Cloudera Runtime 7.1.6

Running Apache Spark Applications

Date published: 2021-02-29
Date modified: 2021-03-19

CLOUDƎRA

https://docs.cloudera.com/

Legal Notice

Contents

Introduction

You can run Spark applications locally or distributed across a cluster, either by using an interactive shell or by submitting an application. Running Spark applications interactively is commonly performed during the data-exploration phase and for ad hoc analysis.

You can:

- Submit interactive statements through the Scala, Python, or R shell, or through a high-level notebook such as Zeppelin.
- Use APIs to create a Spark application that runs interactively or in batch mode, using Scala, Python, R, or Java.

Because of a limitation in the way Scala compiles code, some applications with nested definitions running in an interactive shell may encounter a Task not serializable exception. Cloudera recommends submitting these applications.

To run applications distributed across a cluster, Spark requires a cluster manager. In CDP, Cloudera supports only the YARN cluster manager. When run on YARN, Spark application processes are managed by the YARN ResourceManager and NodeManager roles. Spark Standalone is not supported.

To launch Spark applications on a cluster, you can use the spark-submit script in the /bin directory on a gateway host. You can also use the API interactively by launching an interactive shell for Scala (spark-shell), Python (pyspark), or SparkR. Note that each interactive shell automatically creates SparkContext in a variable called sc, and SparkSession in a variable called spark. For more information about spark-submit, see the Apache Spark documentation Submitting Applications.

Alternately, you can use Livy to submit and manage Spark applications on a cluster. Livy is a Spark service that allows local and remote applications to interact with Apache Spark over an open source REST interface. Livy offers additional multi-tenancy and security functionality. For more information about using Livy to run Spark Applications, see

Running your first Spark application

About this task

⚠ **Important:**

By default, CDH is configured to permit any user to access the Hive Metastore. However, if you have modified the value set for the configuration property hadoop.proxyuser.hive.groups, which can be modified in Cloudera Manager by setting the Hive Metastore Access Control and Proxy User Groups Override property, your Spark application might throw exceptions when it is run. To address this issue, make sure you add the groups that contain the Spark users that you want to have access to the metastore when Spark applications are run to this property in Cloudera Manager:

1. In the Cloudera Manager Admin Console Home page, click the Hive service.
2. On the Hive service page, click the Configuration tab.
3. In the Search well, type hadoop.proxyuser.hive.groups to locate the Hive Metastore Access Control and Proxy User Groups Override property.
4. Click the plus sign (+), enter the groups you want to have access to the metastore, and then click Save Changes. You must restart the Hive Metastore Server for the changes to take effect by clicking the restart icon at the top of the page.

The simplest way to run a Spark application is by using the Scala or Python shells.

Procedure

1. To start one of the shell applications, run one of the following commands:

    - Scala:

    ```
    $ /bin/spark-shell
    ...
    Spark context Web UI available at ...
    Spark context available as 'sc' (master = yarn, app id = ...).
    Spark session available as 'spark'.
    Welcome to
          ____              __
         / __/__  ___ _____/ /__
        _\ \/ _ \/ _ `/ __/  '_/
       /___/ .__/\_,_/_/ /_/\_\   version ...
          /_/
    Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_191)
    Type in expressions to have them evaluated.
    Type :help for more information.

    scala>
    ```

    - Python:

    ```
    $ /bin/pyspark
    Python 2.7.5 (default, Sep 12 2018, 05:31:16)
    [GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux2
    Type "help", "copyright", "credits" or "license" for more information.
    ...
    Welcome to
          ____              __
         / __/__  ___ _____/ /__
        _\ \/ _ \/ _ `/ __/  '_/
       /__ / .__/\_,_/_/ /_/\_\   version ...
          /_/

    Using Python version 2.7.5 (default, Sep 12 2018 05:31:16)
    SparkSession available as 'spark'.
    >>>
    ```

    SPARK_HOME defaults to /opt/cloudera/parcels/CDH/lib/spark in parcel installations. In a Cloudera Manager deployment, the shells are also available from /usr/bin.

    For a complete list of shell options, run spark-shell or pyspark with the -h flag.

2. To run the classic Hadoop word count application, copy an input file to HDFS:

    ```
    hdfs dfs -copyFromLocal input s3a://<bucket_name>/
    ```

3. Within a shell, run the word count application using the following code examples, substituting for *namenode_host*, *path/to/input*, and *path/to/output*:
   Scala:

    ```
    scala> val myfile = sc.textFile("s3a://<bucket_name>/path/to/input")
    scala> val counts = myfile.flatMap(line => line.split(" ")).map(word => (
    word, 1)).reduceByKey(_ + _)
    scala> counts.saveAsTextFile("s3a://<bucket_name>/path/to/output")
    ```

    Python:

    ```
    >>> myfile = sc.textFile("s3a://bucket_name/path/to/input")
    ```

```
>>> counts = myfile.flatMap(lambda line: line.split(" ")).map(lambda word:
 (word, 1)).reduceByKey(lambda v1,v2: v1 + v2)
>>> counts.saveAsTextFile("s3a://<bucket_name>/path/to/output")
```

Running sample Spark applications

About this task

You can use the following sample Spark Pi and Spark WordCount sample programs to validate your Spark installation and explore how to run Spark jobs from the command line and Spark shell.

Spark Pi

You can test your Spark installation by running the following compute-intensive example, which calculates pi by "throwing darts" at a circle. The program generates points in the unit square ((0,0) to (1,1)) and counts how many points fall within the unit circle within the square. The result approximates pi.

Follow these steps to run the Spark Pi example:

1. Authenticate using kinit:

    ```
    kinit <username>
    ```

2. Run the Apache Spark Pi job in yarn-client mode, using code from org.apache.spark:

    ```
    spark-submit --class org.apache.spark.examples.SparkPi \
        --master yarn-client \
        --num-executors 1 \
        --driver-memory 512m \
        --executor-memory 512m \
        --executor-cores 1 \
    ```

```
examples/jars/spark-examples*.jar 10
```

Commonly used options include the following:

--class

> The entry point for your application: for example, org.apache.spark.examples.SparkPi.

--master

> The master URL for the cluster: for example, spark://23.195.26.187:7077.

--deploy-mode

> Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (default is client).

--conf

> Arbitrary Spark configuration property in key=value format. For values that contain spaces, enclose "key=value" in double quotation marks.

<application-jar>

> Path to a bundled jar file that contains your application and all dependencies. The URL must be globally visible inside of your cluster: for instance, an hdfs:// path or a file:// path that is present on all nodes.

<application-arguments>

> Arguments passed to the main method of your main class, if any.

Your job should produce output similar to the following. Note the value of pi in the output.

```
17/03/22 23:21:10 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.s
cala:38, took 1.302805 s
Pi is roughly 3.1445191445191445
```

You can also view job status in a browser by navigating to the YARN ResourceManager Web UI and viewing job history server information. (For more information about checking job status and history, see "Tuning Spark" in this guide.)

Configuring Spark Applications

You can specify Spark application configuration properties as follows:

- Pass properties using the --conf command-line switch; for example:

```
spark-submit \
--class com.cloudera.example.YarnExample \
--master yarn \
--deploy-mode cluster \
--conf "spark.eventLog.dir=hdfs:///user/spark/eventlog" \
lib/yarn-example.jar \
10
```

- Specify properties in spark-defaults.conf.
- Pass properties directly to the SparkConf used to create the SparkContext in your Spark application; for example:
  - Scala:

```
val conf = new SparkConf().set("spark.dynamicAllocation.initialExecutors
", "5")
```

```
val sc = new SparkContext(conf)
```

- Python:

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext
conf = (SparkConf().setAppName('Application name'))
conf.set('spark.hadoop.avro.mapred.ignore.inputs.without.extension',
'false')
sc = SparkContext(conf = conf)
sqlContext = SQLContext(sc)
```

The order of precedence in configuration properties is:

1. Properties passed to SparkConf.
2. Arguments passed to spark-submit, spark-shell, or pyspark.
3. Properties set in spark-defaults.conf.

For more information, see Spark Configuration.

Configuring Spark application properties in spark-defaults.conf

Specify properties in the spark-defaults.conf file in the form property=value.

To create a comment, add a hash mark (#) at the beginning of a line. You cannot add comments to the end or middle of a line.

This example shows an example excerpt of a spark-defaults.conf file:

```
spark.authenticate=false
spark.driver.log.dfsDir=/user/spark/driverLogs
spark.driver.log.persistToDfs.enabled=true
spark.dynamicAllocation.enabled=true
spark.dynamicAllocation.executorIdleTimeout=60
spark.dynamicAllocation.minExecutors=0
spark.dynamicAllocation.schedulerBacklogTimeout=1
spark.eventLog.enabled=true
spark.io.encryption.enabled=false
spark.lineage.enabled=true
spark.network.crypto.enabled=false
spark.serializer=org.apache.spark.serializer.KryoSerializer
spark.shuffle.service.enabled=true
spark.shuffle.service.port=7337
spark.ui.enabled=true
spark.ui.killEnabled=true
spark.yarn.access.hadoopFileSystems=s3a://bucket1,s3a://bucket2
```

Cloudera recommends placing configuration properties that you want to use for every application in spark-defaults.conf. See Application Properties for more information.

Configuring Spark application logging properties

Before you begin

Minimum Required Role: Configurator (also provided by Cluster Administrator, Full Administrator)

About this task

To configure only the logging threshold level, follow the procedure in Configuring Logging Thresholds. To configure any other logging property, do the following:

Procedure

1. In the Cloudera Data Platform (CDP) Management Console, go to Data Hub Clusters.
2. Find and select the cluster you want to configure.
3. Click the link for the Cloudera Manager URL.
4. Go to the Spark service.
5. Click the Configuration tab.
6. Select Scope Gateway .
7. Select Category Advanced .
8. Locate the Spark Client Advanced Configuration Snippet (Safety Valve) for spark-conf/log4j.properties property.
9. Specify log4j properties. If more than one role group applies to this configuration, edit the value for the appropriate role group. See Modifying Configuration Properties Using Cloudera Manager.
10. Enter a Reason for change, and then click Save Changes to commit the changes.
11. Deploy the client configuration.

Submitting Spark applications

To submit an application consisting of a Python file or a compiled and packaged Java or Spark JAR, use the spark-submit script.

spark-submit Syntax

```
spark-submit --option value \
  application jar | python file [application arguments]
```

Example: Running SparkPi on YARN demonstrates how to run one of the sample applications, SparkPi, packaged with Spark. It computes an approximation to the value of pi.

Table 1: spark-submit Arguments

Option	Description
application jar	Path to a JAR file containing a Spark application. For the client deployment mode, the path must point to a local file. For the cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster; see Advanced Dependency Management.
python file	Path to a Python file containing a Spark application. For the client deployment mode, the path must point to a local file. For the cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster; see Advanced Dependency Management.
application arguments	Arguments to pass to the main method of your application.

spark-submit command options

You specify spark-submit options using the form --*option value* instead of --*option=value* . (Use a space instead of an equals sign.)

Option	Description
class	For Java and Scala applications, the fully qualified classname of the class containing the main method of the application. For example, org.apache.spark.examples.SparkPi.
conf	Spark configuration property in *key=value* format. For values that contain spaces, surround "*key=value*" with quotes (as shown).
deploy-mode	Deployment mode: *cluster* and *client*. In cluster mode, the driver runs on worker hosts. In client mode, the driver runs locally as an external client. Use cluster mode with production jobs; client mode is more appropriate for interactive and debugging uses, where you want to see your application output immediately. To see the effect of the deployment mode when running on YARN, see Deployment Modes. Default: client.
driver-class-path	Configuration and classpath entries to pass to the driver. JARs added with --jars are automatically included in the classpath.
driver-cores	Number of cores used by the driver in cluster mode. Default: 1.
driver-memory	Maximum heap size (represented as a JVM string; for example 1024m, 2g, and so on) to allocate to the driver. Alternatively, you can use the spark.driver.memory property.
files	Comma-separated list of files to be placed in the working directory of each executor. For the client deployment mode, the path must point to a local file. For the cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster; see Advanced Dependency Management.
jars	Additional JARs to be loaded in the classpath of drivers and executors in cluster mode or in the executor classpath in client mode. For the client deployment mode, the path must point to a local file. For the cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster; see Advanced Dependency Management.
master	The location to run the application.
packages	Comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths. The local Maven, Maven central, and remote repositories specified in repositories are searched in that order. The format for the coordinates is groupId:artifactId:version.
py-files	Comma-separated list of .zip, .egg, or .py files to place on PYTHONPATH. For the client deployment mode, the path must point to a local file. For the cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster; see Advanced Dependency Management.
repositories	Comma-separated list of remote repositories to search for the Maven coordinates specified in packages.

Table 2: Master Values

Master	Description
local	Run Spark locally with one worker thread (that is, no parallelism).
local[*K*]	Run Spark locally with *K* worker threads. (Ideally, set this to the number of cores on your host.)
local[*]	Run Spark locally with as many worker threads as logical cores on your host.
yarn	Run using a YARN cluster manager. The cluster location is determined by *HADOOP_CONF_DIR* or *YARN_CONF_DIR*. See Configuring the Environment.

Spark cluster execution overview

Spark orchestrates its operations through the driver program. When the driver program is run, the Spark framework initializes executor processes on the cluster hosts that process your data. The following occurs when you submit a Spark application to a cluster:

1. The driver is launched and invokes the main method in the Spark application.

2. The driver requests resources from the cluster manager to launch executors.

3. The cluster manager launches executors on behalf of the driver program.

4. The driver runs the application. Based on the transformations and actions in the application, the driver sends tasks to executors.

5. Tasks are run on executors to compute and save results.

6. If dynamic allocation is enabled, after executors are idle for a specified period, they are released.

7. When driver's main method exits or calls SparkContext.stop, it terminates any outstanding executors and releases resources from the cluster manager.

Canary test for pyspark command

The following example shows a simple `pyspark` session that refers to the SparkContext, calls the collect() function which runs a Spark 2 job, and writes data to HDFS. This sequence of operations helps to check if there are obvious configuration issues that prevent Spark jobs from working at all. For the HDFS path for the output directory, substitute a path that exists on your own system.

```
$ hdfs dfs -mkdir /user/systest/spark
$ pyspark
...
SparkSession available as 'spark'.
>>> strings = ["one","two","three"]
>>> s2 = sc.parallelize(strings)
>>> s3 = s2.map(lambda word: word.upper())
>>> s3.collect()
['ONE', 'TWO', 'THREE']
>>> s3.saveAsTextFile('hdfs:///user/systest/spark/canary_test')
>>> quit()
$ hdfs dfs -ls /user/systest/spark
Found 1 items
drwxr-xr-x   - systest supergroup          0 2016-08-26 14:41 /user/systest/
spark/canary_test
$ hdfs dfs -ls /user/systest/spark/canary_test
Found 3 items
-rw-r--r--   3 systest supergroup          0 2016-08-26 14:41 /user/systest/
spark/canary_test/_SUCCESS
-rw-r--r--   3 systest supergroup          4 2016-08-26 14:41 /user/systest/
spark/canary_test/part-00000
-rw-r--r--   3 systest supergroup         10 2016-08-26 14:41 /user/systest/
spark/canary_test/part-00001
$ hdfs dfs -cat /user/systest/spark/canary_test/part-00000
ONE
$ hdfs dfs -cat /user/systest/spark/canary_test/part-00001
TWO
THREE
```

Fetching Spark Maven dependencies

The Maven coordinates are a combination of groupId, artifactId and version. The groupId and artifactId are the same as for the upstream Apache Spark project. For example, for spark-core, groupId is org.apache.spark, and artifactId is spark-core_2.11, both the same as the upstream project. The version is different for the Cloudera packaging: see the release notes for the exact name depending on which release you are using.

Accessing the Spark History Server

You can access the Spark History Server for your Spark cluster from the Cloudera Data Platform (CDP) Management Console interface.

Procedure

1. In the Management Console, navigate to your Spark cluster (Data Hub Clusters<Cluster Name>).
2. Select the Gateway tab.
3. Click the URL for Spark History Server.

Running Spark applications on YARN

When Spark applications run on a YARN cluster manager, resource management, scheduling, and security are controlled by YARN.

Spark on YARN deployment modes

In YARN, each application instance has an ApplicationMaster process, which is the first container started for that application. The application is responsible for requesting resources from the ResourceManager. Once the resources are allocated, the application instructs NodeManagers to start containers on its behalf. ApplicationMasters eliminate the need for an active client: the process starting the application can terminate, and coordination continues from a process managed by YARN running on the cluster.

Cluster Deployment Mode

In cluster mode, the Spark driver runs in the ApplicationMaster on a cluster host. A single process in a YARN container is responsible for both driving the application and requesting resources from YARN. The client that launches the application does not need to run for the lifetime of the application.
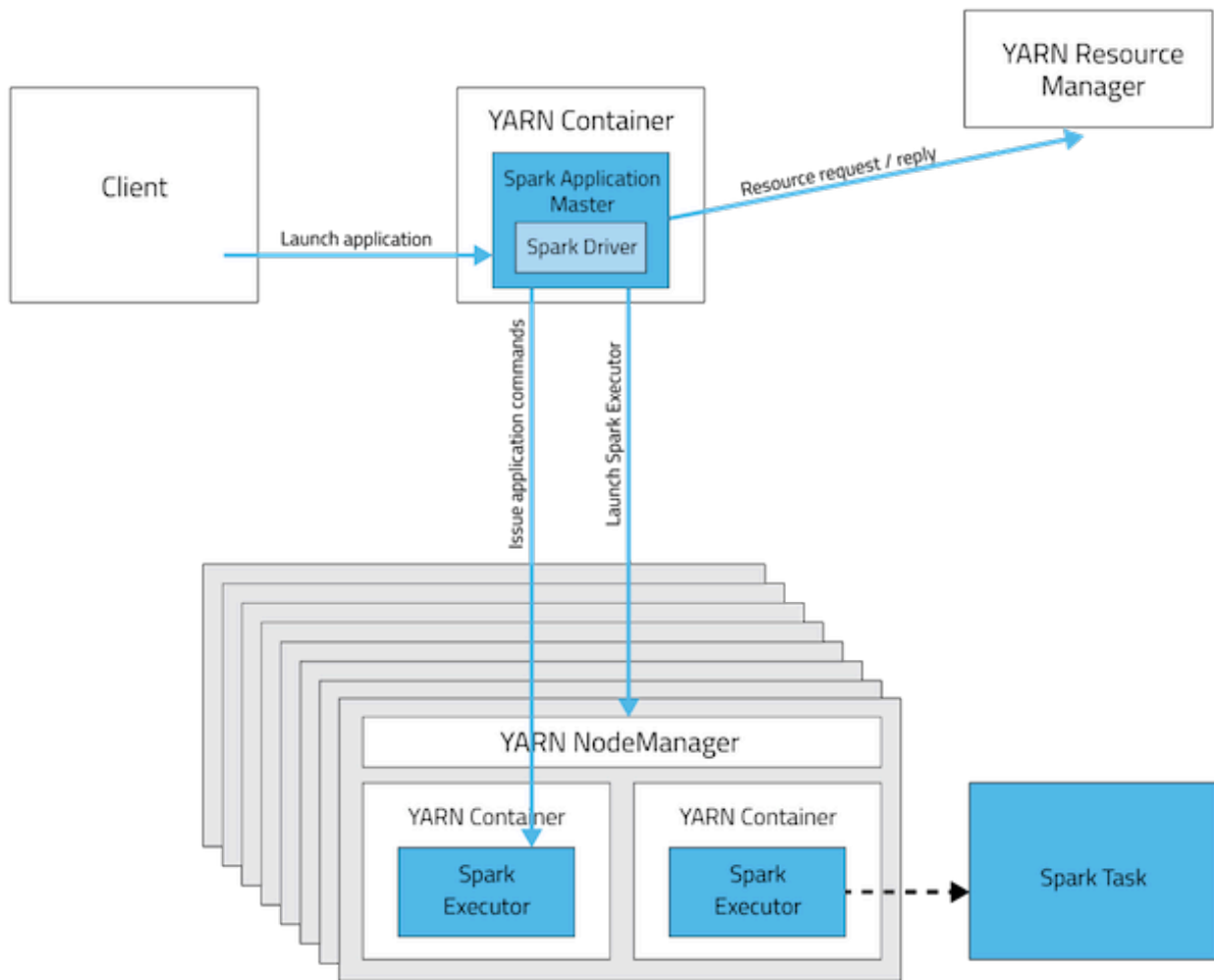
Cluster mode is not well suited to using Spark interactively. Spark applications that require user input, such as spark-shell and pyspark, require the Spark driver to run inside the client process that initiates the Spark application.

Client Deployment Mode

In client mode, the Spark driver runs on the host where the job is submitted. The ApplicationMaster is responsible only for requesting executor containers from YARN. After the containers start, the client communicates with the containers to schedule work.
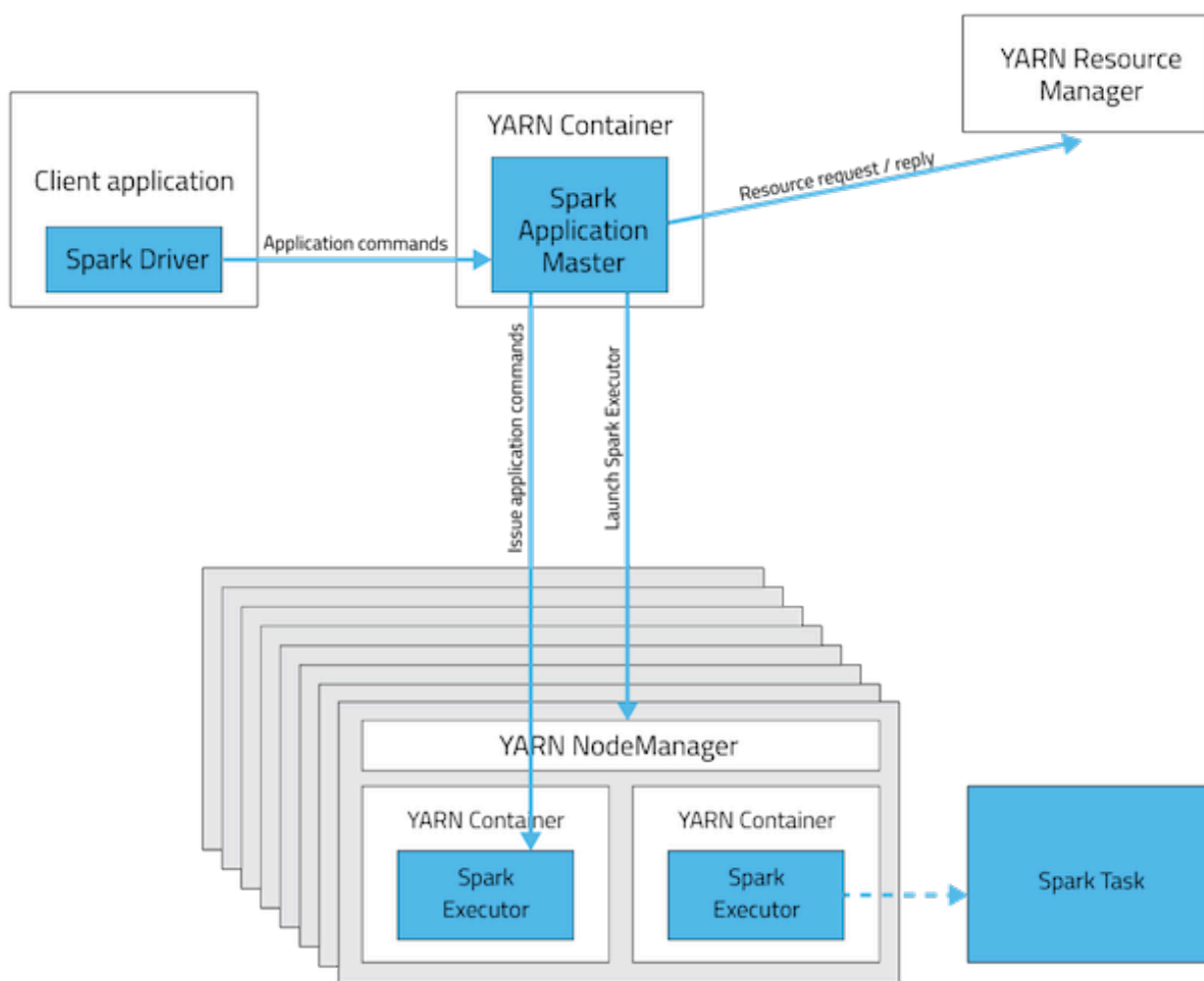
Table 3: Deployment Mode Summary

Mode	YARN Client Mode	YARN Cluster Mode
Driver runs in	Client	ApplicationMaster
Requests resources	ApplicationMaster	ApplicationMaster
Starts executor processes	YARN NodeManager	YARN NodeManager
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers
Supports Spark Shell	Yes	No

Submitting Spark Applications to YARN

To submit an application to YARN, use the spark-submit script and specify the --master yarn flag. For other spark-submit options, see spark-submit command options on page 10.

Monitoring and Debugging Spark Applications

To obtain information about Spark application behavior you can consult YARN logs and the Spark web application UI. These two methods provide complementary information. For information how to view logs created by Spark applications and the Spark web application UI, see Monitoring Spark Applications.

Example: Running SparkPi on YARN

These examples demonstrate how to use spark-submit to submit the SparkPi Spark example application with various options. In the examples, the argument passed after the JAR controls how close to pi the approximation should be.

In a CDP deployment, *SPARK_HOME* defaults to /opt/cloudera/parcels/CDH/lib/spark. The shells are also available from /bin.

Running SparkPi in YARN Cluster Mode

To run SparkPi in cluster mode:

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode cluster /opt/cloudera/parcels/CDH/jars/spark-examples*.jar 10
```

The command prints status until the job finishes or you press `control-C`. Terminating the spark-submit process in cluster mode does not terminate the Spark application as it does in client mode. To monitor the status of the running application, run yarn application   -list.

Running SparkPi in YARN Client Mode

To run SparkPi in client mode:

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode client SPARK_HOME/lib/spark-examples.jar 10
```

Running Python SparkPi in YARN Cluster Mode

1. Unpack the Python examples archive:

   ```
   sudo su gunzip SPARK_HOME/lib/python.tar.gz
   sudo su tar xvf SPARK_HOME/lib/python.tar
   ```

2. Run the pi.py file:

   ```
   spark-submit --master yarn --deploy-mode cluster SPARK_HOME/lib/pi.py 10
   ```

Configuring Spark on YARN Applications

In addition to spark-submit Options, options for running Spark applications on YARN are listed in spark-submit on YARN Options.

Table 4: spark-submit on YARN Options

Option	Description
archives	Comma-separated list of archives to be extracted into the working directory of each executor. For the client deployment mode, the path must point to a local file. For the cluster deployment mode, the path can be either a local file or a URL globally visible inside your cluster; see Advanced Dependency Management.
executor-cores	Number of processor cores to allocate on each executor. Alternatively, you can use the spark.executor.cores property.
executor-memory	Maximum heap size to allocate to each executor. Alternatively, you can use the spark.executor.memory property.
num-executors	Total number of YARN containers to allocate for this application. Alternatively, you can use the spark.executor.instances property. If dynamic allocation is enabled, the initial number of executors is the greater of this value or the spark.dynamicAllocation.initialExecutors value.

Option	Description
queue	YARN queue to submit to. For more information, see Assigning Applications and Queries to Resource Pools. Default: default.

During initial installation, Cloudera Manager tunes properties according to your cluster environment.

In addition to the command-line options, the following properties are available:

Property	Description
spark.yarn.driver.memoryOverhead	Amount of extra off-heap memory that can be requested from YARN per driver. Combined with spark.driver.memory, this is the total memory that YARN can use to create a JVM for a driver process.
spark.yarn.executor.memoryOverhead	Amount of extra off-heap memory that can be requested from YARN, per executor process. Combined with spark.executor.memory, this is the total memory YARN can use to create a JVM for an executor process.

Dynamic allocation

Dynamic allocation allows Spark to dynamically scale the cluster resources allocated to your application based on the workload. When dynamic allocation is enabled and a Spark application has a backlog of pending tasks, it can request executors. When the application becomes idle, its executors are released and can be acquired by other applications.

In Cloudera Data Platform (CDP), dynamic allocation is enabled by default. The table below describes properties to control dynamic allocation.

To disable dynamic allocation, set spark.dynamicAllocation.enabled to false. If you use the --num-executors command-line argument or set the spark.executor.instances property when running a Spark application, the number of initial executors is the greater of spark.executor.instances or spark.dynamicAllocation.initialExecutors.

For more information on how dynamic allocation works, see resource allocation policy.

When Spark dynamic resource allocation is enabled, all resources are allocated to the first submitted job available causing subsequent applications to be queued up. To allow applications to acquire resources in parallel, allocate resources to pools and run the applications in those pools and enable applications running in pools to be preempted.

If you are using Spark Streaming, see the recommendation in Spark Streaming and Dynamic Allocation.

Table 5: Dynamic Allocation Properties

Property	Description
spark.dynamicAllocation.executorIdleTimeout	The length of time executor must be idle before it is removed. Default: 60 s.
spark.dynamicAllocation.enabled	Whether dynamic allocation is enabled. Default: true.
spark.dynamicAllocation.initialExecutors	The initial number of executors for a Spark application when dynamic allocation is enabled. If spark.executor.instances (or its equivalent command-line argument, --num-executors) is set to a higher number, that number is used instead. Default: 1.
spark.dynamicAllocation.minExecutors	The lower bound for the number of executors. Default: 0.
spark.dynamicAllocation.maxExecutors	The upper bound for the number of executors. Default: Integer.MAX_VALUE.

Property	Description
spark.dynamicAllocation.schedulerBacklog Timeout	The length of time pending tasks must be backlogged before new executors are requested. Default: 1 s.

Submitting Spark applications using Livy

Apache Livy is a Spark service that allows local and remote applications to interact with Apache Spark over a REST interface.

You can use Livy to submit and manage Spark jobs on a cluster. Livy extends Spark capabilities, offering additional multi-tenancy and security features. Applications can run code inside Spark without needing to maintain a local Spark context.

Features include the following:

- Jobs can be submitted from anywhere, using the REST API.
- Livy supports user impersonation: the Livy server submits jobs on behalf of the user who submits the requests. Multiple users can share the same server ("user impersonation" support). This is important for multi-tenant environments, and it avoids unnecessary permission escalation.
- Livy supports security features such as Kerberos authentication and wire encryption.

  - REST APIs are backed by SPNEGO authentication, which the requested user should get authenticated by Kerberos at first.
  - RPCs between Livy Server and Remote SparkContext are encrypted with SASL.
  - The Livy server uses keytabs to authenticate itself to Kerberos.

Livy supports programmatic and interactive access to Spark with Scala. For example, you can:

- Use an interactive notebook to access Spark through Livy.
- Develop a Scala, Java, or Python client that uses the Livy API. The Livy REST API supports full Spark functionality including SparkSession, and SparkSession with Hive enabled.
- Run an interactive session, provided by spark-shell, PySpark, or SparkR REPLs.
- Submit batch applications to Spark.

Code runs in a Spark context, either locally or in YARN; YARN cluster mode is recommended.

Using Livy with Spark

Scala Support

Livy supports Scala versions 2.10 and 2.11.

For default Scala builds, Spark 2.0 with Scala 2.11, Livy automatically detects the correct Scala version and associated jar files.

If you require a different Spark-Scala combination, such as Spark 2.0 with Scala 2.10, set livy.spark.scalaVersion to the desired version so that Livy uses the right jar files.

Using Livy with interactive notebooks

You can submit Spark commands through Livy from an interactive Apache Zeppelin notebook:
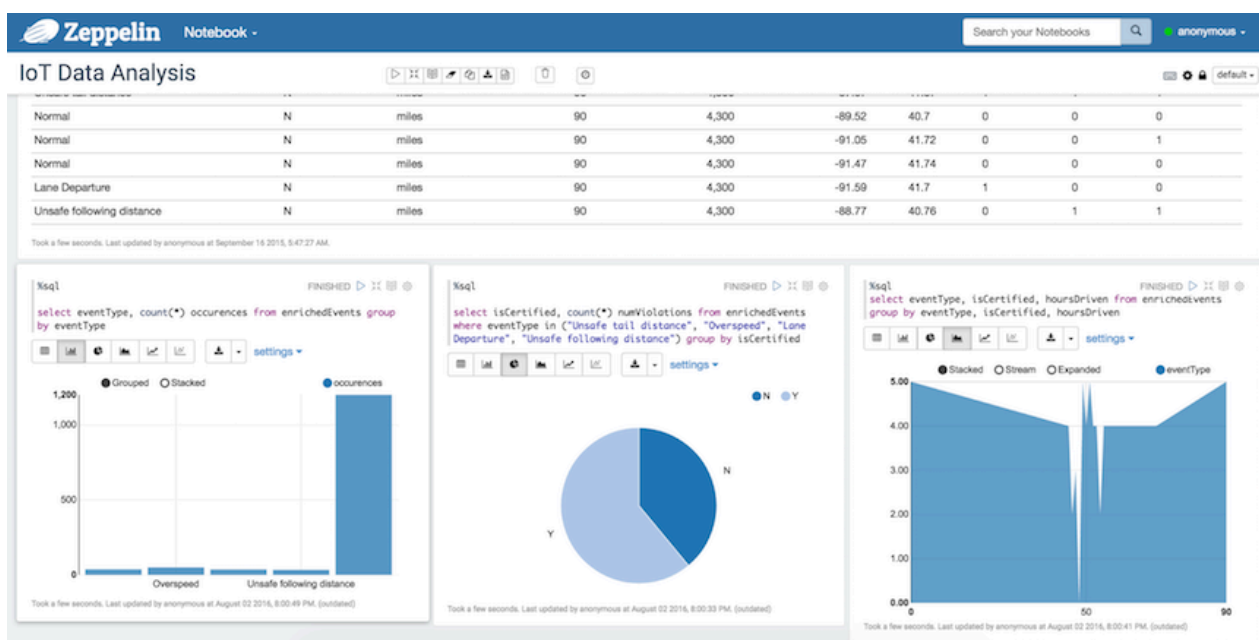
When you run code in a Zeppelin notebook using the %livy directive, the notebook offloads code execution to Livy and Spark:
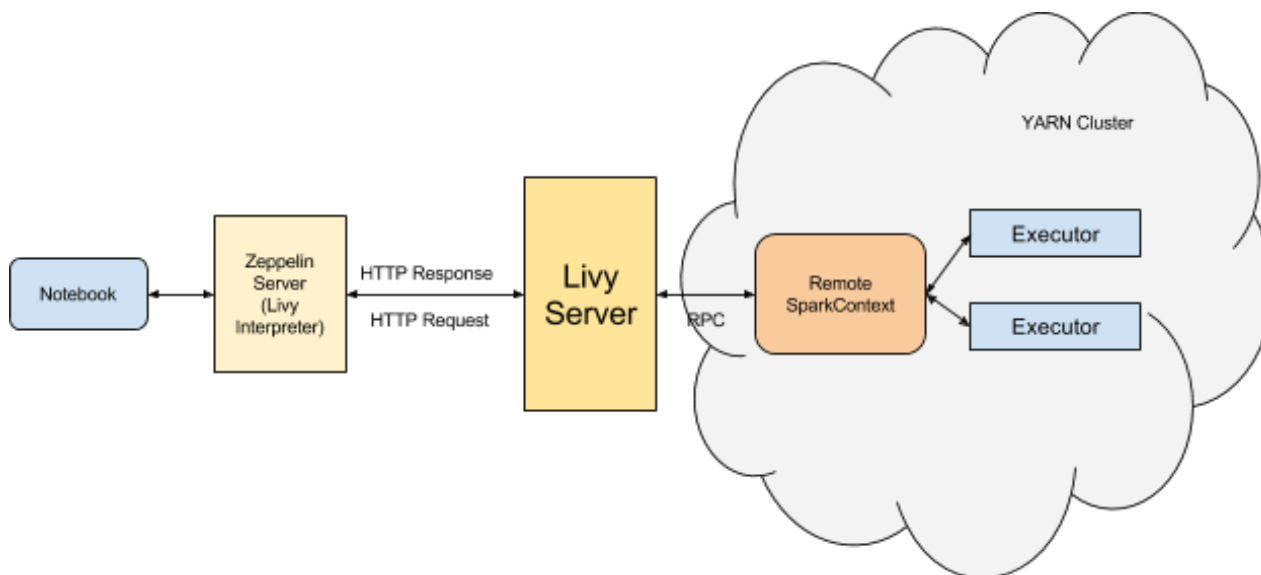
For more information about Zeppelin and Livy, see the CDP Apache Zeppelin guide.

Using the Livy API to run Spark jobs

Using the Livy API to run Spark jobs is similar to using the original Spark API.

The following two examples calculate Pi.

Calculate Pi using the Spark API:

```
def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0
    count = sc.parallelize(xrange(0, NUM_SAMPLES)).map(sample) \
```

```
      .reduce(lambda a, b: a + b)
```

Calculate Pi using the Livy API:

```
def f(_):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 <= 1 else 0
    def pi_job(context):
    count = context.sc.parallelize(range(1, samples + 1), slices).map(f).re
duce(add)
    return 4.0 * count / samples
```

There are two main differences between the two APIs:

- When using the Spark API, the entry point (SparkContext) is created by user who wrote the code. When using the Livy API, SparkContext is offered by the framework; the user does not need to create it.
- The client submits code to the Livy server through the REST API. The Livy server sends the code to a specific Spark cluster for execution.

Architecturally, the client creates a remote Spark cluster, initializes it, and submits jobs through REST APIs. The Livy server unwraps and rewraps the job, and then sends it to the remote SparkContext through RPC. While the job runs the client waits for the result, using the same path. The following diagram illustrates the process:
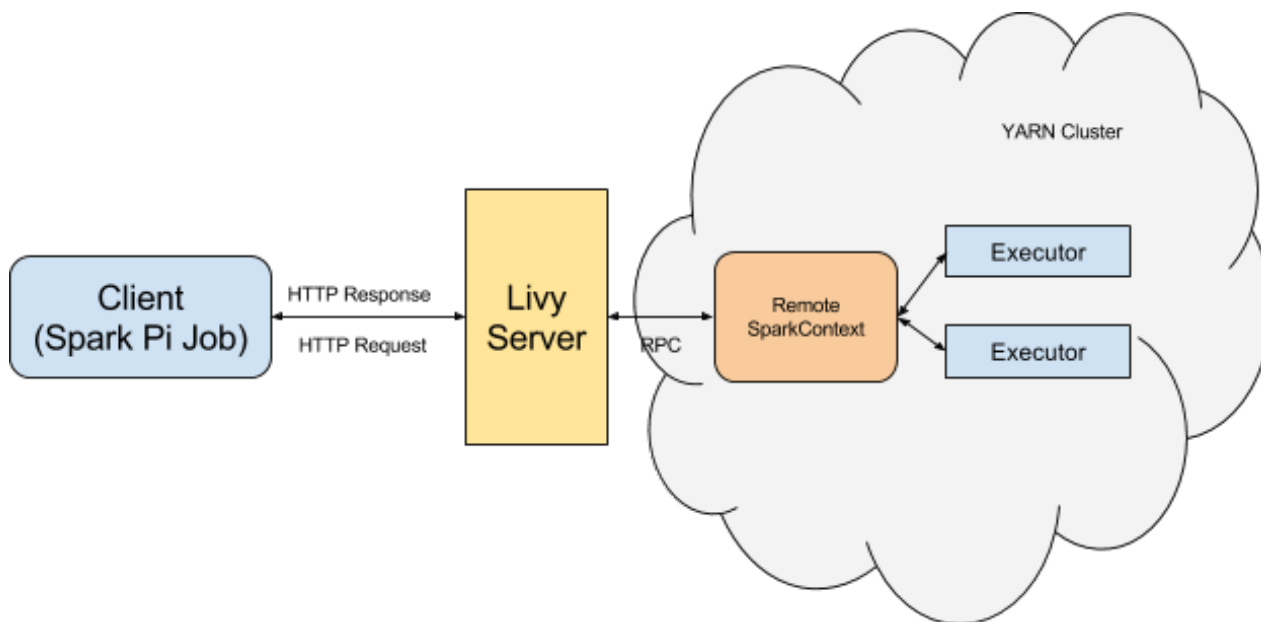
Running an interactive session with the Livy API

About this task

Running an interactive session with Livy is similar to using Spark shell or PySpark, but the shell does not run locally. Instead, it runs in a remote cluster, transferring data back and forth through a network.

The Livy REST API supports GET, POST, and DELETE calls for interactive sessions.

The following example shows how to create an interactive session, submit a statement, and retrieve the result of the statement; the return ID could be used for further queries.

Procedure

1. Create an interactive session. The following POST request starts a new Spark cluster with a remote Spark interpreter; the remote Spark interpreter is used to receive and run code snippets, and return the result.

```
POST /sessions
        host = 'http://localhost:8998'
        data = {'kind': 'spark'}
        headers = {'Content-Type': 'application/json'}
        r = requests.post(host + '/sessions', data=json.dumps(data), heade
rs=headers)
        r.json()

{u'state': u'starting', u'id': 0, u'kind': u'spark'}
```

2. Submit a statement. The following POST request submits a code snippet to a remote Spark interpreter, and returns a statement ID for querying the result after execution is finished.

```
POST /sessions/{sessionId}/statements
        data = {'code': 'sc.parallelize(1 to 10).count()'}
        r = requests.post(statements_url, data=json.dumps(data), header
s=headers)
        r.json()
{u'output': None, u'state': u'running', u'id': 0}
```

3. Get the result of a statement. The following GET request returns the result of a statement in JSON format, which you can parse to extract elements of the result.

```
GET /sessions/{sessionId}/statements/{statementId}
        statement_url = host + r.headers['location']
        r = requests.get(statement_url, headers=headers)
        pprint.pprint(r.json())

{u'id': 0,
   u'output': {u'data': {u'text/plain': u'res0: Long = 10'},
               u'execution_count': 0,
               u'status': u'ok'},
   u'state': u'available'}
```

The remainder of this section describes Livy objects and REST API calls for interactive sessions.

Livy objects for interactive sessions

See the following tables for Livy objects properties for interactive sessions.

Session Object

A session object represents an interactive shell:

Table 6: Session Object

Property	Description	Type
id	A non-negative integer that represents a specific session of interest	int
appId	Application ID for this session	string
owner	Remote user who submitted this session	string
proxyUser	User ID to impersonate when running	string
kind	Session kind (see the "kind" table below for values)	session kind
log	Log file data	list of strings

Property	Description	Type
state	Session state (see the "state" table below for values)	string
appInfo	Detailed application information	key=value map

The following values are valid for the kind property in a session object:

Table 7: Session Kind

Value	Description
spark	Interactive Scala Spark session
pyspark	Interactive Python 2 Spark session
pyspark3	Interactive Python 3 Spark session
sparkr	Interactive R Spark session

The following values are valid for the state property in a session object:

Table 8: Session Object State

Value	Description
not_started	Session has not started
starting	Session is starting
idle	Session is waiting for input
busy	Session is executing a statement
shutting_down	Session is shutting down
error	Session terminated due to an error
dead	Session exited
success	Session successfully stopped

A statement object represents the result of an execution statement.

Table 9: Statement Object

Property	Description	Type
id	A non-negative integer that represents a specific statement of interest	integer
state	Execution state (see the following "state" table for values)	statement state
output	Execution output (see the following "output" table for values)	statement output

The following values are valid for the state property in a statement object:

Table 10: Statement Object State

value	Description
waiting	Statement is queued, execution has not started
running	Statement is running
available	Statement has a response ready

value	Description
error	Statement failed
cancelling	Statement is being cancelled
cancelled	Statement is cancelled

The following values are valid for the output property in a statement object:

Table 11: Statement Object Output

Property	Description	Type
status	Execution status, such as "starting", "idle", or "available".	string
execution_count	Execution count	integer (monotonically increasing)
data	Statement output	An object mapping a mime type to the result. If the mime type is application/json, the value is a JSON value.

Setting Python path variables for Livy

To change the Python executable used by a Livy session, follow the instructions for your version of Python.

pyspark

Livy reads the path from the PYSPARK_PYTHON environment variable (this is the same as PySpark).

- If Livy is running in local mode, simply set the environment variable (this is the same as PySpark).
- If the Livy session is running in yarn-cluster mode, setspark.yarn.appMasterEnv.PYSPARK_PYTHON in the SparkConf file, so that the environment variable is passed to the driver.

pyspark3

Livy reads the path from environment variable PYSPARK3_PYTHON.

- If Livy is running in local mode, simply set the environment variable.
- If the Livy session is running in yarn-cluster mode, setspark.yarn.appMasterEnv.PYSPARK3_PYTHON in Spar kConf file, so that the environment variable is passed to the driver.

Livy API reference for interactive sessions

GET

GET /sessions returns all active interactive sessions.

Request Parameter	Description	Type
from	Starting index for fetching sessions	int
size	Number of sessions to fetch	int

Response	Description	Type
from	Starting index of fetched sessions	int
total	Number of sessions fetched	int
sessions	Session list	list

The following response shows zero active sessions:

```
{"from":0,"total":0,"sessions":[]}
```

GET /sessions/{sessionId} returns information about the specified session.

GET /sessions/{sessionId}/state returns the state of the specified session:

Response	Description	Type
id	A non-negative integer that represents a specific session	int
state	Current state of the session	string

GET /sessions/{sessionId}/logs retrieves log records for the specified session.

Request Parameters	Description	Type
from	Offset	int
size	Maximum number of log records to retrieve	int

Response	Description	Type
id	A non-negative integer that represents a specific session	int
from	Offset from the start of the log file	int
size	Number of log records retrieved	int
log	Log records	list of strings

GET /sessions/{sessionId}/statements returns all the statements in a session.

Response	Description	Type
statements	List of statements in the specified session	list

GET /sessions/{sessionId}/statements/{statementId} returns a specified statement in a session.

Response	Description	Type
statement object (for more information see "Livy Objects for Interactive Sessions")	Statement	statement object

POST

POST /sessions creates a new interactive Scala, Python, or R shell in the cluster.

Request Parameter	Description	Type
kind	Session kind (required)	session kind
proxyUser	User ID to impersonate when starting the session	string
jars	Jar files to be used in this session	list of strings
pyFiles	Python files to be used in this session	list of strings
files	Other files to be used in this session	list of strings
driverMemory	Amount of memory to use for the driver process	string
driverCores	Number of cores to use for the driver process	int
executorMemory	Amount of memory to use for each executor process	string
executorCores	Number of cores to use for each executor process	int
numExecutors	Number of executors to launch for this session	int

Request Parameter	Description	Type
archives	Archives to be used in this session	list of strings
queue	The name of the YARN queue to which the job should be submitted	string
name	Name of this session	string
conf	Spark configuration properties	Map of key=value
heartbeatTimeoutInSecond	Timeout in second to which session be orphaned	int

Response	Description	Type
session object (for more information see "Livy Objects for Interactive Sessions")	The created session	session object

The following response shows a PySpark session in the process of starting:

```
{"id":0,"state":"starting","kind":"pyspark","log":[]}
```

POST /sessions/{sessionId}/statements runs a statement in a session.

Request Parameter	Description	Type
code	The code to run	string

Response	Description	Type
statement object (for more information see "Livy Objects for Interactive Sessions")	Result of an execution statement	statement object

POST   /sessions/{sessionId}/statements/{statementId}/cancel cancels the specified statement in the session.

Response	Description	Type
cancellation message	Reports "cancelled"	string

DELETE

DELETE /sessions/{sessionId} terminates the session.

Submitting batch applications using the Livy API

About this task

Spark provides a spark-submit command for submitting batch applications. Livy provides equivalent functionality through REST APIs, using job specifications specified in a JSON document.

The following example shows a spark-submit command that submits a SparkPi job, followed by an example that uses Livy POST requests to submit the job. The remainder of this subsection describes Livy objects and REST API syntax. For additional examples and information, see the readme.rst file at https://github.com/hortonworks/livy-release/ releases/tag/HDP-2.6.0.3-8-tag.

The following command uses spark-submit to submit a SparkPi job:

```
./bin/spark-submit \
    --class org.apache.spark.examples.SparkPi \
    --master yarn \
    --deploy-mode cluster \
```

```
        --executor-memory 20G \
        /path/to/examples.jar 1000
```

To submit the SparkPi job using Livy, complete the following steps. Note: the POST request does not upload local jars to the cluster. You should upload required jar files to HDFS before running the job. This is the main difference between the Livy API and spark-submit.

Procedure

1. Form a JSON structure with the required job parameters:

```
{ "className": "org.apache.spark.examples.SparkPi",
        "executorMemory": "20g",
        "args": [2000],
        "file": "/path/to/examples.jar"
        }
```

2. Specify master and deploy mode in the livy.conf file.

3. To submit the SparkPi application to the Livy server, use the a POST   /batches request.

4. The Livy server helps launch the application in the cluster.

Livy batch object

Batch session APIs operate on batch objects, defined as follows:

Property	Description	Type
id	A non-negative integer that represents a specific batch session	int
appId	The application ID for this session	String
appInfo	Detailed application info	Map of key=value
log	Log records	list of strings
state	Batch state	string

Livy API reference for batch jobs

GET /batches returns all active batch sessions.

Request Parameters	Description	Type
from	Starting index used to fetch sessions	int
size	Number of sessions to fetch	int

Response	Description	Type
from	Starting index of fetched sessions	int
total	Number of sessions fetched	int
sessions	List of active batch sessions	list

GET /batches/{batchId} returns the batch session information as a batch object.

GET /batches/{batchId}/state returns the state of batch session:

Response	Description	Type
id	A non-negative integer that represents a specific batch session	int

Response	Description	Type
state	The current state of batch session	string

GET /batches/{batchId}/log retrieves log records for the specified batch session.

Request Parameters	Description	Type
from	Offset into log file	int
size	Max number of log lines to return	int

Response	Description	Type
id	A non-negative integer that represents a specific batch session	int
from	Offset from start of the log file	int
size	Number of log records returned	int
log	Log records	list of strings

POST /batches creates a new batch environment and runs a specified application:

Request Body	Description	Type
file	File containing the application to run (required)	path
proxyUser	User ID to impersonate when running the job	string
className	Application Java or Spark main class	string
args	Command line arguments for the application	list of strings
jars	Jar files to be used in this session	list of strings
pyFiles	Python files to be used in this session	list of strings
files	Other files to be used in this session	list of strings
driverMemory	Amount of memory to use for the driver process	string
driverCores	Number of cores to use for the driver process	int
executorMemory	Amount of memory to use for each executor process	string
executorCores	Number of cores to use for each executor	int
numExecutors	Number of executors to launch for this session	int
archives	Archives to be used in this session	list of strings
queue	The name of the YARN queue to which the job should be submitted	string
name	Name of this session	string
conf	Spark configuration properties	Map of key=val

Response	Description	Type
batch object (for more information see "Livy Batch Object")	The created batch object	batch object

DELETE /batches/{batchId} terminates the Batch job.

Using PySpark

Apache Spark provides APIs in non-JVM languages such as Python. Many data scientists use Python because it has a rich variety of numerical libraries with a statistical, machine-learning, or optimization focus.

Running PySpark in a virtual environment

For many PySpark applications, it is sufficient to use --py-files to specify dependencies. However, there are times when --py-files is inconvenient, such as the following scenarios:

- A large PySpark application has many dependencies, including transitive dependencies.
- A large application needs a Python package that requires C code to be compiled before installation.
- You want to run different versions of Python for different applications.

For these situations, you can create a virtual environment as an isolated Python runtime environment. CDP supports VirtualEnv for PySpark in both local and distributed environments, easing the transition from a local environment to a distributed environment.

> **Note:**
>
> This feature is currently only supported in YARN mode.

Running Spark Python applications

Accessing Spark with Java and Scala offers many advantages: platform independence by running inside the JVM, self-contained packaging of code and its dependencies into JAR files, and higher performance because Spark itself runs in the JVM. You lose these advantages when using the Spark Python API.

Managing dependencies and making them available for Python jobs on a cluster can be difficult. 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 transformations you define use any third-party libraries, such as NumPy or nltk, Spark executors require access to those libraries when they run on remote executors.

Setting the Python Path

After the Python packages you want to use are in a consistent location on your cluster, set the appropriate environment variables to the path to your Python executables as follows:

- Specify the Python binary to be used by the Spark driver and executors by setting the PYSPARK_PYTHON environment variable in spark-env.sh. You can also override the driver Python binary path individually using the PYSPARK_DRIVER_PYTHON environment variable. These settings apply regardless of whether you are using yarn-client or yarn-cluster mode.

  Make sure to set the variables using the export statement. For example:

  ```
  export PYSPARK_PYTHON=${PYSPARK_PYTHON:-<path_to_python_executable>}
  ```

  This statement uses shell parameter expansion to set the PYSPARK_PYTHON environment variable to *<path_to_python_executable>* if it is not set to something else already. If it is already set, it preserves the existing value.

  Here are some example Python binary paths:

  - Anaconda parcel: /opt/cloudera/parcels/Anaconda/bin/python
  - Virtual environment: *path/to/virtualenv*/bin/python

- If you are using yarn-cluster mode, in addition to the above, also set spark.yarn.appMasterEnv.PYSPARK_PYTH ON and spark.yarn.appMasterEnv.PYSPARK_DRIVER_PYTHON in spark-defaults.conf (using the safety valve) to the same paths.

In Cloudera Manager, set environment variables in spark-env.sh and spark-defaults.conf as follows:

Minimum Required Role: Configurator (also provided by Cluster Administrator, Full Administrator)

1. Go to the Spark service.
2. Click the Configuration tab.
3. Search for Spark Service Advanced Configuration Snippet (Safety Valve) for spark-conf/spark-env.sh.
4. Add the spark-env.sh variables to the property.
5. Search for Spark Client Advanced Configuration Snippet (Safety Valve) for spark-conf/spark-defaults.conf.
6. Add the spark-defaults.conf variables to the property.
7. Enter a Reason for change, and then click Save Changes to commit the changes.
8. Restart the service.
9. Deploy the client configuration.

Self-Contained Dependencies

In a common situation, a custom Python package contains functionality you want to apply to each element of an RDD. You can use a map() function call to make sure that each Spark executor imports the required package, before calling any of the functions inside that package. The following shows a simple example:

```
def import_my_special_package(x):
  import my.special.package
  return x

int_rdd = sc.parallelize([1, 2, 3, 4])
int_rdd.map(lambda x: import_my_special_package(x))
int_rdd.collect()
```

You create a simple RDD of four elements and call it int_rdd. Then you apply the function import_my_specia l_package to every element of the int_rdd. This function imports my.special.package and then returns the original argument passed to it. Calling this function as part of a map() operation ensures that each Spark executor imports my.special.package when needed.

If you only need a single file inside my.special.package, you can direct Spark to make this available to all executors by using the --py-files option in your spark-submit command and specifying the local path to the file. You can also specify this programmatically by using the sc.addPyFiles() function. If you use functionality from a package that spans multiple files, you can make an egg for the package, because the --py-files flag also accepts a path to an egg file.

If you have a self-contained dependency, you can make the required Python dependency available to your executors in two ways:

- If you depend on only a single file, you can use the --py-files command-line option, or programmatically add the file to the SparkContext with sc.addPyFiles(path) and specify the local path to that Python file.
- If you have a dependency on a self-contained module (a module with no other dependencies), you can create an egg or zip file of that module and use either the --py-files command-line option or programmatically add the module to theSparkContext with sc.addPyFiles(path) and specify the local path to that egg or zip file.

Note: Libraries that are distributed using the Python "wheel" mechanism cannot be used with the --py-files option.

Complex Dependencies

Some operations rely on complex packages that also have many dependencies. Although such a package is too complex to distribute as a *.py file, you can create an egg for it and all of its dependencies, and send the egg file to executors using the --py-files option.

Limitations of Distributing Egg Files on Heterogeneous Clusters

If you are running a heterogeneous cluster, with machines of different CPU architectures, sending egg files is impractical because packages that contain built-in code must be compiled for a single specific CPU architecture. Therefore, distributing an egg for complex, compiled packages like NumPy, SciPy, and pandas often fails. Instead of distributing egg files, install the required Python packages on each host of the cluster and specify the path to the Python binaries for the worker hosts to use.

Installing and Maintaining Python Environments

Installing and maintaining Python environments can be complex but allows you to use the full Python package ecosystem. Ideally, a sysadmin installs the Anaconda distribution or sets up a virtual environment on every host of your cluster with your required dependencies.

If you are using Cloudera Manager, you can deploy the Anaconda distribution as a parcel as follows:

Minimum Required Role: Cluster Administrator (also provided by Full Administrator)

1. Add the following URL https://repo.anaconda.com/pkgs/misc/parcels/ to the Remote Parcel Repository URLs as described in "Parcel Configuration Settings."
2. Download, distribute, and activate the parcel as described in "Managing Parcels."

Anaconda is installed in *parcel directory*/Anaconda, where *parcel directory* is /opt/cloudera/parcels by default, but can be changed in parcel configuration settings. The Anaconda parcel is supported by Continuum Analytics.

If you are not using Cloudera Manager, you can set up a virtual environment on your cluster by running commands on each host using Cluster SSH, Parallel SSH, or Fabric. Assuming each host has Python and pip installed, use the following commands to set up the standard data stack (NumPy, SciPy, scikit-learn, and pandas) in a virtual environment on a RHEL 6-compatible system:

```
# Install python-devel:
yum install python-devel

# Install non-Python dependencies required by SciPy that are not installed
 by default:
yum install atlas atlas-devel lapack-devel blas-devel

# install virtualenv:
pip install virtualenv

# create a new virtualenv:
virtualenv mynewenv

# activate the virtualenv:
source mynewenv/bin/activate

# install packages in mynewenv:
pip install numpy
pip install scipy
pip install scikit-learn
pip install pandas
```

Automating Spark Jobs with Oozie Spark Action

You can use Apache Spark as part of a complex workflow with multiple processing steps, triggers, and interdependencies. You can automate Apache Spark jobs using Oozie Spark action.

Before you begin
Spark2 must be installed on the node where the Oozie server is installed.

About Oozie Spark Action

If you use Apache Spark as part of a complex workflow with multiple processing steps, triggers, and interdependencies, consider using Apache Oozie to automate jobs. Oozie is a workflow engine that executes sequences of actions structured as directed acyclic graphs (DAGs). Each action is an individual unit of work, such as a Spark job or Hive query.

The Oozie "Spark action" runs a Spark job as part of an Oozie workflow. The workflow waits until the Spark job completes before continuing to the next action.

For additional information about Spark action, see Oozie Spark Action Extension in the Apache Oozie documentation. For general information about Oozie, see Overview of Oozie.

Note:

Support for yarn-client execution mode for Oozie Spark action will be removed in a future release. Oozie will continue to support yarn-cluster execution mode for Oozie Spark action.

Configure Oozie Spark Action for Spark

1. Set up .jar file exclusions.

Oozie distributes its own libraries on the ShareLib, which are included on the classpath. These .jar files may conflict with each other if some components require different versions of a library. You can use the oozie.action .sharelib.for.<action_type>.exclude=<value> property to address these scenarios.

Spark uses older jackson-* .jar versions than Oozie, which creates a runtime conflict in Oozie for Spark and generates a NoClassDefFoundError error. This can be resolved by using the oozie.action.sharelib.for.<action_ty pe>.exclude=<value> property to exclude the oozie/jackson.* .jar files from the classpath. Libraries matching the regex pattern provided as the property value will not be added to the distributed cache.

Note: spark2 ShareLib directory will not be created. The named spark directory is used for spark2 libs.

Examples

The following examples show how to use a ShareLib exclude on a Java action.

Actual ShareLib content:

```
    * /user/oozie/share/lib/lib_20180701/oozie/lib-one-1.5.jar
    * /user/oozie/share/lib/lib_20180701/oozie/lib-two-1.5.jar
    * /user/oozie/share/lib/lib_20180701/java/lib-one-2.6.jar
    * /user/oozie/share/lib/lib_20180701/java/lib-two-2.6.jar
    * /user/oozie/share/lib/lib_20180701/java/component-connector.jar
```

Setting the oozie.action.sharelib.for.java.exclude property to oozie/lib-one.*= results in the following distributed cache content:

```
    * /user/oozie/share/lib/lib_20180701/oozie/lib-two-1.5.jar
    * /user/oozie/share/lib/lib_20180701/java/lib-one-2.6.jar
```

```
      * /user/oozie/share/lib/lib_20180701/java/lib-two-2.6.jar
      * /user/oozie/share/lib/lib_20180701/java/component-connector.jar
```

Setting the oozie.action.sharelib.for.java.exclude property to oozie/lib-one.*|component-connector.jar= results in the following distributed cache content:

```
      * /user/oozie/share/lib/lib_20180701/oozie/lib-two-1.5.jar
      * /user/oozie/share/lib/lib_20180701/java/lib-one-2.6.jar
      * /user/oozie/share/lib/lib_20180701/java/lib-two-2.6.jar
```

2. Run the Oozie shareliblist command to verify the configuration. You should see spark in the results.

```
oozie admin –shareliblist spark
```

The following examples show a workflow definition XML file, an Oozie job configuration file, and a Python script for running a Spark2-Pi job.

Sample Workflow.xml file for spark2-Pi:

```
<workflow-app xmlns='uri:oozie:workflow:0.5' name='SparkPythonPi'>
        <start to='spark-node' />

        <action name='spark-node'>
          <spark xmlns="uri:oozie:spark-action:0.1">
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <master>${master}</master>
            <name>Python-Spark-Pi</name>
            <jar>pi.py</jar>
          </spark>
          <ok to="end" />
          <error to="fail" />
        </action>

        <kill name="fail">
          <message>Workflow failed, error message [${wf:errorMessage(wf:
lastErrorNode())}]</message>
        </kill>
        <end name='end' />
      </workflow-app>
```

Sample Job.properties file for spark2-Pi:

```
nameNode=hdfs://host:8020
jobTracker=host:8050
queueName=default
examplesRoot=examples
oozie.use.system.libpath=true
oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/app
s/pyspark
master=yarn-cluster
oozie.action.sharelib.for.spark=spark2
```

Sample Python script, lib/pi.py:

```
import sys
from random import random
from operator import add
from pyspark import SparkContext


if __name__ == "__main__":
"""
```

```
Usage: pi [partitions]
"""
sc = SparkContext(appName="Python-Spark-Pi")
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

count = sc.parallelize(range(1, n + 1), partitions).map(f).reduce(add)
print("Pi is roughly %f" % (4.0 * count / n))

sc.stop()
```

Troubleshooting .jar file conflicts with Oozie Spark action

When using Oozie Spark action, Oozie jobs may fail with the following error if there are .jar file conflicts between the "oozie" ShareLib and the "spark" ShareLib.

```
2018-06-04 13:27:32,652 WARN SparkActionExecutor:523 - SERVER[XXXX] USER[XXX
X] GROUP[-] TOKEN[] APP[XXXX] JOB[0000000-<XXXXX>-oozie-oozi-W] ACTION[00000
00-<XXXXXX>-oozie-oozi-W@spark2] Launcher exception: Attempt to add (hdfs://
XXXX/user/oozie/share/lib/lib_XXXXX/oozie/aws-java-sdk-kms-1.10.6.jar) multi
ple times to the distributed cache.
java.lang.IllegalArgumentException: Attempt to add (hdfs://XXXXX/user/oozie/
share/lib/lib_20170727191559/oozie/aws-java-sdk-kms-1.10.6.jar) multiple ti
mes to the distributed cache.
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources$13$a
nonfun$apply$8.apply(Client.scala:632)
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources$13$anon
fun$apply$8.apply(Client.scala:623)
at scala.collection.mutable.ArraySeq.foreach(ArraySeq.scala:74)
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources$13.ap
ply(Client.scala:623)
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources$13.a
pply(Client.scala:622)
at scala.collection.immutable.List.foreach(List.scala:381)
at org.apache.spark.deploy.yarn.Client.prepareLocalResources(Client.scala:62
2)
at org.apache.spark.deploy.yarn.Client.createContainerLaunchContext(Client.s
cala:895)
at org.apache.spark.deploy.yarn.Client.submitApplication(Client.scala:171)
at org.apache.spark.deploy.yarn.Client.run(Client.scala:1231)
at org.apache.spark.deploy.yarn.Client$.main(Client.scala:1290)
at org.apache.spark.deploy.yarn.Client.main(Client.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.ja
va:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
mpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit
$runMain(SparkSubmit.scala:750)
at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)
at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
at org.apache.oozie.action.hadoop.SparkMain.runSpark(SparkMain.java:311)
at org.apache.oozie.action.hadoop.SparkMain.run(SparkMain.java:232)
at org.apache.oozie.action.hadoop.LauncherMain.run(LauncherMain.java:58)
```

```
at org.apache.oozie.action.hadoop.SparkMain.main(SparkMain.java:62)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.jav
a:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccess
orImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.apache.oozie.action.hadoop.LauncherMapper.map(LauncherMapper.java:2
37)
at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:54)
at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:453)
at org.apache.hadoop.mapred.MapTask.run(MapTask.java:343)
at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild.java:170)
at java.security.AccessController.doPrivileged(Native Method)
at javax.security.auth.Subject.doAs(Subject.java:422)
at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformatio
n.java:1866)
at org.apache.hadoop.mapred.YarnChild.main(YarnChild.java:164)
```

Run the following commands to resolve this issue.

> **Note:**
>
> You may need to perform a backup before running the rm commands.

```
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark/aws*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark/azure*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark/hadoop-aws*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark/hadoop-azure*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark/ok*
hadoop fs -mv /user/oozie/share/lib/lib_<ts>/oozie/jackson* /user/oozie/sha
re/lib/lib_<ts>/oozie.old
```

Next, run the following command to update the Oozie ShareLib:

```
oozie admin -oozie http://<oozie-server-hostname>:11000/oozie -sharelibu
pdate
```