Cloudera Runtime 7.3.3

Running Apache Spark Applications

Date published: 2020-07-28
Date modified: 2026-12-13



https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2026. All rights reserved.

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

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

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

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

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

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

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

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

Contents

Introduction 5

Apache Spark 3.5.8 Requirements 5

Running Apache Spark 3.5.8 applications 6

Running your first Spark application 8

Running sample Spark applications 10

Cross-engine timezone handling for Spark 11

Configuring Spark Applications 13
- Configuring Spark application properties in spark-defaults.conf 14
- Configuring Spark application logging properties 15

Submitting Spark applications 15
- spark-submit command options 16
- Spark cluster execution overview 16
- Canary test for pyspark command 17
- Fetching Spark Maven dependencies 17
- Accessing the Spark History Server 18

Running Spark applications on YARN 18
- Spark on YARN deployment modes 18
- Submitting Spark Applications to YARN 20
- Monitoring and Debugging Spark Applications 20
- Example: Running SparkPi on YARN 21
- Configuring Spark on YARN Applications 21
- Dynamic allocation 22

Submitting Spark applications using Livy 23
- Using the Livy API to run Spark jobs 23
- Running an interactive session with the Livy REST API 24
  - Livy objects for interactive sessions 25
  - Setting Python path variables for Livy 27
  - Livy API reference for interactive sessions 27
- Submitting batch applications using the Livy REST API 29
  - Livy batch object 30
  - Livy API reference for batch jobs 30

Using Livy for Interactive SQL and BI....31
Configuring Indefinite Sessions for Livy....32
Connecting to a Named Livy Session....33
Configuring the Livy Thrift Server....34
Connecting to the Apache Livy Thrift Server....35
Troubleshooting for Livy slowness....36
Livy performance best practices....39

Using PySpark....40
Running PySpark in a virtual environment....40
Running Spark Python applications....40

Automating Spark Jobs with Oozie Spark Action....43

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.
- 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. 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 Submitting Spark applications using Livy on page 23.

Related Information

Migrating Spark applications

Deprecation Notices in Cloudera Runtime

Apache Spark 3.5.8 Requirements

The following sections describe software and hardware requirements for Spark 3.5.8

Software requirements

Each cluster host must have the following software installed:

Java

Cloudera supports JDK 17 for Apache Spark 3.5.8. Remove other JDK versions from all cluster and gateway hosts to ensure proper operation. Refer to *Support Matrix* for supported Java versions.

Scala

Scala 2.12 for Spark 3. When you submit jobs through Livy, Scala 2.12 is supported. For a non-default Spark-Scala pairing, set livy.spark.scalaVersion. See Using the Livy API to run Spark jobs and Submitting Spark applications using Livy.

Python

For Spark 3.5.x, use Python 3.8 through 3.11. For version details by Spark release, see Migrating Apache Spark applications.

For upgrade paths, JDK guidance, and application migration, see Migrating Apache Spark applications.

Hardware requirements

Spark 3 has no specific hardware requirements on top of what is required for Cloudera Runtime deployments.

Running Apache Spark 3.5.8 applications

You can run Apache Spark 3.5.8 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.

Apache Spark 3.5.8 ships in Cloudera Runtime 7.3.2.10000 SP1 and higher on the 7.3.2 line (Apache Spark 3.5.4 in 7.3.2.0). For cluster upgrade steps, see Upgrading Spark and Upgrading Apache Spark 3.5.4 to Spark 3.5.8. For application migration, behavior changes, and JDK 17 guidance, see Migrating Spark applications and the Apache Spark 3.5.8 documentation.



Important:

When Cloudera rebases the Apache Spark version shipped with or CDS, minor and maintenance releases can still change SQL behavior, configuration defaults, and serialization. Recompile Java and Scala applications built against Spark, and retest PySpark and Spark SQL workloads in a non-production environment before you upgrade production clusters.

Review *Upgrading Spark* and *Migrating Spark applications* for steps that match your source and target Spark versions.

Read the *Apache Spark migration guides* for the **exact** Apache Spark version on your cluster (for example 3.5.8 or 3.5.4).



Note: Starting with Spark 3.3, Spark uses the log4j2 dependency. Older Spark lines used log4j1. If you are using custom logging related changes, you must rewrite the original log4j properties files using log4j2 syntax such as XML, JSON, YAML, or properties format.



Important:

Adaptive Query Execution (AQE) is enabled by default in Spark 3. AQE can change join strategies at runtime, including dynamic broadcast join selection, which can differ from Spark 2.4 static plans. Workloads migrated from Spark 2 can encounter unexpected driver or executor OOM errors during joins.

For mitigation options and Cloudera Manager configuration steps, see Troubleshooting in the Spark application migration guide and Known Issues in Apache Spark (INSIGHT-9609).

Submit jobs with Livy

You can submit and manage Spark 3 jobs through Livy using programmatic and REST APIs. Livy supports Scala 2.12 for Spark 3 workloads. For API overview and Scala configuration, see Using the Livy API to run Spark jobs. For submission options and security features, see Submitting Spark applications using Livy.

spark-shell on JDK 17

On clusters that run JDK 17, the Java Platform Module System (JPMS) enforces strong encapsulation on java.base. The interactive Scala REPL used by spark-shell must resolve Java base classes through the application classpath. If the driver JVM does not include -Dscala.usejavacp=true, spark-shell initialization can fail with errors such as:

```
[init] error: error while loading Object, Missing dependency 'object scala.n
ative in compiler mirror', required by /modules/java.base/java/lang/Object.c
lass
Failed to initialize compiler: object scala in compiler mirror not found.
```

Pre-compiled jobs submitted with spark-submit and interactive pyspark sessions are not affected.

Important:

Add -Dscala.usejavacp=true to spark.driver.extraJavaOptions on JDK 17 clusters. On Cloudera Manager 7.13.1 and 7.13.2 with Cloudera Runtime 7.3.1 and 7.3.2, required JDK 17 --add-opens flags are typically pre-configured in spark.driver.extraJavaOptions; append -Dscala.usejavacp=true to that existing value rather than replacing it.

In Cloudera Manager:

1. Navigate to ClustersSpark 3Configuration.
2. Search for Spark 3 Client Advanced Configuration Snippet (Safety Valve) for spark3-conf/spark-defaults.conf (or spark-defaults.conf).
3. Append -Dscala.usejavacp=true to spark.driver.extraJavaOptions. For example:

```
spark.driver.extraJavaOptions=-Dscala.usejavacp=true --add-opens=java.ba
se/java.lang=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --a
dd-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.net=ALL-
UNNAMED --add-opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.base/j
ava.security=ALL-UNNAMED --add-opens=java.base/sun.reflect.annotation=ALL-
UNNAMED
```

4. Click Save Changes.
5. Redeploy client configuration for Spark 3 (Stale Configuration # Deploy Client Configuration).

Unsupported connectors

This release does not support the following connectors:

- SparkR

Limitations of Spark in Cloudera

Limitations of Spark (in comparison to upstream Apache Spark 3.5.8) in Cloudera are described below:

- spark.sql.orc.compression.codec config doesn't accept zsdt value.
- Push-based shuffle is not supported

Note: For a full list of **unsupported** Spark features, refer to Unsupported Apache Spark Features.

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 3 job, and writes data to HDFS. This sequence of operations helps to check if there are obvious configuration issues that prevent Spark 3 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/jdoe/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/jdoe/spark/canary_test')
>>> quit()
$ hdfs dfs -ls /user/jdoe/spark
Found 1 items
drwxr-xr-x   - jdoe spark-users  0 2016-08-26 14:41 /user/jdoe/spark/cana
ry_test
```

```
$ hdfs dfs -ls /user/jdoe/spark/canary_test
Found 3 items
-rw-r--r--   3 jdoe spark-users  0 2016-08-26 14:41 /user/jdoe/spark/canar
y_test/_SUCCESS
-rw-r--r--   3 jdoe spark-users  4 2016-08-26 14:41 /user/jdoe/spark/canary_
test/part-00000
-rw-r--r--   3 jdoe spark-users 10 2016-08-26 14:41 /user/jdoe/spark/canary_
test/part-00001
$ hdfs dfs -cat /user/jdoe/spark/canary_test/part-00000
ONE
$ hdfs dfs -cat /user/jdoe/spark/canary_test/part-00001
TWO
THREE
```

Fetching Spark 3 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_2.12, both the same as the upstream project. The version is different for the Cloudera packaging, it contains the upstream Spark version as well as the Cloudera version: *[**ARTIFACTID***]-[***UPSTREAM SPARK VERSION***].[***CDP VERSION***]-[***CLOUDERA BUILD NUMBER***]*. Use the Cloudera Runtime release notes or Maven repository documentation for your deployment to obtain the exact version string for Spark 3.5.8.

Accessing the Spark 3 History Server

The Spark 3 history server is available on port 18089, rather than port 18088 as with the Spark 2 history server.

Related Information

Upgrading Spark

Migrating Spark applications

Running your first Spark application

About this task



Important:

By default, Cloudera 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 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 2.12.18 (OpenJDK 64-Bit Server VM, Java 17.0.12)
     Type in expressions to have them evaluated.
     Type :help for more information.

     scala>
     ```

   - Python:

     ```
     $ /bin/pyspark
     Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
     Type "help", "copyright", "credits" or "license" for more information.
     ...
     Welcome to
           ____              __
          / __/__  ___ _____/ /__
         _\ \/ _ \/ _ `/ __/  '_/
        /__ / .__/\_,_/_/ /_/\_\   version ...
           /_/

     Using Python version 3.10.12 (main, Nov 20 2023, 15:14:05)
     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 \
       --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.)

Cross-engine timezone handling for Spark

Spark treats TIMESTAMP as an instant with local timezone (LTZ) by default, while Hive, Impala, and Iceberg treat it as a wall-clock value without timezone (NTZ), which can shift displayed hours across engines.

For Spark users, cross-engine consistency in the interpretation of the generic TIMESTAMP keyword is a challenge. In Spark, a TIMESTAMP is traditionally treated as an instant (a specific point in time on a global timeline), whereas Hive, Impala, and Iceberg interpret it as a wall-clock value (a literal date and time regardless of location).

Engine timestamp semantics

The discrepancy creates a shift in data values. Because Spark assumes a timestamp is an instant, it automatically converts the value to UTC when writing and shifts it back to the local session timezone when reading. If the data was originally written as a wall-clock value by Hive or Impala, Spark's attempt to correct for timezones results in the hours shifting. For example, a 12:00 PM entry might suddenly appear as 07:00 AM if the Spark session is set to EST.

Starting in Spark 3.4, Spark SQL introduced a native wall-clock type (TIMESTAMP_NTZ). However, the default behavior of the TIMESTAMP alias remains set to the instant type for backward compatibility. This creates a fundamental mismatch during table creation and metadata sharing across the platform.

Engines generally follow one of two semantic behaviors:

- Wall-clock (no time zone / NTZ): Represents an abstract "what you see is what you get" time (for example, 12:00 PM). It remains identical for all users regardless of their location.
- Instant (local time zone / LTZ): Represents a single global point in time, stored as UTC. When queried, the engine shifts the value to match the viewer's local session timezone (for example, 10:00 AM in New York appears as 3:00 PM in London).

The following table shows how each engine interprets the TIMESTAMP keyword and which specific types they support:

Table 1: Engine timestamp handling comparison

Engine	Default TIMESTAMP behavior	NTZ type name	LTZ type name
Spark lower than 3.4	Instant (LTZ)	Not supported	TIMESTAMP
Spark 3.4 and higher	Instant (LTZ)	TIMESTAMP_NTZ	TIMESTAMP_LTZ
Hive	Wall-clock (NTZ)	TIMESTAMP	TIMESTAMP WITH LOCAL TIME ZONE
Impala	Wall-clock (NTZ)	TIMESTAMP	Not supported
Iceberg	Wall-clock (NTZ)	timestamp	timestamptz
Trino	Instant (LTZ)	TIMESTAMP	TIMESTAMP WITH TIME ZONE

Timestamp handling in tables

Inconsistencies occur when data is written by one engine and read by another, leading to the hour value changing unexpectedly.

Behavior differences by Spark version

- Spark lower than 3.4: Only supports one timestamp type (LTZ), and strips the timezone during the write. Upon reading, Spark assumes the value is UTC and shifts it to the local session timezone, causing an hour offset.
- Spark 3.4 and higher: Introduces native TIMESTAMP_NTZ. However, by default, the TIMESTAMP alias still points to TIMESTAMP_LTZ to maintain backward compatibility. This version allows for alignment with Hive and Impala if configured correctly.

Interoperability matrix

The following table summarizes cross-engine behavior when creating or reading tables:

Table 2: Timestamp interoperability matrix

Action	Spark version	Table format	Resulting behavior
Create table	Lower than 3.4	Iceberg	Created as timestamptz. Impala cannot write to this table. Hive and Spark can read from and write to this table.
Create table	3.4 and higher	Iceberg	Created as timestamptz (default). Impala cannot write to this table.
Create table	3.4 and higher	Iceberg	If spark.sql.timestampType=NTZ is set, created as timestamp. Fully compatible with all engines.
Read table	Any	Hive/Parquet	Spark shifts the wall-clock time to local time. In Impala, Hive, and Spark, the hour displayed shifts due to the change in time zone interpretation, even though the underlying UTC timestamp value remains the same.

Workarounds for Spark lower than 3.4

1. For Iceberg tables:

   In Spark 3.3, Cloudera recommends using timestamp columns in an Iceberg table. Set spark.sql.iceberg.use-timestamp-without-timezone-in-new-tables to true so that when Spark creates an Iceberg table, the Spark type timestamp (with local timezone) is mapped to the Iceberg timestamp (without local timezone) type rather than the timestamptz (with local timezone) type.

   Set spark.sql.iceberg.handle-timestamp-without-timezone to true so that when Spark reads or writes an Iceberg table with an Iceberg timestamp column, the data is treated as Spark timestamp (with local timezone) data. When Hive or Impala read and write the tables created by Spark, they treat them as wall-clock timestamps.

   At the same time, Cloudera recommends setting the Spark session timezone to UTC to achieve consistent results.
2. For Hive tables:

   a. For working with Hive tables written in Parquet file format, see TIMESTAMP compatibility for Parquet files. Setting spark.sql.parquet.int96TimestampConversion=true means that the values are both read and written in a way that is interoperable with Impala.
   b. For any other Hive tables, when a table is created by Spark and read or written from Hive or Impala, or vice versa, Spark treats the timestamp as UTC and converts it to the current Spark session time, whereas Hive and Impala treat it as wall-clock values. A silent shift in timestamps occurs. In this case, Cloudera recommends setting all Spark session timezones to UTC.

Note: This compatibility workaround for Parquet files only applies to Parquet files created by Impala, and has no effect on Parquet files created by Hive, Spark, or other Java components.

Workarounds for Spark 3.4 and higher

1. For Iceberg tables:

   Spark has two timestamp types: TIMESTAMP_LTZ and TIMESTAMP_NTZ. Use the spark.sql.timestampT ype configuration to set the default timestamp type to TIMESTAMP_NTZ or TIMESTAMP_LTZ (the default is TIMESTAMP_LTZ).

   Iceberg tables have a mapping where Spark LTZ maps to Iceberg timestamptz, and Spark NTZ maps to Iceberg timestamp. When Hive or Impala read them, they treat them as wall-clock values.
2. For Hive tables:

   Same as Spark lower than 3.4. The silent shift issue still exists. Cloudera recommends setting all Spark session timezones to UTC.

Timestamp handling in views

Spark views are managed directly by Spark. When a view uses TIMESTAMP_NTZ, this type is written directly to HMS. Because Hive and its libraries do not recognize TIMESTAMP_NTZ, attempts by Hive-based tools to read the view's metadata result in failures or errors.

This lack of Iceberg-based translation for Spark views creates a compatibility gap, restricting their use in mixed-engine environments compared to Iceberg tables. You must be cautious about using timezone-less timestamps in Spark views intended for query by HMS-dependent tools.

When a view is created in Spark 3.4 and higher using the TIMESTAMP_NTZ type:

1. HMS rejection: The Hive Metastore does not recognize the name timestamp_ntz.
2. Spark fallback: To save the view, Spark stores the column in HMS as a dummy type (typically array<string>).
3. Cross-engine failure: While Spark can still read the view using internal metadata, Hive and Impala see the arra y<string> type and fail to query the column or return corrupted data.

The root causes are the lack of HMS support for Spark's TIMESTAMP_NTZ type and the mismatch between Spark and Hive timestamp types.

Workarounds for views

To share views across all engines successfully:

1. Do not create Iceberg views from Spark if TIMESTAMP_NTZ is used. If Iceberg views are created by Spark with TIMESTAMP_NTZ, the views are not readable for Hive or Impala engines.
2. Instead, create views via Hive (standard TIMESTAMP keyword) and use spark.sql.timestampType=NTZ in Spark when querying.

Related Information

TIMESTAMP compatibility for Parquet files

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=7447
spark.ui.enabled=true
spark.ui.killEnabled=true
spark.kerberos.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. Open the cluster's Cloudera Manager URL.
2. Go to the Spark service.
3. Click the Configuration tab.
4. Select Scope Gateway .
5. Select Category Advanced .
6. Locate the Gateway Logging Advanced Configuration Snippet (Safety Valve) for spark-conf/log4j.properties.
7. 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*.
8. Enter a Reason for change, and then click Save Changes to commit the changes.
9. 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 3: 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 4: 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_2.12, 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

Configure HDFS permissions on the Spark event log directory, then access the Spark History Server for your Spark cluster from the Cloudera Management Console interface.

About this task

The Spark History Server reads Spark event logs from HDFS. By default, logs are stored at /user/spark/applicationHistory. Spark 3 clusters may use /user/spark/spark3ApplicationHistory depending on your configuration. The spark user must have permission to read the event log directory before the History Server UI can list applications.

Procedure

1. Grant the spark user read, write, and execute permissions on /user/spark and its contents:

   ```
   hdfs dfs -setfacl -R -m user:spark:rwx /user/spark
   hdfs dfs -setfacl -R -m default:user:spark:rwx /user/spark
   ```

   Default ACLs ensure that all newly created files and subdirectories under /user/spark automatically inherit the same ACLs.
2. Restart the Spark History Server service.
3. In the Cloudera Management Console, navigate to your Spark cluster: Data Hub ClustersCluster Name>
4. Select the Gateway tab.
5. Click the URL for Spark History Server.

Running Spark applications on YARN

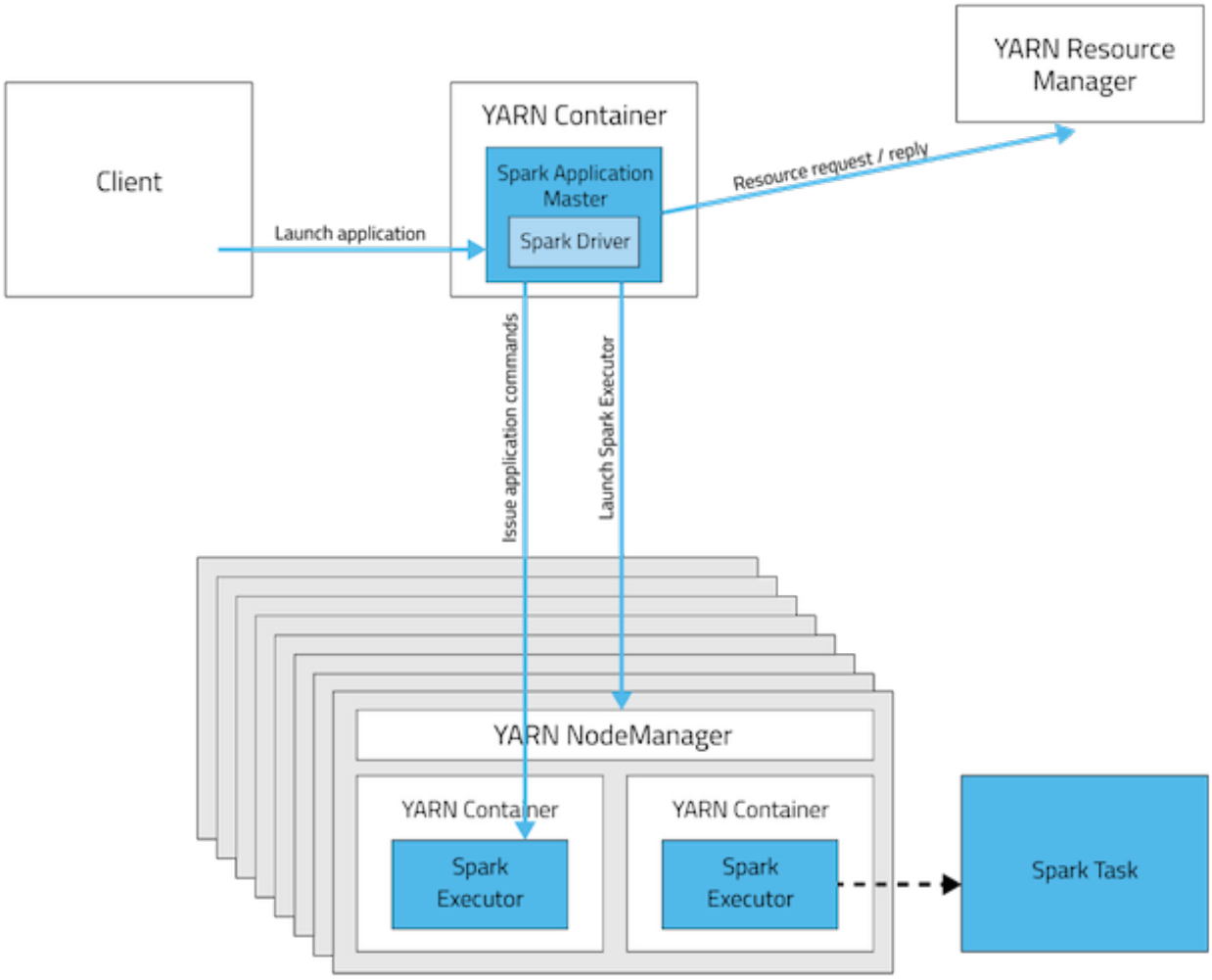
When Spark applications run on a YARN cluster manager, resource management, scheduling, and security are controlled by YARN.

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.

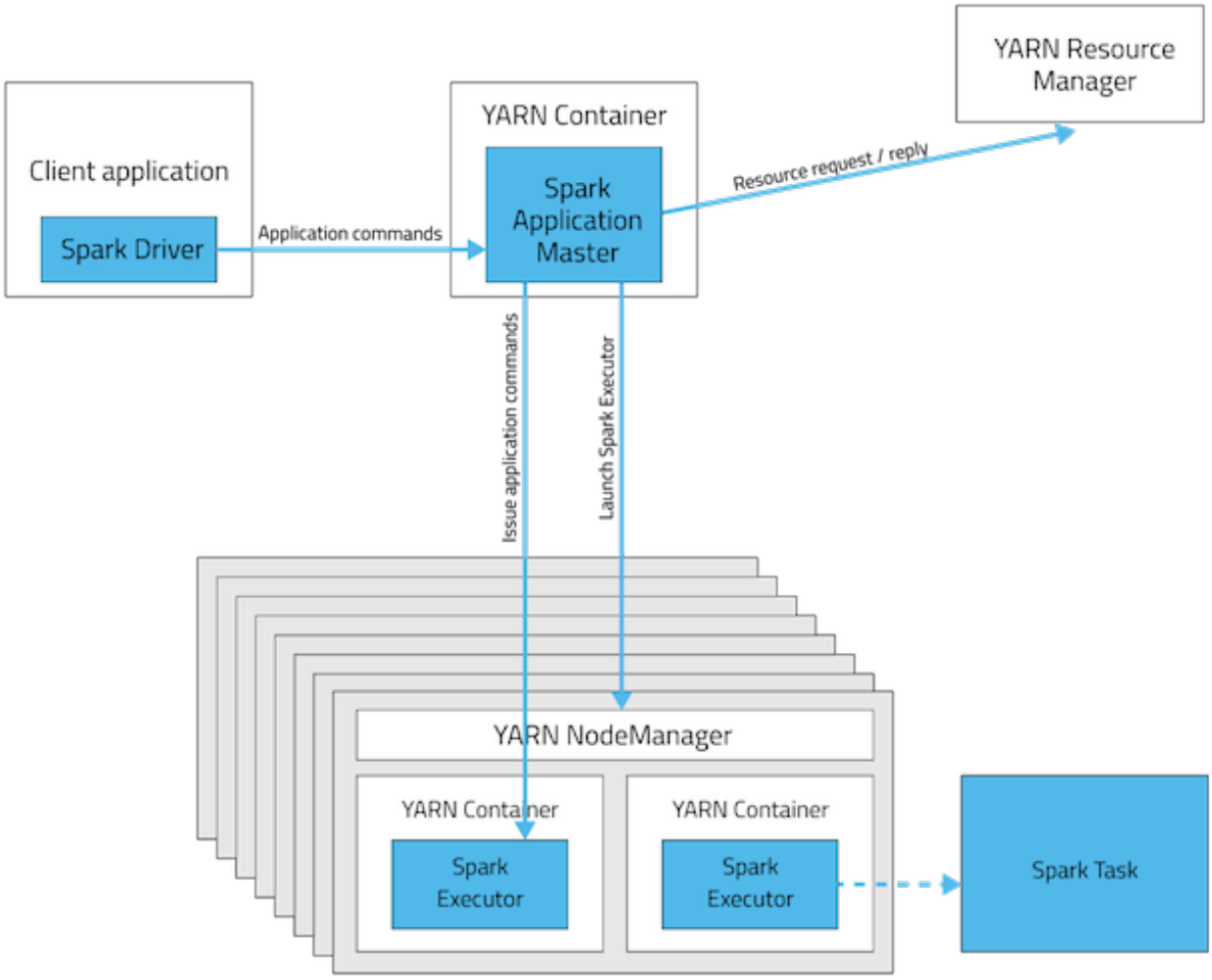


Table 5: Deployment Mode Summary

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 16.

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 Cloudera 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 6: 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, 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 7: 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.schedulerBacklogTimeout	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 the Livy API to run Spark jobs

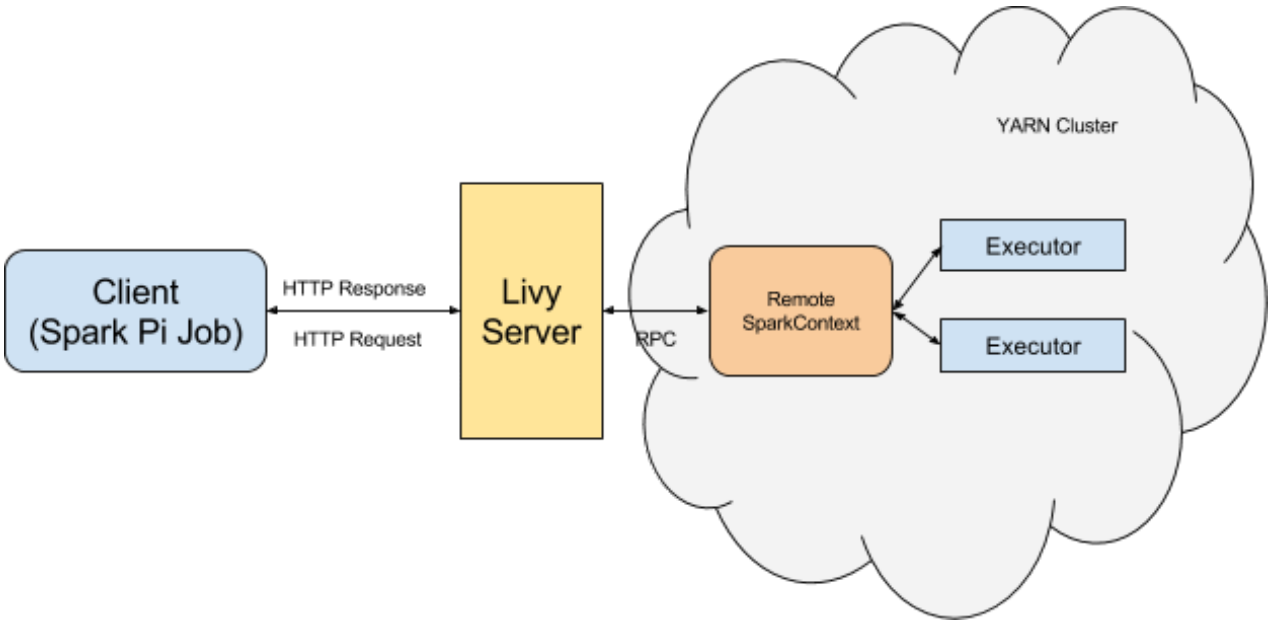
Using the Livy API to run Spark jobs is similar to using the original Spark API. Livy provides programmatic and REST APIs and supports Scala 2.12 for Spark 3 jobs.

Programmatic API

Livy provides a programmatic Java/Scala and Python API that allows applications to run code inside Spark without having to maintain a local Spark context. For more information, see *Using the Programmatic API* documentation.

REST API

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:



Scala support

Livy supports Scala version 2.12 with Spark 3 (Scala 2.12 build).

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

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

Related Information

Using the Programmatic API

Running an interactive session with the Livy REST 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 8: Session Object

Property	Description	Type
name	Name of the session	string
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
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 9: Session Kind

Value	Description
spark	Interactive Scala Spark session

Value	Description
pyspark	Interactive Python Spark session
sparkr	Interactive R Spark session

For Python 3 workloads, use kind pyspark and set the interpreter in session conf (for example spark.pyspark.python and, in yarn-cluster mode, spark.yarn.appMasterEnv.PYSPARK_PYTHON). The session kind pyspark3 is not supported.

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

Table 10: 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 11: 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 12: Statement Object State

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:

Table 13: 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, set environment variables or Spark configuration when you create the session.

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.
- When you create a session through the Livy REST API, set spark.pyspark.python and spark.yarn.appMasterEnv.PYSPARK_PYTHON in the session conf map.

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 the session. This is an optional parameter. The session cannot be created if another session with the same name already exists.	string
conf	Spark configuration properties	Map of key=value

Request Parameter	Description	Type
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 REST 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
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 the batch session. This is an optional parameter. **Important:** The batch session cannot be created if another active session with the same name already exists.	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 Livy for Interactive SQL and BI

You can use Livy to run a persistent, interactive Spark session that clients can connect to using JDBC or ODBC.

Note: This feature is designed to replace the unsupported Spark Thrift Server (STS) for integrating with third-party tools.

Use Cases

Customers who used Spark Thrift Server (STS) to interact with Spark from third-party tools, such as Qlik Sense, can use the Livy named sessions feature to migrate to Livy.

The Livy Named Sessions feature has the following key capabilities:

- The ability to reestablish a connection with a long-running session using a persistent name.
- A Livy thrift session that can run indefinitely, treating Spark as an in-memory database.
- The ability to use a temporary view from multiple client sessions.

How It Works

Instead of relying on a transient session ID, you can assign a unique and persistent **session name** to a Livy session at session creation. The session name acts as a stable alias that clients, such as those using JDBC or ODBC drivers, can use to connect to that specific, long-running session. Session names must be unique and cannot be reused for another active session.

To ensure the session remains available, session timeout checks can be disabled, allowing the session to run indefinitely until explicitly destroyed.

Limitations

The Livy Named Sessions feature has the following limitations:

- Cannot be used in environments with Kerberos enabled for authentication.
- High Availabilty is not supported for JDBC or ODBC Livy Thrift Server sessions

Next Steps

- See *Configuring the Livy Thrift Server* for instructions on how to enable indefinite sessions.
- See *Connecting to the Livy Thrift Server* for an example of how to connect to a named session.

Related Information

Configuring Indefinite Sessions for Livy
Connecting to a Named Livy Session

Configuring Indefinite Sessions for Livy

To support long-running, interactive sessions that do not time out, you can disable the Livy session timeout check. This is particularly useful when replacing the Spark Thrift Server functionality.

Before you begin

- You must have the Livy service configured in your Cloudera Manager environment.

Procedure

1. In Cloudera Manager, navigate to the Clusters page.
2. Select your cluster and click on the Livy service.
3. Go to the Configuration tab.
4. In the Search field, enter the livy.server.session.timeout-check string to find the configuration property.
5. Set the property to false.

   Important: When this property is set to false, sessions will run indefinitely and will not be terminated automatically. You must manually terminate and delete the session using the Livy REST API when the session is no longer needed to conserve cluster resources.
6. Save the changes and restart the Livy service.

Connecting to a Named Livy Session

You can connect a JDBC client to a specific, existing Livy session by referencing its name in the JDBC connection string. This allows you to create a persistent session and connect to it from multiple clients or reconnect after a client application has been restarted.

Before you begin

- The Livy Thrift Server must be enabled and configured. See Configuring the Livy Thrift Server for instructions.

About this task

The Livy Named Sessions feature is useful for sharing temporary data, like a global temporary view, across different client connections.

Procedure

1. Create a named Livy session using the REST API by using cURL to send a REST API request to the Livy server to create a new Spark session with a specific name.

   In this example, persistent-bi-session is the name assigned to the session.

   ```
   LIVY_URL=http://[*** LIVY THRIFT SERVER HOST ***]:8998
   curl -X POST --data '{"kind": "spark", "name": "persistent-bi-session"}' -
   H "Content-Type: application/json" $LIVY_URL/sessions
   ```

2. Create a Global Temporary View in the named Livy session by creating a DataFrame and register it as a global temporary view.

   Note: You must reference the Livy session by its name in the API endpoint.

   ```
   curl -X POST --data '{"code": "val df1 = spark.sparkContext.parallelize(
   Seq((1, \"John\", \"Doe\", 1986))).toDF(\"id\", \"first\", \"last\", \"y
   ear\")"}' -H "Content-Type: application/json" $LIVY_URL/sessions/name/pe
   rsistent-bi-session/statements

   curl -X POST --data '{"code": "df1.createGlobalTempView(\"mytempview\")"}'
    -H "Content-Type: application/json" $LIVY_URL/sessions/name/persistent-bi
   -session/statements
   ```

3. Connect to the existing session through Beeline by using the Beeline JDBC client to connect to the Livy Thrift Server and passing the session name in the livy.session.name property in the connection URL.

   Note: The JDBC user must have access to the session. Cloudera recommends the user to be the same user that created the session or a superuser configured in Livy.

   ```
   beeline -u "jdbc:hive2://[*** LIVY THRIFT SERVER HOST ***]:10090/?livy.ses
   sion.name=persistent-bi-session" -n [*** USERNAME ***]
   ```

4. Query the created global temporary view that is accessible within the Spark session by using the following command:

   ```
   0: jdbc:hive2://[*** ... ***]> SELECT * FROM global_temp.mytempview;
   ```

Configuring the Livy Thrift Server

The Apache Livy Thrift Server provides a JDBC endpoint for submitting Spark SQL queries. The Livy Thrift Server is disabled by default. The following instructions demonstrate how to enable and configure it using Cloudera Manager.

Before you begin

The Livy Thrift Server is supported in Cloudera 7.1.0 and higher. The Thrift Server is part of the Livy Server. There is no separate process.

The Livy Thrift Server inherits the security configuration of the Livy Server. It uses Kerberos for authentication and TLS for encryption if configured. There is no separate security configuration for the Thrift server.

Procedure

1. On the Cloudera Manager home page, click the three-dot menu icon next to the Livy service, and then click Configuration.
2. In the Search field, enter thrift to filter the configuration options and display the Thrift Server configuration settings.
3. Enable the Livy Thrift Server by checking the box labeled Enable Livy Thrift Server.
4. If you want to change the transport mode or port number, modify the Livy Thrift Server Transport Mode and Livy Thrift Server Port settings.
5. Enter a reason for the change in the provided field at the bottom of the page, and then click Save Changes.
6. Return to the Cloudera Manager home page, and then click the stale configuration icon next to the Livy service.
7. Click the Restart Stale Services button.
8. Make sure that Re-deploy client configuration is checked, and then click Restart Now.
9. After the command completes, click Finish.

What to do next

If you want to further configure the Livy Thrift Server, use the Livy Server Advanced Configuration Snippet (Safety Valve) for livy-conf/livy.conf:

1. On the Cloudera Manager home page, click the three-dot menu icon next to the Livy service, and then click Configuration.
2. In the Search field, enter livy.conf to filter the configuration options and display the advanced configuration text box.
3. Enter configuration key pairs, one per line, using the syntax <*PARAMETER*> = <*VALUE*>. For example:

   ```
   livy.server.thrift.limit.connections.per.user = 10
   livy.server.thrift.limit.connections.per.ipaddress = 100
   livy.server.thrift.limit.connections.per.user.ipaddress = 10
   ```

   These parameters limit the number of connections to the Thrift server by user, by IP address, and by the combination of both.
4. Enter a reason for the change in the provided field at the bottom of the page, and then click Save Changes.
5. Return to the Cloudera Manager home page, and then click the stale configuration icon next to the Livy service.
6. Click the Restart Stale Services button.
7. Make sure that Re-deploy client configuration is checked, and then click Restart Now.
8. After the command completes, click Finish.

Connecting to the Apache Livy Thrift Server

You can connect to the Apache Livy Thrift Server using the Beeline client that is included with Apache Hive.

Before you begin

The Livy Thrift Server is disabled by default. Make sure that you have enabled it before proceeding. For instructions, see Configuring the Livy Thrift Server.

The information required to form a JDBC URL is included in the Livy client configuration file, /etc/livy/conf/livy-client.conf. Make sure that you are on a gateway cluster host managed by Cloudera Manager to ensure that the client configuration is included. You can designate a cluster host as a Livy client by assigning it the Gateway role in the Livy service configuration. For more information about role assignments, see Managing Roles.

Procedure

1. On a host designated as a Livy Gateway, view the /etc/livy/conf/livy-client.conf file and note the values of the following configuration parameters:

   livy.uri

   This is the URI for the Livy server. The hostname in this URI is the hostname to use for the Thrift server connection.

   livy.server.thrift.port

   This is the port to use in the JDBC connection URL.

   livy.server.thrift.transport.mode

   This specifies the transport mode (either binary or http).

2.  **Note:** If you are using a cluster with security enabled, you must authenticate with your user principal using kinit before running beeline. If Kerberos authentication is not used, then the "-u <user> -p <password>" options for beeline should be used to connect as a particular user.

   Determine the JDBC connection URL using the following table:

Connection Mode	JDBC Connection URL
binary	jdbc:hive2://<livy_host>:<port>/ <database>
binary with Kerberos	jdbc:hive2://<livy_host>:<port>/ <database>;principal=livy/ <livy_host>@<REALM>
binary with Kerberos and TLS	jdbc:hive2://<livy_host>:<port>/ <database>;principal=livy/ <livy_host>@<REALM>;ssl=true;sslTrustStore=<path_to_truststore>
http	jdbc:hive2://<livy_host>:<port>/ <database>;transportMode=http;httpPath=cliservice
http with Kerberos	jdbc:hive2://<livy_host>:<port>/ <database>;principal=HTTP/ <livy_host>@<REALM>;transportMode=http;httpPath=cliservice
http with Kerberos and TLS	jdbc:hive2://<livy_host>:<port>/ <database>;principal=HTTP/ <livy_host>@<REALM>;ssl=true;sslTrustStore=<path_to_truststore>; \ trustStorePassword=<truststore_password>;transportMode=http;httpPath=cliservice

3. Run the beeline command.

4. At the Beeline prompt, enter !connect <jdbc_url>

   You can set configuration options by appending ?<configuration_list> to the JDBC connection URL. Separate multiple options with a semi-colon (;). Supported options are:

   - livy.session.driverMemory
   - livy.session.driverCores
   - livy.session.executorMemory
   - livy.session.executorCores
   - livy.session.numExecutors
   - livy.session.queue
   - livy.session.name
   - livy.session.heartbeatTimeoutInSecond

   For example:

```
beeline> !connect jdbc:hive2://cdp01.example.com:10090/db01?livy.session
.driverMemory=4g;livy.session.driverCores=2
```

What to do next

Connecting through HAProxy

Important: This load-balancing setup is a validated deployment pattern, not a core Livy feature. HAProxy is third-party software that Cloudera does not build or maintain. If security vulnerabilities or other issues are found in HAProxy itself, you must update the package directly on your operating system. Cloudera only supports HAProxy when it is configured and used as described in this guide.

If the Livy servers run behind HAProxy, connect to the Livy Thrift Server through the load balancer. The connection differs from a direct connection in the following ways:

- Use the http transport mode. HAProxy routes requests by cookie, which is not possible in binary mode.
- Use the load balancer FQDN as the host and the port of the HAProxy frontend for the Thrift Server. That frontend routes by the same SRV_ID cookie values as the REST frontend, to the livy.server.thrift.port of each Livy server.
- With Kerberos, use the HTTP/*LOAD-BALANCER-FQDN* principal. Livy accepts this principal when the Livy Load Balancer Address is set.
- Add httpHeader.Cookie=SRV_ID=*SRV-ID*, where *SRV-ID* identifies the backend that owns the session, and the livy.session.name option of the session to connect to.

For example:

```
jdbc:hive2://livy-lb.example.com:<haproxy_thrift_port>/;principal=HTTP/livy-
lb.example.com@<REALM>;ssl=true;sslTrustStore=<path_to_truststore>;trustStor
ePassword=<truststore_password>;transportMode=http;httpHeader.Cookie=SRV_ID=
s1;httpPath=cliservice?livy.session.name=<session_name>
```

To find the SRV_ID value of a session, see Finding the Livy server that owns a session.

Troubleshooting for Livy slowness

Slow Livy session creation, statement execution, or REST API responses usually come from YARN resource shortages, inefficient Spark workloads, an overloaded Livy server, or slow HDFS storage.

Condition

Users report that Apache Livy is slow. This usually means one or more of the following:

- Slow session creation: POST /sessions takes minutes, and sessions remain in the starting or not_started state.

- Slow statement execution: POST /sessions/{id}/statements returns quickly, but the statement remains in the wait ing or running state for longer than expected.
- Slow REST API responses: calls such as GET /sessions, GET /sessions/{id}, or GET /sessions/{id}/log are slow or time out.

Identify the source of the latency

Follow these checks in order each time Livy slowness is reported. They narrow the cause to one of the scenarios described in the rest of this topic.

1. Identify the kind of latency. Record the time the client sent the request, the time Livy responded, the session ID and statement ID, the Livy server host that handled the request (if Livy runs behind a load balancer or Apache Knox), and the user or proxy user that submitted the request.
2. Check the state that Livy reports. Use the Livy REST API:

   - GET /sessions: check how many sessions are active.
   - GET /sessions/{id}: check the session state and its transitions.
   - GET /sessions/{id}/log: check startup, failure, and waiting messages.

   If a session remains in the starting state, investigate YARN scheduling and the Apache Spark submission path. If a statement remains in the waiting or running state, open the Spark UI for that application. For more information about the endpoints, see *Livy API reference for interactive sessions* and the Apache Livy REST API documentation.
3. Correlate with the YARN and Spark UIs. For the YARN application of the session, check how long the application stays in each state (SUBMITTED, ACCEPTED, RUNNING). A long time in ACCEPTED means that the application is waiting for its ApplicationMaster container. In the Spark UI, check whether the driver is alive, how quickly executors are added, and whether stages show skew, large shuffles, spills, or long garbage collection (GC) times.

   In general:

   - If the application waits to start, the cause is in YARN or cluster resources.
   - If the application runs but its stages are slow, the cause is in the Spark workload.
   - If the YARN and Spark UIs look normal but Livy endpoints are slow, the cause is on the Livy server.
4. Check the health of the Livy server host. Check CPU load, free memory, swap usage, and disk I/O for the log and temporary directories. Check the Livy server JVM for long GC pauses, and collect thread dumps while the slowness is happening.
5. Check storage and node health. Look for HDFS read latency spikes, executor shuffle fetch wait time spikes, and NodeManager disk health alerts. For more information, see *Detecting slow DataNodes*.
6. Check high availability and session recovery. If Livy runs with more than one Livy server or with session recovery enabled, check Apache ZooKeeper latency and session expirations, the latency of the recovery state store, and whether slowness started immediately after a Livy server restart. For more information, see *Livy active/ passive high availability*.
7. Search the logs for the slow period. In Cloudera Manager, search the Livy server logs for the time window of the slowness. Look for repeated session retries, authentication failures and retries, long session creation times, and GC, OutOfMemoryError, or thread pool warnings. For more information, see *Logs*.

Session creation takes 10 to 20 minutes

The YARN queue is at or over capacity, so the ApplicationMaster container of the session is not allocated. Not enough free memory or vcores, unhealthy or decommissioning NodeManagers, full disks, or preemption can also delay container allocation. The session remains in the starting state, and the YARN application remains in the ACCE PTED state.

Solution

- Guarantee resources for interactive workloads, for example, with a dedicated Capacity Scheduler queue for Livy sessions.

- Reduce the default driver and executor sizes so that sessions can start when the cluster is under pressure, and let dynamic allocation add executors after the first statement.
- If many users open notebooks at the same time, start a small number of sessions in advance and reuse them.

All Livy endpoints are slow for every user

The Livy server JVM is struggling. Common causes are long GC pauses because the Livy server heap is too small for the number of sessions, CPU starvation, thread contention from many concurrent sessions and statements, and disk I/O pressure from heavy logging. Authentication overhead (TLS and Kerberos) increases under load. Even simple calls such as GET /sessions are slow.

The Livy server heap can also fill up immediately after a restart when session recovery is enabled and the server recovers a large number of sessions, especially batch sessions, from the state store.

Solution

- Increase the Livy server heap size and tune garbage collection. The G1 garbage collector is usually a good choice.
- Reduce logging volume during normal operation.
- Run Spark sessions in YARN cluster mode so that Spark drivers do not run on the Livy server host.
- Limit the number of concurrent sessions. The livy.server.session.max-creation property limits how many sessions can be starting at the same time.
- Stop sessions that are no longer needed, so that they are not recovered after each restart.

Only some statements are slow

The Spark query itself is slow. Common causes are data skew, an inefficient join strategy, very large shuffles, too few shuffle partitions, many small files, and Apache Hive Metastore latency, which delays the start of SQL queries. The statement remains in the running state, and the Spark UI shows active stages for a long time.

Solution

- Enable Adaptive Query Execution (AQE) for Spark SQL.
- Fix data skew, for example, with salting, skew join hints, or repartitioning.
- Tune the number of shuffle partitions and the executor sizes.

Livy is randomly slow at times

Some queries are fast and others are suddenly slow, because Spark tasks wait on I/O. Common causes are slow DataNodes (partial failures of disks, network interface cards, or controllers, or poor operating system and JVM tuning) and network congestion between compute and storage. Task I/O times spike, and HDFS reports slow nodes.

Solution

- Use slow DataNode detection to find the affected nodes, then fix the node by replacing failing disks, network interface cards, or controllers. For more information, see *Detecting slow DataNodes*.
- Rebalance data and fix operating system tuning on the affected nodes.

Requests are slow after a failover or when a Livy server is stopped

With active/passive high availability, passive Livy servers redirect requests to the active server. If one of the Livy servers is stopped, or ZooKeeper or the recovery state store is slow, requests can be delayed before they reach the active server.

Solution

- Ensure that all Livy server roles are running, or remove the roles that you do not use.
- Ensure that ZooKeeper is healthy and has low latency, and that the recovery state store is fast and reliable.

To prevent Livy slowness, see *Livy performance best practices*.

Related Information

Livy API reference for interactive sessions
Detecting slow DataNodes
Livy active/passive high availability
Livy performance best practices
Logs

Livy performance best practices

Livy performance depends on YARN cluster mode, Livy server sizing, intentional session lifetimes, and dedicated queues for interactive workloads, which together keep sessions responsive for all users.

Run sessions in YARN cluster mode

Run Apache Livy sessions in YARN cluster mode, so that YARN accounts for the resources of each session and Spark drivers do not run on the Livy server host. For more information, see the Apache Livy Getting Started documentation.

Size the Livy server and limit concurrency

The minimum heap size for the Livy server is rarely enough for production workloads with many concurrent sessions. Increase the heap size of the Livy server based on the number of sessions and statements that you expect.

For the minimum resource requirements, see *Livy resource requirements*.

Set operational limits for:

- The number of sessions that each user can keep open.
- The number of concurrent statements in each session.
- The total number of active sessions. The livy.server.session.max-creation property limits how many sessions can be starting at the same time.

Make session lifetimes intentional

Frequent session timeouts cause users to create sessions again and again, which adds load to YARN and to the Livy server.

- Set the session time-to-live (ttl) for each workload when you create the session. For more information, see the Apache Livy REST API documentation.
- Disable the session timeout check only if you also have a process that stops unused sessions, because these sessions are not terminated automatically. For more information, see *Configuring Indefinite Sessions for Livy*.

Isolate interactive workloads

Use a dedicated YARN queue for notebooks and Livy sessions, with guaranteed minimum resources, so that batch jobs do not starve interactive sessions.

Start sessions in advance for predictable peaks

If many users open notebooks at the same time every day, the peak in session creation is predictable. You can start a small number of sessions before the peak and reuse them. Keep the default number of executors small, and let dynamic allocation add executors after the first statement. For more information, see *Dynamic allocation*.

Keep Spark workloads efficient

Apache Spark tuning often gives the biggest improvement in Livy performance:

- Avoid many small files, control the number of shuffle partitions, and use the right join strategies.
- Use the Spark UI as your main tool to profile slow statements.
- Keep the Spark configuration stable. List the Spark properties that users must not override in the Livy list of blocked Spark properties, the spark-blacklist.conf file.

Configure high availability and session recovery carefully

If you run Livy with active/passive high availability:

- Ensure that Apache ZooKeeper is healthy and has low latency.
- Ensure that the recovery state store is fast and reliable.
- Stop sessions that are no longer needed. Every session that is still known to the state store is recovered when a Livy server restarts, which can exhaust the Livy server heap immediately after the restart.

For more information, see *Livy active/passive high availability*. To diagnose slowness, see *Troubleshooting for Livy slowness*.

Related Information

Configuring Indefinite Sessions for Livy
Dynamic allocation
Livy active/passive high availability
Troubleshooting for Livy slowness
Livy resource requirements

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. Cloudera 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_PYTHON 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_special_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/archive/ 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
```

Related Information

Spark Python Supportability Matrix

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

Spark3 (Gateway) 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 latest Oozie Spark3 Action Extension documentation, available on the Oozie UI. For general information about Oozie, see Overview of Oozie.



Important:

Oozie Spark3 actions support both yarn-cluster and yarn-client execution in general. However, there could be use cases which won't work in yarn-client mode.

For Spark3 actions both master and mode tags have to be provided. If master=yarn then mode must be either cluster or client.

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.

Exclusion pattern works well for Spark3 actions, use spark3 as the action type. Jackson should not be an issue for Spark3 actions.

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 Spark3-Pi job.

Sample Workflow.xml file for spark3-Pi:

```
<workflow-app xmlns='uri:oozie:workflow:1.0' name='SparkPythonPi'>
          <start to='spark-node' />

          <action name='spark-node'>
            <spark3 xmlns="uri:oozie:spark3-action:1.0">
              <resource-manager>${resourceManager}</resource-manager>
              <name-node>${nameNode}</name-node>
              <master>${master}</master>
              <mode>${mode}</mode>
              <name>Python-Spark-Pi</name>
              <jar>pi.py</jar>
            </spark3>
            <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 spark3-Pi:

```
nameNode=hdfs://host:8020
resourceManager=host:8050
queueName=default
examplesRoot=examples
oozie.use.system.libpath=true
oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/ap
ps/pyspark
master=yarn
mode=cluster
```

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 an error similar to below if there are .jar file conflicts between the "oozie" ShareLib and the "spark3" ShareLib. In such case, you need to remove the conflicting JAR file.

```
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>/spark3/aws*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark3/azure*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark3/hadoop-aws*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark3/hadoop-azure*
hadoop fs -rm /user/oozie/share/lib/lib_<ts>/spark3/ok*
hadoop fs -mv /user/oozie/share/lib/lib_<ts>/oozie/jackson* /user/oozie/sh
are/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
```