

Using Apache Spark 2 with CML

Date published: 2020-07-16

Date modified: 2023-01-31



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

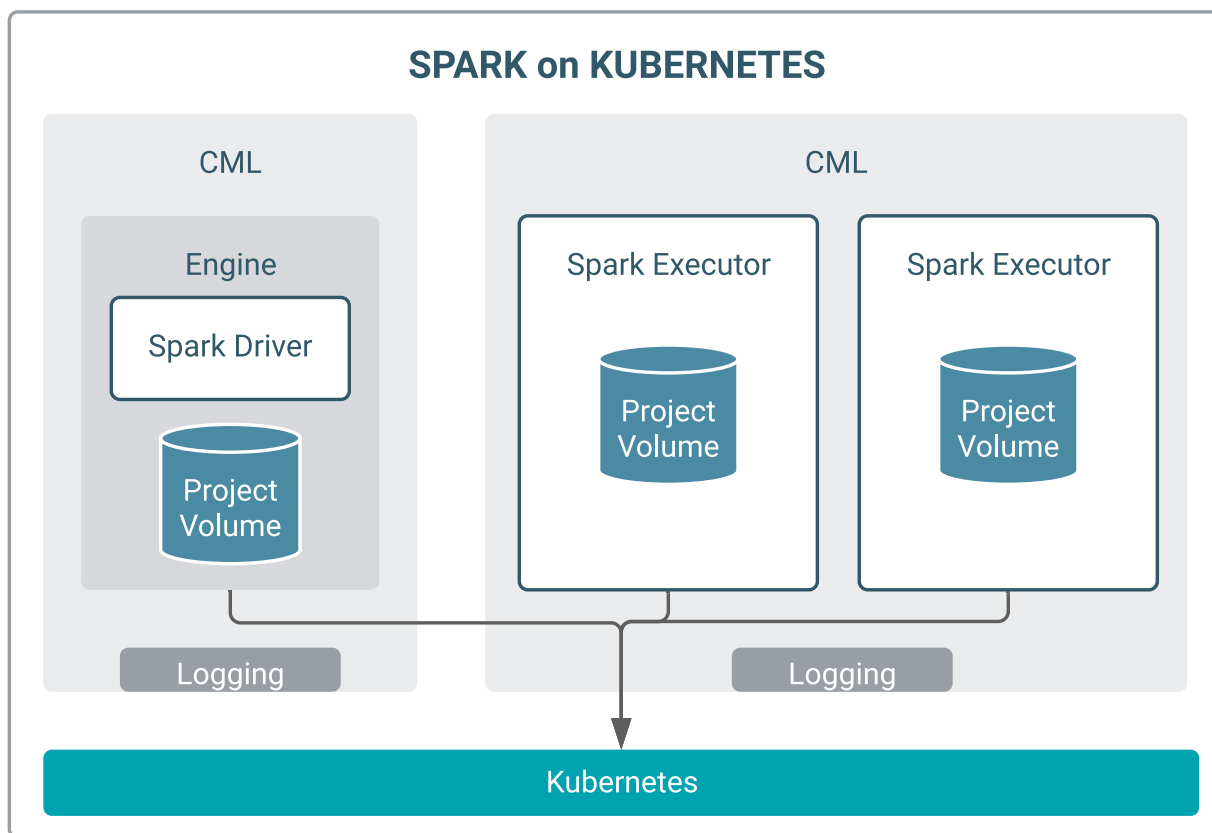
Apache Spark 2 and Spark 3 on CML.....	4
Apache Spark supported versions.....	5
Spark Configuration Files.....	5
Managing Memory Available for Spark Drivers.....	5
Managing Dependencies for Spark 2 Jobs.....	5
Spark Log4j Configuration.....	6
Setting Up an HTTP Proxy for Spark 2.....	7
Spark Web UIs.....	7
Using Spark 2 from Python.....	7
Example: Montecarlo Estimation.....	8
Example: Locating and Adding JARs to Spark 2 Configuration.....	9
Using Spark 2 from R.....	10
Using Spark 2 from Scala.....	10
Managing Dependencies for Spark 2 and Scala.....	11
Running Spark with Yarn on the CDP base cluster.....	12

Apache Spark 2 and Spark 3 on CML

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.

In Cloudera Machine Learning (CML), Spark and its dependencies are bundled directly into the CML engine Docker image.

CML supports fully-containerized execution of Spark workloads via Spark's support for the Kubernetes cluster backend. Users can interact with Spark both interactively and in batch mode.



Dependency Management: In both batch and interactive modes, dependency management, including for Spark executors, is transparently managed by CML and Kubernetes. No extra required configuration is required. In interactive mode, CML leverages your cloud provider for scalable project storage, and in batch mode, CML manages dependencies through container images.

Autoscaling: CML also supports native cloud autoscaling via Kubernetes. When clusters do not have the required capacity to run workloads, they can automatically scale up additional nodes. Administrators can configure autoscaling upper limits, which determine how large a compute cluster can grow. Since compute costs increase as cluster size increases, having a way to configure upper limits gives administrators a method to stay within a budget. Autoscaling policies can also account for heterogeneous node types such as GPU nodes.

Dynamic Resource Allocation: If a Spark job requires increasing memory or CPU resources as it executes a job, Spark can automatically increase the allocation of these resources. Likewise, the resources are automatically returned to the cluster when they are no longer needed. This mechanism is especially useful when multiple applications are sharing the resources of a cluster.

Workload Isolation: In CML, each project is owned by a user or team. Users can launch multiple sessions in a project. Workloads are launched within a separate Kubernetes namespace for each user, thus ensuring isolation between users at the K8s level.

Observability: Monitoring of Spark workloads, such as resources being consumed by Spark executors, can be performed using Grafana dashboards. For more information, see *Monitoring and Alerts* and *Monitoring ML Workspaces*.

Related Information

[Monitoring and Alerts](#)

[Monitoring ML Workspaces](#)

Apache Spark supported versions

Spark 2.4.7 is supported by Engines. Spark 2.4.7 and Spark 3.2.1 are available through Runtime Addons that can be selected when starting a session.



Note: Spark 3 does not work with Scala runtimes.

Spark Configuration Files

Cloudera Machine Learning supports configuring Spark 2 and Spark 3 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 to *Spark Configuration*.

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

Related Information

[Spark Configuration](#)

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 runtime 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).



Note: The memory allocated to a CML session does not include memory taken by Spark executors.

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.scala-lang:scala-library:2.11.2:2.3.0
spark.jars my_sample.jar
spark.files data/test_data_1.csv,data/test_data_2.csv
```

`spark.jars.packages`

The `scala-lang` package will be downloaded from Maven central and included on the Spark driver and executor classpaths.

`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 2 reference documentation](#).

Related Information

[Spark Configuration](#)

[LOG4J Configuration](#)

[Natural Language Toolkit](#)

[Making Python on Apache Hadoop Easier with Anaconda and CDH](#)

Spark Log4j Configuration

Cloudera Machine Learning 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>
```

Setting Up an HTTP Proxy for Spark 2

If you are using an HTTP proxy, you must set the Spark configuration parameter `extraJavaOptions` at runtime to be able to support web-related actions in Spark.

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

Spark Web UIs

This topic describes how to access Spark web UIs from the CML UI.

Spark 2 exposes one web UI for each Spark application driver running in Cloudera Machine Learning. The UI will be running within the container, on the port specified by the environmental variable `CDSW_SPARK_PORT`. By default, `CDSW_SPARK_PORT` is set to 20049. The web UI will exist only as long as a `SparkContext` is active within a session. The port is freed up when the `SparkContext` is shutdown.

Spark 2 web UIs are available in browsers at: `https://spark-<${CDSW_ENGINE_ID}>.<${CDSW_DOMAIN}>`. To access the UI while you are in an active session, click the grid icon in the upper right hand corner of the Cloudera Machine Learning web application, and select Spark UI from the dropdown. Alternatively, the Spark UI is also available as a tab in the session itself. For a job, navigate to the job overview page and click the History tab. Click on a job run to open the session output for the job.

Using Spark 2 from Python

Cloudera Machine Learning supports using Spark 2 from Python via PySpark. This topic describes how to set up and test a PySpark project.

PySpark Environment Variables

The default Cloudera Machine Learning engine currently includes Python 2.7.17 and Python 3.6.9. 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 Machine Learning.

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, 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 Machine Learning includes 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 *Create a Project from a Built-in Template*.

To run a PySpark project, navigate to the project's overview page, open the workbench console and launch a Python session. For detailed instructions, see *Native Workbench Console and Editor*.

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 to the Spark documentation: *Running Spark Application*.

Related Information

[Native Workbench Console and Editor](#)

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 Machine Learning,
# 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.AvroWrapperToJavaConverter",

```

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

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 Machine Learning.

Before you begin

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.

Procedure

1. Install the latest version of sparklyr:

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

2. Optionally, connect to a local or remote Spark 2 cluster:

```
## Connecting to Spark 2
# Connect to an existing Spark 2 cluster in YARN client mode using the
spark_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 *Importing Data into Cloudera Machine Learning*.

Related Information

[sparklyr: R interface for Apache Spark](#)

[sparklyr Requirements](#)

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 Machine Learning 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 `CML_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 the Montecarlo 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'")
```

Managing Dependencies for Spark 2 and Scala

This topic demonstrates how to manage dependencies on local and external files or packages.

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")
```

Adding Remote Packages

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

Running Spark with Yarn on the CDP base cluster

The primary supported way to run Spark workloads on Cloudera Machine Learning uses Spark on Kubernetes. This is different from Cloudera Data Science Workbench, which uses Spark on Yarn to run Spark workloads.

For users who are migrating projects from CDSW to CML, or who have existing Yarn workloads, CML Private Cloud offers a way to run those Spark on Yarn workloads on the CDP base cluster. This is sometimes called "Spark pushdown." This allows the Spark workloads to run without needing to modify them to run on Kubernetes.

The CML Admin must enable this mode for a CML workspace, and each CML workload must enable this mode to run Spark workloads in the attached CDP base cluster.

When this mode is enabled, each newly launched CML workload has port forwarding rules set up in Kubernetes. Additionally, Spark configurations are set in the CML session to allow Spark applications launched in the CML session to run in client mode with Executors in Yarn in the attached base cluster.

Prerequisites

Support

- In CML, Spark on Yarn Pushdown workloads are only supported with ML Runtimes.
- In CML, only Spark 2.x workloads are supported on Yarn (CDSW as well only supports Spark 2.x workloads on Yarn)

General requirements

- Spark pushdown functionality only works with CDE 1.18 Runtime Addons.
- Yarn Service configured and running in your CDP Base Cluster
- Spark On Yarn service configured and running in your CDP Base Cluster
- The CDP Base Cluster must have access to the Spark drivers that run on Data Service Hosts running CML workloads, these are launched on a set of randomized ports in the range: 30000-32768

PySpark requirements

- Python must be installed on all CDP Base Cluster YARN Node Manager nodes which should match the Python version of the selected ML Runtime (i.e. 3.7 or 3.8)
- The python binary available on Yarn Node Manager nodes must be specified in the `PYSPARK_PYTHON` environment variable
 - As an example for 3.7, one could specify the environment variable like this for the CML project with Spark Pushdown enabled:

```
"PYSPARK_PYTHON": "/usr/local/bin/python3.7"
```

- `PYSPARK_PYTHON` - The location of python in executors running in Yarn Nodes
 - Note: In CML `PYSPARK_PYTHON` is by default set to `/usr/local/bin/python3`
 - This should be changed to the appropriate location in Yarn Nodes
- `PYSPARK_DRIVER_PYTHON` = The location of python in the driver running in a CML session

Note: For CML runtimes `PYSPARK_DRIVER_PYTHON` is set to `/usr/local/bin/python3`

Enabling Spark on the base cluster

Spark can be enabled on the base cluster both site-wide and project-specific.

- Site Administration > Settings

Select Allow users to enable Spark Pushdown Configuration for Projects.

- A project-specific setting to enable spark pushdown for all newly launched workloads in the project. Each project that intends to use the CDP Base Cluster Yarn for spark workloads must enable this setting.

In Project Settings, select Settings > Enable Spark Pushdown.

Spark Application Dependencies

Due to the unique running mode of Spark on Yarn in CML, how dependencies are handled differ greatly from running the same jobs while on the base 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.

Refer to the following Spark configurations to determine how dependencies can be made available to executors.

Jars:

- spark.yarn.jars
 - By default, this is unset in a CML Project Spark Pushdown project to ensure that all spark jars loaded from the CML Spark Runtime Addon is made available to yarn executors.
 - This configuration should not be overridden within your CML projects. Consider using spark.yarn.dist.jars to indicate external references to jars.
 - (Add note about added transfer time at beginning of workloads)
- spark.yarn.dist.jars
 - This is not configured by CML.

Python:

- spark.submit.pyFiles
 - By default, this is set to /opt/spark/python/lib/*.zip to ensure that the pyspark and py4j zips included in CML Spark Runtime Addons are available to executors.
 - (Can be overridden, keeping original)

Extra files:

- spark.yarn.dist.archives - This is not configured by CML.
- spark.yarn.dist.files - This is not configured by CML.

User-Specified Spark Application Configurations

spark-defaults.conf

Multiple Spark configuration sources are appended to a single file for Spark Pushdown in CML PVC. This occurs in the following order (lower has higher precedence as the contents of /etc/spark/conf/spark-defaults.conf are loaded from top-down):

- Base Cluster Spark spark-defaults.conf Defaults and Safety valves are included here
- CML system-specific configurations injection
- CML Project spark-defaults.conf

Check the contents of /etc/spark/conf/spark-defaults.conf inside the CML Session for the final configuration used by the spark driver.

CML-Injected Spark Application Configurations

There are a number of Spark Configurations which are applied by CML in order to enable or simplify Spark on Basecluster Yarn workloads.



Warning: Do not to override these settings in your project spark-defaults.conf:

- spark.driver.host
- spark.driver.port
- spark.blockmanager.port

Spark Environment Variables

Multiple environment variable sources are considered when setting up the CML session which will run the interactive spark driver.

For spark-env.sh

- Base Cluster Spark spark-env.sh Defaults and Safety valves are included here
- CML system-specific spark envs overriding

For CML Session Environment

- Contents of constructed spark-env.sh (see above)
- Workspace env vars
- Project env vars
- User env vars