

# **HPCC / Integração com Spark**

**Equipe de documentação de Boca Raton**



## HPCC / Integração com Spark

Equipe de documentação de Boca Raton

Copyright © 2023 HPCC Systems®. All rights reserved

Sua opinião e comentários sobre este documento são muito bem-vindos e podem ser enviados por e-mail para <docfeedback@hpccsystems.com>

Inclua a frase **Feedback sobre documentação** na linha de assunto e indique o nome do documento, o número das páginas e número da versão atual no corpo da mensagem.

LexisNexis e o logotipo Knowledge Burst são marcas comerciais registradas da Reed Elsevier Properties Inc., usadas sob licença.

HPCC Systems® é uma marca registrada da LexisNexis Risk Data Management Inc.

Os demais produtos, logotipos e serviços podem ser marcas comerciais ou registradas de suas respectivas empresas.

Todos os nomes e dados de exemplo usados neste manual são fictícios. Qualquer semelhança com pessoas reais, vivas ou mortas, é mera coincidência.

2023 Version 8.10.70-1

HPCC / Instalação e Configuração com Spark .....	4
Instalação do Spark .....	4
Configuração do Spark .....	5
O conector Spark HPCC Systems .....	8
Visão geral .....	8
Classes Primárias .....	11
Classes Adicionais de Interesse .....	14
Exemplos .....	15

# HPCC / Instalação e Configuração com Spark

O plug-in Spark do HPCC Systems, `hpccsystems-plugin-spark` integra o Spark a sua plataforma HPCC Systems. Uma vez instalado e configurado, o componente Sparkthor gerencia o cluster Spark Integrado. Ele configura, inicia e interrompe dinamicamente o cluster do Spark Integrado ao iniciar ou interromper a plataforma do HPCC Systems.

## Instalação do Spark

Para adicionar a integração do Spark ao seu cluster HPCC System, você deve ter um cluster HPCC executando a versão 7.0.0 ou posterior. O Java 8 também é necessário. Você precisará configurar o componente Sparkthor. O componente Sparkthor precisa estar associado a um cluster Thor existente válido. Os nós de trabalho do Spark serão criados ao lado de cada nós de trabalho Thor. O nó do Integrated Spark Manager será designado durante a configuração, junto com quaisquer outros recursos do nó do Spark. Em seguida, o componente Sparkthor gerará um cluster Spark Integrado na inicialização. Você também terá um conector jar SPARK-HPCC disponível.

Para obter o componente Spark Integrado, os pacotes e plug-ins estão disponíveis no portal do HPCC Systems®: <https://hpccsystems.com/download/>

Faça o download do pacote `hpccsystems-plugin-spark` no Portal do HPCC Systems.

## Instalando o Plug-in Spark

O processo de instalação e o pacote a ser feito o download variam de acordo com o sistema operacional que você planeja usar. Os pacotes de instalação não serão instalados com sucesso se suas dependências não estiverem presentes no sistema de destino. Para instalar o pacote, siga as instruções de instalação apropriadas para o seu sistema operacional:

### CentOS/Red Hat

Para sistemas baseados em RPM, você pode instalar utilizando o yum.

```
sudo yum install <hpccsystems-plugin-spark>
```

### Ubuntu/Debian

Para instalar um pacote Ubuntu/Debian, use:

```
sudo dpkg -i <hpccsystems-plugin-spark>
```

Após instalar o pacote, execute o seguinte para atualizar quaisquer dependências.

```
sudo apt-get install -f
```

- Você precisa copiar e instalar o plug-in em todos os nós. Isso pode ser feito usando o script `install-cluster.sh` fornecido com o HPCC. Use o comando a seguir:

```
sudo /opt/HPCCSystems/sbin/install-cluster.sh <hpccsystems-plugin-spark>
```

Mais detalhes, incluindo outras opções que podem ser usadas com este comando, estão incluídos no apêndice Instalando e executando a plataforma HPCC, também disponível no portal da web HPCC Systems®.

# Configuração do Spark

Para configurar seu HPCC System para integrar o Spark, instale o pacote `hpccsystems-plugin-spark` e modifique o ambiente existente (arquivo) para adicionar o componente Sparkthor.

1. Caso esteja em execução, pare o HPCC System usando este comando em uma janela de terminal:

```
sudo systemctl stop hpccsystems-platform.target
```



Este comando pode ser usado para confirmar que os processos do HPCC foram interrompidos:

```
sudo systemctl status hpccsystems-platform.target
```

2. Inicie o serviço do Gerenciador de Configurações.

```
sudo /opt/HPCCSystems/sbin/configmgr
```

```
node219008 ~]$ sudo /opt/HPCCSystems/sbin/configmgr
Using default filename /etc/HPCCSystems/source/environment.xml and default port
"8015"
Validating environment file /etc/HPCCSystems/source/environment.xml using config
gen ... Success
Verifying configmgr startup ... Success
Exit by pressing ctrl-c...
```

3. Deixe esta janela aberta. Se desejar, você pode minimizá-la.
4. Usando um navegador de Internet, acesse a interface do Gerenciador de Configurações:

```
http://<node ip>:8015
```


5. Marque a caixa Advanced View e selecione o arquivo de ambiente a ser editado.
6. Ative o acesso de gravação (caixa de seleção na parte superior direita da página)
7. Clique com o botão direito no painel Navigator no lado esquerdo.

Escolher **Novos componentes**, em seguida, escolha **Sparkthor**

8. Configure os atributos da sua instância do Spark:


atributo	descrição	default	obrigatório
name	Nome para este processo	mysparkthor	obrigatório
ThorClusterName	Cluster Thor para se conectar a*	mythor*	obrigatório
SPARK_EXECUTOR_CORES	Número de núcleos para executores	1	opcional
SPARK_EXECUTOR_MEMORY	Memória por executor	1G	opcional
SPARK_MASTER_WEBUI_PORT	Porta base a ser usada para a interface web principal	8080	opcional
SPARK_MASTER_PORT	Porta base a ser usada pela principal	7077	opcional
SPARK_WORKER_CORES	Número de núcleos para workers	1	opcional
SPARK_WORKER_MEMORY	Memória por worker	1G	opcional
SPARK_WORKER_PORT	Porta base a ser usada para os workers	7071	opcional

\*ThorClusterName segmenta um cluster Thor existente. Ao configurar, você deve escolher um cluster Thor válido existente para o cluster Spark Integrado espelhar.


	<b>OBSERVAÇÃO:</b> Você deve deixar pelo menos dois núcleos abertos para o HPCC fornecer o Spark com dados. O número de núcleos e memória alocados para o Spark dependerá da carga de trabalho. Não tente alocar muitos recursos para o Spark, onde você poderia ter problemas com o HPCC e o Spark conflitando em busca de recursos.
--	---

9. Especifique um Nó Principal do Spark; Selecione a aba Instances. Clique com o botão direito do mouse na tabela Instances e escolha **Add Instances**

Adicione a instância do nó principal do Spark.

	<b>OBSERVAÇÃO:</b> Você só pode ter uma instância principal do Spark
---	--

10. Salve o arquivo do ambiente. Saia do configmgr (Ctrl+C). Copie o arquivo de ambiente do diretório de origem para o diretório /etc/HPCCSystems.

	Certifique-se de que o sistema não esteja em execução antes de tentar mover o arquivo environment.xml.
---	--

```
sudo cp /etc/HPCCSystems/source/<new environment file.xml>
/etc/HPCCSystems/environment.xml
```

e distribua o arquivo do ambiente para todos os nós em seu cluster

Você pode optar por usar o script hpcc-push.sh fornecido para implantar o novo arquivo de ambiente. Por exemplo:

```
sudo /opt/HPCCSystems/sbin/hpcc-push.sh -s <sourcefile> -t <destinationfile>
```

Agora você pode iniciar seu cluster HPCC Systems e verificar se o Sparkthor está ativo.

Para iniciar seu sistema HPCC.

```
sudo systemctl start hpccsystems-platform.target
```

Usando um navegador, navegue para a sua instância do Spark's Manager (a instância que você adicionou acima) em execução na porta 8080 do seu HPCC Systems.

Por exemplo, `http://nnn.nnn.nnn.nnn:8080`, em que `nnn.nnn.nnn.nnn` é o endereço IP do nó do Integrated Spark Manager.

```
https://192.168.56.101:8080
```

## Opções de configuração Cluster Integrado do Spark.

Além das opções de configuração disponíveis através do gerenciador de configuração do HPCC Systems, existem opções para casos extremos e configurações mais avançadas. Para customizar seu ambiente de cluster do Spark Integrado para utilizar essas opções adicionais, use o script **spark-env.sh** fornecido .

```
/etc/HPCCSystems/externals/spark-hadoop/spark-env.sh
```

Para obter mais informações sobre as opções do Spark Cluster, consulte as páginas a seguir.

- <https://spark.apache.org/docs/latest/spark-standalone.html#cluster-launch-scripts>
- <https://spark.apache.org/docs/latest/configuration.html#environment-variables>

## Exemplo de Casos de Uso

- O Spark atualmente requer que o Java 8 seja executado. Em um sistema em que a instalação padrão do Java não é o Java 8. A variável de ambiente `JAVA_HOME` pode ser usada para definir a versão do Spark Java para o Java 8.
- Geralmente, quando um job é executado em um cluster Spark, ele assume a propriedade de todos os nós de trabalho. Em um ambiente de cluster compartilhado, isso pode não ser o ideal. Usando o atributo `SPARK_MASTER_OPTS` é possível definir um limite para o número de nós de workers que um job pode utilizar.

# O conector Spark HPCC Systems

## Visão geral

O conector distribuído Spark-HPCCSystems é uma biblioteca Java que facilita o acesso de um cluster do Spark aos dados armazenados em um cluster do HPCC Systems. A biblioteca de conectores emprega o recurso de leitura de arquivos remotos padrão do HPCC Systems para ler dados de conjuntos de datasets sequenciais ou indexados do HPCC.

Os dados em um cluster HPCC são particionados horizontalmente, com dados em cada nó do cluster. Depois de configurados, os dados do HPCC estão disponíveis para acesso em paralelo pelo cluster do Spark.

No repositório do GitHub (<https://github.com/hpcc-systems/Spark-HPCC>) você pode encontrar o código-fonte e exemplos. Existem vários artefatos na pasta `DataAccess/src/main/java` de interesse primário. A classe `org.hpccsystems.spark.HpccFile` é a fachada de um arquivo em um cluster HPCC. O `org.hpccsystems.spark.HpccRDD` é um dataset distribuído resiliente derivado dos dados no cluster HPCC e é criado pelo método `org.hpccsystems.spark.HpccFile.getRDD(...)`. A classe `HpccFile` suporta o carregamento de dados para construir um objeto `Dataset <Row>` para a interface Spark. Isso primeiro carregará os dados em um RDD `<Row>` e, em seguida, converterá esse RDD em um `DataSet <Row>` por meio dos mecanismos internos do Spark.

Existem vários artefatos adicionais de algum interesse. A classe `org.hpccsystems.spark.ColumnPruner` é fornecida para permitir a recuperação somente das colunas de interesse do cluster HPCC. O artefato `targetCluster` permite especificar o cluster HPCC no qual o arquivo de destino existe. A classe `org.hpccsystems.spark.thor.FileFilter` é fornecida para facilitar a filtragem de registros de interesse do cluster HPCC.

O repositório git inclui dois exemplos na pasta `Examples/src/main/scala` folder. Os exemplos (`org.hpccsystems.spark_examples.DataFrame_Iris_LR` e `org.hpccsystems.spark_examples.Iris_LR`) são Scala Objects com uma função `main()`. Ambos os exemplos usam o dataset clássico da Iris. O dataset pode ser obtido de uma variedade de fontes, incluindo o repositório HPCC-Systems/ecl-ml. `IrisDs.ecl` (pode ser encontrado na pasta `ML/Tests/Explanatory`: <https://github.com/hpcc-systems/Spark-HPCC/blob/master/Examples/src/main/ecl/IrisDS.ecl>) pode ser executado para gerar o dataset Iris no HPCC. Um passo a passo dos exemplos é fornecido na seção Exemplos.

O conector distribuído Spark-HPCCSystems também suporta o PySpark. Ele usa as mesmas classes/API que o Java.



Como é comum na comunicação do cliente Java por TLS, os conectores Spark-HPCC direcionados a um cluster HPCC por TLS precisarão importar os certificados apropriados para o keystore Java local.

\*Uma maneira de fazer isso é usar o keytool fornecido com as instalações Java. Consulte a documentação do keytool para uso.



## Considerações Especiais

### Estouro de valor não assinado

Java não suporta um tipo de inteiro não assinado, portanto, a leitura de valores UNSIGNED8 dos dados do HPCC pode causar um estouro de inteiro em Java. Os valores de UNSIGNED8 são frequentemente usados como identificadores exclusivos em datasets, caso em que o overflow seria aceitável, pois o valor do transbordamento ainda será exclusivo.

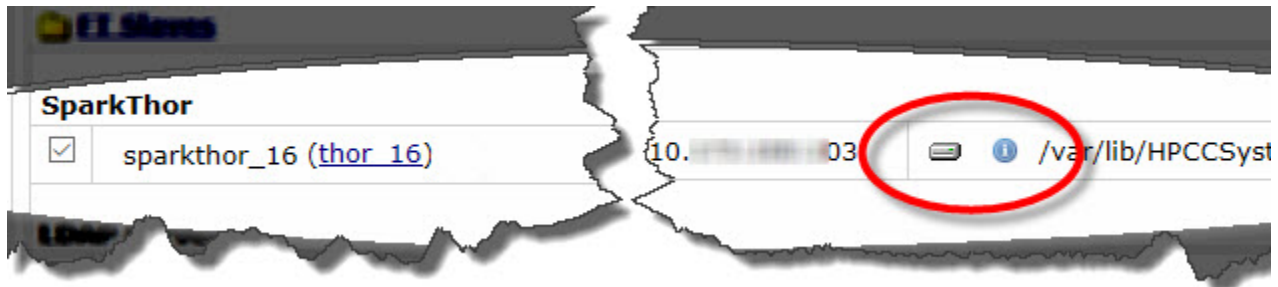
O conector Spark-HPCC permite que os valores não assinados sejam excedidos em Java e não relatará uma exceção. O chamador é responsável por interpretar o valor com base no sinalizador recef **isunsigned**.

## Suporte Spark no ECL Watch

Como parte dos HPCC Systems, o conector Spark pode ser monitorado a partir da interface ECL Watch. Consulte a seção *ECL Watch* consulte o manual Como usar o ECL Watch para obter mais detalhes.

O cluster do **SparkThor** é listado na página Servidores do Sistema do ECL Watch. Para acessar a página Systems Servers:

- No ECL Watch, clique no ícone/link de operações
- Clique na aba System Servers



Isso informa informações para o nó principal do cluster Spark integrado. Aqui você pode verificar se o cluster está ativo e em execução e executar a verificação prévia selecionando o cluster do SparkThor e pressionando o botão Submit na parte inferior da página.

Clique no ícone do disco ao lado do cluster do SparkThor para acessar seus registros.

Clique no ícone azul de informações para visualizar informações mais integradas do cluster do Spark.

## Classes Primárias

A classe *HpccFile* e as classes *HpccRDD* são discutidas em mais detalhes abaixo. Essas são as classes principais usadas para acessar dados de um cluster HPCC. A classe *HpccFile* suporta o carregamento de dados para construir um objeto *Dataset <Row>* para a interface Spark. Isso primeiro carregará os dados em um RDD *<Row>* e, em seguida, converterá esse RDD em um *DataSet <Row>* por meio dos mecanismos internos do Spark.

A classe *org.hpccsystems.spark.HpccFile* possui vários construtores. Todos os construtores recebem informações sobre o cluster e o nome do dataset de interesse. As classes JAPI WS-Client são usadas para acessar informações detalhadas do arquivo. Uma definição usada para selecionar as colunas a serem retornadas e uma definição para selecionar as linhas a serem retornadas também podem ser fornecidas. Eles são discutidos na seção *Classes Adicionais de Interesse* abaixo. A classe tem dois métodos de interesse primário: o método *getRDD(...)* e o método *getDataframe(...)*, que são ilustrados na seção *Exemplo*.

O método *getRecordDefinition()* da classe *HpccFile* pode ser usado para recuperar uma definição do arquivo. O método *getFileParts()* pode ser usado para ver como o arquivo é particionado no cluster HPCC. Esses métodos retornam as mesmas informações que podem ser encontradas na aba DEF da página de detalhes do dataset do ECL Watch e na aba PARTS respectivamente.

A classe *org.hpccsystems.spark.HpccRDD* estende a classe de modelo *RDD<Record>*. A classe emprega a *org.hpccsystems.spark.HpccPart* para as partições Spark. A classe *org.hpccsystems.spark.Record* é usada como o contêiner para os campos do cluster HPCC. A classe *Record* pode criar uma instância *Row* com um esquema.

Os objetos de partição *HpccRDD HpccPart* leem cada um blocos de dados do cluster HPCC independentemente uns dos outros. A leitura inicial busca o primeiro bloco de dados, solicita o segundo bloco de dados e retorna o primeiro registro. Quando o bloco estiver esgotado, o próximo bloco deverá estar disponível no soquete e uma nova solicitação de leitura será emitida.

O *HpccFileWriter* é outra classe primária usada para gravar dados em um cluster HPCC. Tem um único construtor com a seguinte assinatura:

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

O primeiro parâmetro *connectionString* contém as mesmas informações que o *HpccFile*. Deve estar no seguinte formato: {http|https}://{ECLWATCHHOST}:{ECLWATCHPORT}

O construtor tentará se conectar ao HPCC. Esta conexão será então usada para quaisquer chamadas subsequentes para o *saveToHPCC*.

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

O método *saveToHPCC* suporta apenas os tipos RDD<row>. Você pode precisar modificar sua representação de dados para usar essa funcionalidade. No entanto, essa representação de dados é usada pelo Spark SQL e pelo HPCC. Isso só é suportado gravando em uma configuração co-localizada. Assim, o Spark e o HPCC devem ser instalados nos mesmos nós. A leitura suporta apenas a leitura de dados de um cluster HPCC remoto.

O *clusterName*, conforme usado no caso acima, é o cluster desejado para gravar dados, por exemplo, no cluster Thor "mitor". Atualmente, há suporte apenas para gravação em clusters do Thor. A gravação em um cluster Roxie não é suportada e retornará uma exceção. O nome do arquivo usado no exemplo acima está no formato HPCC, por exemplo: "~example::text".

Internamente, o método *saveToHPCC* gerará múltiplos jobs do Spark. Atualmente, isso gera dois jobs. O primeiro job mapeia o local das partições no cluster do Spark para fornecer essas informações ao HPCC. O segundo job faz a gravação real dos arquivos. Há também algumas chamadas internamente ao ESP para lidar com coisas como iniciar

o processo de gravação usando *DFUCreateFile* e publicar o arquivo depois de ter sido escrito chamando *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")
```

Exemplo de código Scala:

```
// 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")
```

Exemplo de código R:

```
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)
```

## Classes Adicionais de Interesse

As principais classes de interesse para esta seção são a remoção de colunas e a filtragem de arquivos. Além disso, há uma classe auxiliar para remapear informações de IP quando necessário, e isso também é discutido abaixo.

As informações de seleção da coluna são fornecidas como uma string para o objeto *org.hpccsystems.spark.ColumnPruner*. A string é uma lista de nomes de campos separados por vírgulas. Um campo de interesse pode conter um conjunto de dados de linha ou filho e a notação de nome pontilhada é usada para oferecer suporte à seleção de campos filho individuais. O *ColumnPruner* analisa a cadeia em uma instância da classe *TargetColumn* raiz que contém as colunas de destino de nível superior. Um *TargetColumn* pode ser um campo simples ou pode ser um conjunto de dados filho e, portanto, ser um objeto raiz para o layout do registro filho.

O filtro de linha é implementado na classe *org.hpccsystems.spark.thor.FileFilter*. Uma instância de *FileFilter* é restrita a partir de uma matriz de objetos *org.hpccsystems.spark.thor.FieldFilter*. Cada instância de *FieldFilter* é composta de um nome de campo (em notação pontuada para nomes compostos) e uma matriz de objetos *org.hpccsystems.spark.thor.FieldFilterRange*. Cada instância de *FieldFilterRange* pode ser um intervalo aberto, ou fechado ou um valor único. O registro é selecionado quando pelo menos um *FieldFilterRange* corresponde para cada uma das instâncias do *FieldFilter* na matriz.

Os valores *FieldFilterRange* podem ser cadeias ou números. Existem métodos fornecidos para construir os seguintes testes de intervalo: igual, não igual, menor que, menor que ou igual a, maior que, e maior que ou igual a. Além disso, um teste de inclusão de conjunto é suportado para cadeias de caracteres. Se o arquivo for um índice, os campos de filtro, que são campos-chave, são utilizados para uma pesquisa de índice. Qualquer campo de filtro não mencionado é tratado como desconhecido.

A arquitetura de implantação usual para os Clusters HPCC consiste em uma coleção de nós em uma rede. As informações de gerenciamento de arquivos incluem os endereços IP dos nós que contêm as partições do arquivo. As classes do conector Spark-HPCC usam esses endereços IP para estabelecer conexões de soquete para a leitura remota. Um cluster HPCC pode ser implantado como um cluster virtual com endereços IP privados. Isso funciona para os componentes do cluster porque eles estão todos na mesma LAN privada. No entanto, os nós do cluster Spark podem não estar na mesma LAN. Nesse caso, a classe *org.hpccsystems.spark.RemapInfo* é usada para definir as informações necessárias para alterar o endereçamento. Existem duas opções que podem ser usadas. A primeira opção é que cada nó de trabalho do Thor pode receber um IP visível para o cluster do Spark. Esses endereços devem ser um intervalo contíguo. A segunda opção é atribuir um IP e um intervalo contíguo de números de porta. O objeto *RemapInfo* é fornecido como um parâmetro.

## Exemplos

Vamos percorrer os dois exemplos abaixo, utilizando um ambiente Spark. Além disso, o repositório fornece programas de teste (na pasta `DataAccess/src/test`) que podem ser executados como exemplos stand-alone.

Esses programas de teste devem ser executados a partir de um IDE de desenvolvimento, como o Eclipse, por meio do aplicativo `Spark-submit`, enquanto os exemplos abaixo são dependentes do shell do Spark.

Os exemplos a seguir assumem um Spark Shell. Você pode usar o comando `spark-submit` se você pretende compilar e empacotar esses exemplos. Para conectar corretamente seu shell ao cluster Spark Integrado, forneça os seguintes parâmetros ao iniciá-lo:

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

### Iris\_LR

Este exemplo pressupõe que você tenha o Spark Shell em execução. O próximo passo é estabelecer o seu `HpccFile` e seu RDD para esse arquivo. Você precisa do nome do arquivo, do protocolo (`http` ou `https`), do nome ou IP do ESP, da porta do ESP (normalmente 8010) e da sua conta de usuário e senha. O valor `sc` é o objeto `SparkContext` fornecido pelo shell.

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

Agora temos um RDD dos dados. Na verdade, nada aconteceu nesse ponto porque o Spark executa uma avaliação lenta e ainda não há nada para acionar uma avaliação.

O Spark MLLib possui um pacote de regressão logística. A Regressão Logística MLLib espera que os dados sejam fornecidos como registros formatados em Ponto Rotulado. Isso é comum em implementações de treinamento supervisionado no MLLib. Precisamos de rótulos de coluna, então criamos uma matriz de nomes. Em seguida, fazemos um RDD de ponto rotulado a partir do nosso RDD. Isso também é apenas uma definição. Finalmente, definimos a Regressão Logística que queremos executar. Os nomes das colunas são os nomes dos campos na definição de registro ECL do arquivo, incluindo o nome "class", que é o nome do campo que contém o código de classificação.

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

A próxima etapa é definir o modelo, que é uma ação e fará com que o Spark avalie as definições.

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

Agora nós temos um modelo. Utilizaremos esse modelo para pegar o dataset original e usar o modelo para produzir novos rótulos. A maneira correta de fazer isso é ter amostrado aleatoriamente alguns dados em espera. Vamos usar o conjunto de dados original porque é mais fácil mostrar como usar o conector. Em seguida, pegamos nossos dados originais e usamos uma função de mapa definida em linha para criar um novo registro com nosso valor de previsão e a classificação original.

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

A classe *MulticlassMetrics* agora pode ser usada para produzir uma matriz de confusão.

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

## Dataframe\_Iris\_LR

O *Dataframe\_Iris\_LR* é semelhante ao *Iris\_LR*, exceto que um *Dataframe* é usado e as novas classes do ML Spark são usadas em vez das antigas classes MLLib. Como o ML não está completamente pronto, voltamos para uma classe MLLib para criar nossa matriz de confusão.

Uma vez que o shell Spark é criado, precisamos das classes de importação.

```
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
```

O próximo passo é estabelecer o objeto *HpccFile* e criar o *Dataframe*. O *valor* do *spark* é um objeto *SparkSession* fornecido pelo shell e é usado em vez do objeto *SparkContext*.

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

As classes de aprendizado de máquina do Spark *ml* usam classes de contêiner de dados diferentes. No caso da Regressão Logística, precisamos transformar nossas linhas de dados em uma linha com uma coluna chamada "recursos" contendo os recursos e uma coluna chamada "rótulo" contendo o rótulo de classificação. Lembre-se de que nossa linha tem "class", "sepal\_width", "sepal\_length", "petal\_width" e "petal\_length" como os nomes das colunas. Esse tipo de transformação pode ser realizado com uma classe *VectorAssembler*.

```
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")
```

Agora que os dados (*iris\_fv*) estão prontos, definimos nosso modelo e ajustamos os dados.

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

Agora queremos aplicar nossa previsão e avaliar os resultados. Como observado anteriormente, usaríamos um dataset de validação para realizar a avaliação. Nós vamos ser preguiçosos e apenas usar os dados originais para evitar a tarefa de amostragem. Usamos a função *transform(...)* para o modelo para adicionar a previsão. A função adiciona uma coluna chamada "previsão" e define um novo conjunto de dados. A nova implementação de Aprendizado de Máquina não possui capacidade de métricas para produzir uma matriz de confusão, portanto, vamos pegar nosso dataset com a coluna de *previsão* e criar um novo RDD com um conjunto de dados para uma classe *MulticlassMetrics*.

```
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
```