HPCC / Spark Connector

Boca Raton Documentation Team

HPCC / Spark Connector

Boca Raton Documentation Team

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com>

Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Version Number in the text of the message.

2023 Version 9.0.0-1

The Spark HPCC Systems Connector

Overview

The Spark-HPCCSystems Distributed Connector is a Java library that facilitates access from a Spark cluster to data stored on an HPCC Systems cluster. The connector library employs the standard HPCC Systems remote file read facility to read data from either sequential or indexed HPCC datasets.

The data on an HPCC cluster is partitioned horizontally, with data on each cluster node. Once configured, the HPCC data is available for reading in parallel by the Spark cluster.

In the GitHub repository (https://github.com/hpcc-systems/Spark-HPCC) you can find the source code and examples. There are several artifacts in the DataAccess/src/main/java folder of primary interest. The *org.hpccsystems.spark.HpccFile* class is the façade of a file on an HPCC Cluster. The *org.hpccsystems.spark.HpccRDD* is a resilient distributed dataset derived from the data on the HPCC Cluster and is created by the *org.hpccsystems.spark.HpccFile.getRDD*(…) method. The *HpccFile* class supports loading data to construct a *Dataset<Row>* object for the Spark interface. This will first load the data into an RDD<Row> and then convert this RDD to a Dataset<Row> through internal Spark mechanisms.

There are several additional artifacts of some interest. The *org.hpccsystems.spark.ColumnPruner* class is provided to enable retrieving only the columns of interest from the HPCC Cluster. The *targetCluster* artifact allows you to specify the HPCC cluster on which the target file exists. The *org.hpccsystems.spark.thor.FileFilter* class is provided to facilitate filtering records of interest from the HPCC Cluster.

The git repository includes two examples under the Examples/src/main/scala folder. The examples (*org.hpccsystems.spark_examples.Dataframe_Iris_LR* and *org.hpccsystems.spark_examples.Iris_LR*) are Scala Objects with a main() function. Both examples use the classic Iris dataset. The dataset can be obtained from a variety of sources, including the HPCC-Systems/ecl-ml repository. IrisDs.ecl (can be found under the ML/Tests/Explanatory folder: https://github.com/hpcc-systems/Spark-HPCC/blob/master/Examples/src/main/ecl/IrisDS.ecl) can be executed to generate the Iris dataset in HPCC. A walk-through of the examples is provided in the Examples section.

The Spark-HPCCSystems Distributed Connector also supports PySpark. It uses the same classes/API as Java does.

As is common in Java client communication over TLS, Spark-HPCC connectors targeting an HPCC cluster over TLS will need to import the appropriate certificates to local Java keystore.

*One way to accomplish this is to use the keytool packaged with Java installations. Refer to the keytool documentation for usage.

Spark Integration

The HPCC integrated Spark plugin is no longer supported as of version 9.0.0 in favor of stand-alone user-managed Spark clusters linked to the HPCC platform using the Spark-HPCC connector.

Special considerations

Unsigned Value Overflow

Java does not support an unsigned integer type so reading UNSIGNED8 values from HPCC data can cause an integer overflow in Java. UNSIGNED8 values are often used as unique identifiers in datasets, in which case overflowing would be acceptable as the overflowed value will still be unique.

The Spark-HPCC connector allows unsigned values to overflow in Java and will not report a exception. The caller is responsible for interpreting the value based on the recdef **isunsigned** flag.

Primary Classes

The *HpccFile* class and the *HpccRDD* classes are discussed in more detail below. These are the primary classes used to access data from an HPCC Cluster. The *HpccFile* class supports loading data to construct a *Dataset<Row>* object for the Spark interface. This will first load the data into an RDD<Row> and then convert this RDD to a Dataset<Row> through internal Spark mechanisms.

The *org.hpccsystems.spark.HpccFile* class has several constructors. All of the constructors take information about the Cluster and the name of the dataset of interest. The JAPI WS-Client classes are used to access file detail information. A definition used to select the columns to be returned and a definition to select the rows to be returned could also be supplied. These are discussed in the *Additional Classes of Interest* section below. The class has two methods of primary interest: the *getRDD(…)* method and the *getDataframe(…)* method, which are illustrated in the *Example* section.

The *HpccFile* class *getRecordDefinition()* method can be used to retrieve a definition of the file. The *getFileParts()* method can be used to see how the file is partitioned on the HPCC Cluster. These methods return the same information as can be found on the ECL Watch dataset details page DEF tab and the PARTS tab respectively.

The *org.hpccsystems.spark.HpccRDD* class extends the *RDD<Record>* templated class. The class employs the *org.hpccsystems.spark.HpccPart* class for the Spark partitions. The *org.hpccsystems.spark.Record* class is used as the container for the fields from the HPCC Cluster. The *Record* class can create a *Row* instance with a schema.

The *HpccRDD* HpccPart partition objects each read blocks of data from the HPCC Cluster independently from each other. The initial read fetches the first block of data, requests the second block of data, and returns the first record. When the block is exhausted, the next block should be available on the socket and new read request is issued.

The *HpccFileWriter* is another primary class used for writing data to an HPCC Cluster. It has a single constructor with the following signature:

```
public HpccFileWriter(String connectionString, String user, String pass) throws Exception {
```

The first parameter *connectionString* contains the same information as *HpccFile*. It should be in the following format: {http|https}://{ECLWATCHHOST}:{ECLWATCHPORT}

The constructor will attempt to connect to HPCC. This connection will then be used for any subsequent calls to *saveToHPCC*.

```
public long saveToHPCC(SparkContext sc, RDD<Row> scalaRDD, String clusterName,
                       String fileName) throws Exception {
```

The *saveToHPCC* method only supports RDD<row> types. You may need to modify your data representation to use this functionality. However, this data representation is what is used by Spark SQL and by HPCC. This is only supported by writing in a co-located setup. Thus Spark and HPCC must be installed on the same nodes. Reading only supports reading data in from a remote HPCC cluster.

The *clusterName* as used in the above case is the desired cluster to write data to, for example, the "mythor" Thor cluster. Currently there is only support for writing to Thor clusters. Writing to a Roxie cluster is not supported and will return an exception. The filename as used in the above example is in the HPCC format, for example: "~example::text".

Internally the saveToHPCC method will Spawn multiple Spark jobs. Currently, this spawns two jobs. The first job maps the location of partitions in the Spark cluster so it can provide this information to HPCC. The second job does the actual writing of files. There are also some calls internally to ESP to handle things

like starting the writing process by calling *DFUCreateFile* and publishing the file once it has been written by calling *DFUPublishFile*.

Using the Spark Datasource API to Read and Write

Example Python code:

```
# Connect to HPCC and read a file
df = spark.read.load(format="hpcc",
                     host="127.0.0.1:8010",
                     password="",
                     username="",
                     limitPerFilePart=100,
                                    # Limit the number of rows to read from each file part
                     projectList="field1, field2, field3.childField1",
                                    # Comma separated list of columns to read
                     fileAccessTimeout=240,
                     path="example::file")
# Write the file back to HPCC
df.write.save(format="hpcc",
              mode="overwrite",
                   # Left blank or not specified results in an error if the file exists
              host="127.0.0.1:8010",
              password="",
              username="",
              cluster="mythor",
              path="example::file")
```

Example Scala code:

```
// Read a file from HPCC
val dataframe = spark.read.format("hpcc")
                .option("host","127.0.0.1:8010")
                .option("password", "")
                .option("username", "")
                .option("limitPerFilePart",100)
                .option("fileAccessTimeout",240)
                .option("projectList","field1, field2, field3.childField")
                .load("example::file")
// Write the dataset back
   dataframe.write.mode("overwrite")
                  .format("hpcc")
                  .option("host","127.0.0.1:8010")
                  .option("password", "")
                  .option("username", "")
                  .option("cluster","mythor")
                  .save("example::file")
```

Example R code:

```
df <- read.df(source = "hpcc",
              host = "127.0.0.1:8010",
              path = "example::file",
              password = "",
              username = "",
              limitPerFilePart = 100,
              fileAccessTimeout = 240,
              projectList = "field1, field2, field3.childField")
write.df(df, source = "hpcc",
             host = "127.0.0.1:8010",
             cluster = "mythor",
             path = "example::file",
             mode = "overwrite",
             password = "",
             username = "",
             fileAccessTimeout = 240)
```

Additional Classes of interest

The main classes of interest for this section are column pruning and file filtering. In addition there is a helper class to remap IP information when required, and this is also discussed below.

The column selection information is provided as a string to the *org.hpccsystems.spark.ColumnPruner* object. The string is a list of comma separated field names. A field of interest could contain a row or child dataset, and the dotted name notation is used to support the selection of individual child fields. The *ColumnPruner* parses the string into a root *TargetColumn* class instance which holds the top level target columns. A *TargetColumn* can be a simple field or can be a child dataset and so be a root object for the child record layout.

The row filter is implemented in the *org.hpccsystems.spark.thor.FileFilter* class. A *FileFilter* instance is constricted from an array of *org.hpccsystems.spark.thor.FieldFilter* objects. Each *FieldFilter* instance is composed of a field name (in doted notation for compound names) and an array of *org.hpccsystems.spark.thor.FieldFilterRange* objects. Each *FieldFilterRange* instance can be an open or closed interval or a single value. The record is selected when at least one *FieldFilterRange* matches for each of the *FieldFilter* instances in the array.

The *FieldFilterRange* values may be either strings or numbers. There are methods provided to construct the following range tests: equals, not equals, less than, less than or equals, greater than, and a greater than or equals. In addition, a set inclusion test is supported for strings. If the file is an index, the filter fields that are key fields are used for an index lookup. Any filter field unmentioned is treated as wild.

The usual deployment architecture for HPCC Clusters consists of a collection of nodes on a network. The file management information includes the IP addresses of the nodes that hold the partitions of the file. The Spark-HPCC connector classes use these IP addresses to establish socket connections for the remote read. An HPCC Cluster may be deployed as a virtual cluster with private IP addresses. This works for the cluster components because they are all on the same private LAN. However, the Spark cluster nodes might not be on that same LAN. In this case, the *org.hpccsystems.spark.RemapInfo* class is used to define the information needed to change the addressing. There are two options that can be used. The first option is that each Thor worker node can be assigned an IP that is visible to the Spark cluster. These addresses must be a contiguous range. The second option is to assign an IP and a contiguous range of port numbers. The *RemapInfo* object is supplied as a parameter.

Examples

We will walk through the two examples below utilizing a Spark environment. Additionally, the repository provides testing programs (in the DataAccess/src/test folder) that can be executed as stand-alone examples.

These test programs are intended to be run from a development IDE such as Eclipse via the Spark-submit application whereas the examples below are dependent on the Spark shell.

The following examples assume a Spark Shell. You can use the spark-submit command if you intend to compile and package these examples. To properly connect your spark shell to the Integrated Spark cluster, provide the following parameters when starting the shell:

```
bin/spark-shell  \
 --master=<spark://{remotesparkhost-IP}:{sparkport}> --conf="spark.driver.host={localhost-ip}"
```

Iris_LR

This example assumes that you have Spark Shell running. The next step is to establish your HpccFile and your RDD for that file. You need the name of the file, the protocol (http or https), the name or IP of the ESP, the port for the ESP (usually 8010), and your user account and password. The *sc* value is the *SparkContext* object provided by the shell.

```
val espcon = new Connection("http", "myeclwatchhost", "8010");
espcon.setUserName("myuser");
espcon.setPassword("mypass");
val file = new HpccFile("myfile",espcon);
```

Now we have an RDD of the data. Nothing has actually happened at this point because Spark performs lazy evaluation and there is nothing yet to trigger an evaluation.

The Spark MLLib has a Logistic Regression package. The MLLib Logistic Regression expects that the data will be provided as Labeled Point formatted records. This is common for supervised training implementations in MLLib. We need column labels, so we make an array of names. We then make a labeled point RDD from our RDD. This is also just a definition. Finally, we define the Logistic Regression that we want to run. The column names are the field names in the ECL record definition for the file, including the name “class” which is the name of the field holding the classification code.

```
val names = Array("petal_length","petal_width", "sepal_length",
                  "sepal_width")
var lpRDD = myRDD.makeMLLibLabeledPoint("class", names)
val lr = new LogisticRegressionWithLBFGS().setNumClasses(3)
```

The next step is to define the model, which is an action and will cause Spark to evaluate the definitions.

```
val iris_model = lr.run(lpRDD)
```

Now we have a model. We will utilize this model to take our original dataset and use the model to produce new labels. The correct way to do this is to have randomly sampled some hold out data. We are just going to use the original dataset because it is easier to show how to use the connector. We then take our original data and use a map function defined inline to create a new record with our prediction value and the original classification.

```
val predictionAndLabel = lpRDD.map {
  case LabeledPoint(label, features) =>
    val prediction = iris_model.predict(features)
  (prediction, label)
}
```

The *MulticlassMetrics* class can now be used to produce a confusion matrix.

```
val metrics = new MulticlassMetrics(predictionAndLabel)
metrics.confusionMatrix
```

Dataframe_Iris_LR

The Dataframe_Iris_LR is similar to the Iris_LR, except that a Dataframe is used and the new ML Spark classes are used instead of the old MLLib classes. Since ML is not completely done, we do fall back to an MLLib class to create our confusion matrix.

Once the Spark shell is brought up, we need our import classes.

```
import org.hpccsystems.spark.HpccFile
import org.apache.spark.sql.Dataset
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.mllib.evaluation.MulticlassMetrics
```

The next step is to establish the *HpccFile* object and create the Dataframe. The *spark* value is a *SparkSession* object supplied by the shell and is used instead of the *SparkContext* object.

```
val espcon = new Connection("http", "myeclwatchhost", "8010");
espcon.setUserName("myuser");
espcon.setPassword("mypass");
val file = new HpccFile("myfile",espcon);
```

The Spark *ml* Machine Learning classes use different data container classes. In the case of Logistic Regression, we need to transform our data rows into a row with a column named “features” holding the features and a column named “label” holding the classification label. Recall that our row has “class”, “sepal_width”, “sepal_length”, “petal_width”, and “petal_length” as the column names. This kind of transformation can be accomplished with a *VectorAssembler* class.

```
val assembler = new VectorAssembler()
assembler.setInputCols(Array("petal_length","petal_width",
                             "sepal_length", "sepal_width"))
assembler.setOutputCol("features")
val iris_fv = assembler.transform(my_df)
                             .withColumnRenamed("class", "label")
```

Now that the data (*iris_fv*) is ready, we define our model and fit the data.

```
val lr = new LogisticRegression()
val iris_model = lr.fit(iris_fv)
```

We now want to apply our prediction and evaluate the results. As noted before, we would use a holdout dataset to perform the evaluation. We are going to be lazy and just use the original data to avoid the sampling task. We use the *transform(…)* function for the model to add the prediction. The function adds a column named “prediction” and defines a new dataset. The new Machine Learning implementation lacks a metrics capability to produce a confusion matrix, so we will then take our dataset with the *prediction* column and create a new RDD with a dataset for a *MulticlassMetrics* class.

```
val with_preds = iris_model.transform(iris_fv)
val predictionAndLabel = with_preds.rdd.map(
r => (r.getDouble(r.fieldIndex("prediction")),
r.getDouble(r.fieldIndex("label"))))
val metrics = new MulticlassMetrics(predictionAndLabel)
metrics.confusionMatrix
```