement	Data Program	Data BY Statement
YES	NO	NO	NO

Table 1. SAS In-Database Code Accelerator Case 1

The following code sample is the simplest case where the DS2 thread program runs in one single phase and tasks are executed in parallel.

```
proc ds2 ds2accel=yes;
  thread work.workthread / overwrite=yes;
    method run();
      set hive.cars;
      output;
    end;
  endthread;

  data hive.dgcarsout (overwrite=yes);
    dcl thread work.workthread m;
    method run();
      set from m;
      output;
    end;
  enddata;
run; quit;
```

Figure 5 illustrates the steps performed by the generated Scala program when the input method is a file.

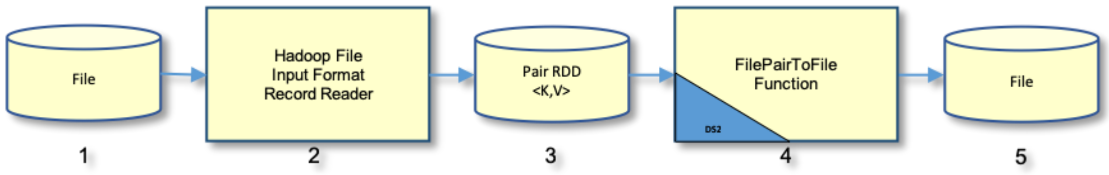

Figure 5. Case 1 - Input from File

Data is read from a file (1) into a key/value pair RDD (3) using the Hadoop File Input Format and Record Reader framework (2). The record key is ignored, and the value contains the record data. The pair RDD (3) is applied to the specialized EP function *FilePairToFileFunction* (4), where the DS2 thread program is executed in parallel. The output produced by the DS2 thread program is written directly to a file (5) stored in HDFS.

Figure 6 illustrates the steps performed by the generated Scala program when the input method is a table.

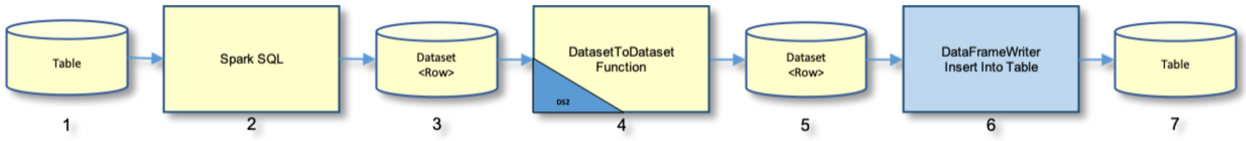

Figure 6. Case 1 - Input from Table

Using Spark SQL (2), the data are read from a table (1) into a Dataset of Spark Row objects (3). Each Row object represents a record. The Dataset (3) is applied to the specialized EP function *DatasetToDatasetFunction* (4), where the DS2 thread program is executed in parallel. The output produced by the DS2 thread program is stored in the output Dataset of Row objects (5). The output Dataset rows (5) are inserted into the output table (7) using the *DataFrameWriter* interface (6).

SAS IN-DATABASE CODE ACCELERATOR CASE 2

Table 2 illustrates the deciding factors for Case 2:

Thread Program	Thread BY Statement	Data Program	Data BY Statement
YES	NO	YES	NO

Table 2. SAS In-Database Code Accelerator Case 2

The entire DS2 program runs in two phases. The DS2 thread program runs in phase one and its tasks are executed in parallel. The DS2 data program runs in phase two using one single task. Here is an example of a case 2 DS2 program:

```
proc ds2 ds2accel=yes;
  thread work.workthread / overwrite=yes;
    method run();
      set hive.cars;
      output;
    end;
  endthread;

  data hive.dgcarsout (overwrite=yes);
    dcl thread work.workthread m;
    dcl double count;
    keep count make model;
    method run();
      set from m;
      count+1;
      output;
    end;
  enddata;
run; quit;
```

Figure 7 illustrates the steps performed by the generated Scala program when the input method is a file.

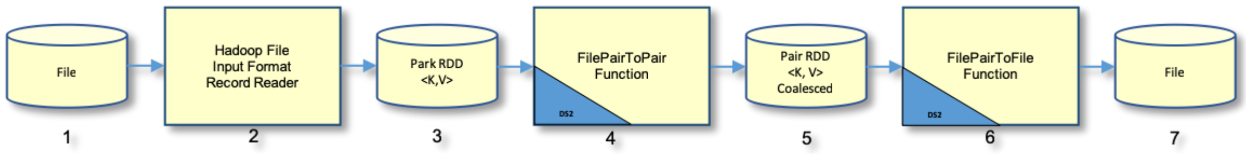

Figure 7. Case 2 - Input from File

In phase one, data are read from a file (1) into a key/value pair RDD (3) using the Hadoop File Input Format and Record Reader framework (2). The record key is ignored, and the value contains the record data. The pair RDD (3) is applied to the specialized EP function *FilePairToPairFunction* (4), where the DS2 thread program is executed in parallel. The output produced by the DS2 thread program is coalesced into a key/value pair RDD (5).

In phase two, the coalesced pair RDD (5) is applied to the specialized EP function, *FilePairToFileFunction* (6), where the DS2 data program is executed as a single task. The output produced by the DS2 data program is written directly to a file (7) stored in HDFS.

Figure 8 illustrates the steps performed by the generated Scala program when the input method is a table.

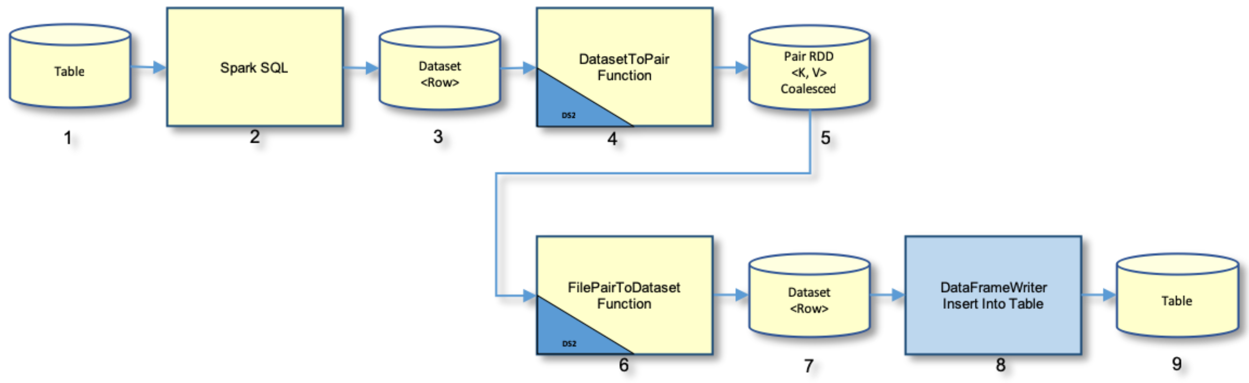
Table
1
Spark SQL
2
Dataset <Row>
3
DatasetToPair Function
DS2
4
Pair RDD <K, V> Coalesced
5
FilePairToDataset Function
DS2
6
Dataset <Row>
7
DataFrameWriter Insert Into Table
8
Table
9

Figure 8. Case 2 - Input from Table

In phase one, using Spark SQL (2), data is read from a table (1) into a Dataset of Row objects (3). Each Row object represents a record. The Dataset (3) is applied to the specialized EP function *DatasetToPairFunction* (4), where the DS2 thread program is executed in parallel. The data produced by the DS2 thread program is coalesced into a key/value pair RDD (5).

In phase two, the coalesced pair RDD (5) is applied to the specialized EP function *FilePairToDatasetFunction* (6), where the DS2 data program is executed as a single task. The output produced by the DS2 data program is stored in the output Dataset of Row objects (7). The output Dataset rows (7) are inserted into the output table (9) using the *DataFrameWriter* interface (8).

SAS IN-DATABASE CODE ACCELERATOR CASE 3

Table 3 illustrates the deciding factors for Case 3:

Thread Program	Thread BY Statement	Data Program	Data BY Statement
YES	NO	YES	YES

Table 3. SAS In-Database Code Accelerator Case 3

Here is an example of a case 3 DS2 program that uses a BY statement in the DS2 data program:

```
proc ds2 ds2accel=yes;
  thread work.workthread / overwrite=yes;
    method run();
      set hive.cars;
      output;
    end;
  endthread;

  data hive.dgcarsout (overwrite=yes);
    dcl thread work.workthread m;
```

```
      dcl double count;
      keep count make model;
      method run();
        set from m;
        by make model;
        count+1;
        output;
      end;
    enddata;
  run; quit;
```

The entire DS2 program runs in two phases. The DS2 thread program runs in phase one and its tasks are executed in parallel. Output data from the DS2 thread program is sorted by the columns specified in the BY statement of the DS2 data program. The DS2 data program runs in phase two using one single task.

Figure 9 illustrates the steps performed by the generated Scala program when the input method is a file.

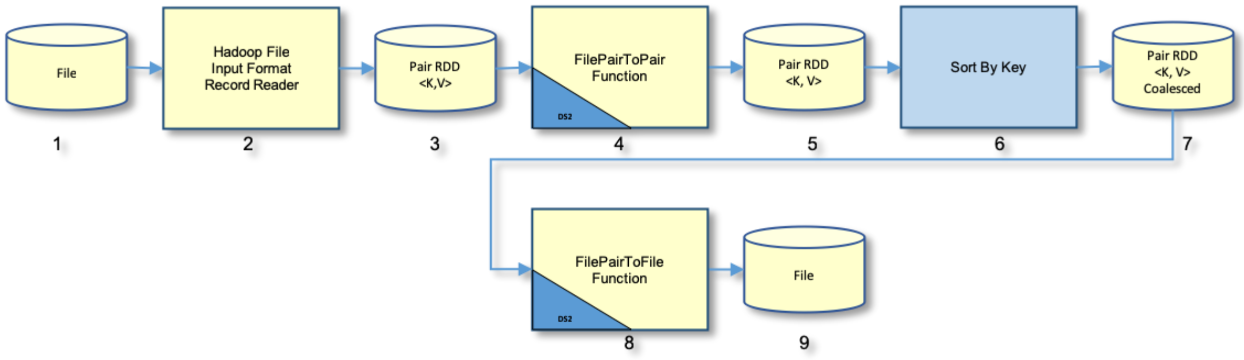

Figure 9. Case 3 - Input from File

In phase one, data is read from a file (1) into a key/value pair RDD (3) using the Hadoop File Input Format and Record Reader framework (2). The key is ignored, and the value contains the record data. The pair RDD (3) is applied to the specialized EP function *FilePairToPairFunction* (4), where the DS2 thread program is executed in parallel. The output produced by the DS2 thread program is stored in a key/value pair RDD (5). The pair RDD’s (5) key is the columns specified in the BY statement of the DS2 data program. The pair RDD (5) is sorted and coalesced (6) into another pair RDD (7) that is used as input for the next phase.

In phase two, the coalesced pair RDD (7) is applied to the specialized EP function *FilePairToFileFunction* (8), where the DS2 data program is executed as a single task. The output produced by the DS2 data program is written directly to a file (9) stored in HDFS.

Figure 10 illustrates the steps performed by the generated Scala program when the input method is a table.

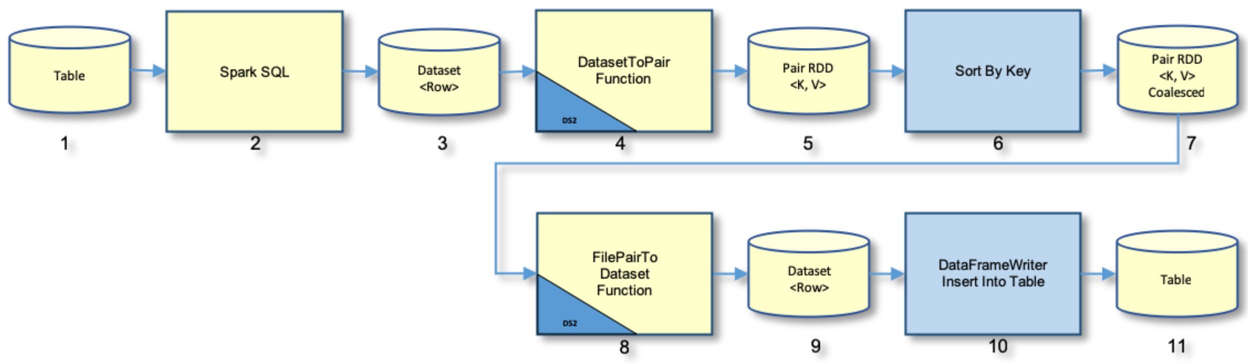

Figure 10. Case 3 - Input from Table

In phase one, using Spark SQL (2), the data is read from a table (1) into a Dataset of Row objects (3). Each Row object represents a record. The Dataset (3) is applied to the specialized EP function *DatasetToPairFunction* (4), where the DS2 thread program is executed in parallel. The output produced by the DS2 thread program is stored in a key/value pair RDD (5). The pair RDD's (5) key is the columns specified in the BY statement of the DS2 data program. The pair RDD (5) is sorted and coalesced (6) into another pair RDD (7) that is used as input for the next phase.

In phase two, the coalesced pair RDD (7) is applied to the specialized EP function *FilePairToDatasetFunction* (8), where the DS2 data program is executed as a single task. The output produced by the DS2 data program is stored in the output Dataset of Row objects (9). The output rows are inserted into the output table (11) using the *DataFrameWriter* interface (10).

SAS IN-DATABASE CODE ACCELERATOR CASE 4

Table 4 illustrates the deciding factors for Case 4:

Thread Program	Thread BY Statement	Data Program	Data BY Statement
YES	YES	NO	NO

Table 4. SAS In-Database Code Accelerator Case 4

Here is an example of a case 4 DS2 program that uses a BY statement in the DS2 thread program. The data program in the example below does not contain logic worth accelerating. Therefore, the data program is not executed in accelerated mode inside Spark.

```
proc ds2 ds2accel=yes;
  thread work.thread1 / overwrite=yes;
    dcl double count;
    dcl double averagemsrp;
    dcl double totalmsrp;
    keep make type averagemsrp;

    method run();
      set hive.cars;
      by make type;

      if first.type then do;
        count=0;
        totalmsrp=0;
```

```
          end;

          count+1;
          totalmsrp+msrp;

          if last.type and count > 0 then do;
            averagemsrp = totalmsrp / count;
            output;
          end;
        end;
      endthread;

      data hive.carsmsrpout (overwrite=yes);
        dcl thread work.thread1 p;
        method run();
          set from p;
          output;
        end;
      enddata;
    run; quit;
```

The input data are partitioned and sorted within the partition by the columns specified in the BY statement of the DS2 thread program. All records with same key end up in the same partition. DS2 thread program runs in parallel using multiple tasks.

Figure 11 illustrates the steps performed by the generated Scala program when the input method is file.

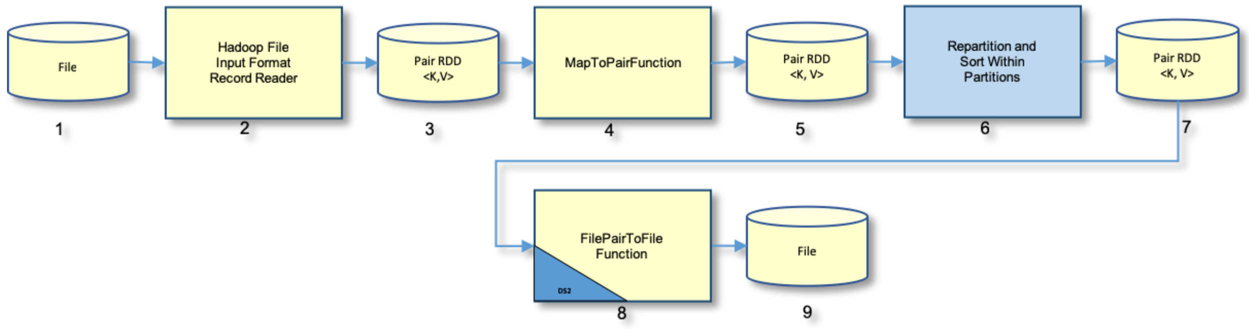

Figure 11. Case 4 - Input from File

Data is read from a file (1) into a key/value pair RDD (3) using the Hadoop File Input Format and Record Reader framework (2). Using the columns specified in the DS2 thread program BY statement, the pair RDD (3) is applied to the specialized function *MapToPairFunction* (4) to create a mapped pair RDD (5) with proper key and value pair. Using the key, the mapped pair RDD (5) is repartitioned with sorting within partitions (6). This results in a new pair RDD (7) partitioned by the key, where the keys within the partitions are sorted. The partitioned pair RDD (7) is applied to the specialized EP function *FilePairToFileFunction* (8), where the DS2 thread program runs in parallel. The output produced by the DS2 thread program is written directly to a file (9) stored in HDFS.

Figure 12 illustrates the steps performed by the generated Scala program when the input method is a table.

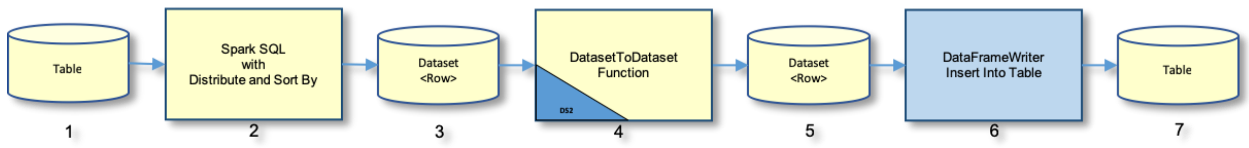
Table
1
Spark SQL with Distribute and Sort By
2
Dataset <Row>
3
DatasetToDataset Function
DS2
4
Dataset <Row>
5
DataFrameWriter Insert Into Table
6
Table
7

Figure 12. Case 4 - Input from Table

The columns specified in the BY statement of the DS2 thread program are used in the Spark SQL SELECT statement with the DISTRIBUTE BY and SORT BY clauses (2). Data is read from a table (1) into a partitioned and sorted within partition Dataset of Row objects (3). The Dataset (3) is applied to the specialized EP function *DatasetToDatasetFunction* (4), where the DS2 thread program is executed in parallel. The output produced by the DS2 thread program is stored in the output Dataset of Row objects (5). The output rows are inserted into the output table (7) using the *DataFrameWriter* interface (6).

SAS IN-DATABASE CODE ACCELERATOR CASE 5

Table 5 illustrates the deciding factors for Case 5:

Thread Program	Thread BY Statement	Data Program	Data BY Statement
YES	YES	YES	NO

Table 5. SAS In-Database Code Accelerator Case 5

The entire DS2 program runs in two phases. Case 5 phase one is similar to the single phase executed in Case 4. The DS2 thread program contains a BY statement and its tasks are executed in parallel. The DS2 data program contains logic worth accelerating. Therefore, it runs in phase two using one single task. Here is an example of a case 5 DS2 program:

```
proc ds2 ds2accel=yes;
  thread work.thread1 / overwrite=yes;
    dcl double count;
    dcl double averagemsrp;
    dcl double totalmsrp;
    keep make type averagemsrp;

    method run();
      set hive.cars;
      by make type;

      if first.type then do;
        count=0;
        totalmsrp=0;
      end;

      count+1;
      totalmsrp+msrp;

      if last.type and count > 0 then do;
        averagemsrp = totalmsrp / count;
        output;
      end;
    end;
  endthread;
```

```
    data hive.carsmsrpout (overwrite=yes);
      dcl thread work.thread1 p;
      dcl int i;
      method run();
        set from p;
        i+1;
        output;
      end;
    enddata;
  run; quit;
```

Figure 13 illustrates the steps performed by the generated Scala program when the input method is a file.

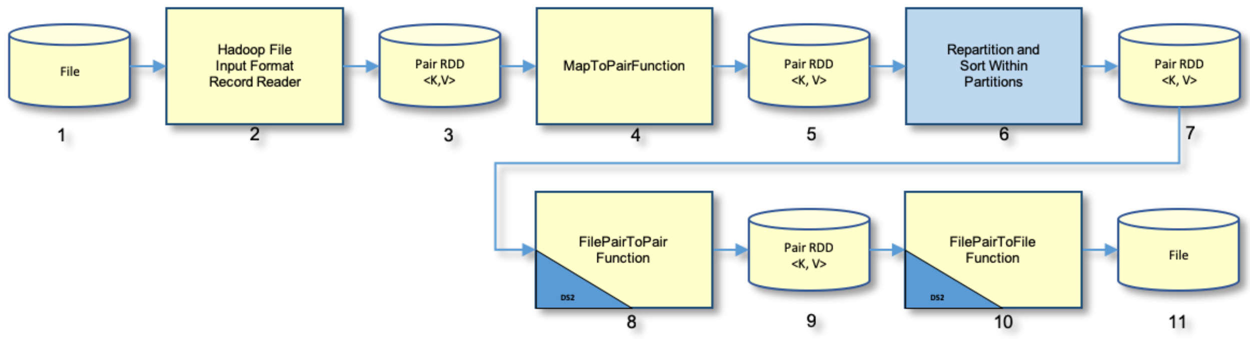

Figure 13. Case 5 - Input from File

In phase one, data is read from a file (1) using the Hadoop File Input Format and Record Reader framework (2) into a key/value pair RDD (3). Using the columns specified in the BY statement of the DS2 thread program, the pair RDD (3) is applied to the specialized function *MapToPairFunction* (4) to create a mapped pair RDD (5) with proper key and value pair. Using the key, the mapped pair RDD (5) is repartitioned with sorting within partitions (6) resulting in a new pair RDD (7) partitioned by the key, where the keys within the partitions are sorted. The partitioned pair RDD (7) is applied to the specialized EP function *FilePairToPairFunction* (8), where the DS2 thread program runs in parallel.

In phase two, the pair RDD (9) produced by the DS2 thread program is applied to the specialized EP function *FilePairToFileFunction* (10), where the DS2 data program runs a single task. The output produced by the DS2 data program is written directly to a file (11) stored in HDFS.

Figure 14 illustrates the steps performed by the generated Scala program when the input method is a table.

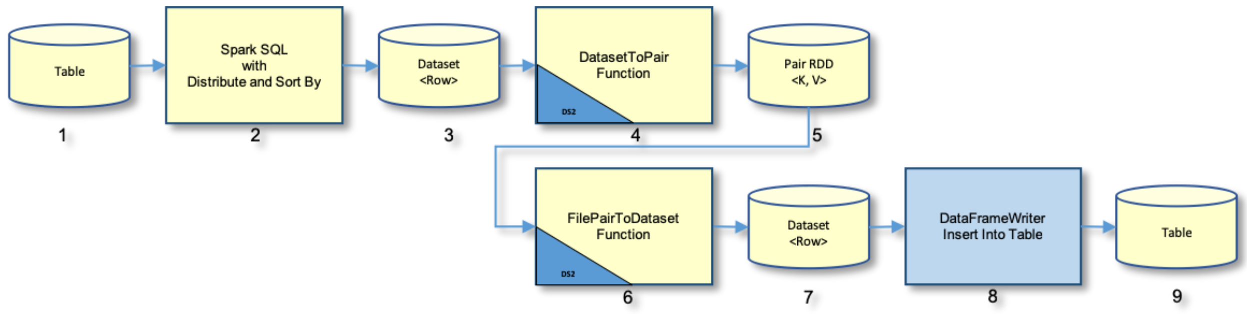
Table
1
Spark SQL with Distribute and Sort By
2
Dataset <Row>
3
DatasetToPair Function
DS2
4
Pair RDD <K, V>
5
FilePairToDataset Function
DS2
6
Dataset <Row>
7
DataFrameWriter Insert Into Table
8
Table
9

Figure 14. Case 5 - Input from Table

In phase one, the columns specified in the BY statement of the DS2 thread program are used in the Spark SQL SELECT statement with the DISTRIBUTE BY and SORT BY clauses (2). Data is read from a table (1) into a partitioned and sorted within partition Dataset of Row objects (3). The Dataset (3) is applied to the specialized EP function *DatasetToPairFunction* (4), where the DS2 thread program is executed in parallel.

In phase two, the pair RDD (5) produced by the DS2 thread program is applied to the specialized EP function *FilePairToDatasetFunction* (6), where the DS2 data program runs a single task. The rows in the output Dataset (7) produced by the DS2 data program are inserted into the output table (9) using the *DataFrameWriter* interface (8).

SAS IN-DATABASE CODE ACCELERATOR CASE 6

Table 6 illustrates the deciding factors for Case 6:

Thread Program	Thread BY Statement	Data Program	Data BY Statement
YES	YES	YES	YES

Table 6. SAS In-Database Code Accelerator Case 6

This is the most complex case where both DS2 thread and data program contain a BY statement and the entire DS2 program runs in two phases. Case 6 phase one is similar to Case 5 phase one. The DS2 thread program is executed in parallel. The DS2 data program is executed in one single task. Here is an example of a Case 6 DS2 program:

```
proc ds2 ds2accel=yes;
  thread work.thread1 / overwrite=yes;
    dcl double count;
    dcl double averagemsrp;
    dcl double totalmsrp;
    keep make type averagemsrp;

    method run();
      set hive.cars;
      by make type;

      if first.type then do;
        count=0;
        totalmsrp=0;
      end;

      count+1;
      totalmsrp+msrp;

      if last.type and count > 0 then do;
        averagemsrp = totalmsrp / count;
        output;
      end;
    end;
  endthread;

  data hive.carsmsrpout (overwrite=yes);
    dcl thread work.thread1 p;
    method run();
      set from p;
      by make;
```

```
         output;
      end;
   enddata;
run; quit;
```

Figure 15 illustrates the steps performed by the generated Scala program when the input method is file.

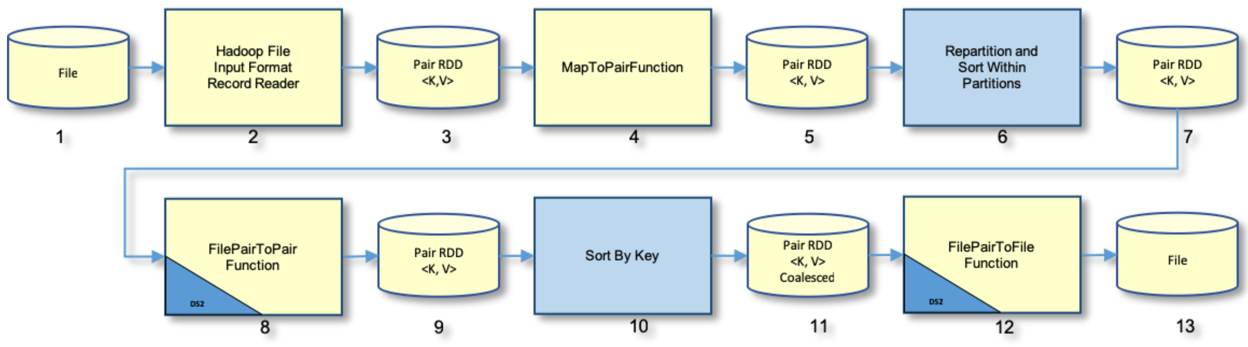

Figure 15. Case 6 - Input from File

In phase one, data are read from a file (1) using the Hadoop File Input Format and Record Reader framework (2) into a key/value pair RDD (3). Using the columns specified in the BY statement of the DS2 thread program, the pair RDD (3) is applied to the specialized function *MapToPairFunction* (4) to create a mapped pair RDD (5) with proper key and value pair. Using the key, the mapped pair RDD (5) is repartitioned with sorting within partitions (6) resulting in a new pair RDD (7) partitioned by the key, where the keys within the partitions are sorted. The partitioned pair RDD (7) is applied to the specialized EP function *FilePairToPairFunction* (8), where the DS2 thread program runs in parallel.

In phase two, the pair RDD (9) produced by the DS2 thread program is sorted and coalesced (10) into a new pair RDD (11), which is applied to the specialized EP function *FilePairToFileFunction* (12), where the DS2 data program runs a single task. The output produced by the DS2 data program is written directly to a file (13) stored in HDFS.

Figure 16 illustrates the steps performed by the generated Scala program when the input method is a file.

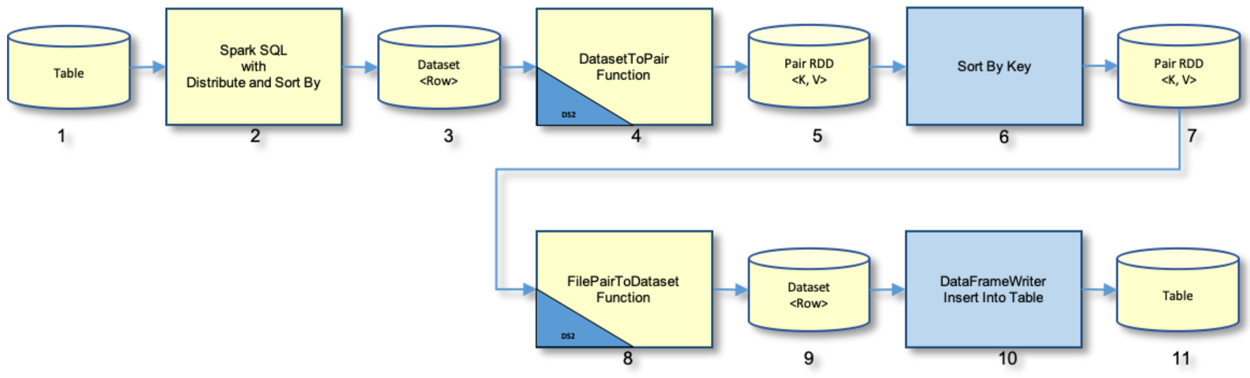

Figure 16. Case 6 - Input from Table

In phase one, the columns specified in the BY statement of the DS2 thread program are used in the Spark SQL SELECT statement with the DISTRIBUTE BY and SORT BY clauses (2). Data is read from a table (1) into a partitioned and sorted within partition Dataset of

Row objects (3). The Dataset (3) is applied to the specialized EP function *DatasetToPairFunction* (4), where the DS2 thread program is executed in parallel.

In phase two, the pair RDD (5) produced by the DS2 thread program is sorted and coalesced (6) into a new pair RDD (7), which is applied to the specialized EP function *FilePairToDatasetFunction* (8), where the DS2 data program runs a single task. The rows in the output Dataset (9) produced by the DS2 data program are inserted into the output table (11) using the *DataFrameWriter* interface (10).

CONCLUSION

The SAS In-Database Code Accelerator enables you to run your DS2 code inside the Spark framework. Parallel processing and data proximity are fundamental factors to achieve faster results. The challenges imposed by the big data era can be minimized by using SAS In-Database Technologies for Spark.

SAS provides the platform you need to process your data in an effective and efficient manner by using the massively parallel processing power of Apache Spark. You are now ready to collect, store, and process your data with confidence using the power of SAS.

This paper has prepared you to explore the SAS In-Database Code Accelerator for Hadoop using Apache Spark as the execution platform. It has explained the internal components of the SAS Embedded Process and the many different ways Code Accelerator is executed within it.

REFERENCES

- 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.
- Secosky, Jason, et al. 2014. "Parallel Data Preparation with the DS2 Programming Language", *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available: http://support.sas.com/resources/papers/proceedings14/SAS329-2014.pdf.
- Ray, Robert, and William Eason. 2016. "Data Analysis with User-Written DS2 Packages", *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available: http://support.sas.com/resources/papers/proceedings16/SAS6462-2016.pdf
- Apache Spark Documentation. Available: https://spark.apache.org/docs/latest/configuration.html

RECOMMENDED READING

- SAS Institute Inc. *SAS® DS2 Programmer's Guide*. Cary, NC: SAS Institute, Inc. Available: https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=ds2pg&docsetTarget=ds2pgwhatsnew94.htm
- SAS Institute Inc. *SAS® In-Database User's Guide*. Eighth Edition. Cary, NC: SAS Institute, Inc. Available: https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=indbug&docsetTarget=titlepage.htm
- SAS Institute Inc. *SAS® In-Database Administrator's Guide*. Ninth Edition. Cary, NC: SAS Institute, Inc. Available: https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=indbag&docsetTarget=titlepage.htm

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.
(919) 531-7416
david.ghazaleh@sas.com