

Hortonworks Data Platform

Apache Spark Component Guide

(October 30, 2017)

Hortonworks Data Platform: Apache Spark Component Guide

Copyright © 2012-2017 Hortonworks, Inc. Some rights reserved.

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, [training](#) and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the [Hortonworks Data Platform](#) page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the [Support](#) or [Training](#) page. Feel free to [contact us](#) directly to discuss your specific needs.



Except where otherwise noted, this document is licensed under
Creative Commons Attribution ShareAlike 4.0 License.
<http://creativecommons.org/licenses/by-sa/4.0/legalcode>

Table of Contents

1. Analyzing Data with Apache Spark	1
2. Installing Spark	3
2.1. Installing Spark Using Ambari	3
2.2. Installing Spark Manually	6
2.3. Verifying Spark Configuration for Hive Access	7
2.4. Installing the Spark Thrift Server After Deploying Spark	7
2.5. Validating the Spark Installation	8
3. Configuring Spark	9
3.1. Configuring the Spark Thrift Server	9
3.1.1. Enabling Spark SQL User Impersonation for the Spark Thrift Server	9
3.1.2. Customizing the Spark Thrift Server Port	11
3.2. Configuring the Livy Server	11
3.2.1. Configuring SSL for the Livy Server	11
3.2.2. Configuring High Availability for the Livy Server	12
3.3. Configuring the Spark History Server	12
3.4. Configuring Dynamic Resource Allocation	12
3.4.1. Customizing Dynamic Resource Allocation Settings on an Ambari- Managed Cluster	13
3.4.2. Configuring Cluster Dynamic Resource Allocation Manually	14
3.4.3. Configuring a Job for Dynamic Resource Allocation	15
3.4.4. Dynamic Resource Allocation Properties	15
3.5. Configuring Spark for Wire Encryption	16
3.5.1. Configuring Spark for Wire Encryption	17
3.5.2. Configuring Spark2 for Wire Encryption	18
3.6. Configuring Spark for a Kerberos-Enabled Cluster	20
3.6.1. Configuring the Spark History Server	21
3.6.2. Configuring the Spark Thrift Server	21
3.6.3. Setting Up Access for Submitting Jobs	21
4. Running Spark	24
4.1. Specifying Which Version of Spark to Run	24
4.2. Running Sample Spark 1.x Applications	25
4.2.1. Spark Pi	26
4.2.2. WordCount	27
4.3. Running Sample Spark 2.x Applications	28
4.3.1. Spark Pi	29
4.3.2. WordCount	30
5. Submitting Spark Applications Through Livy	32
5.1. Using Livy with Spark Versions 1 and 2	32
5.2. Using Livy with Interactive Notebooks	33
5.3. Using the Livy API to Run Spark Jobs: Overview	34
5.4. Running an Interactive Session With the Livy API	35
5.4.1. Livy Objects for Interactive Sessions	36
5.4.2. Setting Path Variables for Python	37
5.4.3. Livy API Reference for Interactive Sessions	38
5.5. Submitting Batch Applications Using the Livy API	40
5.5.1. Livy Batch Object	41
5.5.2. Livy API Reference for Batch Jobs	41
6. Running PySpark in a Virtual Environment	43

7. Automating Spark Jobs with Oozie Spark Action	44
7.1. Configuring Oozie Spark Action for Spark 1	44
7.2. Configuring Oozie Spark Action for Spark 2	46
8. Developing Spark Applications	49
8.1. Using the Spark DataFrame API	49
8.2. Using Spark SQL	51
8.2.1. Accessing Spark SQL through the Spark Shell	52
8.2.2. Accessing Spark SQL through JDBC or ODBC: Prerequisites	52
8.2.3. Accessing Spark SQL through JDBC	53
8.2.4. Accessing Spark SQL through ODBC	54
8.2.5. Spark SQL User Impersonation	54
8.3. Calling Hive User-Defined Functions	60
8.3.1. Using Built-in UDFs	60
8.3.2. Using Custom UDFs	61
8.4. Using Spark Streaming	61
8.4.1. Prerequisites	62
8.4.2. Building and Running a Secure Spark Streaming Job	62
8.4.3. Running Spark Streaming Jobs on a Kerberos-Enabled Cluster	64
8.4.4. Sample <code>pom.xml</code> File for Spark Streaming with Kafka	65
8.5. HBase Data on Spark with Connectors	68
8.5.1. Selecting a Connector	68
8.5.2. Using the Connector with Apache Phoenix	69
8.6. Accessing HDFS Files from Spark	69
8.6.1. Specifying Compression	69
8.6.2. Accessing HDFS from PySpark	70
8.7. Accessing ORC Data in Hive Tables	70
8.7.1. Accessing ORC Files from Spark	70
8.7.2. Enabling Predicate Push-Down Optimization	71
8.7.3. Loading ORC Data into DataFrames by Using Predicate Push-Down	72
8.7.4. Optimizing Queries Through Partition Pruning	72
8.7.5. Enabling Vectorized Query Execution	73
8.7.6. Reading Hive ORC Tables	73
8.7.7. Additional Resources	73
8.8. Using Custom Libraries with Spark	74
9. Using Spark from R: SparkR	75
9.1. Prerequisites	75
9.2. SparkR Example	75
10. Tuning Spark	77
10.1. Provisioning Hardware	77
10.2. Checking Job Status	77
10.3. Checking Job History	77
10.4. Improving Software Performance	78
10.4.1. Configuring YARN Memory Allocation for Spark	78

List of Tables

1.1. Spark and Livy Feature Support by HDP Version	1
3.1. Dynamic Resource Allocation Properties	16
3.2. Optional Dynamic Resource Allocation Properties	16
8.1. Comparison of the Spark-HBase Connectors	69

1. Analyzing Data with Apache Spark

Hortonworks Data Platform (HDP) supports Apache Spark, a fast, large-scale data processing engine.

Deep integration of Spark with YARN allows Spark to operate as a cluster tenant alongside Apache engines such as Hive, Storm, and HBase, all running simultaneously on a single data platform. Instead of creating and managing a set of dedicated clusters for Spark applications, you can store data in a single location, access and analyze it with multiple processing engines, and leverage your resources.

Spark on YARN leverages YARN services for resource allocation, runs Spark executors in YARN containers, and supports workload management and Kerberos security features. It has two modes:

- YARN-cluster mode, optimized for long-running production jobs
- YARN-client mode, best for interactive use such as prototyping, testing, and debugging

Spark shell and the Spark Thrift server run in YARN-client mode only.

HDP 2.6 supports Spark versions 1.6 and 2.0; Livy, for local and remote access to Spark through the Livy REST API; and Apache Zeppelin, for browser-based notebook access to Spark. (For more information about Zeppelin, see the [Zeppelin Component Guide](#).)

Table 1.1. Spark and Livy Feature Support by HDP Version

HDP Version(s)	2.6.3	2.6.2	2.6.1	2.6.0	2.5.0, 2.5.3	2.4.3	2.4.2	2.4.0
Spark Version	1.6.3, 2.2.0	1.6.3, 2.1.1	1.6.3, 2.1.1	1.6.3, 2.1.0	1.6.2	1.6.2	1.6.1	1.6.0
Support for Livy	0.3	0.3	0.3	0.3				
Support for Hive	1.2.1	1.2.1	1.2.1	1.2.1	1.2.1	1.2.1	1.2.1	1.2.1
Spark Core	#	#	#	#	#	#	#	#
Spark on YARN	#	#	#	#	#	#	#	#
Spark on YARN for Kerberos-enabled clusters	#	#	#	#	#	#	#	#
Spark history server	#	#	#	#	#	#	#	#
Spark MLlib	#	#	#	#	#	#	#	#
ML Pipeline API	#	#	#	#	#	#	#	#
DataFrame API	#	#	#	#	#	#	#	#
Optimized Row Columnar (ORC) Files	#	#	#	#	#	#	#	#
PySpark	#	#	#	#	#	#	#	#
SparkR	#	#	#	#	#	TP	TP	TP
Spark SQL	#	#	#	#	#	#	#	#
Spark SQL Thrift server for JDBC, ODBC access	#	#	#	#	#	#	#	#
Spark Streaming	#	#	#	#	#	#	#	#
Structured Streaming	TP							
Dynamic resource allocation	#*	#*	#*	#*	#*	#*	#*	#*

HDP Version(s)	2.6.3	2.6.2	2.6.1	2.6.0	2.5.0, 2.5.3	2.4.3	2.4.2	2.4.0
Spark Version	1.6.3, 2.2.0	1.6.3, 2.1.1	1.6.3, 2.1.1	1.6.3, 2.1.0	1.6.2	1.6.2	1.6.1	1.6.0
HBase connector	#	#	#	#	#	TP	TP	
GraphX	TP	TP	TP	TP	TP	TP	TP	TP
DataSet API	TP	TP	TP	TP	TP	TP	TP	

* Dynamic Resource Allocation does not work with Spark Streaming.

TP: Technical Preview. Technical previews are considered under development. Do not use these features in production systems. If you have questions regarding these features, contact Support through the Hortonworks Support Portal, <https://support.hortonworks.com>.

The following features and associated tools are not officially supported by Hortonworks:

- Spark Standalone
- Spark on Mesos
- Jupyter Notebook (formerly IPython)

2. Installing Spark

Before installing Spark, ensure that your cluster meets the following prerequisites:

- HDP cluster stack version 2.6.0 or later
- (Optional) Ambari version 2.5.0 or later
- HDFS and YARN deployed on the cluster

You can choose to install Spark version 1, Spark version 2, or both. (To specify which version of Spark runs a job, see [Specifying Which Version of Spark to Run](#).)

Additionally, note the following requirements and recommendations for optional Spark services and features:

- Spark Thrift server requires Hive deployed on the cluster.
- SparkR requires R binaries installed on all nodes.
- SparkR is not currently supported on SLES.
- Spark access through Livy requires the Livy server installed on the cluster.
 - For clusters managed by Ambari, see [Installing Spark Using Ambari](#).
 - For clusters not managed by Ambari, see "Installing and Configuring Livy" in the [Spark](#) or [Spark 2](#) chapter of the *Command Line Installation Guide*, depending on the version of Spark installed on your cluster.
- PySpark and associated libraries require Python version 2.7 or later, or Python version 3.4 or later, installed on all nodes.
- For optimal performance with MLlib, consider installing the [netlib-java](#) library.

2.1. Installing Spark Using Ambari

The following diagram shows the Spark installation process using Ambari. Before you install Spark using Ambari, refer to [Adding a Service](#) in the *Ambari Operations Guide* for background information about how to install Hortonworks Data Platform (HDP) components using Ambari.



Caution

During the installation process, Ambari creates and edits several configuration files. If you configure and manage your cluster using Ambari, do not edit these

files during or after installation. Instead, use the Ambari web UI to revise configuration settings.

To install Spark using Ambari, complete the following steps.

1. Click the Ambari "Services" tab.
2. In the Ambari "Actions" menu, select "Add Service."

This starts the Add Service wizard, displaying the Choose Services page. Some of the services are enabled by default.

3. Scroll through the alphabetic list of components on the Choose Services page, and select "Spark", "Spark2", or both:

<input type="checkbox"/> Spark	1.6.x	Apache Spark is a fast and general engine for large-scale data processing.
<input type="checkbox"/> Spark2	2.x	Apache Spark is a fast and general engine for large-scale data processing

4. Click "Next" to continue.
5. On the Assign Masters page, review the node assignment for the Spark History Server or Spark2 History Server, depending on which Spark versions you are installing. Modify the node assignment if desired, and click "Next":

Spark History Server:

Spark2 History Server:

6. On the Assign Slaves and Clients page:
 - a. Scroll to the right and choose the "Client" nodes that you want to run Spark clients. These are the nodes from which Spark jobs can be submitted to YARN.
 - b. To install the optional Livy server, for security and user impersonation features, check the "Livy Server" box for the desired node assignment on the Assign Slaves and Clients page, for the version(s) of Spark you are deploying.
 - c. To install the optional Spark Thrift server at this time, for ODBC or JDBC access, review Spark Thrift Server node assignments on the Assign Slaves and Clients page and assign one or two nodes to it, as needed for the version(s) of Spark you are deploying. (To install the Thrift server later, see [Installing the Spark Thrift Server after Deploying Spark.](#))

Deploying the Thrift server on multiple nodes increases scalability of the Thrift server. When specifying the number of nodes, take into consideration the cluster capacity allocated to Spark.

Assign Slaves and Clients

Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with *.
"Client" will install Spark Client and Spark2 Client.

	all	none
Server	<input type="checkbox"/> Ranger Tagsync	<input checked="" type="checkbox"/> Livy Server
	<input checked="" type="checkbox"/> Spark Thrift Server	<input checked="" type="checkbox"/> Livy for Spark2 Server
	<input checked="" type="checkbox"/> Spark2 Thrift Server	<input checked="" type="checkbox"/> Client

Show: 25 1 - 1 of 1

[← Back](#) [Next →](#)

7. Click "Next" to continue.

8. Unless you are installing the Spark Thrift server now, use the default values displayed on the Customize Services page. Note that there are two tabs, one for Spark settings, and one for Spark2 settings.

Customize Services

We have come up with recommended configurations for the services you selected. Customize them as you see fit.

HDFS YARN MapReduce2 Tez Hive HBase Pig Oozie ZooKeeper Ambari Infra Ambari Metrics Kafka

Log Search Ranger Ranger KMS SmartSense **Spark** **Spark2** Zeppelin Notebook Slider Misc

There are 4 configuration changes in 3 services [Show Details](#)

Group: Default (1) [Manage Config Groups](#) Filter...

- Advanced livy-conf
- Advanced livy-env
- Advanced livy-log4j-properties

9. If you are installing the Spark Thrift server at this time, complete the following steps:
- Click the "Spark" or "Spark2" tab on the Customize Services page, depending on which version of Spark you are installing.
 - Navigate to the "Advanced spark-thrift-sparkconf" group.
 - Set the `spark.yarn.queue` value to the name of the YARN queue that you want to use.

10. Click "Next" to continue.

11. If Kerberos is enabled on the cluster, review principal and keytab settings on the Configure Identities page, modify settings if desired, and then click Next.

12. When the wizard displays the Review page, ensure that all HDP components correspond to HDP 2.6.0 or later. Scroll down and check the node assignments for selected services; for example:

Services:	
Spark	
Livy Server :	1 host
History Server :	lg-hdp260.field.hortonworks.com
Thrift Server :	1 host

13. Click "Deploy" to begin installation.

14. When Ambari displays the Install, Start and Test page, monitor the status bar and messages for progress updates:

Install, Start and Test

Please wait while the selected services are installed and started.

51 % overall

Show: All (1) In Progress (1) Warning (0) Success (0) Fail (0)		
Host	Status	Message
lgcl001-hdp260.field.hortonworks.com	<div></div> 51%	Starting Spark2 History Server

1 of 1 hosts showing - [Show All](#)

Show: 25
1 - 1 of 1

Next →

15. When the wizard presents a summary of results, click "Complete" to finish installing Spark.

2.2. Installing Spark Manually

If you want to install Spark or Spark 2 on a cluster that is not managed by Ambari, see [Installing and Configuring Apache Spark](#) or [Installing and Configuring Apache Spark 2](#), in the *Command Line Installation Guide*.

If you previously installed Spark on a cluster not managed by Ambari, and you want to move to Spark 2:

1. Install Spark 2 according to the Spark 2 instructions in the *Command Line Installation Guide*.
2. Ensure that each version of Spark uses a different port.
3. Test your Spark jobs on Spark 2. To direct a job to Spark 2 when Spark 1 is the default version, see [Specifying Which Version of Spark to Run](#).

4. When finished testing, optionally remove Spark 1 from the cluster: stop all services and then uninstall Spark. Manually check to make sure all library and configuration directories have been removed.

2.3. Verifying Spark Configuration for Hive Access

When you install Spark using Ambari, the `hive-site.xml` file is automatically populated with the Hive metastore location.

If you move Hive to a different server, edit the `SPARK_HOME/conf/hive-site.xml` file so that it contains only the `hive.metastore.uris` property. Make sure that the host name points to the URI where the Hive metastore is running, and that the Spark copy of `hive-site.xml` contains only the `hive.metastore.uris` property.

```
<configuration>
  <property>
    <name>hive.metastore.uris</name>
    <!-- hostname must point to the Hive metastore URI in your cluster -->
    <value>thrift://hostname:9083</value>
    <description>URI for client to contact metastore server</description>
  </property>
</configuration>
```

2.4. Installing the Spark Thrift Server After Deploying Spark

To install the Spark Thrift server after deploying Spark over Ambari, add the Thrift service to the specified host or hosts. Deploying the Thrift server on multiple hosts increases scalability of the Thrift server; the number of hosts should take into consideration the cluster capacity allocated to Spark.

1. On the Summary tab, click "+ Add" and choose the Spark Thrift server:

✓ c6401.ambari.apache.org
← Back

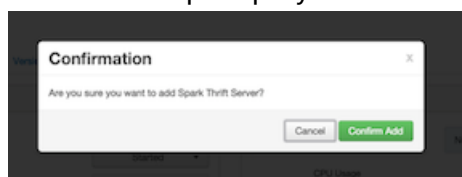
Summary Configs Alerts 0 Versions

Components + Add

- ✓ App Timeline Server / YARN
- ✓ History Server / MapReduce2
- ✓ Hive Metastore / Hive
- ✓ HiveServer2 / Hive
- ✓ MySQL Server / Hive
- ✓ NameNode / HDFS
- ✓ ResourceManager / YARN
- ✓ SNameNode / HDFS
- ✓ Spark History Server / Spark

Spark Thrift Server: Started

2. When Ambari prompts you to confirm the selection, click **Confirm All**:



The installation process runs in the background until it is complete:

Operations	Start Time	Duration	Show: All (7)
✓ Install Spark Thrift Server	Today 08:09	3.58 secs	100%
✓ Stop Spark Thrift Server	Today 08:05	6.42 secs	100%
✓ Start All Services	Mon Dec 14 2015 07:38	274.72 secs	100%
✓ Restart all components with Stale Configs for YARN	Fri Dec 11 2015 12:52	43.80 secs	100%
✓ Stop Spark Thrift Server	Fri Dec 11 2015 12:48	9.22 secs	100%
✓ Start Services	Fri Dec 11 2015 12:26	565.27 secs	100%
✓ Install Services	Fri Dec 11 2015 12:21	333.80 secs	100%

☐ Do not show this dialog again when starting a background operation OK

2.5. Validating the Spark Installation

To validate the Spark or Spark2 installation process, run the Spark Pi and WordCount jobs supplied with the Spark package. For more information, see [Running Spark Applications](#).

3. Configuring Spark

This chapter describes how to configure the following Apache Spark services and features:

- Spark thrift server for JDBC and ODBC clients
- Livy server for remote access to Spark through the Livy REST API
- Spark history server
- Dynamic resource allocation
- Wire encryption
- Configuring Spark for a Kerberos-enabled cluster
- Row- and column-level access control for Spark SQL

3.1. Configuring the Spark Thrift Server

Spark Thrift server is a service that allows JDBC and ODBC clients to run Spark SQL queries. The Spark Thrift server is a variant of HiveServer2.

This subsection describes optional Spark Thrift Server features and configuration steps:

- Enabling user impersonation, so that SQL queries run under the identity of the user who originated the query. (By default, queries run under the account associated with the Spark Thrift server.)
- Customizing the Spark Thrift server port.

For information about configuring the Thrift server on a Kerberos-enabled cluster, see [Configuring the Spark Thrift Server](#) in "Configuring Spark for a Kerberos-Enabled Cluster."

3.1.1. Enabling Spark SQL User Impersonation for the Spark Thrift Server

By default, the Spark Thrift server runs queries under the identity of the operating system account running the Spark Thrift server. In a multi-user environment, queries often need to run under the identity of the end user who originated the query; this capability is called "user impersonation."

When user impersonation is enabled, Spark Thrift server runs Spark SQL queries as the submitting user. By running queries under the user account associated with the submitter, the Thrift server can enforce user level permissions and access control lists. Associated data cached in Spark is visible only to queries from the submitting user.

User impersonation enables granular access control for Spark SQL queries at the level of files or tables.

The user impersonation feature is controlled with the `doAs` property. When `doAs` is set to `true`, Spark Thrift server launches an on-demand Spark application to handle user queries. These queries are shared only with connections from the same user. Spark Thrift server forwards incoming queries to the appropriate Spark application for execution, making the Spark Thrift server extremely lightweight: it merely acts as a proxy to forward requests and responses. When all user connections for a Spark application are closed at the Spark Thrift server, the corresponding Spark application also terminates.

Prerequisites

Spark SQL user impersonation is supported for Apache Spark 1 versions 1.6.3 and later.

If storage-based authorization is to be enabled, complete the instructions in [Configuring Storage-based Authorization](#) in the *Data Access Guide* before enabling user impersonation.

Enabling User Impersonation on an Ambari-managed Cluster

To enable user impersonation for the Spark Thrift server on an Ambari-managed cluster, complete the following steps:

1. Enable `doAs` support. Navigate to the “Advanced spark-hive-site-override” section and set `hive.server2.enable.doAs=true`.
2. Add DataNucleus jars to the Spark Thrift server classpath. Navigate to the “Custom spark-thrift-sparkconf” section and set the `spark.jars` property as follows:

```
spark.jars=/usr/hdp/current/spark-thriftserver/lib/datanucleus-api-jdo-3.2.6.jar,/usr/hdp/current/spark-thriftserver/lib/datanucleus-core-3.2.10.jar,/usr/hdp/current/spark-thriftserver/lib/datanucleus-rdbms-3.2.9.jar
```

3. (Optional) Disable Spark Yarn application for Spark Thrift server master. Navigate to the “Advanced spark-thrift-sparkconf” section and set `spark.master=local`. This prevents launching a spark-client HiveThriftServer2 application master, which is not needed when `doAs=true` because queries are executed by the Spark AM, launched on behalf of the user. When `spark.master` is set to `local`, `SparkContext` uses only the local machine for driver and executor tasks.

(When the Thrift server runs with `doAs` set to `false`, you should set `spark.master` to `yarn-client`, so that query execution leverages cluster resources.)

4. Restart the Spark Thrift server.

Enabling User Impersonation on a Cluster Not Managed by Ambari

To enable user impersonation for the Spark Thrift server on a cluster not managed by Ambari, complete the following steps:

1. Enable `doAs` support. Add the following setting to the `/usr/hdp/current/spark-thriftserver/conf/hive-site.xml` file:

```
<property>
  <name>hive.server2.enable.doAs</name>
  <value>true</value>
</property>
```

2. Add DataNucleus jars to the Spark Thrift server classpath. Add the following setting to the `/usr/hdp/current/spark-thriftserver/conf/spark-thrift-sparkconf.conf` file:

```
spark.jars=/usr/hdp/current/spark-thriftserver/lib/datanucleus-api-jdo-3.2.6.jar,/usr/hdp/current/spark-thriftserver/lib/datanucleus-core-3.2.10.jar,/usr/hdp/current/spark-thriftserver/lib/datanucleus-rdbms-3.2.9.jar
```

3. (Optional) Disable Spark Yarn application for Spark Thrift server master. Add the following setting to the `/usr/hdp/current/spark-thriftserver/conf/spark-thrift-sparkconf.conf` file:

```
spark.master=local
```

This prevents launching an unused spark-client HiveThriftServer2 application master, which is not needed when `doAs=true` because queries are executed by the Spark AM, launched on behalf of the user. When `spark.master` is set to `local`, `SparkContext` uses only the local machine for driver and executor tasks.

(When the Thrift server runs with `doAs` set to `false`, you should set `spark.master` to `yarn-client`, so that query execution leverages cluster resources.)

4. Restart the Spark Thrift server.

For more information about user impersonation for the Spark Thrift Server, see [Using Spark SQL](#).

3.1.2. Customizing the Spark Thrift Server Port

The default Spark Thrift server port is 10015. To specify a different port, you can navigate to the `hive.server2.thrift.port` setting in the "Advanced spark-hive-site-override" category of the Spark configuration section and update the setting with your preferred port number.

3.2. Configuring the Livy Server

On a cluster managed by Ambari, to configure the optional [Livy service](#), complete the following steps:

1. Navigate to Spark > Configs, Custom livy-conf category.
2. Add `livy.superusers` as a property, and set it to the Zeppelin service account.

For a cluster not managed by Ambari, see "Installing and Configuring Livy" in the [Spark](#) or [Spark 2](#) chapter of the *Command Line Installation Guide*, depending on the version of Spark installed on your cluster.

3.2.1. Configuring SSL for the Livy Server

To enable SSL for Livy, configure the following parameters for the SSL certificate and key, the keystore password, and the key password, respectively:


```
livy.keystore=<keystore_file>  
livy.keystore.password = <storePassword>  
livy.key-password = <KeyPassword>
```

For background information about configuring SSL for Spark or Spark2, see [Configuring Spark for Wire Encryption](#).

3.2.2. Configuring High Availability for the Livy Server

By default, if the Livy server fails, all connected Spark clusters are terminated. This means that all jobs and data will disappear immediately.

For deployments that require high availability, Livy supports session recovery, which ensures that a Spark cluster remains available if the Livy server fails. After a restart, the Livy server can connect to existing sessions and roll back to the state before failing.

Livy uses several property settings for recovery behavior related to high availability. If your cluster is managed by Ambari, Ambari manages these settings. If your cluster is not managed by Ambari, or for a list of recovery properties, see instructions for enabling Livy recovery in the [Spark](#) or [Spark2](#) chapter of the *Command Line Installation Guide*.

3.3. Configuring the Spark History Server

The Spark history server is a monitoring tool that displays information about completed Spark applications. This information is pulled from the data that applications by default write to a directory on Hadoop Distributed File System (HDFS). The information is then presented in a web UI at <host>:<port>. (The default port is 18080.)

For information about configuring optional history server properties, see the [Apache Monitoring and Instrumentation](#) document.

3.4. Configuring Dynamic Resource Allocation

When the dynamic resource allocation feature is enabled, an application's use of executors is dynamically adjusted based on workload. This means that an application can relinquish resources when the resources are no longer needed, and request them later when there is more demand. This feature is particularly useful if multiple applications share resources in your Spark cluster.

Dynamic resource allocation is available for use by the Spark Thrift server and general Spark jobs.



Note

Dynamic Resource Allocation does not work with Spark Streaming.

You can configure dynamic resource allocation at either the cluster or the job level:

- Cluster level:
 - On an Ambari-managed cluster, the Spark Thrift server uses dynamic resource allocation by default. The Thrift server increases or decreases the number of running

executors based on a specified range, depending on load. (In addition, the Thrift server runs in YARN mode by default, so the Thrift server uses resources from the YARN cluster.) The associated shuffle service starts automatically, for use by the Thrift server and general Spark jobs.

- On a manually installed cluster, dynamic resource allocation is not enabled by default for the Thrift server or for other Spark applications. You can enable and configure dynamic resource allocation and start the shuffle service during the Spark manual installation or upgrade process.
- Job level: You can customize dynamic resource allocation settings on a per-job basis. Job settings override cluster configuration settings.

Cluster configuration is the default, unless overridden by job configuration.

The following subsections describe each configuration approach, followed by a list of dynamic resource allocation properties and a set of instructions for customizing the Spark Thrift server port.

3.4.1. Customizing Dynamic Resource Allocation Settings on an Ambari-Managed Cluster

On an Ambari-managed cluster, dynamic resource allocation is enabled and configured for the Spark Thrift server as part of the Spark installation process. Dynamic resource allocation is not enabled by default for general Spark jobs.

You can review dynamic resource allocation for the Spark Thrift server, and enable and configure settings for general Spark jobs, by choosing **Services > Spark** and then navigating to the "Advanced spark-thrift-sparkconf" group:

▼ Advanced spark-thrift-sparkconf

spark.dynamicAllocation.enabled	true	🔒	🔄	📄
spark.dynamicAllocation.initialExecutors	0	🔒	🔄	📄
spark.dynamicAllocation.maxExecutors	10	🔒	🔄	📄
spark.dynamicAllocation.minExecutors	0	🔒	🔄	📄
spark.eventLog.dir	{{spark_history_dir}}	🔒	🔄	📄
spark.eventLog.enabled	true	🔒	🔄	📄
spark.executor.memory	1g	🔒	🔄	📄
spark.history.fs.logDirectory	{{spark_history_dir}}	🔒	🔄	📄
spark.history.provider	org.apache.spark.deploy.history.FsHistoryProvider	🔒	🔄	📄
spark.master	{{spark_thrift_master}}	🔒	🔄	📄
spark.scheduler.allocation.file	{{spark_conf}}/spark-thrift-fairscheduler.xml	🔒	🔄	📄
spark.scheduler.mode	FAIR	🔒	🔄	📄
spark.shuffle.service.enabled	true	🔒	🔄	📄
spark.yarn.am.memory	512m	🔒	🔄	📄
spark.yarn.queue	default	🔒	🔄	📄

The "Advanced spark-thrift-sparkconf" group lists required settings. You can specify optional properties in the custom section. For a complete list of DRA properties, see [Dynamic Resource Allocation Properties](#).

Dynamic resource allocation requires an external shuffle service that runs on each worker node as an auxiliary service of NodeManager. If you installed your cluster using Ambari, the service is started automatically for use by the Thrift server and general Spark jobs; no further steps are needed.

3.4.2. Configuring Cluster Dynamic Resource Allocation Manually

To configure a cluster to run Spark jobs with dynamic resource allocation, complete the following steps:

1. Add the following properties to the `spark-defaults.conf` file associated with your Spark installation (typically in the `$SPARK_HOME/conf` directory):
 - Set `spark.dynamicAllocation.enabled` to `true`.

- Set `spark.shuffle.service.enabled` to `true`.
2. (Optional) To specify a starting point and range for the number of executors, use the following properties:

- `spark.dynamicAllocation.initialExecutors`
- `spark.dynamicAllocation.minExecutors`
- `spark.dynamicAllocation.maxExecutors`

Note that `initialExecutors` must be greater than or equal to `minExecutors`, and less than or equal to `maxExecutors`.

For a description of each property, see [Dynamic Resource Allocation Properties](#).

3. Start the shuffle service on each worker node in the cluster:
- a. In the `yarn-site.xml` file on each node, add `spark_shuffle` to `yarn.nodemanager.aux-services`, and then set `yarn.nodemanager.aux-services.spark_shuffle.class` to `org.apache.spark.network.yarn.YarnShuffleService`.
 - b. Review and, if necessary, edit `spark.shuffle.service.*` configuration settings.

For more information, see the Apache [Spark Shuffle Behavior](#) documentation.
 - c. Restart all NodeManagers in your cluster.

3.4.3. Configuring a Job for Dynamic Resource Allocation

There are two ways to customize dynamic resource allocation properties for a specific job:

- Include property values in the `spark-submit` command, using the `-conf` option.

This approach loads the default `spark-defaults.conf` file first, and then applies property values specified in your `spark-submit` command. Here is an example:

```
spark-submit --conf "property_name=property_value"
```

- Create a job-specific `spark-defaults.conf` file and pass it to the `spark-submit` command.

This approach uses the specified properties file, without reading the default property file. Here is an example:

```
spark-submit --properties-file <property_file>
```

3.4.4. Dynamic Resource Allocation Properties

See the following tables for more information about basic and optional dynamic resource allocation properties. For more information, see the Apache [Dynamic Resource Allocation](#) documentation.

Table 3.1. Dynamic Resource Allocation Properties

Property Name	Value	Meaning
<code>spark.dynamicAllocation.enabled</code>	Default is <code>true</code> for the Spark Thrift server, and <code>false</code> for Spark jobs.	Specifies whether to use dynamic resource allocation, which scales the number of executors registered for an application up and down based on workload. Note that this feature is currently only available in YARN mode.
<code>spark.shuffle.service.enabled</code>	<code>true</code>	Enables the external shuffle service, which preserves shuffle files written by executors so that the executors can be safely removed. This property must be set to <code>true</code> if <code>spark.dynamicAllocation.enabled</code> is <code>true</code> .
<code>spark.dynamicAllocation.initialExecutors</code>	Default is <code>spark.dynamicAllocation.minExecutors</code>	The initial number of executors to run if dynamic resource allocation is enabled. This value must be greater than or equal to the <code>minExecutors</code> value, and less than or equal to the <code>maxExecutors</code> value.
<code>spark.dynamicAllocation.maxExecutors</code>	Default is infinity	Specifies the upper bound for the number of executors if dynamic resource allocation is enabled.
<code>spark.dynamicAllocation.minExecutors</code>	Default is 0	Specifies the lower bound for the number of executors if dynamic resource allocation is enabled.

Table 3.2. Optional Dynamic Resource Allocation Properties

Property Name	Value	Meaning
<code>spark.dynamicAllocation.executorIdleTimeout</code>	Default is 60 seconds (60s)	If dynamic resource allocation is enabled and an executor has been idle for more than this time, the executor is removed.
<code>spark.dynamicAllocation.cachedExecutorIdleTimeout</code>	Default is infinity	If dynamic resource allocation is enabled and an executor with cached data blocks has been idle for more than this time, the executor is removed.
<code>spark.dynamicAllocation.schedulerBacklogTimeout</code>	1 second (1s)	If dynamic resource allocation is enabled and there have been pending tasks backlogged for more than this time, new executors are requested.
<code>spark.dynamicAllocation.sustainedSchedulerBacklogTimeout</code>	Default is <code>spark.dynamicAllocation.schedulerBacklogTimeout</code>	Same as <code>spark.dynamicAllocation.schedulerBacklogTimeout</code> , but used only for subsequent executor requests.

3.5. Configuring Spark for Wire Encryption

You can configure Spark to protect sensitive data in transit, by enabling wire encryption.

In general, encryption protects data by making it unreadable without a phrase or digital key to access the data. Data can be encrypted while it is in transit and when it is at rest:

- "In transit" encryption refers to data that is encrypted when it traverses a network. The data is encrypted between the sender and receiver process across the network. Wire encryption is a form of "in transit" encryption.
- "At rest" or "transparent" encryption refers to data stored in a database, on disk, or on other types of persistent media.

Apache Spark supports "in transit" wire encryption of data for Apache Spark jobs. When encryption is enabled, Spark encrypts all data that is moved across nodes in a cluster on behalf of a job, including the following scenarios:

- Data that is moving between executors and drivers, such as during a `collect()` operation.
- Data that is moving between executors, such as during a shuffle operation.

Spark does not support encryption for connectors accessing external sources; instead, the connectors must handle any encryption requirements. For example, the Spark HDFS connector supports transparent encrypted data access from HDFS: when transparent encryption is enabled in HDFS, Spark jobs can use the HDFS connector to read encrypted data from HDFS.

Spark does not support encrypted data on local disk, such as intermediate data written to a local disk by an executor process when the data does not fit in memory. Additionally, wire encryption is not supported for shuffle files, cached data, and other application files. For these scenarios you should enable local disk encryption through your operating system.

In Spark 2.0, enabling wire encryption also enables HTTPS on the History Server UI, for browsing historical job data.

The following two subsections describe how to configure Spark and Spark2 for wire encryption, respectively.

3.5.1. Configuring Spark for Wire Encryption

Use the following commands to configure Spark (version 1) for wire encryption:

1. On each node, create keystore files, certificates, and truststore files.

- a. Create a keystore file:

```
keytool -genkey \  
-alias <host> \  
-keyalg RSA \  
-keysize 1024 \  
-dname CN=<host>,OU=hw,O=hw,L=paloalto,ST=ca,C=us \  
-keypass <KeyPassword> \  
-keystore <keystore_file> \  
-storepass <storePassword>
```

- b. Create a certificate:

```
keytool -export \  
-alias <host> \  
-keystore <keystore_file> \  
-rfc -file <cert_file> \  
-storepass <StorePassword>
```

c. Create a truststore file:

```
keytool -import \  
-noprompt \  
-alias <host> \  
-file <cert_file> \  
-keystore <truststore_file> \  
-storepass <truststorePassword>
```

2. Create one truststore file that contains the public keys from all certificates.

a. Log on to one host and import the truststore file for that host:

```
keytool -import \  
-noprompt \  
-alias <hostname> \  
-file <cert_file> \  
-keystore <all_jks> \  
-storepass <allTruststorePassword>
```

b. Copy the <all_jks> file to the other nodes in your cluster, and repeat the `keytool` command on each node.

3. Enable Spark authentication.

a. Set `spark.authenticate` to `true` in the `yarn-site.xml` file:

```
<property>  
  <name>spark.authenticate</name>  
  <value>true</value>  
</property>
```

b. Set the following properties in the `spark-defaults.conf` file:

```
spark.authenticate true  
spark.authenticate.enableSaslEncryption true
```

4. Enable Spark SSL.

Set the following properties in the `spark-defaults.conf` file:

```
spark.ssl.enabled true  
spark.ssl.enabledAlgorithms TLS_RSA_WITH_AES_128_CBC_SHA,  
TLS_RSA_WITH_AES_256_CBC_SHA  
spark.ssl.keyPassword <KeyPassword>  
spark.ssl.keyStore <keystore_file>  
spark.ssl.keyStorePassword <storePassword>  
spark.ssl.protocol TLS  
spark.ssl.trustStore <all_jks>  
spark.ssl.trustStorePassword <allTruststorePassword>
```

3.5.2. Configuring Spark2 for Wire Encryption

Use the following commands to configure Spark2 for wire encryption:

1. On each node, create keystore files, certificates, and truststore files.

a. Create a keystore file:

```
keytool -genkey \  
-alias <host> \  
-keyalg RSA \  
-keysize 1024 \  
-dname CN=<host>,OU=hw,O=hw,L=paloalto,ST=ca,C=us \  
-keypass <KeyPassword> \  
-keystore <keystore_file> \  
-storepass <storePassword>
```

b. Create a certificate:

```
keytool -export \  
-alias <host> \  
-keystore <keystore_file> \  
-rfc -file <cert_file> \  
-storepass <StorePassword>
```

c. Create a truststore file:

```
keytool -import \  
-noprompt \  
-alias <host> \  
-file <cert_file> \  
-keystore <truststore_file> \  
-storepass <truststorePassword>
```

2. Create one truststore file that contains the public keys from all certificates.

a. Log on to one host and import the truststore file for that host:

```
keytool -import \  
-noprompt \  
-alias <hostname> \  
-file <cert_file> \  
-keystore <all_jks> \  
-storepass <allTruststorePassword>
```

b. Copy the <all_jks> file to the other nodes in your cluster, and repeat the keytool command on each node.

3. Enable Spark2 authentication.

a. Set spark.authenticate to true in the yarn-site.xml file:

```
<property>  
  <name>spark.authenticate</name>  
  <value>true</value>  
</property>
```

b. Set the following properties in the spark-defaults.conf file:

```
spark.authenticate true  
spark.authenticate.enableSaslEncryption true
```

4. Enable Spark2 SSL.

Set the following properties in the spark-defaults.conf file:


```
spark.ssl.enabled true
spark.ssl.enabledAlgorithms TLS_RSA_WITH_AES_128_CBC_SHA,
TLS_RSA_WITH_AES_256_CBC_SHA
spark.ssl.keyPassword <KeyPassword>
spark.ssl.keyStore <keystore_file>
spark.ssl.keyStorePassword <storePassword>
spark.ssl.protocol TLS
spark.ssl.trustStore <all_jks>
spark.ssl.trustStorePassword <allTruststorePassword>
```

5. Enable HTTPS for the Spark2 UI.

Set `spark.ui.https.enabled` to `true` in the `spark-defaults.conf` file:

```
spark.ui.https.enabled true
```

Note: In Spark2, enabling wire encryption also enables HTTPS on the History Server UI, for browsing job history data.

6. (Optional) If you want to enable optional on-disk block encryption, which applies to both shuffle and RDD blocks on disk, complete the following steps:

a. Add the following properties to the `spark-defaults.conf` file for Spark2:

```
spark.io.encryption.enabled true
spark.io.encryption.keySizeBits 128
spark.io.encryption.keygen.algorithm HmacSHA1
```

b. Enable RPC encryption.

For more information, see the [Shuffle Behavior](#) section of Apache Spark Properties documentation, and the Apache Spark [Security](#) documentation.

3.6. Configuring Spark for a Kerberos-Enabled Cluster

Before running Spark jobs on a Kerberos-enabled cluster, configure additional settings for the following modules and scenarios:

- Spark history server
- Spark Thrift server
- Individuals who submit jobs
- Processes that submit jobs without human interaction

Each of these scenarios is described in the following subsections.

When Kerberos is enabled on an Ambari-managed cluster, Livy configuration for Kerberos is handled automatically.

3.6.1. Configuring the Spark History Server

The Spark history server daemon must have a Kerberos account and keytab to run on a Kerberos-enabled cluster.

When you enable Kerberos for a Hadoop cluster with Ambari, Ambari configures Kerberos for the history server and automatically creates a Kerberos account and keytab for it. For more information, see [Enabling Kerberos Authentication Using Ambari](#) in the *HDP Security Guide*.

If your cluster is not managed by Ambari, or if you plan to enable Kerberos manually for the history server, see [Creating Service Principals and Keytab Files for HDP](#) in the *HDP Security Guide*.

3.6.2. Configuring the Spark Thrift Server

If you are installing the Spark Thrift server on a Kerberos-enabled cluster, note the following requirements:

- The Spark Thrift server must run in the same host as `HiveServer2`, so that it can access the `hiveserver2` keytab.
- Permissions in `/var/run/spark` and `/var/log/spark` must specify read/write permissions to the Hive service account.
- You must use the Hive service account to start the `thriftserver` process.

If you access Hive warehouse files through `HiveServer2` on a deployment with fine-grained access control, run the Spark Thrift server as user `hive`. This ensures that the Spark Thrift server can access Hive keytabs, the Hive metastore, and HDFS data stored under user `hive`.



Important

If you read files from HDFS directly through an interface such as Hive CLI or Spark CLI (as opposed to `HiveServer2` with fine-grained access control), you should use a different service account for the Spark Thrift server. Configure the account so that it can access Hive keytabs and the Hive metastore. Use of an alternate account provides a more secure configuration: when the Spark Thrift server runs queries as user `hive`, all data accessible to user `hive` is accessible to the user submitting the query.

For Spark jobs that are not submitted through the Thrift server, the user submitting the job must have access to the Hive metastore in secure mode, using the `kinit` command.

3.6.3. Setting Up Access for Submitting Jobs

Accounts that submit jobs on behalf of other processes must have a Kerberos account and keytab. End users should use their own keytabs (instead of using a headless keytab) when submitting a Spark job. The following two subsections describe both scenarios.

Setting Up Access for an Account

When access is authenticated without human interaction (as happens for processes that submit job requests), the process uses a headless keytab. Security risk is mitigated by ensuring that only the service that should be using the headless keytab has permission to read it.

The following example creates a headless keytab for a `spark` service user account that will submit Spark jobs on node `blue1@example.com`:

1. Create a Kerberos service principal for user `spark`:

```
kadmin.local -q "addprinc -randkey spark/blue1@EXAMPLE.COM"
```

2. Create the keytab:

```
kadmin.local -q "xst -k /etc/security/keytabs/spark.keytab
spark/blue1@EXAMPLE.COM"
```

3. For every node of your cluster, create a `spark` user and add it to the `hadoop` group:

```
useradd spark -g hadoop
```

4. Make `spark` the owner of the newly created keytab:

```
chown spark:hadoop /etc/security/keytabs/spark.keytab
```

5. Limit access by ensuring that user `spark` is the only user with access to the keytab:

```
chmod 400 /etc/security/keytabs/spark.keytab
```

In the following example, user `spark` runs the Spark Pi example in a Kerberos-enabled environment:

```
su spark
kinit -kt /etc/security/keytabs/spark.keytab spark/blue1@EXAMPLE.COM
cd /usr/hdp/current/spark-client/
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn-cluster \
  --num-executors 1 \
  --driver-memory 512m \
  --executor-memory 512m \
  --executor-cores 1 \
  lib/spark-examples*.jar 10
```

Setting Up Access for an End User

Each person who submits jobs must have a Kerberos account and their own keytab; end users should use their own keytabs (instead of using a headless keytab) when submitting a Spark job. This is a best practice: submitting a job under the end user keytab delivers a higher degree of audit capability.

In the following example, end user `$USERNAME` has their own keytab and runs the Spark Pi job in a Kerberos-enabled environment:

```
su $USERNAME
kinit USERNAME@YOUR-LOCAL-REALM.COM
cd /usr/hdp/current/spark-client/
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn-cluster \
  --num-executors 3 \
  --driver-memory 512m \
  --executor-memory 512m \
  --executor-cores 1 \
  lib/spark-examples*.jar 10
```

4. Running Spark

You can run Spark interactively or from a client program:

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

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

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

This chapter describes how to specify Spark version for a Spark application, and how to run Spark 1 and Spark 2 sample programs.

4.1. Specifying Which Version of Spark to Run

More than one version of Spark can run on a node. If your cluster runs Spark 1, you can install Spark 2 and test jobs on Spark 2 in parallel with a Spark 1 working environment. After verifying that all scripts and jobs run successfully with Spark 2 (including any changes for backward compatibility), you can then step through transitioning jobs from Spark 1 to Spark 2. For more information about installing a second version of Spark, see [Installing Spark](#).

Use the following guidelines for determining which version of Spark runs a job by default, and for specifying an alternate version if desired.

- By default, if only one version of Spark is installed on a node, your job runs with the installed version.
- By default, if more than one version of Spark is installed on a node, your job runs with the default version for your HDP package. In HDP 2.6, the default is Spark version 1.6.
- If you want to run jobs on the non-default version of Spark, use one of the following approaches:
 - If you use full paths in your scripts, change `spark-client` to `spark2-client`; for example:

```
change /usr/hdp/current/spark-client/bin/spark-submit
```

to `/usr/hdp/current/spark2-client/bin/spark-submit`.

- If you do not use full paths, but instead launch jobs from the path, set the `SPARK_MAJOR_VERSION` environment variable to the desired version of Spark before you launch the job.

For example, if Spark 1.6.3 and Spark 2.0 are both installed on a node and you want to run your job with Spark 2.0, set

```
SPARK_MAJOR_VERSION=2.
```

You can set `SPARK_MAJOR_VERSION` in automation scripts that use Spark, or in your manual settings after logging on to the shell.

Note: The `SPARK_MAJOR_VERSION` environment variable can be set by any user who logs on to a client machine to run Spark. The scope of the environment variable is local to the user session.

The following example submits a SparkPi job to Spark 2, using `spark-submit` under `/usr/bin`:

1. Navigate to a host where Spark 2.0 is installed.

2. Change to the Spark2 client directory:

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

3. Set the `SPARK_MAJOR_VERSION` environment variable to 2:

```
export SPARK_MAJOR_VERSION=2
```

4. Run the Spark Pi example:

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \  
  --master yarn-client \  
  --num-executors 1 \  
  --driver-memory 512m \  
  --executor-memory 512m \  
  --executor-cores 1 \  
  examples/jars/spark-examples*.jar 10
```

Note that the path to `spark-examples-*.jar` is different than the path used for Spark 1.x.

To change the environment variable setting later, either remove the environment variable or change the setting to the newly desired version.

4.2. Running Sample Spark 1.x Applications

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

4.2.1. Spark Pi

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

Follow these steps to run the Spark Pi example:

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

When the job runs, the library is uploaded into HDFS, so the user running the job needs permission to write to HDFS.

2. Navigate to a node with a Spark client and access the `spark-client` directory:

```
cd /usr/hdp/current/spark-client
su spark
```

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

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn-client \
  --num-executors 1 \
  --driver-memory 512m \
  --executor-memory 512m \
  --executor-cores 1 \
  lib/spark-examples*.jar 10
```

Commonly used options include the following:

<code>--class</code>	The entry point for your application: for example, <code>org.apache.spark.examples.SparkPi</code> .
<code>--master</code>	The master URL for the cluster: for example, <code>spark://23.195.26.187:7077</code> .
<code>--deploy-mode</code>	Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (default is client).
<code>--conf</code>	Arbitrary Spark configuration property in <code>key=value</code> format. For values that contain spaces, enclose “ <code>key=value</code> ” in double quotation marks.
<code><application-jar></code>	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 <code>hdfs://</code> path or a <code>file://</code> path that is present on all nodes.
<code><application-arguments></code>	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.

```
17/03/12 14:28:35 INFO scheduler.DAGScheduler: Job 0 finished: reduce at
SparkPi.scala:36, took 1.721177 s
Pi is roughly 3.141296
17/03/12 14:28:35 INFO spark.ContextCleaner: Cleaned accumulator 1
```

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 and Troubleshooting Spark](#).)

4.2.2. WordCount

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

The following example submits WordCount code to the Scala shell:

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

You can use any text file as input.

2. Log on as a user with HDFS access: for example, your `spark` user (if you defined one) or `hdfs`.

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

```
cd /usr/hdp/current/spark-client/
su spark
```

3. Upload the input file to HDFS:

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

4. Run the Spark shell:

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

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

[illegible]

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

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

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

- View output in the Scala shell:

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

- View the full output from within the Scala shell:

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

- View the output using HDFS:

- a. Exit the Scala shell.

- b. View WordCount job status:

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

You should see output similar to the following:

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

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

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

4.3. Running Sample Spark 2.x Applications

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

4.3.1. Spark Pi

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

Follow these steps to run the Spark Pi example:

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

When the job runs, the library is uploaded into HDFS, so the user running the job needs permission to write to HDFS.

2. Navigate to a node with a Spark client and access the `spark2-client` directory:

```
cd /usr/hdp/current/spark2-client  
  
su spark
```

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

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \  
  --master yarn-client \  
  --num-executors 1 \  
  --driver-memory 512m \  
  --executor-memory 512m \  
  --executor-cores 1 \  
  examples/jars/spark-examples*.jar 10
```

Commonly used options include the following:

<code>--class</code>	The entry point for your application: for example, <code>org.apache.spark.examples.SparkPi</code> .
<code>--master</code>	The master URL for the cluster: for example, <code>spark://23.195.26.187:7077</code> .
<code>--deploy-mode</code>	Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (default is client).
<code>--conf</code>	Arbitrary Spark configuration property in <code>key=value</code> format. For values that contain spaces, enclose “key=value” in double quotation marks.
<code><application-jar></code>	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 <code>hdfs://</code> path or a <code>file://</code> path that is present on all nodes.
<code><application-arguments></code>	Arguments passed to the main method of your main class, if any.

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

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

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

4.3.2. WordCount

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

The following example submits WordCount code to the Scala shell:

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

You can use any text file as input.

2. Log on as a user with HDFS access: for example, your `spark` user (if you defined one) or `hdfs`.

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

```
cd /usr/hdp/current/spark2-client/
su spark
```

- ### 3. Upload the input file to HDFS:

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

4. Run the Spark shell:

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

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

```
Spark context Web UI available at http://172.26.236.247:4041
Spark context available as 'sc' (master = yarn, app id =
  application_1490217230866_0002).
Spark session available as 'spark'.
Welcome to
```

[illegible]

```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_112)
```

```
Type in expressions to have them evaluated.  
Type :help for more information.  
  
scala>
```

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

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

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

- View output in the Scala shell:

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

- View the full output from within the Scala shell:

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

- View the output using HDFS:

- a. Exit the Scala shell.

- b. View WordCount job status:

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

You should see output similar to the following:

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

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

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

5. Submitting Spark Applications Through Livy

Livy is a Spark service that allows local and remote applications to interact with Apache Spark over an open source REST interface. You can use Livy to submit and manage Spark jobs on a cluster. Livy extends Spark capabilities, offering additional multi-tenancy and security features. Applications can run code inside Spark without needing to maintain a local Spark context.

Features include the following:

- Jobs can be submitted from anywhere, using the REST API.
- Livy supports user impersonation: the Livy server submits jobs on behalf of the user who submits the requests. Multiple users can share the same server ("user impersonation" support). This is important for multi-tenant environments, and it avoids unnecessary permission escalation.
- Livy supports security features such as Kerberos authentication and wire encryption.
 - REST APIs are backed by SPNEGO authentication, which the requested user should get authenticated by Kerberos at first.
 - RPCs between Livy Server and Remote SparkContext are encrypted with SASL.
 - The Livy server uses keytabs to authenticate itself to Kerberos.

Livy 0.3.0 supports programmatic and interactive access to Spark1 and Spark2 with Scala 2.10, and Scala 2.11:

- 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 1 and Spark 2 functionality including SparkSession, and SparkSession with Hive enabled.
- Run an interactive session, provided by spark-shell, PySpark, or SparkR REPLs.
- Submit batch applications to Spark.

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

To install Livy on an Ambari-managed cluster, see [Installing Spark Using Ambari](#). To install Livy on a cluster not managed by Ambari, see the Spark sections of the *Command Line Installation Guide*. For additional configuration steps, see [Configuring the Livy Server](#).

5.1. Using Livy with Spark Versions 1 and 2

In HDP 2.6, Livy supports both Spark1 and Spark2 in one build. To specify which version of Spark to use, set `SPARK_HOME` to `Spark1` or `Spark2`. Livy automatically differentiates between the two.

Here is a sample `export` statement in the `livy-env.sh` file:

```
export SPARK_HOME=<path-to>/spark-2.1.0-bin-hadoop2.6
```

Scala Support

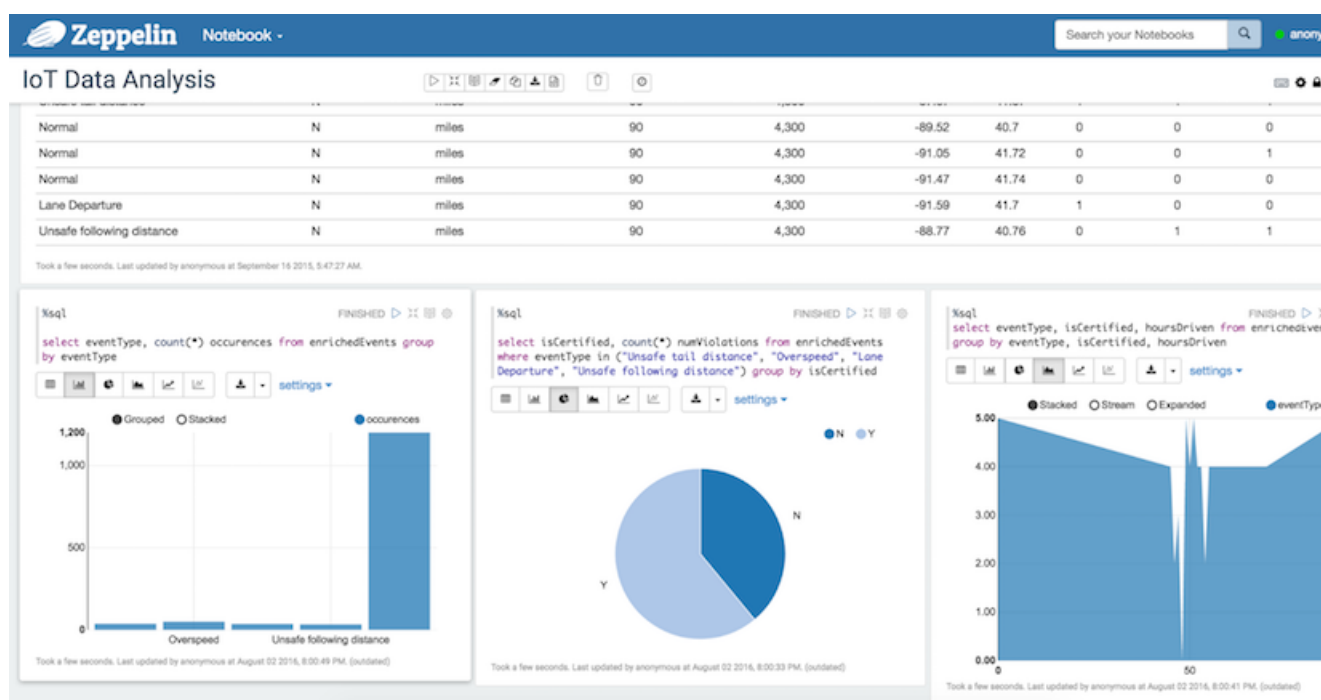
Livy supports Scala versions 2.10 and 2.11.

For default Scala builds, Spark 1.6 with Scala 2.10 or Spark 2.0 with Scala 2.11, Livy automatically detects the correct Scala version and associated jar files.

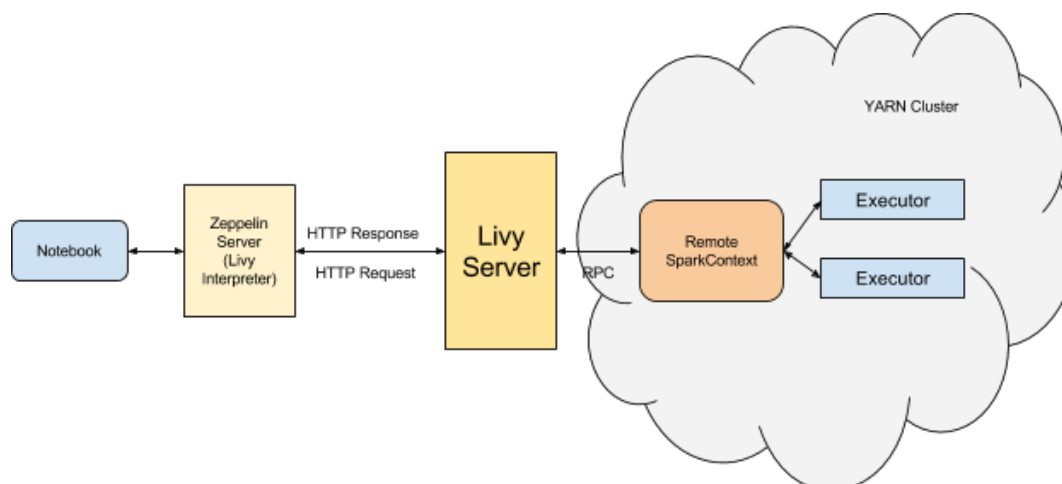
If you require a different Spark-Scala combination, such as Spark 2.0 with Scala 2.10, set `livy.spark.scalaVersion` to the desired version so that Livy uses the right jar files.

5.2. Using Livy with Interactive Notebooks

You can submit Spark commands through Livy from an interactive Apache Zeppelin notebook:



When you run code in a Zeppelin notebook using the `%livy` directive, the notebook offloads code execution to Livy and Spark:



For more information about Zeppelin and Livy, see the [Zeppelin Component Guide](#).

5.3. Using the Livy API to Run Spark Jobs: Overview

Using the Livy API to run Spark jobs is similar to using the original Spark API. The following two examples [calculate Pi](#); the first example uses the Spark API, and the second example uses the Livy API.

Calculate Pi using the Spark API:

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

Calculate Pi using the Livy API:

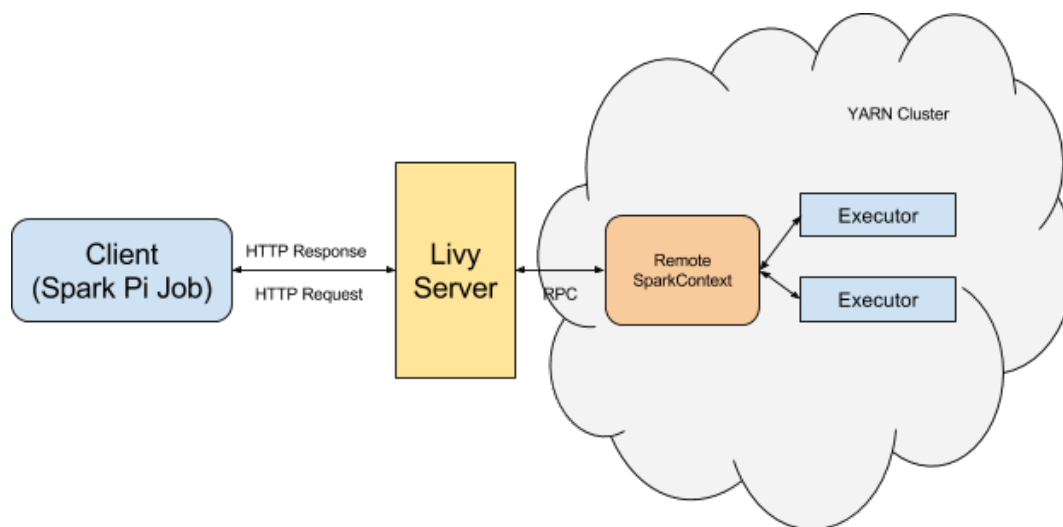
```
def f(_):
    x = random() * 2 - 1
    y = random() * 2 - 1
    return 1 if x ** 2 + y ** 2 <= 1 else 0
def pi_job(context):
    count = context.sc.parallelize(range(1, samples + 1), slices).map(f).
reduce(add)
    return 4.0 * count / samples
```

There are two main differences between the two APIs:

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

Architecturally, the client creates a remote Spark cluster, initializes it, and submits jobs through REST APIs. The Livy server unwraps and rewraps the job, and then sends it to the

remote SparkContext through RPC. While the job runs the client waits for the result, using the same path. The following diagram illustrates the process:



5.4. Running an Interactive Session With the Livy API

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.

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

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

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

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


```
POST /sessions/{sessionId}/statements
  data = {'code': 'sc.parallelize(1 to 10).count()'}
  r = requests.post(statements_url, data=json.dumps(data), headers=
headers)
  r.json()

{'u'output': None, u'state': u'running', u'id': 0}
```

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

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

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

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

5.4.1. Livy Objects for Interactive Sessions

Session Object

A session object represents an interactive shell:

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

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

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

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

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

Statement Object

A statement object represents the result of an execution statement.

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

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

value	Description
waiting	Statement is queued, execution has not started
running	Statement is running
available	Statement has a response ready
error	Statement failed
cancelling	Statement is being cancelled
cancelled	Statement is cancelled

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

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

5.4.2. Setting Path Variables for Python

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

pyspark

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

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

pyspark3

Livy reads the path from environment variable `PYSPARK3_PYTHON`.

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

5.4.3. 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

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

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

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

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

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

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

POST

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

Request Parameter	Description	Type
kind	Session kind (required)	session kind
proxyUser	User ID to impersonate when starting the session	string
jars	Jar files to be used in this session	list of strings
pyFiles	Python files to be used in this session	list of strings
files	Other files to be used in this session	list of strings
driverMemory	Amount of memory to use for the driver process	string
driverCores	Number of cores to use for the driver process	int
executorMemory	Amount of memory to use for each executor process	string
executorCores	Number of cores to use for each executor process	int
numExecutors	Number of executors to launch for this session	int
archives	Archives to be used in this session	list of strings
queue	The name of the YARN queue to which the job should be submitted	string
name	Name of this session	string
conf	Spark configuration properties	Map of key=value
heartbeatTimeoutInSecond	Timeout in second to which session be orphaned	int

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

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

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

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

Request Parameter	Description	Type
code	The code to execute	string

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

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

Response	Description	Type
cancellation message	Reports "cancelled"	string

DELETE

DELETE /sessions/{sessionId} terminates the session.

5.5. Submitting Batch Applications Using the Livy API

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`.

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.

5.5.1. 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
appld	The application ID for this session	String
applInfo	Detailed application info	Map of key=value
log	Log records	list of strings
state	Batch state	string

5.5.2. 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 this session	string
conf	Spark configuration properties	Map of key=val

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

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

6. Running PySpark in a Virtual Environment

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

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

For these situations, you can create a virtual environment as an isolated Python runtime environment. HDP 2.6 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.

For more information, see [Using VirtualEnv with PySpark](#).

7. Automating Spark Jobs with Oozie Spark Action

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

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

For additional information about Spark action, see the Apache Oozie [Spark Action Extension](#) documentation. For general information about Oozie, see [Using HDP for Workflow and Scheduling with Oozie](#). For general information about using Workflow Manager, see the [Workflow Management Guide](#).



Note

In HDP 2.6, Oozie works with either Spark 1 or Spark 2 (not side-by-side deployments), but Spark 2 support for Oozie Spark action is available as a technical preview; it is not ready for production deployment. Configuration is through manual steps (not Ambari).

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

7.1. Configuring Oozie Spark Action for Spark 1

To place a Spark job into an Oozie workflow, you need two configuration files:

- A workflow XML file that defines workflow logic and parameters for running the Spark job. Some of the elements in a Spark action are specific to Spark; others are common to many types of actions.
- A `job.properties` file for configuring the Oozie job.

You can configure a Spark action manually, or on an Ambari-managed cluster you can use the Spark action editor in the Ambari Oozie Workflow Manager (WFM). The Workflow Manager is designed to help build powerful workflows.

For two examples that use Oozie Workflow Manager—one that creates a new Spark action, and another that imports and runs an existing Spark workflow—see the Hortonworks Community Connection article [Apache Ambari Workflow Manager View for Apache Oozie: Part 7 \(Spark Action & PySpark\)](#).

Here is the basic structure of a workflow definition XML file for a Spark action:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:oozie:workflow:0.3">
  ...
  <action name="[NODE-NAME]">
```

```

    <spark xmlns="uri:oozie:spark-action:0.1">
      <job-tracker>[JOB-TRACKER]</job-tracker>
      <name-node>[NAME-NODE]</name-node>
      <prepare>
        <delete path="[PATH]" />
        ...
        <mkdir path="[PATH]" />
        ...
      </prepare>
      <job-xml>[SPARK SETTINGS FILE]</job-xml>
      <configuration>
        <property>
          <name>[PROPERTY-NAME]</name>
          <value>[PROPERTY-VALUE]</value>
        </property>
        ...
      </configuration>
      <master>[SPARK MASTER URL]</master>
      <mode>[SPARK MODE]</mode>
      <name>[SPARK JOB NAME]</name>
      <class>[SPARK MAIN CLASS]</class>
      <jar>[SPARK DEPENDENCIES JAR / PYTHON FILE]</jar>
      <spark-opts>[SPARK-OPTIONS]</spark-opts>
      <arg>[ARG-VALUE]</arg>
      ...
      <arg>[ARG-VALUE]</arg>
      ...
    </spark>
    <ok to="[NODE-NAME]" />
    <error to="[NODE-NAME]" />
  </action>
  ...
</workflow-app>

```

The following examples show a workflow definition XML file and an Oozie job configuration file for running a SparkPi job (Spark version 1.x).

Sample Workflow.xml file for SparkPi app:

```

<workflow-app xmlns='uri:oozie:workflow:0.5# name='SparkWordCount'>
  <start to='spark-node' />
  <action name='spark-node'>
    <spark xmlns="uri:oozie:spark-action:0.1">
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="${nameNode}/user/${wf:user()}/${examplesRoot}/
output-data" />
      </prepare>
      <master>${master}</master>
      <name>SparkPi</name>
      <class>org.apache.spark.examples.SparkPi</class>
      <jar>lib/spark-examples.jar</jar>
      <spark-opts>--executor-memory 20G --num-executors 50</spark-opts>
      <arg>value=10</arg>
    </spark>
    <ok to="end" />
    <error to="fail" />
  </action>
  <kill name="fail">

```

```

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

```

Sample Job.properties file for SparkPi app:

```

nameNode=hdfs://host:8020
jobTracker=host:8050
queueName=default
examplesRoot=examples
oozie.use.system.libpath=true
oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/apps/
pyspark
master=yarn-cluster

```

7.2. Configuring Oozie Spark Action for Spark 2

To use Oozie Spark action with Spark 2 jobs, create a spark2 ShareLib directory, copy associated files into it, and then point Oozie to spark2. (The Oozie ShareLib is a set of libraries that allow jobs to run on any node in a cluster.)

1. Create a spark2 ShareLib directory under the Oozie ShareLib directory associated with the oozie service user:

```
hdfs dfs -mkdir /user/oozie/share/lib/lib_<ts>/spark2
```

2. Copy spark2 jar files from the spark2 jar directory to the Oozie spark2 ShareLib:

```

hdfs dfs -put \
    /usr/hdp/<version>/spark2/jars/* \
    /user/oozie/share/lib/lib_<ts>/spark2/

```

3. Copy the oozie-sharelib-spark jar file from the spark ShareLib directory to the spark2 ShareLib directory:

```

hdfs dfs -cp \
    /user/oozie/share/lib/lib_<ts>/spark/oozie-sharelib-spark-<version>.jar \
    /user/oozie/share/lib/lib_<ts>/spark2/

```

4. Copy the hive-site.xml file from the current spark ShareLib to the spark2 ShareLib:

```

hdfs dfs -cp \
    /user/oozie/share/lib/lib_<ts>/spark/hive-site.xml \
    /user/oozie/share/lib/lib_<ts>/spark2/

```

5. Copy Python libraries to the spark2 ShareLib:

```

hdfs dfs -put \
    /usr/hdp/<version>/spark2/python/lib/py* \
    /user/oozie/share/lib/lib_<ts>/spark2/

```

6. Run the Oozie sharelibupdate command:

```
oozie admin -sharelibupdate
```

To verify the configuration, run the Oozie `shareliblist` command. You should see `spark2` in the results.

```
oozie admin -shareliblist spark2
```

To run a Spark job with the `spark2` ShareLib, add the `action.sharelib.for.spark` property to the `job.properties` file, and set its value to `spark2`:

```
oozie.action.sharelib.for.spark=spark2
```

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

Sample Workflow.xml file for spark2-Pi:

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

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

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

Sample Job.properties file for spark2-Pi:

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

Sample Python script, lib/pi.py:

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

if __name__ == "__main__":
    """
    Usage: pi [partitions]
    """
```

```
sc = SparkContext(appName="Python-Spark-Pi")
partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
n = 100000 * partitions

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

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

sc.stop()
```

8. Developing Spark Applications

Apache Spark enables you to quickly develop applications and process jobs. It is designed for fast application development and processing. Spark Core is the underlying execution engine; other services, such as Spark SQL, MLlib, and Spark Streaming, are built on top of the Spark Core.

Depending on your use case, you can extend your use of Spark into several domains, including the following described in this chapter:

- Spark DataFrames
- Spark SQL
- Calling Hive user-defined functions from Spark SQL
- Spark Streaming
- Accessing HBase tables, HDFS files, and ORC data (Hive)
- Using custom libraries

Additional resources:

- To get started with Spark, see the Apache Spark [Quick Start](#) and the [Spark 1.6.3](#) and [Spark 2.0](#) overviews.
- For more information about application development, see the Apache [Spark Programming Guide](#).
- For more information about using Livy to submit Spark jobs, see [Submitting Spark Applications Through Livy](#).

8.1. Using the Spark DataFrame API

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R or in the Python `pandas` library. You can construct DataFrames from a wide array of sources, including structured data files, Apache Hive tables, and existing Spark resilient distributed datasets (RDD). The Spark DataFrame API is available in Scala, Java, Python, and R.

This subsection contains several examples of DataFrame API use.

To list JSON file contents as a DataFrame:

1. As user `spark`, upload the `people.txt` and `people.json` sample files to the Hadoop Distributed File System (HDFS):

```
cd /usr/hdp/current/spark-client
su spark
hdfs dfs -copyFromLocal examples/src/main/resources/people.txt people.txt
hdfs dfs -copyFromLocal examples/src/main/resources/people.json people.json
```

2. Launch the Spark shell:

```
cd /usr/hdp/current/spark-client
su spark
./bin/spark-shell --num-executors 1 --executor-memory 512m --master yarn-
client
```

3. At the Spark shell, type the following:

```
scala> val df = sqlContext.read.format("json").load("people.json")
```

4. Using `df.show`, display the contents of the DataFrame:

```
scala> df.show
17/03/12 11:24:10 INFO YarnScheduler: Removed TaskSet 2.0, whose tasks have
all completed, from pool

+----+-----+
| age|   name|
+----+-----+
| null|Michael|
|   30|   Andy|
|   19|  Justin|
+----+-----+
```

The following examples use Scala to access DataFrame `df` defined in the previous subsection:

```
// Import the DataFrame functions API
scala> import org.apache.spark.sql.functions._

// Select all rows, but increment age by 1
scala> df.select(df("name"), df("age") + 1).show()

// Select people older than 21
scala> df.filter(df("age") > 21).show()

// Count people by age
scala> df.groupBy("age").count().show()
```

The following example uses the DataFrame API to specify a schema for `people.txt`, and then retrieves names from a temporary table associated with the schema:

```
import org.apache.spark.sql._

val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val people = sc.textFile("people.txt")
val schemaString = "name age"

import org.apache.spark.sql.types.{StructType, StructField, StringType}

val schema = StructType(schemaString.split(" ").map(fieldName =>
  StructField(fieldName, StringType, true)))
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

peopleDataFrame.registerTempTable("people")

val results = sqlContext.sql("SELECT name FROM people")
```

```
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

This produces output similar to the following:

```
17/03/12 14:36:49 INFO cluster.YarnScheduler: Removed TaskSet 13.0, whose
tasks have all completed, from pool
17/03/12 14:36:49 INFO scheduler.DAGScheduler: ResultStage 13 (collect at :33)
finished in 0.129 s
17/03/12 14:36:49 INFO scheduler.DAGScheduler: Job 10 finished: collect
at :33, took 0.162827 s
Name: Michael
Name: Andy
Name: Justin
```

8.2. Using Spark SQL

Using `SQLContext`, Apache Spark SQL can read data directly from the file system. This is useful when the data you are trying to analyze does not reside in Apache Hive (for example, JSON files stored in HDFS).

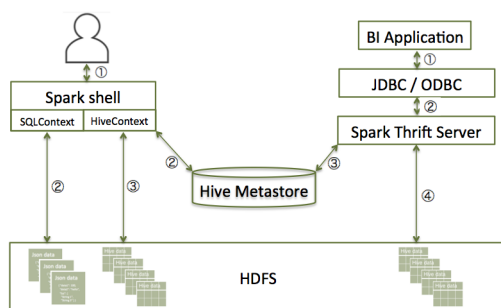
Using `HiveContext`, Spark SQL can also read data by interacting with the Hive MetaStore. If you already use Hive, you should use `HiveContext`; it supports all Hive data formats and user-defined functions (UDFs), and it enables you to have full access to the HiveQL parser. `HiveContext` extends `SQLContext`, so `HiveContext` supports all `SQLContext` functionality.

There are two ways to interact with Spark SQL:

- Interactive access using the Spark shell (see [Accessing Spark SQL through the Spark Shell](#)).
- From an application, operating through one of the following two APIs and the Spark Thrift server:
 - JDBC, using your own Java code or the [Beeline](#) JDBC client
 - ODBC, through the Simba ODBC driver

For more information about JDBC and ODBC access, see [Accessing Spark SQL through JDBC: Prerequisites](#) and [Accessing Spark SQL through JDBC and ODBC](#).

The following diagram illustrates the access process, depending on whether you are using the Spark shell or business intelligence (BI) application:



The following subsections describe how to access Spark SQL through the Spark shell, and through JDBC and ODBC.

8.2.1. Accessing Spark SQL through the Spark Shell

The following sample command launches the Spark shell on a YARN cluster:

```
./bin/spark-shell --num-executors 1 --executor-memory 512m --  
master yarn-client
```

To read data directly from the file system, construct a `SQLContext`. For an example that uses `SQLContext` and the Spark `DataFrame` API to access a JSON file, see [Using the Spark DataFrame API](#).

To read data by interacting with the Hive Metastore, construct a `HiveContext` instance (`HiveContext` extends `SQLContext`). For an example of the use of `HiveContext` (instantiated as `val sqlContext`), see [Accessing ORC Files from Spark](#).

8.2.2. Accessing Spark SQL through JDBC or ODBC: Prerequisites

Using the Spark Thrift server, you can remotely access Spark SQL over JDBC (using the JDBC [Beeline](#) client) or ODBC (using the Simba driver).

The following prerequisites must be met before accessing Spark SQL through JDBC or ODBC:

- The Spark Thrift server must be deployed on the cluster.
 - For an Ambari-managed cluster, deploy and launch the Spark Thrift server using the Ambari web UI (see [Installing and Configuring Spark Over Ambari](#)).
 - For a cluster that is not managed by Ambari, see [Starting the Spark Thrift Server](#) in the *Non-Ambari Cluster Installation Guide*.
- Ensure that `SPARK_HOME` is defined as your Spark directory:

```
export SPARK_HOME=/usr/hdp/current/spark-client
```

If you want to enable user impersonation for the Spark Thrift server, so that the Thrift server runs Spark SQL jobs as the submitting user, see [Configuring the Spark Thrift server](#).

Before accessing Spark SQL through JDBC or ODBC, note the following caveats:

- The Spark Thrift server works in YARN client mode only.
- ODBC and JDBC client configurations must match Spark Thrift server configuration parameters. For example, if the Thrift server is configured to listen in binary mode, the client should send binary requests and use HTTP mode when the Thrift server is configured over HTTP.
- All client requests coming to the Spark Thrift server share a `SparkContext`.

Additional Spark Thrift Server Commands

To list available Thrift server options, run `./sbin/start-thriftserver.sh --help`.

To manually stop the Spark Thrift server, run the following commands:

```
su spark
./sbin/stop-thriftserver.sh
```

8.2.3. Accessing Spark SQL through JDBC

To access Spark SQL through JDBC, you need a JDBC URL connection string to supply connection information to the JDBC data source. Connection strings for the Spark SQL JDBC driver have the following format:

```
jdbc:hive2://<host>:<port>/<dbName>;<sessionConfs>?
<hiveConfs>#<hiveVars>
```

JDBC Parameter	Description
host	The node hosting the Thrift server
port	The port number on which the Thrift server listens
dbName	The name of the Hive database to run the query against
sessionConfs	Optional configuration parameters for the JDBC or ODBC driver in the following format: <key1>=<value1>;<key2>=<key2>...;
hiveConfs	Optional configuration parameters for Hive on the server in the following format: <key1>=<value1>;<key2>=<key2>; ... These settings last for the duration of the user session.
hiveVars	Optional configuration parameters for Hive variables in the following format: <key1>=<value1>;<key2>=<key2>; ... These settings persist for the duration of the user session.



Note

The Spark Thrift server is a variant of HiveServer2, so you can use many of the same settings. For more information about JDBC connection strings, including transport and security settings, see [Hive JDBC and ODBC Drivers](#) in the *HDP Data Access Guide*.

The following connection string accesses Spark SQL through JDBC on a Kerberos-enabled cluster:

```
beeline> !connect jdbc:hive2://localhost:10002/default;httpPath=/;principal=
hive/hdp-team.example.com@EXAMPLE.COM
```

The following connection string accesses Spark SQL through JDBC over HTTP transport on a Kerberos-enabled cluster:

```
beeline> !connect jdbc:hive2://localhost:10002/default;transportMode=
http;httpPath=/;principal=hive/hdp-team.example.com@EXAMPLE.COM
```

To access Spark SQL, complete the following steps:

1. Connect to the Thrift server over the Beeline JDBC client.
 - a. From the SPARK_HOME directory, launch Beeline:

```
su spark
./bin/beeline
```

- b. At the Beeline prompt, connect to the Spark SQL Thrift server with the JDBC connection string:

```
beeline> !connect jdbc:hive2://localhost:10015
```

The host port must match the host port on which the Spark Thrift server is running.

You should see output similar to the following:

```
beeline> !connect jdbc:hive2://localhost:10015
Connecting to jdbc:hive2://localhost:10015
Enter username for jdbc:hive2://localhost:10015:
Enter password for jdbc:hive2://localhost:10015:
...
Connected to: Spark SQL (version 1.6.3)
Driver: Spark Project Core (version 1.6.3.2.4.0.0-169)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10015>
```

2. When connected, issue a Spark SQL statement.

The following example executes a SHOW TABLES query:

```
0: jdbc:hive2://localhost:10015> show tables;
+-----+-----+
| tableName | isTemporary |
+-----+-----+
| sample_07 | false       |
| sample_08 | false       |
| testtable | false       |
+-----+-----+
3 rows selected (2.399 seconds)
0: jdbc:hive2://localhost:10015>
```

8.2.4. Accessing Spark SQL through ODBC

If you want to access Spark SQL through ODBC, first download the ODBC Spark driver for the operating system you want to use for the ODBC client. After downloading the driver, refer to the *Hortonworks ODBC Driver with SQL Connector for Apache Spark User Guide* for installation and configuration instructions.

Drivers and associated documentation are available in the "Hortonworks Data Platform Add-Ons" section of the Hortonworks downloads page (<http://hortonworks.com/downloads/>) under "Hortonworks ODBC Driver for SparkSQL." If the latest version of HDP is newer than your version, check the Hortonworks Data Platform Archive area of the add-ons section for the version of the driver that corresponds to your version of HDP.

8.2.5. Spark SQL User Impersonation

When user impersonation is enabled for Spark SQL through the Spark Thrift server, the Thrift server runs queries as the submitting user. By running queries under the user account associated with the submitter, the Thrift Server can enforce user level permissions and access control lists. This enables granular access control to Spark SQL at the level of files or tables. Associated data cached in Spark is visible only to queries from the submitting user.

Spark SQL user impersonation is supported for Apache Spark 1 versions 1.6.3 and later. To enable user impersonation, see [Enabling User Impersonation for the Spark Thrift Server](#). The following paragraphs illustrate several features of user impersonation.

8.2.5.1. Permissions and ACL Enforcement

When user impersonation is enabled, permissions and ACL restrictions are applied on behalf of the submitting user. In the following example, “foo_db” database has a table “drivers”, which only user “foo” can access:

```
[[root@sandbox spark]# hdfs dfs -ls /apps/hive/warehouse/foo_db.db
Found 1 items
drwx----- - foo hdfs          0 2017-02-22 14:59 /apps/hive/warehouse/
[root@sandbox spark]#
```

A Beeline session running as user “foo” can access the data, read the drivers table, and create a new table based on the table:

```
[[foo@sandbox ~]$ beeline
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
beeline> !connect jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM
Connecting to jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM
Enter username for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM:
Enter password for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM:
Connected to: Spark SQL (version 1.6.3)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://sandbox.hortonworks.com:10015> show tables;
+-----+-----+
| tableName | isTemporary |
+-----+-----+
| drivers   | false       |
+-----+-----+
1 row selected (0.244 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015> select * from drivers limit 2;
+-----+-----+-----+-----+-----+-----+
| driverid | name          | ssn      | location          | certified | wageplan |
+-----+-----+-----+-----+-----+-----+
| 10       | George Vetticaden | 621011971 | 244-4532 Nulla Rd. | N        | miles    |
| 11       | Jamie Engesser   | 262112338 | 366-4125 Ac Street | N        | miles    |
+-----+-----+-----+-----+-----+-----+
2 rows selected (7.505 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015> create table driver_data like drivers;
+-----+
| result |
+-----+
+-----+
No rows selected (0.658 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015> insert into driver_data select * from drivers;
+-----+
| Result |
+-----+
+-----+
No rows selected (1.33 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015> select * from driver_data limit 2;
+-----+-----+-----+-----+-----+-----+
| driverid | name          | ssn      | location          | certified | wageplan |
+-----+-----+-----+-----+-----+-----+
| 10       | George Vetticaden | 621011971 | 244-4532 Nulla Rd. | N        | miles    |
| 11       | Jamie Engesser   | 262112338 | 366-4125 Ac Street | N        | miles    |
+-----+-----+-----+-----+-----+-----+
2 rows selected (0.331 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015>
```

Spark queries run in a YARN application as user “foo”:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Contain
<u>application_1487790747426_0008</u>	foo	User SparkThriftServerApp	SPARK	default	0	Wed Feb 22 12:03:00 -0800 2017	N/A	RUNNING	UNDEFINED	1

All user permissions and access control lists are enforced while accessing tables, data or other resources. In addition, all output generated is for user "foo".

For the table created in the preceding Beeline session, the owner is user "foo":

```
[[root@sandbox spark]# hdfs dfs -ls /apps/hive/warehouse/foo_db.db
Found 2 items
drwxr-xr-x - foo hdfs 0 2017-02-22 15:45 /apps/hive/warehouse/foo_db.db/driver_data
drwx----- - foo hdfs 0 2017-02-22 14:59 /apps/hive/warehouse/foo_db.db/drivers
[root@sandbox spark]#
```

The per-user Spark Application Master ("AM") caches data in memory without other users being able to access the data—cached data and state are restricted to the Spark AM running the query. Data and state information are not stored in the Spark Thrift server, so they are not visible to other users. Spark master runs as yarn-cluster, but query execution works as though it is yarn-client (essentially a yarn-cluster user program that accepts queries from STS indefinitely).

8.2.5.2. Spark Thrift Server as Proxy

The Spark Thrift server does not run user queries; it forwards them to the appropriate user-specific Spark AM. This improves the scalability and fault tolerance of the Spark Thrift server.

<u>application_1487790747426_0013</u>	bar	User SparkThriftServerApp	SPARK	default	0	V
<u>application_1487790747426_0010</u>	foo	User SparkThriftServerApp	SPARK	default	0	V

When user impersonation is enabled for the Spark Thrift server, the Thrift Server is responsible for the following features and capabilities:

- Authorizing incoming user connections (SASL authorization that validates the user Beeline/socket connection).
- Managing Spark applications launched on behalf of users:
 - Launching Spark application if no appropriate application exists for the incoming request.
 - Terminating the Spark AM when all associated user connections are closed at the Spark Thrift server.
- Acting as a proxy and forwarding requests/responses to the appropriate user's Spark AM.

- Ensuring that long-running Spark SQL sessions persist, by keeping the Kerberos state valid.
- The Spark Thrift server and Spark AM, when launched on behalf of a user, can be long-running applications in clusters with Kerberos enabled.
- The submitter's principal and keytab are not required for long-running Spark AM processes, although the Spark Thrift server requires the Hive principal and keytab.

8.2.5.3. Enhancements for Connection URL Support.

The connection URL format for Hive is described in [Apache Hive](#) documentation. In user impersonation mode, the Spark Thrift server supports a default database and `hivevar` variables.

Specifying Default Database in the Connection URL

Specifying the connection URL as `jdbc:hive2://$HOST:$PORT/my_db` results in an implicit "use my_db" when a user connects.

For an example, see the preceding Beeline example where the `!connect` command specifies the connection URL for "foo_db".

Support for `hivevar` Variables

Hive variables can be used to parameterize queries. To set a Hive variable, use the `set hivevar` command:

```
set hivevar:key=value
```

You can also set a Hive variable as part of the connection URL. In the following Beeline example, `plan=miles` is appended to the connection URL. The variable is referenced in the query as `${hivevar:plan}`.

```
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
[beeline> !connect jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=
Connecting to jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=
[Enter username for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=
[Enter password for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=
Connected to: Spark SQL (version 1.6.3)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://sandbox.hortonworks.com:10015>
[0: jdbc:hive2://sandbox.hortonworks.com:10015> select * from drivers where
+-----+-----+-----+-----+-----+
| driverid | name | ssn | location | cert |
+-----+-----+-----+-----+-----+
| 10 | George Vetticaden | 621011971 | 244-4532 Nulla Rd. | N |
| 11 | Jamie Engesser | 262112338 | 366-4125 Ac Street | N |
+-----+-----+-----+-----+-----+
2 rows selected (8.897 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015> █
```

8.2.5.4. Advanced Connection Management Features

By default, all connections for a user are forwarded to the same Spark AM to execute the query. In some cases, it is necessary to exercise finer-grained control.

Specifying Named Connections

When user impersonation is enabled, Spark supports user-named connections identified by a user-specified `connectionId` (a Hive `conf` parameter in the connection URL). This can be useful when overriding Spark configurations such as queue, memory configuration, or executor configuration settings.

Every Spark AM managed by the Spark Thrift server is associated with a user and `connectionId`. Connection IDs are not globally unique; they are specific to the user.

You can specify `connectionId` to control which Spark AM executes queries. If you not specify `connectionId`, a default `connectionId` is associated with the Spark AM.

To explicitly name a connection, set the Hive `conf` parameter to `spark.sql.thriftServer.connectionId`, as shown in the following session:

```
[[foo@sandbox ~]$ beeline
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
[beeline> !connect jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer
Connecting to jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer
Enter username for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer
Enter password for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer
Connected to: Spark SQL (version 1.6.3)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://sandbox.hortonworks.com:10015>
```

Note: Named connections allow users to specify their own Spark AM connections. They are scoped to individual users, and do not allow a user to access the Spark AM associated with another user.

application_1487790747426_0020	foo	User SparkThriftServerApp connection id = conn1	SPARK	default	0	Wed Feb 22 13:57:15 -0800 2017	N/A	RUNNING	UNDEFINED	1
application_1487790747426_0019	bar	User SparkThriftServerApp connection id = conn1	SPARK	default	0	Wed Feb 22 13:57:01 -0800 2017	N/A	RUNNING	UNDEFINED	1

If the Spark AM is available, the connection is associated with the existing Spark AM.

Data Sharing and Named Connections

Each `connectionId` for a user identifies a different Spark AM.

For a user, cached data is shared and available only within a single AM, not across Spark AM's.

Different user connections on the same Spark AM can leverage previously cached data. Each user connection has its own Hive session (which maintains the current database, Hive variables, and so on), but shares the underlying cached data, executors, and Spark application.

The following example shows a session for the first connection from user "foo" to named connection "conn1":


```

[[foo@sandbox ~]$ beeline
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
[beeline> !connect jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
Connecting to jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
[Enter username for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
[Enter password for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
Connected to: Spark SQL (version 1.6.3)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
[0: jdbc:hive2://sandbox.hortonworks.com:10015>
[0: jdbc:hive2://sandbox.hortonworks.com:10015> select count(*) from drivers;
+-----+---+
| _c0    |
+-----+---+
| 34     |
+-----+---+
1 row selected (8.32 seconds)
[0: jdbc:hive2://sandbox.hortonworks.com:10015> cache table drivers;
+-----+---+
| Result  |
+-----+---+
+-----+---+
No rows selected (2.242 seconds)
[0: jdbc:hive2://sandbox.hortonworks.com:10015> select count(*) from drivers;
+-----+---+
| _c0    |
+-----+---+
| 34     |
+-----+---+
1 row selected (0.323 seconds)
[0: jdbc:hive2://sandbox.hortonworks.com:10015> █

```

After caching the 'drivers' table, the query runs an order of magnitude faster.

A second connection to the same `connectionId` from user "foo" leverages the cached table from the other active Beeline session, significantly increasing query execution speed:

```

[[foo@sandbox ~]$ beeline
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
[beeline> !connect jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
Connecting to jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
[Enter username for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
[Enter password for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=pool;hiveconf=hive.hadoopi
Connected to: Spark SQL (version 1.6.3)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
[0: jdbc:hive2://sandbox.hortonworks.com:10015> select count(*) from drivers;
+-----+---+
| _c0    |
+-----+---+
| 34     |
+-----+---+
1 row selected (0.402 seconds)
[0: jdbc:hive2://sandbox.hortonworks.com:10015> █

```


Overriding Spark Configuration Settings

If the Spark Thrift server is unable to find an existing Spark AM for a user connection, by default the Thrift server launches a new Spark AM to service user queries. This is applicable to named connections and unnamed connections. When a new Spark AM is to be launched, you can override current Spark configuration settings by specifying them in the connection URL. Specify Spark configuration settings as `hiveconf` variables prepended by the `sparkconf` prefix:

```
Beeline version 1.2.1000.2.5.3.0-37 by Apache Hive
[beeline>
[beeline> !connect jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=
Connecting to jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=
Enter username for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=
Enter password for jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer=
Connected to: Spark SQL (version 1.6.3)
Driver: Hive JDBC (version 1.2.1000.2.5.3.0-37)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://sandbox.hortonworks.com:10015> select * from drivers where wageplan="miles";
+-----+-----+-----+-----+-----+-----+
| driverid | name          | ssn      | location          | certified | wageplan |
+-----+-----+-----+-----+-----+-----+
| 10       | George Vetticaden | 621011971 | 244-4532 Nulla Rd. | N        | miles    |
| 11       | Jamie Engesser   | 262112338 | 366-4125 Ac Street | N        | miles    |
+-----+-----+-----+-----+-----+-----+
2 rows selected (10.617 seconds)
0: jdbc:hive2://sandbox.hortonworks.com:10015>
```

The following connection URL includes a `spark.executor.memory` setting of 4 GB:

```
jdbc:hive2://sandbox.hortonworks.com:10015/foo_db;principal=hive/_HOST@REALM.COM?spark.sql.thriftServer.connectionId=my_conn;sparkconf.spark.executor.memory=4g
```

The environment tab of the Spark application shows the appropriate value:

spark.eventLog.enabled
spark.executor.extraLibraryPath
spark.executor.id
spark.executor.memory
spark.externalBlockStore.folderName
spark.hadoop.cacheConf

8.3. Calling Hive User-Defined Functions

You can call built-in Hive UDFs, UDAFs, and UDTFs and custom UDFs from Spark SQL applications if the functions are available in the standard Hive .jar file. When using Hive UDFs, use `HiveContext` (not `SQLContext`).

8.3.1. Using Built-in UDFs

The following interactive example reads and writes to HDFS under Hive directories, using `hiveContext` and the built-in `collect_list(col)` UDF. The `collect_list(col)` UDF returns a list of objects with duplicates. In a production environment, this type of

operation runs under an account with appropriate HDFS permissions; the following example uses `hdfs` user.

1. Launch the Spark Shell on a YARN cluster:

```
su hdfs
cd $SPARK_HOME
./bin/spark-shell --num-executors 2 --executor-memory 512m --master yarn-
client
```

2. At the Scala REPL prompt, construct a `HiveContext` instance:

```
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

3. Invoke the Hive `collect_list` UDF:

```
scala> hiveContext.sql("from TestTable SELECT key, collect_list(value) group
by key order by key").collect.foreach(println)
```

8.3.2. Using Custom UDFs

You can register custom functions in Python, Java, or Scala, and use them within SQL statements.

When using a custom UDF, ensure that the `.jar` file for your UDF is included with your application, or use the `--jars` command-line option to specify the file.

The following example uses a custom Hive UDF. This example uses the more limited `SQLContext`, instead of `HiveContext`.

1. Launch `spark-shell` with `hive-udf.jar` as its parameter:

```
./bin/spark-shell --jars <path-to-your-hive-udf>.jar
```

2. From `spark-shell`, define a function:

```
scala> sqlContext.sql("""create temporary function balance as 'org.package.
hiveudf.BalanceFromRechargesAndOrders'""");
```

3. From `spark-shell`, invoke your UDF:

```
scala> sqlContext.sql("""
create table recharges_with_balance_array as
select
  reseller_id,
  phone_number,
  phone_credit_id,
  date_recharge,
  phone_credit_value,
  balance(orders,'date_order', 'order_value', reseller_id, date_recharge,
  phone_credit_value) as balance
from orders
""");
```

8.4. Using Spark Streaming

Spark Streaming is an extension of the core `spark` package. Using Spark Streaming, your applications can ingest data from sources such as Apache Kafka and Apache Flume; process

the data using complex algorithms expressed with high-level functions like `map`, `reduce`, `join`, and `window`; and send results to file systems, databases, and live dashboards.

Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches:



See the Apache [Spark Streaming Programming Guide](#) for conceptual information; programming examples in Scala, Java, and Python; and performance tuning recommendations.

Apache Spark 1.6 has built-in support for the Apache Kafka 0.8 API. If you want to access a Kafka 0.10 cluster using new Kafka 0.10 APIs (such as wire encryption support) from Spark 1.6 streaming jobs, the [spark-kafka-0-10-connector](#) package supports a Kafka 0.10 connector for Spark 1.x streaming. See the package readme file for additional documentation.

The remainder of this subsection describes general steps for developers using Spark Streaming with Kafka on a Kerberos-enabled cluster; it includes a sample `pom.xml` file for Spark Streaming applications with Kafka. For additional examples, see the Apache GitHub example repositories for [Scala](#), [Java](#), and [Python](#).



Important

Dynamic Resource Allocation does not work with Spark Streaming.

8.4.1. Prerequisites

Before running a Spark Streaming application, Spark and Kafka must be deployed on the cluster.

Unless you are running a job that is part of the Spark examples package installed by Hortonworks Data Platform (HDP), you must add or retrieve the HDP `spark-streaming-kafka` .jar file and associated .jar files before running your Spark job.

8.4.2. Building and Running a Secure Spark Streaming Job

Depending on your compilation and build processes, one or more of the following tasks might be required before running a Spark Streaming job:

- If you are using `maven` as a compile tool:
 1. Add the Hortonworks repository to your `pom.xml` file:

```
<repository>
  <id>hortonworks</id>
  <name>hortonworks repo</name>
  <url>http://repo.hortonworks.com/content/repositories/releases/</url>
</repository>
```

2. Specify the Hortonworks version number for Spark streaming Kafka and streaming dependencies to your `pom.xml` file:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming-kafka_2.10</artifactId>
  <version>1.6.3.2.4.2.0-90</version>
</dependency>

<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming_2.10</artifactId>
  <version>1.6.3.2.4.2.0-90</version>
  <scope>provided</scope>
</dependency>
```

Note that the correct version number includes the Spark version and the HDP version.

3. (Optional) If you prefer to pack an uber .jar rather than use the default ("provided"), add the `maven-shade-plugin` to your `pom.xml` file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.3</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <filters>
      <filter>
        <artifact>*:*</artifact>
        <excludes>
          <exclude>META-INF/*.SF</exclude>
          <exclude>META-INF/*.DSA</exclude>
          <exclude>META-INF/*.RSA</exclude>
        </excludes>
      </filter>
    </filters>
    <finalName>uber-${project.artifactId}-${project.version}</finalName>
  </configuration>
</plugin>
```

- Instructions for submitting your job depend on whether you used an uber .jar file or not:

- If you kept the default .jar scope and you can access an external network, use `--packages` to download dependencies in the runtime library:

```
spark-submit --master yarn-client \  
  --num-executors 1 \  
  --packages org.apache.spark:spark-streaming-kafka_2.10:1.6.3.2.4.2.  
0-90 \  
  --repositories http://repo.hortonworks.com/content/repositories/  
releases/ \  
  --class <user-main-class> \  
  <user-application.jar> \  
  <user arg lists>
```

The artifact and repository locations should be the same as specified in your `pom.xml` file.

- If you packed the .jar file into an uber .jar, submit the .jar file in the same way as you would a regular Spark application:

```
spark-submit --master yarn-client \  
  --num-executors 1 \  
  --class <user-main-class> \  
  <user-uber-application.jar> \  
  <user arg lists>
```

For a sample `pom.xml` file, see [Sample pom.xml file for Spark Streaming with Kafka](#).

8.4.3. Running Spark Streaming Jobs on a Kerberos-Enabled Cluster

To run a Spark Streaming job on a Kerberos-enabled cluster, complete the following steps:

1. Select or create a user account to be used as principal.

This should not be the `kafka` or `spark` service account.

2. Generate a keytab for the user.
3. Create a Java Authentication and Authorization Service (JAAS) login configuration file: for example, `key.conf`.
4. Add configuration settings that specify the user keytab.

The keytab and configuration files are distributed using YARN local resources. Because they reside in the current directory of the Spark YARN container, you should specify the location as `./v.keytab`.

The following example specifies keytab location `./v.keytab` for principal `vagrant@example.com`:

```
KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="/v.keytab"
  storeKey=true
  useTicketCache=false
  serviceName="kafka"
  principal="vagrant@EXAMPLE.COM";
};
```

5. In your `spark-submit` command, pass the JAAS configuration file and keytab as local resource files, using the `--files` option, and specify the JAAS configuration file options to the JVM options specified for the driver and executor:

```
spark-submit \
  --files key.conf#key.conf,v.keytab#v.keytab \
  --driver-java-options "-Djava.security.auth.login.config=./key.conf" \
  --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.
config=./key.conf" \
  ...
```

6. Pass any relevant Kafka security options to your streaming application.

For example, the `KafkaWordCount` example accepts `PLAINTEXTSASL` as the last option in the command line:

```
KafkaWordCount /vagrant/spark-examples.jar c6402:2181 abc ts 1
PLAINTEXTSASL
```

8.4.4. Sample `pom.xml` File for Spark Streaming with Kafka

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>test</groupId>
  <artifactId>spark-kafka</artifactId>
  <version>1.0-SNAPSHOT</version>

  <repositories>
    <repository>
      <id>hortonworks</id>
      <name>hortonworks repo</name>
      <url>http://repo.hortonworks.com/content/repositories/releases/</
url>
    </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-streaming-kafka_2.10</artifactId>
      <version>1.6.3.2.4.2.0-90</version>
    </dependency>
```

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming_2.10</artifactId>
  <version>1.6.3.2.4.2.0-90</version>
  <scope>provided</scope>
</dependency>
</dependencies>
<build>
  <defaultGoal>package</defaultGoal>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
    <resource>
      <directory>src/test/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-resources-plugin</artifactId>
      <configuration>
        <encoding>UTF-8</encoding>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>copy-resources</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <recompileMode>incremental</recompileMode>
        <args>
          <arg>-target:jvm-1.7</arg>
        </args>
        <javacArgs>
          <javacArg>-source</javacArg>
          <javacArg>1.7</javacArg>
          <javacArg>-target</javacArg>
          <javacArg>1.7</javacArg>
        </javacArgs>
      </configuration>
      <executions>
        <execution>
          <id>scala-compile</id>
          <phase>process-resources</phase>
          <goals>
            <goal>compile</goal>
          </goals>
        </execution>
        <execution>
          <id>scala-test-compile</id>
```

```

        <phase>process-test-resources</phase>
        <goals>
            <goal>testCompile</goal>
        </goals>
    </execution>
</executions>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.7</source>
        <target>1.7</target>
    </configuration>

    <executions>
        <execution>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.3</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <filters>
            <filter>
                <artifact>*:*</artifact>
                <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                </excludes>
            </filter>
        </filters>
        <finalName>uber-${project.artifactId}-${project.version}</
finalName>
    </configuration>
</plugin>

</plugins>

</build>
</project>

```


8.5. HBase Data on Spark with Connectors

Software connectors are architectural elements in the cluster that facilitate interaction between different Hadoop components. For real-time and near-real-time data analytics, there are connectors that bridge the gap between the HBase key-value store and complex relational SQL queries that Spark supports. Developers can enrich applications and interactive tools with connectors because connectors allow operations such as complex SQL queries on top of an HBase table inside Spark and table JOINS against data frames.



Important

The HDP bundle includes two different connectors that extract datasets out of HBase and streams them into Spark:

- `Hortonworks Spark-HBase Connector`
- `RDD-Based Spark-HBase Connector`: a connector from Apache HBase that uses resilient distributed datasets (RDDs)

8.5.1. Selecting a Connector

The two connectors are designed to meet the needs of different workloads. In general, use the `Hortonworks Spark-HBase Connector` for SparkSQL, `DataFrame`, and other fixed schema workloads. Use the `RDD-Based Spark-HBase Connector` for RDDs and other flexible schema workloads.

`Hortonworks Spark-HBase Connector`

When using the connector developed by Hortonworks, the underlying context is data frame, with support for optimizations such as partition pruning, predicate pushdowns, and scanning. The connector is highly optimized to push down filters into the HBase level, speeding up workload. The tradeoff is limited flexibility because you must specify your schema upfront. The connector leverages the standard Spark `DataSource` API for query optimization.

The connector is open-sourced for the community. The `Hortonworks Spark-HBase Connector` library is available as a downloadable Spark package at <https://github.com/hortonworks-spark/shc>. The repository `readme` file contains information about how to use the package with Spark applications.

For more information about the connector, see [A Year in Review](#) blog.

`RDD-Based Spark-HBase Connector`

The RDD-based connector is developed by the Apache community. The connector is designed with full flexibility in mind: you can define schema on read and therefore it is suitable for workloads where schema is undefined at ingestion time. However, the architecture has some tradeoffs when it comes to performance.

Refer to the following table for other factors that might affect your choice of connector, source repos, and code examples.

Table 8.1. Comparison of the Spark-HBase Connectors

	Hortonworks Spark-HBase Connector	RDD-Based Spark-HBase Connector
Source	Hortonworks	Apache HBase community
Apache Open Source?	Yes	Yes
Requires a Schema?	Yes: Fixed schema	No: Flexible schema
Suitable Data for Connector	SparkSQL or DataFrame	RDD
Main Repo	shc git repo	Apache hbase-spark git repo
Sample Code for Java	Not available	Apache hbase.git repo
Sample Code for Scala	shc git repo	Apache hbase.git repo
Sample Code for SparkSQL	Perform DataFrame git repo	Apache hbase.git repo

8.5.2. Using the Connector with Apache Phoenix

If you use a Spark-HBase connector in an environment that uses Apache Phoenix as a SQL skin, be aware that both connectors use only HBase .jar files by default. If you want to submit jobs on an HBase cluster with Phoenix enabled, you must include `--jars phoenix-server.jar` in your `spark-submit` command. For example:

```
./bin/spark-submit --class your.application.class \
--master yarn-client \
--num-executors 2 \
--driver-memory 512m \
--executor-memory 512m --executor-cores 1 \
--packages com.hortonworks:shc:1.0.0-1.6-s_2.10 \
--repositories http://repo.hortonworks.com/content/groups/
public/ \
--jars /usr/hdp/current/phoenix-client/phoenix-server.jar \
--files /etc/hbase/conf/hbase-site.xml /To/your/application/jar
```

8.6. Accessing HDFS Files from Spark

This subsection contains information for running Spark jobs over HDFS data.

8.6.1. Specifying Compression

To add a compression library to Spark, you can use the `--jars` option. For an example, see [Adding Libraries to Spark](#).

To save a Spark RDD to HDFS in compressed format, use code similar to the following (the example uses the GZip algorithm):

```
rdd.saveAsHadoopFile("/tmp/spark_compressed",
                    "org.apache.hadoop.mapred.TextOutputFormat",
                    compressionCodecClass="org.apache.hadoop.io.compress.
GzipCodec")
```

For more information about supported compression algorithms, see [Configuring HDFS Compression](#) in the *HDFS Administration Guide*.

8.6.2. Accessing HDFS from PySpark

When accessing an HDFS file from PySpark, you must set `HADOOP_CONF_DIR` in an environment variable, as in the following example:

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
[hrt_ga@ip-172-31-42-188 spark]$ pyspark
[hrt_ga@ip-172-31-42-188 spark]$ >>>lines = sc.textFile("hdfs://
ip-172-31-42-188.ec2.internal:8020/tmp/PySparkTest/file-01")
.....
```

If `HADOOP_CONF_DIR` is not set properly, you might see the following error:

Error from secure cluster

```
2016-08-22 00:27:06,046|t1.machine|INFO|1580|140672245782272|MainThread|
Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.
PythonRDD.collectAndServe.
2016-08-22 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|: org.
apache.hadoop.security.AccessControlException: SIMPLE authentication is not
enabled. Available:[TOKEN, KERBEROS]
2016-08-22 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|at
sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
2016-08-22 00:27:06,047|t1.machine|INFO|1580|140672245782272|
MainThread|at sun.reflect.NativeConstructorAccessorImpl.
newInstance(NativeConstructorAccessorImpl.java:57)
2016-08-22 00:27:06,048|t1.machine|INFO|1580|140672245782272|MainThread|at
{code}
```

8.7. Accessing ORC Data in Hive Tables

Apache Spark on HDP supports the Optimized Row Columnar (ORC) file format, a self-describing, type-aware, column-based file format that is one of the primary file formats supported in Apache Hive. ORC reduces I/O overhead by accessing only the columns that are required for the current query. It requires significantly fewer seek operations because all columns within a single group of row data (known as a [stripe](#)) are stored together on disk.

Spark ORC data source supports [ACID transactions](#), snapshot isolation, built-in indexes, and complex data types (such as array, map, and struct), and provides read and write access to ORC files. It leverages the Spark SQL Catalyst engine for common optimizations such as column pruning, predicate push-down, and partition pruning.

This subsection has several examples of Spark ORC integration, showing how ORC optimizations are applied to user programs.

8.7.1. Accessing ORC Files from Spark

To start using ORC, you can define a `SparkSession` instance:

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()
import spark.implicits._
```

The following example uses data structures to demonstrate working with complex types. The `Person` struct data type has a name, an age, and a sequence of contacts, which are themselves defined by names and phone numbers.

1. Define `Contact` and `Person` data structures:

```
case class Contact(name: String, phone: String)
case class Person(name: String, age: Int, contacts: Seq[Contact])
```

2. Create 100 `Person` records:

```
val records = (1 to 100).map { i =>
  Person(s"name_{$i}", i, (0 to 1).map { m => Contact(s"contact_{$m}", s"phone_{$m}") })
}
```

In the physical file, these records are saved in columnar format. When accessing ORC files through the `DataFrame` API, you see rows.

3. To write person records as ORC files to a directory named “people”, you can use the following command:

```
records.toDF().write.format("orc").save("people")
```

4. Read the objects back:

```
val people = sqlContext.read.format("orc").load("people.json")
```

5. For reuse in future operations, register the new “people” directory as temporary table “people”:

```
people.createOrReplaceTempView("people")
```

6. After you register the temporary table “people”, you can query columns from the underlying table:

```
sqlContext.sql("SELECT name FROM people WHERE age < 15").count()
```

In this example the physical table scan loads only columns **name** and **age** at runtime, without reading the **contacts** column from the file system. This improves read performance.

You can also use Spark [DataFrameReader](#) and [DataFrameWriter](#) methods to access ORC files.

8.7.2. Enabling Predicate Push-Down Optimization

The columnar nature of the ORC format helps avoid reading unnecessary columns, but it is still possible to read unnecessary rows. The example in this subsection reads all rows in which the age value is between 0 and 100, even though the query requested rows in which the age value is less than 15 ("...WHERE age < 15"). Such full table scanning is an expensive operation.

ORC avoids this type of overhead by using predicate push-down, with three levels of built-in indexes within each file: *file level*, *stripe level*, and *row level*:

- File-level and stripe-level statistics are in the file footer, making it easy to determine if the rest of the file must be read.

- Row-level indexes include column statistics for each row group and position, for finding the start of the row group.

ORC uses these indexes to move the filter operation to the data loading phase by reading only data that potentially includes required rows.

This combination of predicate push-down with columnar storage reduces disk I/O significantly, especially for larger datasets in which I/O bandwidth becomes the main bottleneck to performance.

By default, ORC predicate push-down is disabled in Spark SQL. To obtain performance benefits from predicate push-down, you must enable it explicitly, as follows:

```
spark.sql("set spark.sql.orc.filterPushdown=true")
```

8.7.3. Loading ORC Data into DataFrames by Using Predicate Push-Down

DataFrames look similar to Spark RDDs but have higher-level semantics built into their operators. This allows optimization to be pushed down to the underlying query engine.

Here is the Scala API version of the SELECT query used in the previous section, using the DataFrame API:

```
val spark = SparkSession.builder().getOrCreate()  
spark.sql("set spark.sql.orc.filterPushdown=true")  
val people = spark.read.format("orc").load("peoplePartitioned")  
people.filter(people("age") < 15).select("name").show()
```

DataFrames are not limited to Scala. There is a Java API and, for data scientists, a Python API binding:

```
from pyspark.sql import SparkSession  
spark = SparkSession.builder.getOrCreate()  
people = spark.read.format("orc").load("peoplePartitioned")  
people.filter(people.age < 15).select("name").show()
```

8.7.4. Optimizing Queries Through Partition Pruning

When predicate push-down optimization is not applicable—for example, if all stripes contain records that match the predicate condition—a query with a *WHERE* clause might need to read the entire data set. This becomes a bottleneck over a large table. Partition pruning is another optimization method; it exploits query semantics to avoid reading large amounts of data unnecessarily.

Partition pruning is possible when data within a table is split across multiple logical partitions. Each partition corresponds to a particular value of a partition column and is stored as a subdirectory within the table root directory on HDFS. Where applicable, only the required partitions (subdirectories) of a table are queried, thereby avoiding unnecessary I/O.

Spark supports saving data in a partitioned layout seamlessly, through the **partitionBy** method available during data source write operations. To partition the "people" table by the "age" column, you can use the following command:

```
people.write.format("orc").partitionBy("age").save("peoplePartitioned")
```

As a result, records are automatically partitioned by the age field and then saved into different directories: for example, `peoplePartitioned/age=1/`, `peoplePartitioned/age=2/`, and so on.

After partitioning the data, subsequent queries can omit large amounts of I/O when the partition column is referenced in predicates. For example, the following query automatically locates and loads the file under `peoplePartitioned/age=20/` and omits all others:

```
val peoplePartitioned = spark.read.format("orc").load("peoplePartitioned")
peoplePartitioned.createOrReplaceTempView("peoplePartitioned")
spark.sql("SELECT * FROM peoplePartitioned WHERE age = 20")
```

8.7.5. Enabling Vectorized Query Execution

Vectorized query execution is a feature that greatly reduces the CPU usage for typical query operations such as scans, filters, aggregates, and joins. Vectorization is also implemented for the ORC format. Spark also uses Whole Stage Codegen and this vectorization (for Parquet) since Spark 2.0 (released on July 26, 2016).

Use the following steps to implement the new ORC format and enable vectorization for ORC files with SparkSQL.

1. On the Ambari Dashboard, select **Spark2 > Configs**. For Spark shells and applications, click **Custom spark2-defaults**, then add the following properties. For the Spark Thrift Server, add these properties under **Custom spark2-thrift-sparkconf**.
 - `spark.sql.orc.enabled=true` – Enables the new ORC format to read/write Spark data source tables and files.
 - `spark.sql.hive.convertMetastoreOrc=true` – Enables the new ORC format to read/write Hive tables.
 - `spark.sql.orc.char.enabled=true` – Enables the new ORC format to use CHAR types to read Hive tables. By default, STRING types are used for performance reasons. This is an optional configuration for Hive compatibility.
2. Click **Save**, then restart Spark and any other components that require a restart.

8.7.6. Reading Hive ORC Tables

For existing Hive tables, Spark can read them without `createOrReplaceTempView`. If the table is stored as ORC format, predicate Push-down, partition pruning, and vectorized query execution are also applied according to the configuration.

```
spark.sql("SELECT * FROM hiveTable WHERE age = 20")
```

8.7.7. Additional Resources

- Apache ORC website: <https://orc.apache.org/>
- ORC performance:

- <http://hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance/>
- https://www.slideshare.net/Hadoop_Summit/performance-update-when-apache-orc-met-apache-spark-81023199
- Get Started with Spark: <http://hortonworks.com/hadoop/spark/get-started/>

8.8. Using Custom Libraries with Spark

Spark comes equipped with a selection of libraries, including Spark SQL, Spark Streaming, and MLlib.

If you want to use a custom library, such as a compression library or [Magellan](#), you can use one of the following two `spark-submit` script options:

- The `--jars` option, which transfers associated `.jar` files to the cluster. Specify a list of comma-separated `.jar` files.
- The `--packages` option, which pulls files directly from Spark packages. This approach requires an internet connection.

For example, you can use the `--jars` option to add codec files. The following example adds the LZO compression library:

```
spark-submit --driver-memory 1G \  
  --executor-memory 1G \  
  --master yarn-client \  
  --jars /usr/hdp/2.6.0.3-8/hadoop/lib/hadoop-lzo-0.6.0.2.6.0.3-8.jar \  
  test_read_write.py
```

For more information about the two options, see [Advanced Dependency Management](#) on the Apache Spark "Submitting Applications" web page.



Note

If you launch a Spark job that references a codec library without specifying where the codec resides, Spark returns an error similar to the following:

```
Caused by: java.lang.IllegalArgumentException: Compression codec  
com.hadoop.compression.lzo.LzoCodec not found.
```

To address this issue, specify the codec file with the `--jars` option in your job submit command.

9. Using Spark from R: SparkR

SparkR is an R package that provides a lightweight front end for using Apache Spark from R, supporting large-scale analytics on Hortonworks Data Platform (HDP) from the R language and environment.

SparkR provides a distributed data frame implementation that supports operations like selection, filtering, and aggregation on large datasets. In addition, SparkR supports distributed machine learning through MLlib.

This chapter lists prerequisites, followed by a SparkR example. Here are several links to additional information:

- For information about SparkR architecture and the use of SparkR in a data science workflow, see [Integrate SparkR and R for Better Data Science Workflow](#).
- For information about how to install and use R packages with SparkR, see [Using R Packages with SparkR](#).
- For additional SparkR information, see the Apache [SparkR documentation](#) for your version of Apache Spark (the link is for Spark 1, version 1.6.3).

9.1. Prerequisites

Before you run SparkR, ensure that your cluster meets the following prerequisites:

- R must be installed on all nodes. Commands for installing R are specific to the operating system. For example, for CentOS you would log on as root and run the following command:

```
yum install R
```

- JAVA_HOME must be set on all nodes.

Note: SparkR is not currently supported on SLES.

9.2. SparkR Example

The following example launches SparkR, and then uses R to create a `people` DataFrame in Spark 1.6. The example then lists part of the DataFrame, and reads the DataFrame. (For more information about Spark DataFrames, see "Using the Spark DataFrame API").

1. Launch SparkR:

```
su spark
cd /usr/hdp/2.6.0.0-598/spark/bin
./sparkR
```

Output similar to the following displays:


```
Welcome to
  ____
 /  _ \  _  _  _  _  _  _  _  _
/_  _ \/_  _ \/_  _ \/_  _ \
/_  _ \/_  _ \/_  _ \/_  _ \ version 1.6.3
/_  _ \/_  _ \/_  _ \/_  _ \

Spark context is available as sc, SQL context is available as sqlContext
>
```

2. From your R prompt (not the Spark shell), initialize `SQLContext`, create a `DataFrame`, and list the first few rows:

```
sqlContext <- sparkRSQL.init(sc)
df <- createDataFrame(sqlContext, faithful)
head(df)
```

You should see results similar to the following:

```
...
eruptions waiting
1      3.600      79
2      1.800      54
3      3.333      74
4      2.283      62
5      4.533      85
6      2.883      55
```

- ### 3. Read the `people` DataFrame:

```
people <- read.df(sqlContext, "people.json", "json")
head(people)
```

You should see results similar to the following:

	age	name
1	NA	Michael
2	30	Andy
3	19	Justin

10. Tuning Spark

When tuning Apache Spark applications, it is important to understand how Spark works and what types of resources your application requires. For example, machine learning tasks are usually CPU intensive, whereas extract, transform, load (ETL) operations are I/O intensive.

This chapter provides an overview of approaches for assessing and tuning Spark performance.

10.1. Provisioning Hardware

For general information about Spark memory use, including node distribution, local disk, memory, network, and CPU core recommendations, see the Apache Spark [Hardware Provisioning](#) document.

10.2. Checking Job Status

If a job takes longer than expected or does not finish successfully, check the following to understand more about where the job stalled or failed:

- To list running applications by ID from the command line, use `yarn application - list`.
- To see a description of a resilient distributed dataset (RDD) and its recursive dependencies (useful for understanding how jobs are executed) use `toDebugString()` on the RDD.
- To check the query plan when using the DataFrame API, use `DataFrame#explain()`.

10.3. Checking Job History

You can use the following resources to view job history:

- Spark history server UI: view information about Spark jobs that have completed.
 1. On an Ambari-managed cluster, in the Ambari Services tab, select Spark.
 2. Click Quick Links.
 3. Choose the Spark history server UI.

Ambari displays a list of jobs.

 4. Click "App ID" for job details.
- Spark history server web UI: view information about Spark jobs that have completed.

In a browser window, navigate to the history server web UI. The default host port is `<host>:18080`.

- YARN web UI: view job history and time spent in various stages of the job:

```
http://<host>:8088/proxy/<job_id>/environment/
```

```
http://<host>:8088/proxy/<app_id>/stages/
```

- `yarn logs` command: list the contents of all log files from all containers associated with the specified application.

```
yarn logs -applicationId <app_id>.
```

- Hadoop Distributed File System (HDFS) shell or API: view container log files.

For more information, see "Debugging your Application" in the Apache document [Running Spark on YARN](#).

10.4. Improving Software Performance

To improve Spark performance, assess and tune the following operations:

- Minimize shuffle operations where possible.
- Match join strategy (ShuffledHashJoin vs. BroadcastHashJoin) to the table.

This requires manual configuration.

- Consider switching from the default serializer to the Kryo serializer to improve performance.

This requires manual configuration and class registration.

- Adjust YARN memory allocation

The following subsection describes YARN memory allocation in more detail.

10.4.1. Configuring YARN Memory Allocation for Spark

This section describes how to manually configure YARN memory allocation settings based on node hardware specifications.

YARN evaluates all available compute resources on each machine in a cluster and negotiates resource requests from applications running in the cluster. YARN then provides processing capacity to each application by allocating containers. A container is the basic unit of processing capacity in YARN; it is an encapsulation of resource elements such as memory (RAM) and CPU.

In a Hadoop cluster, it is important to balance the use of RAM, CPU cores, and disks so that processing is not constrained by any one of these cluster resources.

When determining the appropriate YARN memory configurations for Spark, note the following values on each node:

- RAM (amount of memory)

- CORES (number of CPU cores)

When configuring YARN memory allocation for Spark, consider the following information:

- Driver memory does not need to be large if the job does not aggregate much data (as with a `collect()` action).
- There are tradeoffs between `num-executors` and `executor-memory`.

Large executor memory does not imply better performance, due to JVM garbage collection. Sometimes it is better to configure a larger number of small JVMs than a small number of large JVMs.

- Executor processes are not released if the job has not finished, even if they are no longer in use.

Therefore, do not overallocate executors above your estimated requirements.

In `yarn-cluster` mode, the Spark driver runs inside an application master process that is managed by YARN on the cluster. The client can stop after initiating the application. The following example shows starting a YARN client in `yarn-cluster` mode, specifying the number of executors and associated memory and core, and driver memory. The client starts the default Application Master, and SparkPi runs as a child thread of the Application Master. The client periodically polls the Application Master for status updates and displays them on the console.

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \  
  --master yarn-cluster \  
  --num-executors 3 \  
  --driver-memory 4g \  
  --executor-memory 2g \  
  --executor-cores 1 \  
  lib/spark-examples*.jar 10
```

In `yarn-client` mode, the driver runs in the client process. The application master is used only to request resources for YARN. To launch a Spark application in `yarn-client` mode, replace `yarn-cluster` with `yarn-client`. The following example launches the Spark shell in `yarn-client` mode and specifies the number of executors and associated memory:

```
./bin/spark-shell --num-executors 32 \  
  --executor-memory 24g \  
  --master yarn-client
```