

# Configuring CDS 2.x Powered by Apache Spark

## 2

Date published: 2020-02-28

Date modified:

The Cloudera logo, consisting of the word "CLOUDERA" in a bold, orange, sans-serif font. The letter "E" is stylized with a horizontal bar through its middle.

# Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

<b>Spark 3 on CDSW.....</b>	<b>4</b>
<b>Using CDS 2.x Powered by Apache Spark.....</b>	<b>4</b>
<b>Configuring CDS 2.x Powered by Apache Spark 2.....</b>	<b>6</b>
Spark Configuration Files.....	6
Configuring Global Properties Using Cloudera Manager.....	6
Configuring Spark Environment Variables Using Cloudera Manager.....	6
Managing Memory Available for Spark Drivers.....	7
Managing Dependencies for Spark 2 Jobs.....	7
Spark Logging Configuration.....	8
Running Spark Jobs on an HDP Cluster.....	8
<b>Setting Up an HTTP Proxy for Spark 2.....</b>	<b>8</b>
<b>Using Spark 2 from Python.....</b>	<b>9</b>
Setting Up a PySpark Project.....	9
Spark on ML Runtimes.....	10
Example: Montecarlo Estimation.....	10
Example: Locating and Adding JARs to Spark 2 Configuration.....	11
Example: Distributing Dependencies on a PySpark Cluster.....	12
<b>Using Spark 2 from R.....</b>	<b>14</b>
Installing sparklyr.....	14
Connecting to Spark 2.....	15
<b>Using Spark 2 from Scala.....</b>	<b>15</b>
Accessing Spark 2 from the Scala Engine.....	15
Example: Read Files from the Cluster Local Filesystem.....	16
Example: Using External Packages by Adding Jars or Dependencies.....	16
Adding Remote Packages.....	16
Adding Remote or Local JARs.....	16

## Spark 3 on CDSW

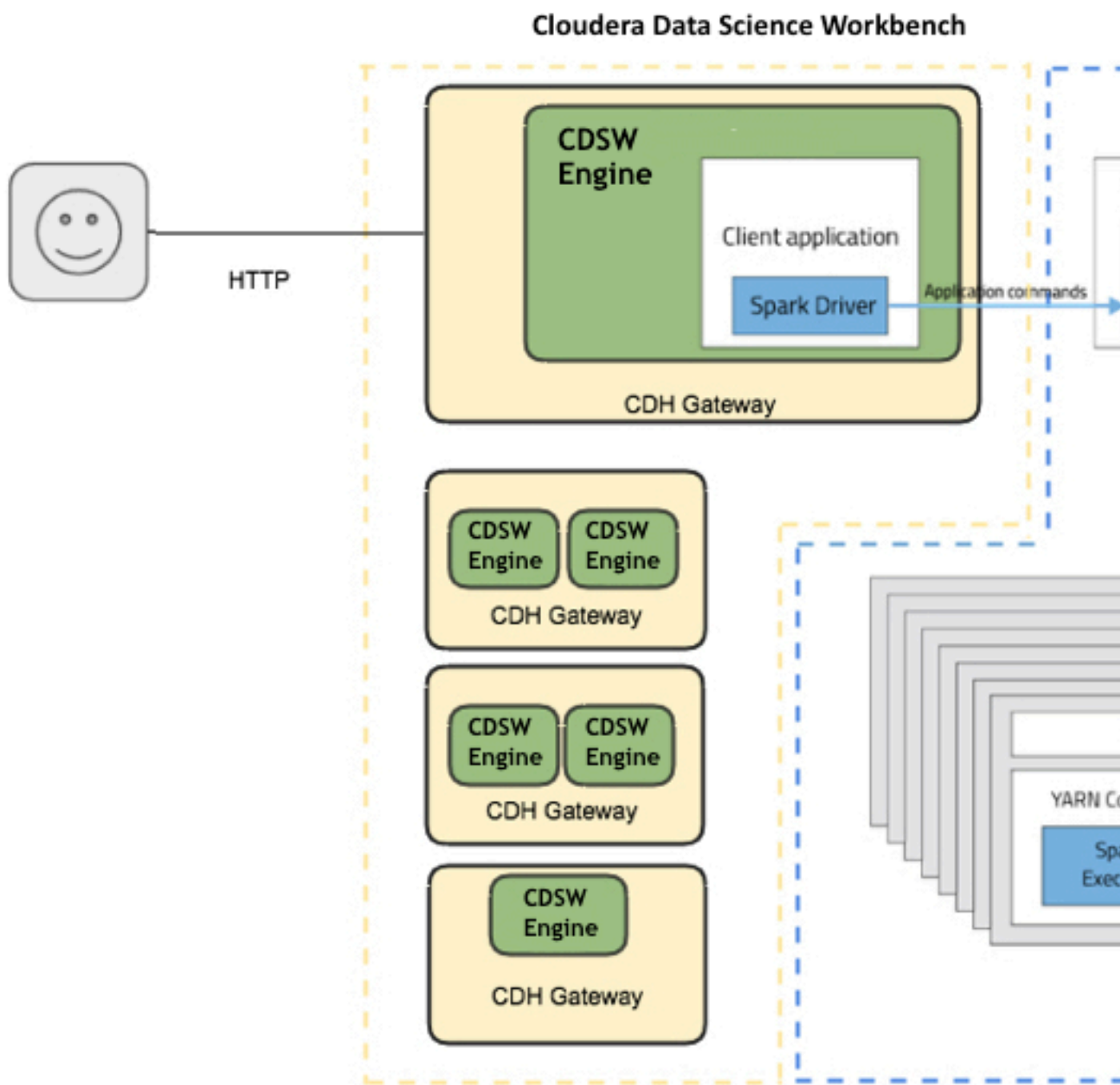
Spark 3 is not supported on CDSW. To use Spark 3 you must upgrade to CML.

## Using CDS 2.x Powered by Apache Spark

Apache Spark is a general purpose framework for distributed computing that offers high performance for both batch and stream processing. It exposes APIs for Java, Python, R, and Scala, as well as an interactive shell for you to run jobs.

Cloudera Data Science Workbench provides interactive and batch access to Spark 2. Connections are fully secure without additional configuration, with each user accessing Spark using their Kerberos principal. With a few extra lines of code, you can do anything in Cloudera Data Science Workbench that you might do in the Spark shell, as well as leverage all the benefits of the workbench. Your Spark applications will run in an isolated project workspace.

Cloudera Data Science Workbench's interactive mode allows you to launch a Spark application and work iteratively in R, Python, or Scala, rather than the standard workflow of launching an application and waiting for it to complete to view the results. Because of its interactive nature, Cloudera Data Science Workbench works with Spark on YARN's client mode, where the driver persists through the lifetime of the job and runs executors with full access to the CDH cluster resources. This architecture is illustrated the following figure:



The rest of this guide describes how to set Spark 2 environment variables, manage package dependencies, and how to configure logging. It also consists of instructions and sample code for running R, Scala, and Python projects from Spark 2.

Cloudera Data Science Workbench allows you to access the Spark History Server and even transient per-session UIs for Spark 2 directly from the workbench console. For more details, see [Accessing Web User Interfaces from Cloudera Data Science Workbench](#).

## Configuring CDS 2.x Powered by Apache Spark 2

This topic describes how to set Spark 2 environment variables, manage package dependencies for Spark 2 jobs, and how to configure logging.

### Spark Configuration Files

Cloudera Data Science Workbench supports configuring Spark 2 properties on a per project basis with the `spark-defaults.conf` file.

If there is a file called `spark-defaults.conf` in your project root, this will be automatically be added to the global Spark defaults. To specify an alternate file location, set the environmental variable, `SPARK_CONFIG`, to the path of the file relative to your project. If you're accustomed to submitting a Spark job with key-values pairs following a `--conf` flag, these can also be set in a `spark-defaults.conf` file instead. For a list of valid key-value pairs, refer the [Spark configuration reference documentation](#).

Administrators can set environment variable paths in the `/etc/spark2/conf/spark-env.sh` file.

You can also use Cloudera Manager to configure `spark-defaults.conf` and `spark-env.sh` globally for all Spark applications as follows.



**Important:** Cloudera Data Science Workbench does not automatically detect configuration changes on the CDH cluster. Therefore, any changes made to CDH services must be followed by a full reset of Cloudera Data Science Workbench. Review the associated known issue in *CDH Integration Issues*.

### Configuring Global Properties Using Cloudera Manager

Configure client configuration properties for all Spark applications in `spark-defaults.conf` as follows.

#### Procedure

1. Go to the Cloudera Manager Admin Console.
2. Navigate to the Spark service.
3. Click the Configuration tab.
4. Search for the Spark Client Advanced Configuration Snippet (Safety Valve) for `spark-conf/spark-defaults.conf` property.
5. Specify properties described in [Application Properties](#). If more than one role group applies to this configuration, edit the value for the appropriate role group.
6. Click Save Changes to commit the changes.
7. Deploy the client configuration.
8. Restart Cloudera Data Science Workbench.

#### What to do next

For more information on using a `spark-defaults.conf` file for Spark jobs, visit the [Apache Spark 2 reference documentation](#).

### Configuring Spark Environment Variables Using Cloudera Manager

Configure service-wide environment variables for all Spark applications in `spark-env.sh` as follows.

#### Procedure

1. Go to the Cloudera Manager Admin Console.

2. Navigate to the Spark 2 service.
3. Click the Configuration tab.
4. Search for the Spark Service Advanced Configuration Snippet (Safety Valve) for spark-conf/spark-env.sh property and add the paths for the environment variables you want to configure.
5. Click Save Changes to commit the changes.
6. Restart the service.
7. Deploy the client configuration.
8. Restart Cloudera Data Science Workbench.

## Managing Memory Available for Spark Drivers

By default, the amount of memory allocated to Spark driver processes is set to a 0.8 fraction of the total memory allocated for the engine container.

If you want to allocate more or less memory to the Spark driver process, you can override this default by setting the `spark.driver.memory` property in `spark-defaults.conf` (as described above).

## Managing Dependencies for Spark 2 Jobs

As with any Spark job, you can add external packages to the executor on startup.

To add external dependencies to Spark jobs, specify the libraries you want added by using the appropriate configuration parameter in a `spark-defaults.conf` file. The following table lists the most commonly used configuration parameters for adding dependencies and how they can be used:

Property	Description
<code>spark.files</code>	Comma-separated list of files to be placed in the working directory of each Spark executor.
<code>spark.submit.pyFiles</code>	Comma-separated list of .zip, .egg, or .py files to place on PYTHONPATH for Python applications.
<code>spark.jars</code>	Comma-separated list of local jars to include on the Spark driver and Spark executor classpaths.
<code>spark.jars.packages</code>	Comma-separated list of Maven coordinates of jars to include on the Spark driver and Spark executor classpaths. When configured, Spark will search the local Maven repo, and then Maven central and any additional remote repositories configured by <code>spark.jars.ivy</code> . The format for the coordinates are <code>groupId:artifactId:version</code> .
<code>spark.jars.ivy</code>	Comma-separated list of additional remote repositories to search for the coordinates given with <code>spark.jars.packages</code> .

Example `spark-defaults.conf`

Here is a sample `spark-defaults.conf` file that uses some of the Spark configuration parameters discussed in the previous section to add external packages on startup.

```
spark.jars.packages org.scalaj:scalaj-http_2.11:2.3.0
spark.jars my_sample.jar
spark.files data/test_data_1.csv,data/test_data_2.csv
```

### **spark.jars**

The pre-existing jar, `my_sample.jar`, residing in the root of this project will be included on the Spark driver and executor classpaths.

### **spark.files**

The two sample data sets, `test_data_1.csv` and `test_data_2.csv`, from the `/data` directory of this project will be distributed to the working directory of each Spark executor.

For more advanced configuration options, visit the [Apache Spark 2 reference documentation](#).

## Spark Logging Configuration

Cloudera Data Science Workbench allows you to update Spark's internal logging configuration on a per-project basis. Spark 2 uses Apache Log4j, which can be configured through a properties file.

By default, a `log4j.properties` file found in the root of your project will be appended to the existing Spark logging properties for every session and job. To specify a custom location, set the environmental variable `LOG4J_CONFIG` to the file location relative to your project.

The [Log4j documentation](#) has more details on logging options.

Increasing the log level or pushing logs to an alternate location for troublesome jobs can be very helpful for debugging. For example, this is a `log4j.properties` file in the root of a project that sets the logging level to INFO for Spark jobs.

```
shell.log.level=INFO
```

PySpark logging levels should be set as follows:

```
log4j.logger.org.apache.spark.api.python.PythonGatewayServer=<LOG_LEVEL>
```

And Scala logging levels should be set as:

```
log4j.logger.org.apache.spark.repl.Main=<LOG_LEVEL>
```

## Running Spark Jobs on an HDP Cluster

Execution errors might occur due to inadequate resource threshold values set for YARN.

### About this task

In order to run Spark jobs on an HDP cluster, apply the following changes:

### Procedure

1. From Ambari, go to `Views YARN Queue Manager` service.
2. Choose or add a queue. You can configure root, default, or individual queues.
3. Under the Resources section, increase the value in the Maximum AM Resource field so that the queues have enough resources to run. For example, you can set it to 80%.  
Child queues can inherit the settings from the parent queue.

## Setting Up an HTTP Proxy for Spark 2

In Cloudera Data Science Workbench clusters that use an HTTP proxy, follow these steps to support web-related actions in Spark. You must set the Spark configuration parameter `extraJavaOptions` on your gateway hosts.

### Procedure

1. Log in to Cloudera Manager.
2. Go to `Spark2 Configuration`.

3. Filter the properties with `Scope Gateway` and `Category Advanced`.
4. Scroll down to `Spark 2 Client Advanced Configuration Snippet (Safety Valve) for spark2-conf/spark-defaults.conf`.
5. Enter the following configuration code, substituting your proxy host and port values:

```
spark.driver.extraJavaOptions= \
-Dhttp.proxyHost=<YOUR HTTP PROXY HOST> \
-Dhttp.proxyPort=<HTTP PORT> \
-Dhttps.proxyHost=<YOUR HTTPS PROXY HOST> \
-Dhttps.proxyPort=<HTTPS PORT>
```

6. Click `Save Changes`.
7. Choose `Actions Deploy Client Configuration`.

## Using Spark 2 from Python

Cloudera Data Science Workbench supports using Spark 2 from Python via PySpark.

### Setting Up a PySpark Project

The default Cloudera Data Science Workbench engine currently includes Python 2.7.18 and Python 3.6.10.

#### PySpark Environmental Variables

To use PySpark with lambda functions that run within the CDH cluster, the Spark executors must have access to a matching version of Python. For many common operating systems, the default system Python will not match the minor release of Python included in Data Science Workbench.

To ensure that the Python versions match, Python can either be installed on every CDH host or made available per job run using Spark's ability to distribute dependencies. Given the size of a typical isolated Python environment and the desire to avoid repeated uploads from gateway hosts, Cloudera recommends installing Python 2.7 and 3.6 on the cluster if you are using PySpark with lambda functions.

You can install Python 2.7 and 3.6 on the cluster using any method and set the corresponding `PYSPARK_PYTHON` environment variable in your project. Cloudera Data Science Workbench 1.3 (and higher) include a separate environment variable for Python 3 sessions called `PYSPARK3_PYTHON`. Python 2 sessions continue to use the default `PYSPARK_PYTHON` variable. This will allow you to run Python 2 and Python 3 sessions in parallel without either variable being overridden by the other.

#### Creating and Running a PySpark Project

To get started quickly, use the PySpark template project to create a new project. For instructions, see [Creating a Project with Legacy Engine Variants](#) or [Creating a Project with ML Runtimes Variants](#)

To run a PySpark project, navigate to the project's overview page, open the workbench console and launch a Python session.

#### Testing a PySpark Project in Spark Local Mode

Spark's local mode is often useful for testing and debugging purposes. Use the following sample code snippet to start a PySpark session in local mode.

```
from pyspark.sql import SparkSession

spark = SparkSession\
    .builder \
    .appName("LocalSparkSession") \
    .master("local") \
```

```
.getOrCreate()
```

For more details, refer the Spark documentation: [Running Spark Applications](#).

### Spark on ML Runtimes

Spark is supported for ML Runtimes with Python 3.6 and Python 3.7 kernels given that the following workaround is applied on the cluster:

- Python must be installed on the CDH cluster YARN Node Manager nodes which should match the Python version of the selected ML Runtime (i.e., 3.6 or 3.7)
- This Python version must be specified by its path for Spark using the `pyspark_python` environment variable
- As an example for 3.7, one could specify the environment variable like this for the CDSW project:
  - "PYSPARK\_PYTHON": "/usr/local/bin/python3.7"

## Spark on ML Runtimes

Only certain Spark versions are supported on ML Runtimes.

Spark is supported for ML Runtimes with Python 3.7 and above. Python 3.6 is no longer supported because it has reached end-of-life. Different ML Runtimes support different versions of Python. Review the list of *Pre-Installed Packages* in ML Runtimes to determine which Runtime supports which specific Python kernel.

To use Python, ensure that:

- The Python installed on the CDH cluster Node Manager nodes matches the Python version of the selected ML Runtime (for example, 3.7 or above)
- The `/usr/bin/python` symlink on all Node Managers should point to the path where Python is installed.

If multiple versions of Python 3 need to be used, then:

1. Install each version of Python into the same location on all Node Manager nodes.
2. Set the `/usr/bin/python` on all nodes to the Python version that will be used by default.
3. If a non-default version of Python needs to be used, in the Project Settings Advanced Environment Variables section, set `PYSPARK_PYTHON` to the path to the non-default Python version. For example:
  - `PYSPARK_PYTHON=/user/bin/python3.9`

## Example: Montecarlo Estimation

Within the template PySpark project, `pi.py` is a classic example that calculates Pi using the [Montecarlo Estimation](#).

What follows is the full, annotated code sample that can be saved to the `pi.py` file.

```
# # Estimating $\pi$
#
# This PySpark example shows you how to estimate $\pi$ in parallel
# using Monte Carlo integration.

from __future__ import print_function
import sys
from random import random
from operator import add
# Connect to Spark by creating a Spark session
from pyspark.sql import SparkSession
spark = SparkSession\
    .builder\
    .appName("PythonPi")\
    .getOrCreate()
```

```

partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
n = 100000 * partitions

def f(_):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 < 1 else 0

# To access the associated SparkContext
count = spark.sparkContext.parallelize(range(1, n + 1), partitions).map(f)
    .reduce(add)
print("Pi is roughly %f" % (4.0 * count / n))

spark.stop()

```

## Example: Locating and Adding JARs to Spark 2 Configuration

This example shows how to discover the location of JAR files installed with Spark 2, and add them to the Spark 2 configuration.

```

# # Using Avro data
#
# This example shows how to use a JAR file on the local filesystem on
# Spark on Yarn.

from __future__ import print_function
import os,sys
import os.path
from functools import reduce
from pyspark.sql import SparkSession
from pyspark.files import SparkFiles

# Add the data file to HDFS for consumption by the Spark executors.
!hdfs dfs -put resources/users.avro /tmp

# Find the example JARs provided by the Spark parcel. This parcel
# is available on both the driver, which runs in Cloudera Data Science Wor
kbench, and the
# executors, which run on Yarn.
exampleDir = os.path.join(os.environ["SPARK_HOME"], "examples/jars")
exampleJars = [os.path.join(exampleDir, x) for x in os.listdir(exampleDir)]

# Add the Spark JARs to the Spark configuration to make them available for
use.
spark = SparkSession\
    .builder\
    .config("spark.jars", ",".join(exampleJars))\
    .appName("AvroKeyInputFormat")\
    .getOrCreate()
sc = spark.sparkContext
# Read the schema.
schema = open("resources/user.avsc").read()
conf = {"avro.schema.input.key": schema }

avro_rdd = sc.newAPIHadoopFile(
    "/tmp/users.avro", # This is an HDFS path!
    "org.apache.avro.mapreduce.AvroKeyInputFormat",
    "org.apache.avro.mapred.AvroKey",
    "org.apache.hadoop.io.NullWritable",
    keyConverter="org.apache.spark.examples.pythonconverters.AvroWrapperToJ
avaConverter",

```

```

    conf=conf)
output = avro_rdd.map(lambda x: x[0]).collect()
for k in output:
    print(k)
spark.stop()

```

## Example: Distributing Dependencies on a PySpark Cluster

Although Python is a popular choice for data scientists, it is not straightforward to make a Python library available on a distributed PySpark cluster. To determine which dependencies are required on the cluster, you must understand that Spark code applications run in Spark executor processes distributed throughout the cluster. If the Python code you are running uses any third-party libraries, Spark executors require access to those libraries when they run on remote executors.

### About this task

This example demonstrates a way to run the following Python code (`nltk_sample.py`), that includes pure Python libraries ([nltk](#)), on a distributed PySpark cluster.

### Procedure

#### 1. (Prerequisites)

- a) Make sure the [Anaconda parcel](#) has been [distributed and activated](#) on your cluster.
- b) Create a new project in Cloudera Data Science Workbench. In that project, create a new file called `nltk_sample.py` with the following sample script.

`nltk_sample.py`

```

# This code uses NLTK, a Python natural language processing library.
# NLTK is not installed with conda by default.
# You can use 'import' within your Python UDFs, to use Python libraries.
# The goal here is to distribute NLTK with the conda environment.
import os
import sys
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("spark-nltk") \
    .getOrCreate()

data = spark.sparkContext.textFile('1970-Nixon.txt')

def word_tokenize(x):
    import nltk
    return nltk.word_tokenize(x)

def pos_tag(x):
    import nltk
    return nltk.pos_tag([x])

words = data.flatMap(word_tokenize)
words.saveAsTextFile('nixon_tokens')

pos_word = words.map(pos_tag)
pos_word.saveAsTextFile('nixon_token_pos')

```

2. Go to the project you created and launch a new PySpark session.

3. Click Terminal Access and run the following command to pack the Python environment into conda.

```
conda create -n nltk_env --copy -y -q python=2.7.11 nltk numpy
```

The `--copy` option allows you to copy whole dependent packages into certain directory of a conda environment. If you want to add extra pip packages without conda, you should copy packages manually after using `pip install`. In Cloudera Data Science Workbench, pip will install packages into the `~/local` directory in a project.

```
pip install some-awesome-package
cp -r ~/.local/lib ~/.conda/envs/nltk_env/
```

Zip the conda environment for shipping on PySpark cluster.

```
cd ~/.conda/envs
zip -r ../../nltk_env.zip nltk_env
```

4. (Specific to NLTK) For this example, you can use NLTK data as input.

```
cd ~/
source activate nltk_env

# download nltk data
(nltk_env)$ python -m nltk.downloader -d nltk_data all
(nltk_env)$ hdfs dfs -put nltk_data/corpora/state_union/1970-Nixon.txt ./

# archive nltk data for distribution
cd ~/nltk_data/tokenizers/
zip -r ../../tokenizers.zip *
cd ~/nltk_data/taggers/
zip -r ../../taggers.zip *
```

5. Set spark-submit options in spark-defaults.conf.

```
spark.yarn.appMasterEnv.PYSPARK_PYTHON=./NLTK/nltk_env/bin/python
spark.yarn.appMasterEnv.NLTK_DATA=./
spark.executorEnv.NLTK_DATA=./
spark.yarn.dist.archives=nltk_env.zip#NLTK,tokenizers.zip#tokenizers,taggers.zip#taggers
```

With these settings, PySpark unzips `nltk_env.zip` into the NLTK directory. `NLTK_DATA` is the environmental variable where NLTK data is stored.

6. In Cloudera Data Science Workbench, set the `PYSPARK_PYTHON` environment variable to the newly-created environment.
  - a) To do this, navigate back to the Project Overview page and click **Settings Engine Environment Variables**.
  - b) Set `PYSPARK_PYTHON` to `./NLTK/nltk_env/bin/python` and click **Add**.
  - c) Then click **Save Environment**.
7. Launch a new PySpark session and run the `nltk_sample.py` script in the workbench. You can test whether the script ran successfully using the following command:

```
!hdfs dfs -cat ./nixon_tokens/* | head -n 20
```

```
Annual
Message
to
the
Congress
on
the
State
```

```

of
the
Union
.
January
22
,
1970
Mr.
Speaker
,
Mr.

! hdfs dfs -cat nixon_token_pos/* | head -n 20
[(u'Annual', 'JJ')]
[(u'Message', 'NN')]
[(u'to', 'TO')]
[(u'the', 'DT')]
[(u'Congress', 'NNP')]
[(u'on', 'IN')]
[(u'the', 'DT')]
[(u'State', 'NNP')]
[(u'of', 'IN')]
[(u'the', 'DT')]
[(u'Union', 'NN')]
[(u'.', '.')]
[(u'January', 'NNP')]
[(u'22', 'CD')]
[(u',', ',')]
[(u'1970', 'CD')]
[(u'Mr.', 'NNP')]
[(u'Speaker', 'NN')]
[(u',', ',')]
[(u'Mr.', 'NNP')]

```

## Using Spark 2 from R

R users can access Spark 2 using [sparklyr](#). Although Cloudera does not ship or support sparklyr, we do recommend using sparklyr as the R interface for Cloudera Data Science Workbench.

### Installing sparklyr

Install the latest version of sparklyr as follows.

#### Procedure

Install the latest version of sparklyr as follows;

```
install.packages("sparklyr")
```



**Note:** The `spark_apply()` function requires the R Runtime environment to be pre-installed on your cluster. This will likely require intervention from your cluster administrator. For details, refer the [RStudio documentation](#).

## Connecting to Spark 2

You can connect to local instances of Spark 2 as well as remote clusters.

```
## Connecting to Spark 2
# Connect to an existing Spark 2 cluster in YARN client mode using the spar
k_connect function.
library(sparklyr)
system.time(sc <- spark_connect(master = "yarn-client"))
# The returned Spark 2 connection (sc) provides a remote dplyr data source
to the Spark 2 cluster.
```

For a complete example, see [Sparklyr \(R\)](#).

## Using Spark 2 from Scala

This topic describes how to set up a Scala project for CDS 2.x Powered by Apache Spark along with a few associated tasks. Cloudera Data Science Workbench provides an interface to the Spark 2 shell (v 2.0+) that works with Scala 2.11.

## Accessing Spark 2 from the Scala Engine

This topic describes how to set up a Scala project for CDS 2.x Powered by Apache Spark along with a few associated tasks. Cloudera Data Science Workbench provides an interface to the Spark 2 shell (v 2.0+) that works with Scala 2.11.

Unlike PySpark or Sparklyr, you can access a `SparkContext` assigned to the `spark` (`SparkSession`) and `sc` (`SparkContext`) objects on console startup, just as when using the Spark shell. By default, the application name will be set to `CDSW_sessionID`, where `sessionID` is the id of the session running your Spark code. To customize this, set the `spark.app.name` property to the desired application name in a `spark-defaults.conf` file.

`Pi.scala` is a classic starting point for calculating Pi using [Monte Carlo Estimation](#).

This is the full, annotated code sample.

```
//Calculate pi with Monte Carlo estimation
import scala.math.random
//make a very large unique set of 1 -> n
val partitions = 2
val n = math.min(100000L * partitions, Int.MaxValue).toInt
val xs = 1 until n

//split up n into the number of partitions we can use
val rdd = sc.parallelize(xs, partitions).setName("'N values rdd'")

//generate a random set of points within a 2x2 square
val sample = rdd.map { i =>
  val x = random * 2 - 1
  val y = random * 2 - 1
  (x, y)
}.setName("'Random points rdd'")

//points w/in the square also w/in the center circle of r=1
val inside = sample.filter { case (x, y) => (x * x + y * y < 1) }.setName(
"'Random points inside circle'")
val count = inside.count()
```

```
//Area(circle)/Area(square) = inside/n => pi=4*inside/n  
println("Pi is roughly " + 4.0 * count / n)
```

Key points to note:

- `import scala.math.random`

Importing included packages works just as in the shell, and need only be done once.

- Spark context (`sc`).

You can access a `SparkContext` assigned to the variable `sc` on console startup.

```
val rdd = sc.parallelize(xs, partitions).setName("'N values rdd'")
```

## Example: Read Files from the Cluster Local Filesystem

Use the following command in the terminal to read text from the local filesystem.

The file must exist on all hosts, and the same path for the driver and executors. In this example you are reading the file `ebay-xbox.csv`.

```
sc.textFile("file:///tmp/ebay-xbox.csv")
```

## Example: Using External Packages by Adding Jars or Dependencies

External libraries are handled through line magics. Line magics in the Toree kernel are prefixed with `%`.

### Adding Remote Packages

You can use Apache Toree's `AddDeps` magic to add dependencies from Maven central.

You must specify the company name, artifact ID, and version. To resolve any transitive dependencies, you must explicitly specify the `--transitive` flag.

```
%AddDeps org.scalaj scalaj-http_2.11 2.3.0  
import scalaj.http._  
val response: HttpResponse[String] = Http("http://www.omdbapi.com/").param(  
  "t","crimson tide").asString  
response.body  
response.code  
response.headers  
response.cookies
```

### Adding Remote or Local JARs

You can use the `AddJars` magic to distribute local or remote JARs to the kernel and the cluster.

Using the `-f` option ignores cached JARs and reloads.

```
%AddJar http://example.com/some_lib.jar -f  
%AddJar file:/path/to/some/lib.jar
```