Apache Spark 3

Running Apache Spark Applications

Date of Publish: 2018-04-01

http://docs.hortonworks.com

Contents

Introduction

You can run Spark interactively or from a client program:

- 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.

To launch Spark applications on a cluster, you can use the spark-submit script in the Spark bin directory. 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. For more informationa about spark-submit, see the Apache Spark document "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 "Submitting Spark Applications through Livy" in this guide.

Related Information

Submitting Applications

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. Log in as a user with Hadoop Distributed File System (HDFS) access: for example, your spark user, if you defined one, or hdfs.

   When the job runs, the library is uploaded into HDFS, so the user running the job needs permission to write to HDFS.
2. Navigate to a node with a Spark client and access the spark2-client directory:

   cd /usr/hdp/current/spark2-client

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

```
./bin/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.scala: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.)

WordCount

WordCount is a simple program that counts how often a word occurs in a text file. The code builds a dataset of (String, Int) pairs called counts, and saves the dataset to a file.

The following example submits WordCount code to the Scala shell:

1. Select an input file for the Spark WordCount example.

   You can use any text file as input.
2. Log on as a user with HDFS access: for example, your spark user (if you defined one) or hdfs.

   The following example uses log4j.properties as the input file:

   cd /usr/hdp/current/spark2-client/

   su spark
3. Upload the input file to HDFS:

```
hadoop fs -copyFromLocal /etc/hadoop/conf/log4j.properties
          /tmp/data
```

4. Run the Spark shell:

```
./bin/spark-shell --master yarn-client --driver-memory 512m --executor-
memory
```

```
                512m
```

You should see output similar to the following (with additional status messages):

```
Spark context Web UI available at http://172.26.236.247:4041
Spark context available as 'sc' (master = yarn, app id =
 application_1490217230866_0002).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0.2.6.0.0-598
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java
 1.8.0_112)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

5. At the scala> prompt, submit the job by typing the following commands, replacing node names, file name, and file location with your own values:

```
val file = sc.textFile("/tmp/data")
val counts = file.flatMap(line => line.split(" ")).map(word => (word,
 1)).reduceByKey(_ + _)
counts.saveAsTextFile("/tmp/wordcount")
```

6. Use one of the following approaches to view job output:

   - View output in the Scala shell:

     ```
     scala> counts.count()
     ```

   - View the full output from within the Scala shell:

     ```
     scala> counts.toArray().foreach(println)
     ```

   - View the output using HDFS:

     a. Exit the Scala shell.

     b. View WordCount job status:

     ```
     hadoop fs -ls /tmp/wordcount
     ```

     You should see output similar to the following:

     ```
     /tmp/wordcount/_SUCCESS
     /tmp/wordcount/part-00000
     /tmp/wordcount/part-00001
     ```

     c. Use the HDFS cat command to list WordCount output:

     ```
     hadoop fs -cat /tmp/wordcount/part-00000
     ```

Running Spark in Docker Containers on YARN

About this task

Apache Spark applications usually have a complex set of required software dependencies. Spark applications may require specific versions of these dependencies (such as Pyspark and R) on the Spark executor hosts, sometimes with conflicting versions. Installing these dependencies creates package isolation and organizational challenges, which have typically been managed by specialized operations teams. Virtualization solutions such as Virtualenv or Conda can be complex and inefficient due to per-application dependency downloads.

Docker support in Apache Hadoop 3 enables you to containerize dependencies along with an application in a Docker image, which makes it much easier to deploy and manage Spark applications on YARN.

Before you begin

To enable Docker support in YARN, refer to the following documentation:

"Configure YARN for running Docker containers" in the HDP Managing Data Operating System guide.

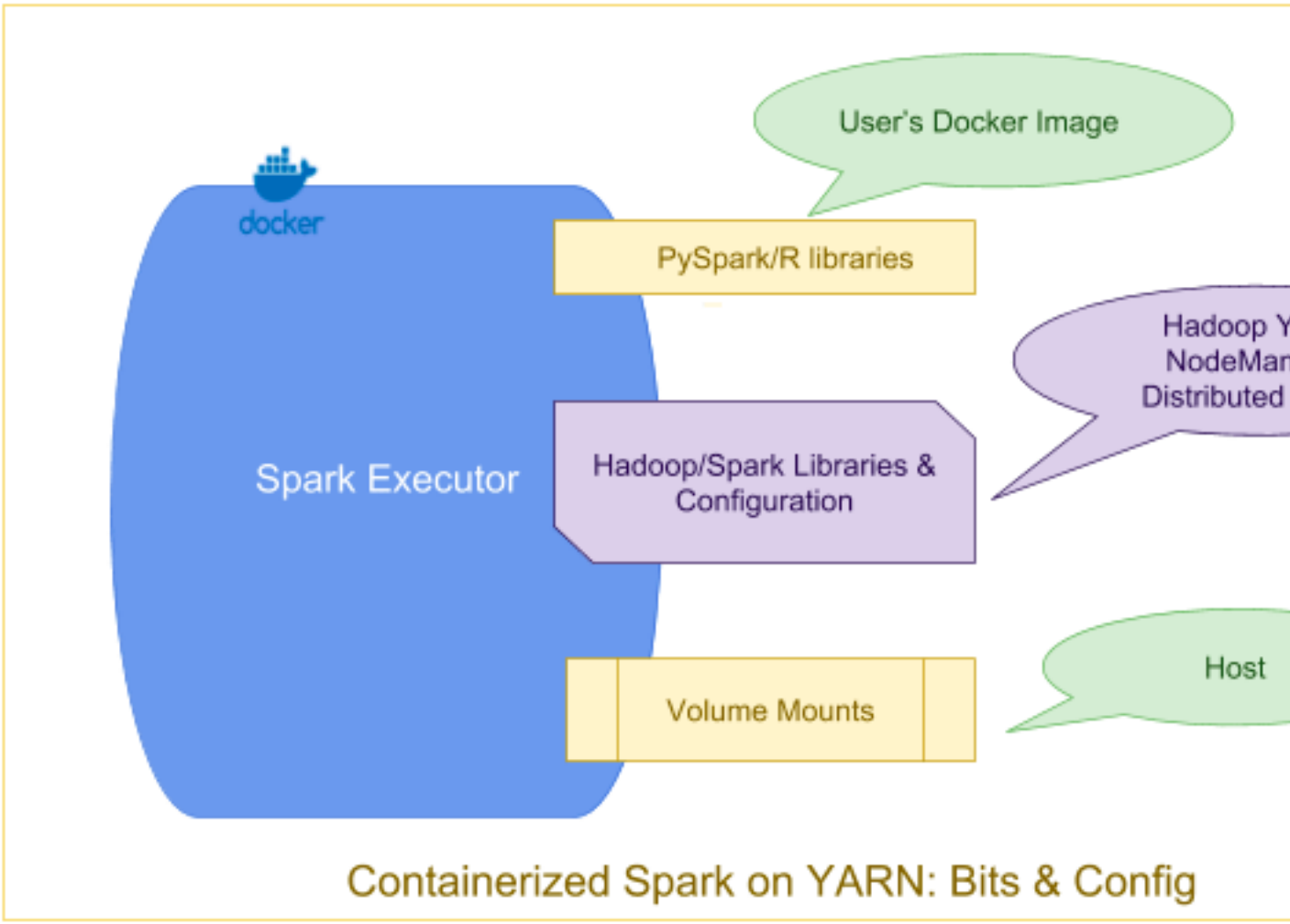
"Launching Applications Using Docker Containers" in the Apache Hadoop 3.1.0 YARN documentation.

Links to these documents are available at the bottom of this topic.

Containerized Spark: Bits and Configuration

The base Spark and Hadoop libraries and related configurations installed on the gateway hosts are distributed automatically to all of the Spark hosts in the cluster using the Hadoop distributed cache, and are mounted into the Docker containers automatically by YARN.

In addition, any binaries (–files, –jars, etc.) explicitly included by the user when the application is submitted are also made available via the distributed cache.



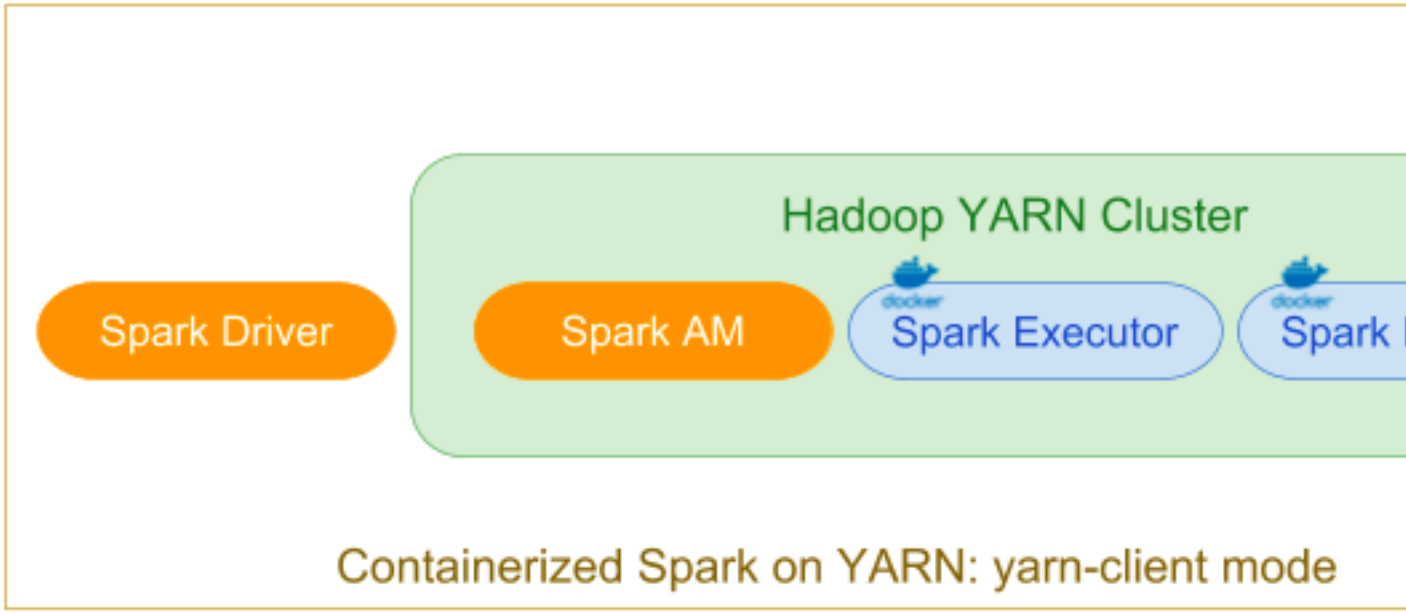
Spark Configuration

YARN Client Mode

In YARN client mode, the driver runs in the submission client's JVM on the gateway machine. Spark client mode is typically used through Spark-shell.

The YARN application is submitted as part of the SparkContext initialization at the driver. In YARN Client mode the ApplicationMaster is a proxy for forwarding YARN allocation requests, container status, etc., from and to the driver.

In this mode, the Spark driver runs on the gateway hosts as a java process, and not in a YARN container. Hence, specifying any driver-specific YARN configuration to use Docker or Docker images will not take effect. Only Spark executors will run in Docker containers.



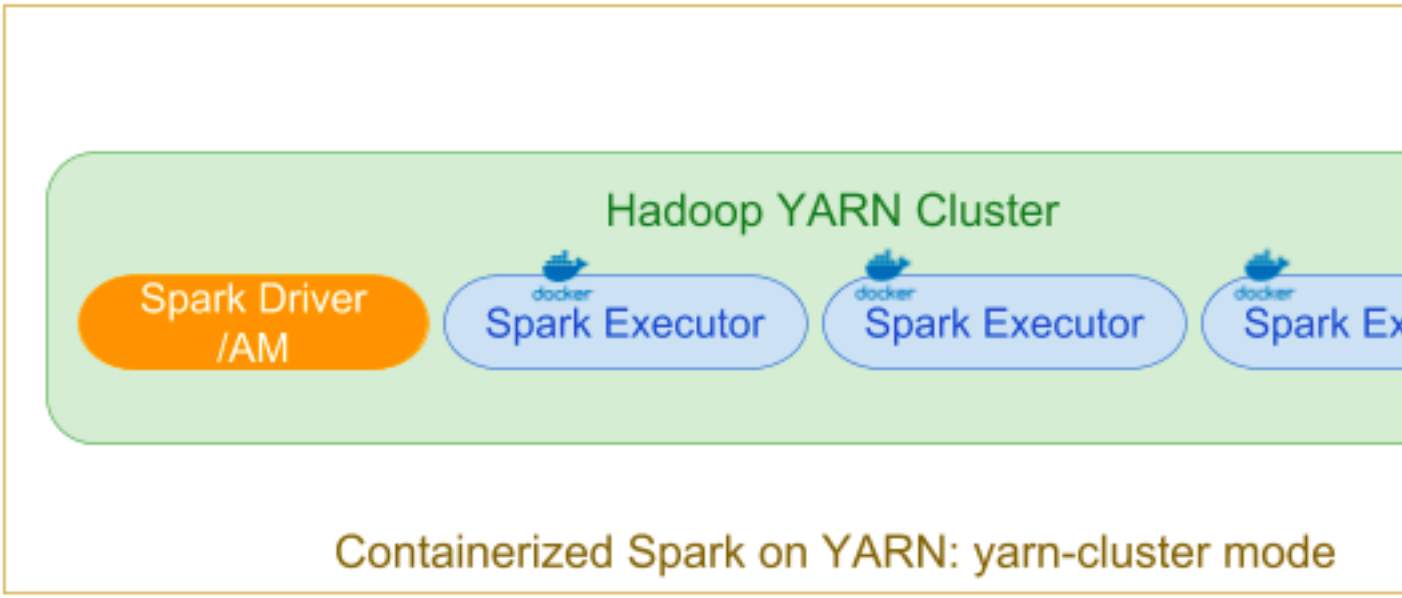
During submission, deploy mode is specified as client using –deploy-mode=client with the following executor container environment variables:

Settings for Executors

```
spark.executorEnv.YARN_CONTAINER_RUNTIME_TYPE=docker

spark.executorEnv.YARN_CONTAINER_RUNTIME_DOCKER_IMAGE=<spark executor's
 docker-image>

spark.executorEnv.YARN_CONTAINER_RUNTIME_DOCKER_MOUNTS=<any volume mounts
 needed by the spark application>
```

YARN Cluster Mode

In the "classic" distributed application YARN cluster mode, a user submits a Spark job to be executed, which is scheduled and executed by YARN. The ApplicationMaster hosts the Spark driver, which is launched on the cluster in a Docker container.



During submission, deploy mode is specified as cluster using –deploy-mode=cluster. Along with the executor's Docker container configurations, the driver/app master's Docker configurations can be set through environment variables during submission. Note that the driver's Docker image can be customized with settings that are different than the executor's image.

Additional Settings for Driver

```
spark.yarn.appMasterEnv.YARN_CONTAINER_RUNTIME_TYPE=docker

spark.yarn.appMasterEnv.YARN_CONTAINER_RUNTIME_DOCKER_IMAGE=<docker-image>

spark.yarn.appMasterEnv.YARN_CONTAINER_RUNTIME_DOCKER_MOUNTS=/etc/passwd:/
etc/passwd:ro
```

In the remainder of this topic, we will use YARN client mode.

Spark-R Example

In this example, Spark-R is used (in YARN client mode) with a Docker image that includes the R binary and the necessary R packages (rather than installing these on the host).

Spark-R Shell

```
/usr/hdp/current/spark2-client/bin/sparkR --master yarn

--conf spark.executorEnv.YARN_CONTAINER_RUNTIME_TYPE=docker

--conf spark.executorEnv.YARN_CONTAINER_RUNTIME_DOCKER_IMAGE=spark-r-demo

--conf spark.executorEnv.YARN_CONTAINER_RUNTIME_DOCKER_MOUNTS=/etc/passwd:/
etc/passwd:ro
```


Welcome to
version 2.2.0.3.0.0.0-829
SparkSession available as 'spark'.
During startup - Warning message:
In SparkR::sparkR.session() :
Version mismatch between Spark JVM
arkR package. JVM vers
was 2.2.0
> algorithms <- c("Hartigan-W
, "Lloyd", "MacQueen")
> train2 <- function(alg
hm) {
+ set.seed(42)
+ model <- kmeans(x = iris[1:4], centers = 3, algorithm = alg
+ model$withinss
+ }
>
> model.withinss <- spark.lapply(algorithms, train2)
>
> # Print the within-cluster sum of squares for the first model
> print(sort(model.withinss[[1]]))
[1] 15.15100 23.87947 39.82097
>
R binaries and Kmeans/algo
available inside the execut
docker container

Dockerfile

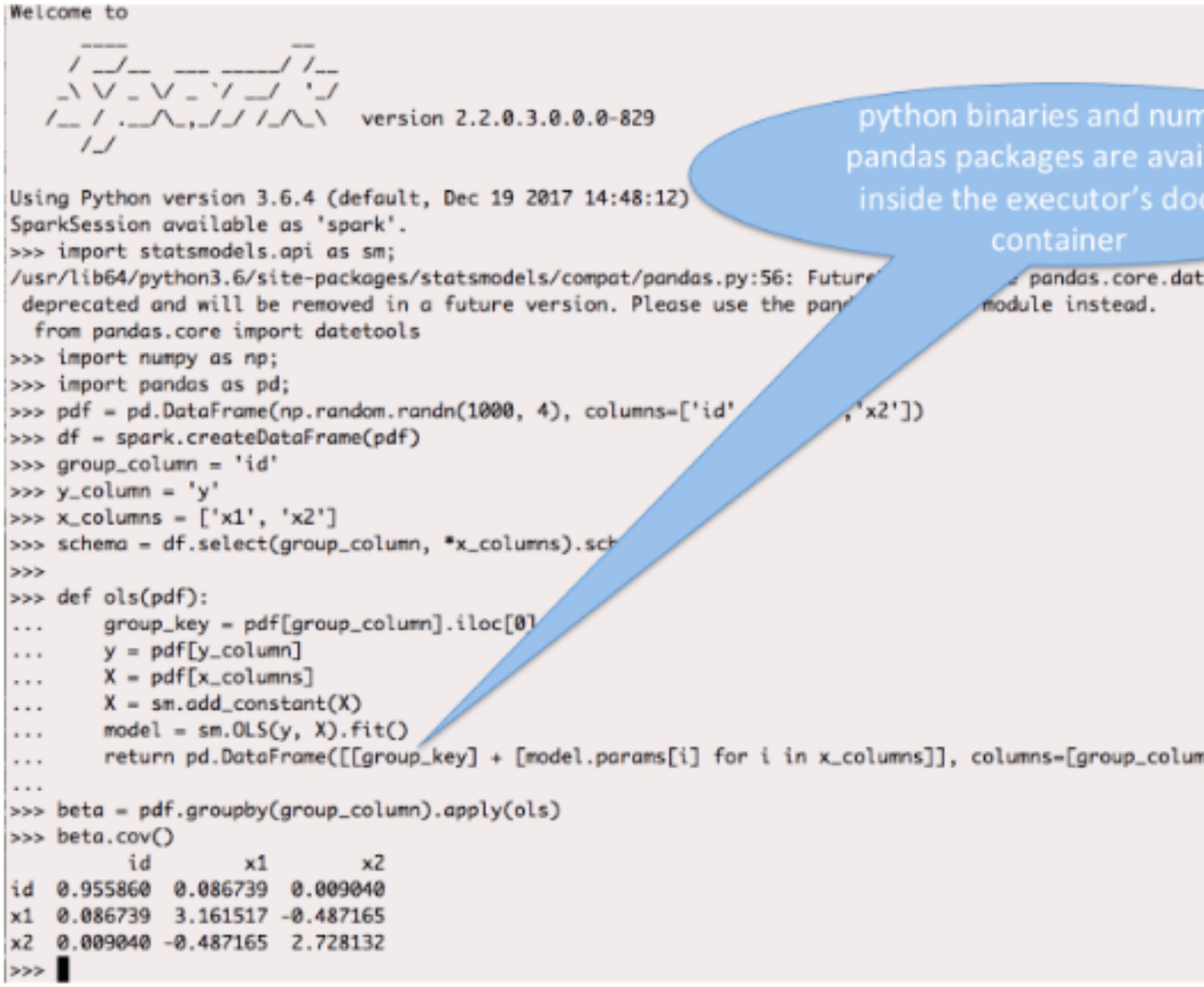
```
FROM centos

RUN yum install -y epel-release
RUN yum -y install java-1.8.0-openjdk java-1.8.0-openjdk-devel
RUN yum -y install R R-devel openssl-devel

#setup R configs
RUN echo "r <- getOption('repos'); r['CRAN'] <- 'http://cran.us
org'; options(repos = r);" > ~/.Rprofile

#Install necessary R packages
RUN Rscript -e "install.packages('yhatr')"
RUN Rscript -e "install.packages('ggplot2')"
RUN Rscript -e "install.packages('plyr')"
RUN Rscript -e "install.packages('reshape2')"
RUN Rscript -e "install.packages('forecast')"
RUN Rscript -e "install.packages('stringr')"
RUN Rscript -e "install.packages('lubridate')"
RUN Rscript -e "install.packages('randomForest')"
RUN Rscript -e "install.packages('rpart')"
RUN Rscript -e "install.packages('e1071')"
RUN Rscript -e "install.packages('kknn')"
```

PySpark Example

This example shows how to use PySpark (in YARN client mode) with Python3 (which is part of the Docker image and is not installed on the executor host) to run OLS linear regression for each group using statsmodels with all the dependencies isolated through the Docker image.

The Python version can be customized using the PYSPARK_DRIVER_PYTHON and PYSPARK_PYTHON environment variables on the Spark driver and executor respectively.

```
PYSPARK_DRIVER_PYTHON=python3.6 PYSPARK_PYTHON=python3.6 pyspark --master
 yarn --conf
spark.executorEnv.YARN_CONTAINER_RUNTIME_TYPE=docker --conf
 spark.executorEnv.
YARN_CONTAINER_RUNTIME_DOCKER_IMAGE=pandas-demo --conf spark.executorEnv.
YARN_CONTAINER_RUNTIME_DOCKER_MOUNTS=/etc/passwd:/etc/passwd:ro
```


Welcome to
version 2.2.0.3.0.0.0-829
Using Python version 3.6.4 (default, Dec 19 2017 14:48:12)
SparkSession available as 'spark'.
>>> import statsmodels.api as sm;
/usr/lib64/python3.6/site-packages/statsmodels/compat/pandas.py:56: Future
pandas.core.dat
deprecated and will be removed in a future version. Please use the pan
module instead.
from pandas.core import datetools
>>> import numpy as np;
>>> import pandas as pd;
>>> pdf = pd.DataFrame(np.random.randn(1000, 4), columns=['id'
,'x2'])
>>> df = spark.createDataFrame(pdf)
>>> group_column = 'id'
>>> y_column = 'y'
>>> x_columns = ['x1', 'x2']
>>> schema = df.select(group_column, *x_columns).sch
>>>
>>> def ols(pdf):
... group_key = pdf[group_column].iloc[0]
... y = pdf[y_column]
... X = pdf[x_columns]
... X = sm.add_constant(X)
... model = sm.OLS(y, X).fit()
... return pd.DataFrame([[group_key] + [model.params[i] for i in x_columns]], columns=[group_colum
...
>>> beta = pdf.groupby(group_column).apply(ols)
>>> beta.cov()
id x1 x2
id 0.955860 0.086739 0.009040
x1 0.086739 3.161517 -0.487165
x2 0.009040 -0.487165 2.728132
>>>
python binaries and num
pandas packages are avai
inside the executor's do
container

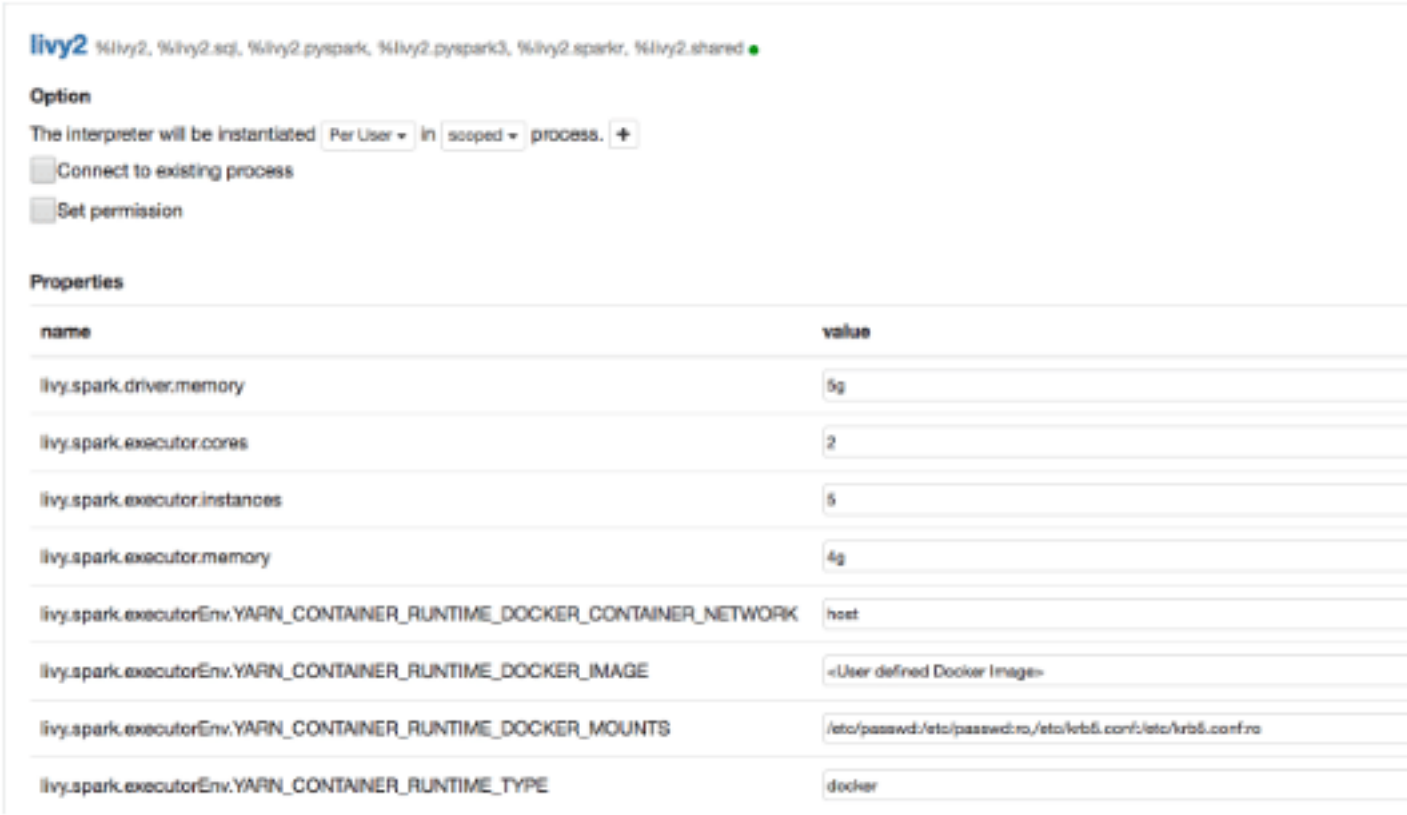
Dockerfile

```
FROM mybasecentos:latest

ENV PYTHON_VERSION 36u
RUN yum -y install python$PYTHON_VERSION python$PYTHON_VERSION-dev python$PYTHON_VERSION-pip  python$PYTHON_

ENV PYSPARK_PYTHON python3.6
ENV PYSPARK_DRIVER_PYTHON python3.6

RUN ln -s /usr/bin/python3.6 /usr/local/bin/python

RUN wget https://bootstrap.pypa.io/get-pip.py

RUN python get-pip.py

RUN pip3.6 install numpy
RUN pip3.6 install pandas
RUN pip3.6 install --upgrade --no-deps statsmodels
RUN pip3.6 install patsy
RUN pip3.6 install scikit-learn
```

Running Containerized Spark Jobs Using Zeppelin

To run containerized Spark using Apache Zeppelin, configure the Docker image, the runtime volume mounts, and the network as shown below in the Zeppelin Interpreter settings (under User (e.g.: admin) > Interpreter) in the Zeppelin UI.

Configuring the Livy Interpreter



You can also configure Docker images, volume, etc. for other Zeppelin interpreters.

You must restart the interpreter(s) in order for these new settings to take effect. You can then submit Spark applications as before in Zeppelin to launch them using Docker containers.

Related Information

Configure YARN for running Docker containers

Launching Applications Using Docker Containers

Submitting Spark Applications Through Livy

Livy is a Spark service that allows local and remote applications to interact with Apache Spark over an open source 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:

- 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.

To install Livy on an Ambari-managed cluster, see "Installing Spark Using Ambari" in this guide. For additional configuration steps, see "Configuring the Livy Server" in this guide.

Using Livy with Spark

Scala Support

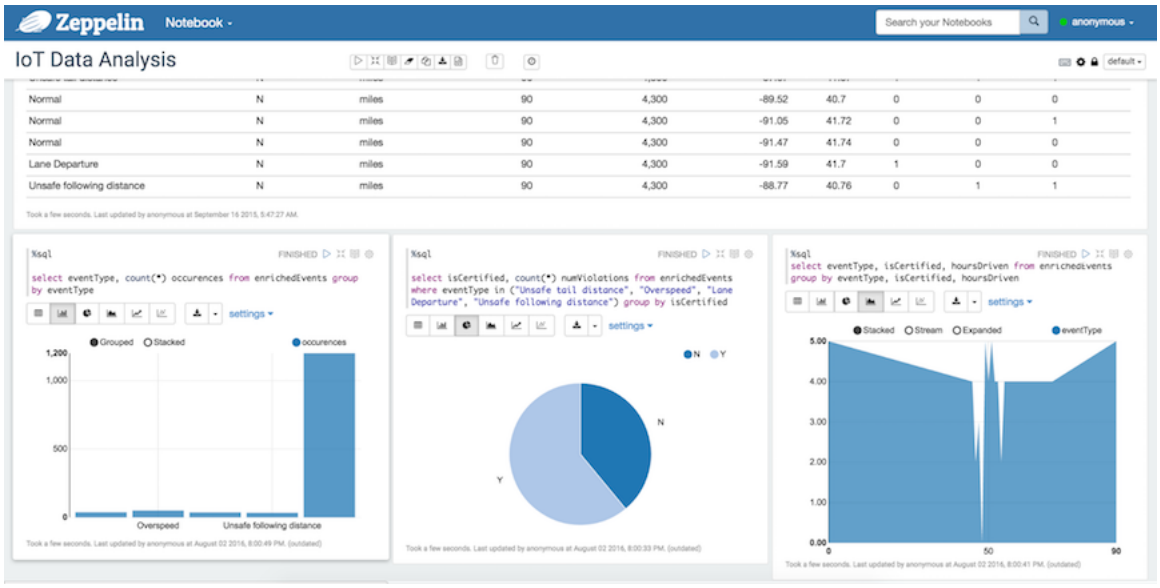
Livy supports Scala versions 2.10 and 2.11.

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

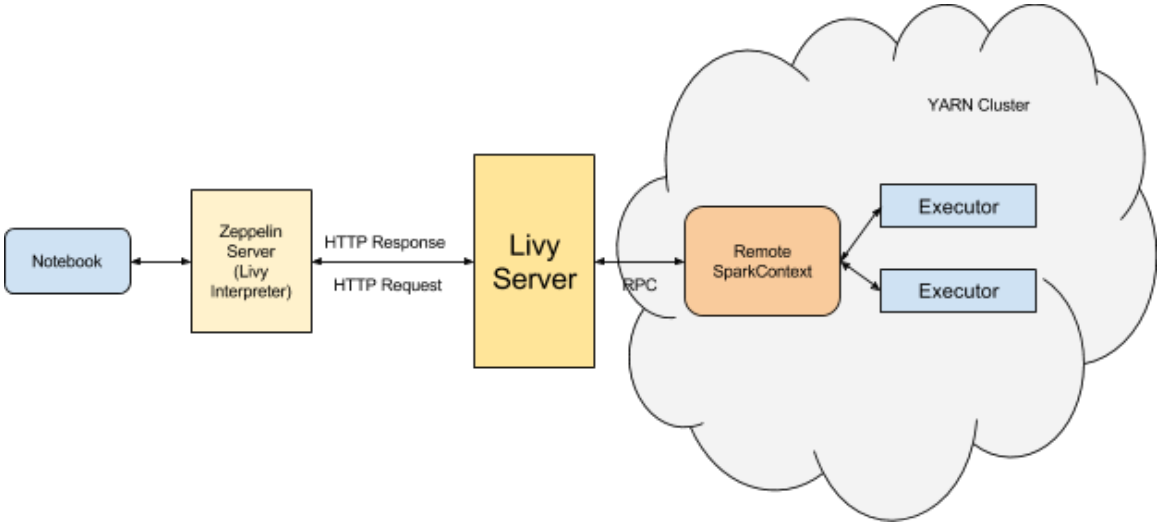
If you require a different Spark-Scala combination, such as Spark 2.0 with Scala 2.10, set livy.spark.scalaVersion to the desired version so that Livy uses the right jar files.

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 HDP Apache Zeppelin guide.

Using the Livy API to run Spark jobs: overview

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)
```

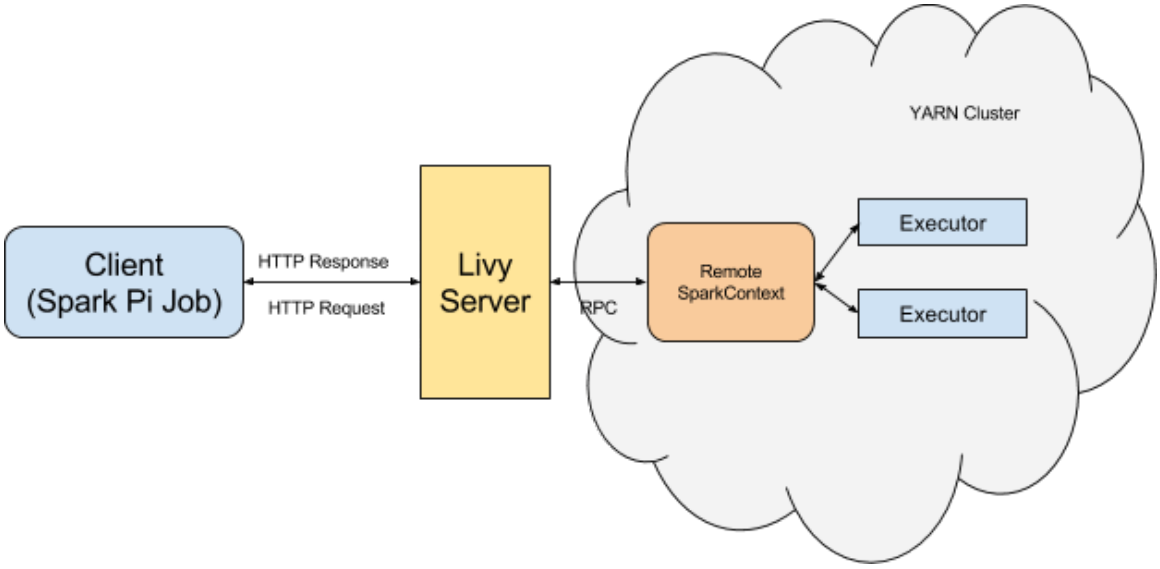
Calculate Pi using the Livy API:

```
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).reduce(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:



Related Information

Apache Spark Examples

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 execute 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),
 headers=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),
 headers=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

Session Object

A session object represents an interactive shell:

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 following "kind" table for values)	session kind
log	Log file data	list of strings
state	Session state (see the following "state" table for values)	string

Property	Description	Type
appInfo	Detailed application information	key=value map

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

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:

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

Statement Object

A statement object represents the result of an execution statement.

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:

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

Property	Description	Type
status	Execution status, such as "starting", "idle", or "available".	string

Property	Description	Type
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.

Set Path Variables for Python

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 SparkConf 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
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 execute	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
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

Response	Description	Type
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.

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. HDP 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.

Related Information

Using VirtualEnv with PySpark

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 the Apache "Oozie Spark Action Extension" documentation. For general information about Oozie and Workflow Manager, see Workflow Management under Ambari documentation.

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.

   In HDP-3.x, Spark2 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_type>.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}/
apps/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[XXXX] GROUP[-] TOKEN[] APP[XXXX] JOB[0000000-<XXXXX>-oozie-oozi-W]
 ACTION[0000000-<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) multiple 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
 times to the distributed cache.
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources
$13$anonfun$apply$8.apply(Client.scala:632)
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources
$13$anonfun$apply$8.apply(Client.scala:623)
at scala.collection.mutable.ArraySeq.foreach(ArraySeq.scala:74)
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources
$13.apply(Client.scala:623)
at org.apache.spark.deploy.yarn.Client$anonfun$prepareLocalResources
$13.apply(Client.scala:622)
at scala.collection.immutable.List.foreach(List.scala:381)
at
 org.apache.spark.deploy.yarn.Client.prepareLocalResources(Client.scala:622)
at
 org.apache.spark.deploy.yarn.Client.createContainerLaunchContext(Client.scala:895)
at org.apache.spark.deploy.yarn.Client.submitApplication(Client.scala:171)
at org.apache.spark.deploy.yarn.Client.run(Client.scala:1231)
at org.apache.spark.deploy.yarn.Client$.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.java:62)
at
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.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.java:62)
at
 sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at
 org.apache.oozie.action.hadoop.LauncherMapper.map(LauncherMapper.java:237)
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(UserGroupInformation.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/
share/lib/lib_<ts>/oozie.old
```

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

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

Related Information

Oozie Spark Action Extension