

Hortonworks Data Platform

Spark Guide

(May 9, 2016)

Hortonworks Data Platform: Spark Guide

Copyright © 2012-2016 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. Introduction	1
2. Prerequisites	4
3. Installing and Configuring Spark	5
3.1. Installing and Configuring Spark Over Ambari	5
3.1.1. (Optional) Configuring Spark for Hive Access	9
3.1.2. (Optional) Installing the Spark Thrift Server After Deploying Spark	10
3.2. Configuring Dynamic Resource Allocation and Thrift Server Settings	11
3.2.1. Customizing Cluster Dynamic Resource Allocation Settings (Ambari)	12
3.2.2. Configuring Cluster Dynamic Resource Allocation Manually	13
3.2.3. Configuring a Job for Dynamic Resource Allocation	14
3.2.4. Dynamic Resource Allocation Properties	14
3.2.5. Customizing the Spark Thrift Server Port	16
3.3. (Optional) Configuring Spark for a Kerberos-Enabled Cluster	16
3.3.1. Configuring the Spark Thrift Server on a Kerberos-Enabled Cluster	17
3.3.2. Configuring Spark Streaming with Kafka in a Kerberos-Enabled Environment	18
3.4. (Optional) Configuring the Spark History Server	18
3.5. Validating the Spark Installation	18
4. Developing Spark Applications	19
4.1. Spark Pi Program	19
4.2. WordCount Program	20
5. Using the Spark DataFrame API	23
5.1. Additional DataFrame API Examples	23
5.2. Specify Schema Programmatically	24
6. Accessing ORC Files from Spark	25
6.1. Accessing ORC in Spark	25
6.2. Reading and Writing with ORC	25
6.3. Column Pruning	26
6.4. Predicate Push-down	26
6.5. Partition Pruning	26
6.6. DataFrame Support	27
6.7. Additional Resources	27
7. Using Spark SQL	29
7.1. Accessing Spark SQL Through the Spark Shell	29
7.2. Accessing Spark SQL through JDBC and ODBC	30
7.2.1. Accessing Spark SQL through JDBC	30
7.2.2. Accessing Spark SQL through ODBC	31
7.3. Forming JDBC Connection Strings for Spark SQL	31
7.4. Calling Hive User-Defined Functions	32
7.4.1. Using Custom UDFs	33
8. Using Spark Streaming	34
8.1. Using Spark Streaming with Kafka on a Kerberos-Enabled Cluster	34
8.1.1. Adding the <code>spark-kafka-streaming</code> jar File to a Deployed Build	34
8.1.2. Running Spark Streaming - Kafka Jobs on a Kerberos-Enabled Cluster	36
9. Adding Libraries to Spark	38
10. Using Spark with HDFS	39

10.1. Specifying Compression	39
10.2. Accessing HDFS from PySpark: Setting <code>HADOOP_CONF_DIR</code>	39
11. Tuning Spark	40
11.1. Provisioning Hardware	40
11.2. Checking Job Status	40
11.3. Checking Job History	40
11.4. Improving Software Performance	41
11.5. Configuring YARN Memory Allocation for Spark	41
11.6. Specifying codec Files	42
12. Appendix: Sample <code>pom.xml</code> File for Spark Streaming with Kafka	44

List of Tables

1.1. Spark - HDP Version Support	2
1.2. Spark Feature Support by Version	2
2.1. Prerequisites for Running Spark	4
3.1. Dynamic Resource Allocation Properties	14
3.2. Dynamic Resource Allocation: Optional Settings	15

1. Introduction

Hortonworks Data Platform 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 other engines such as Hive, Storm, and HBase, all running simultaneously on a single data platform. YARN allows flexibility: you can choose the right processing tool for the job. 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. In a modern data architecture with multiple processing engines using YARN and accessing data in HDFS, Spark on YARN is the leading Spark deployment mode.

Spark Features

Spark on HDP supports the following features:

- Spark on YARN
- Spark Core
- Spark SQL
- Spark SQL Thrift Server (JDBC/ODBC)
- Spark MLlib
- Spark Streaming
- Spark History Server
- DataFrame API
- Optimized Row Columnar (ORC) files
- ML Pipeline API
- Support for Hive 1.2.1
- PySpark
- Dynamic Resource Allocation

The following features and associated tools are available as technical previews:

- SparkR
- GraphX
- Apache [Zeppelin](#)
- DataSet API

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

- Spark Standalone
- Spark on Mesos
- Jupyter/iPython Notebook

Spark on YARN leverages YARN services for resource allocation, and runs Spark Executors in YARN containers. Spark on YARN 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 Spark Thrift Server both run in YARN-client mode only.

Table 1.1. Spark - HDP Version Support

HDP	Ambari	Spark
2.4.2	2.2.2	1.6.1
2.4.0	2.2.1	1.6.0
2.3.4	2.2.0	1.5.2
2.3.2	2.1.2	1.4.1
2.3.0	2.1.1	1.3.1
2.2.9	2.1.1	1.3.1
2.2.8	2.1.1	1.3.1
2.2.6	2.1.1	1.2.1
2.2.4	2.0.1	1.2.1

Table 1.2. Spark Feature Support by Version

Feature	1.2.1	1.3.1	1.4.1	1.5.2	1.6.0	1.6.1
Spark Core	Yes	Yes	Yes	Yes	Yes	Yes
Spark on YARN	Yes	Yes	Yes	Yes	Yes	Yes
Spark on YARN, Kerberos-enabled clusters	Yes	Yes	Yes	Yes	Yes	Yes
Spark History Server	Yes	Yes	Yes	Yes	Yes	Yes
Spark MLLib	Yes	Yes	Yes	Yes	Yes	Yes
Hive 13 (or later) support, including collect_list UDF		Hive version 0.13.1	Hive version 0.13.1	Hive version 1.2.1	Hive version 1.2.1	Hive version 1.2.1
ML Pipeline API			Yes	Yes	Yes	Yes
DataFrame API		TP	Yes	Yes	Yes	Yes
ORC Files		TP	Yes	Yes	Yes	Yes
PySpark		TP	Yes	Yes	Yes	Yes
Spark SQL	TP	TP	TP	Yes	Yes	Yes
Spark Thrift Server (JDBC/ODBC)		TP	TP	Yes	Yes	Yes
Spark Streaming	TP	TP	TP	Yes	Yes	Yes
Dynamic Resource Allocation		TP	TP	Yes*	Yes*	Yes*

Feature	1.2.1	1.3.1	1.4.1	1.5.2	1.6.0	1.6.1
SparkR			TP	TP	TP	TP
GraphX				TP	TP	TP
Spark HBase connector						TP

TP: Tech Preview

* Note: Dynamic Resource Allocation does not work with Spark Streaming.

2. Prerequisites

Before installing Spark, make sure your cluster meets the following prerequisites.

Table 2.1. Prerequisites for Running Spark

Prerequisite	Description
HDP Cluster Stack Version	<ul style="list-style-type: none">• 2.4.2 or later
(Optional) Ambari Version	<ul style="list-style-type: none">• 2.2.2 or later
Software dependencies	<ul style="list-style-type: none">• Spark requires HDFS and YARN• PySpark requires Python to be installed on all nodes• (Optional) The Spark Thrift Server requires Hive to be deployed on your cluster• (Optional) For optimal performance with MLlib, consider installing the netlib-java library.• SparkR (tech preview) requires R binaries to be installed on all nodes



Note

When you upgrade your cluster to HDP 2.4.2, Spark is automatically upgraded to version 1.6.1.

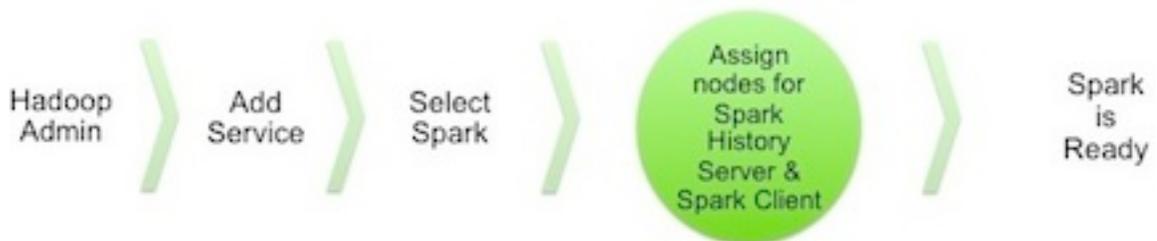
3. Installing and Configuring Spark

To install Spark manually, see [Installing and Configuring Apache Spark](#) in the *Non-Ambari Cluster Installation Guide*.

The next section in this chapter describes how to install and configure Spark on an Ambari-managed cluster, followed by configuration topics that apply to both types of clusters (Ambari-managed and not).

3.1. Installing and Configuring Spark Over Ambari

The following diagram shows the Spark installation process using Ambari. (For general information about installing HDP components using Ambari, see [Adding a Service](#) in the Ambari Documentation Suite.)



To install Spark using Ambari, complete the following steps.



Note

If you wish to install the Spark Thrift Server, you can install it during component installation (described in this subsection) or at any time after Spark has been installed and deployed. To install the Spark Thrift Server later, add the optional STS service to the specified host. For more information, see "Installing the Spark Thrift Server after Installing Spark" (later in this chapter).

Before installing the Spark Thrift Server, make sure that Hive is deployed on your cluster.

1. Choose the Ambari "Services" tab.

In the Ambari "Actions" pulldown menu, choose "Add Service." This will start the Add Service Wizard. You'll see the Choose Services screen.

Select "Spark", and click "Next" to continue.

Choose Services

Choose which services you want to install on your cluster.

<input type="checkbox"/> Service	Version	Description
<input checked="" type="checkbox"/> HDFS	2.7.1.2.4	Apache Hadoop Distributed File System
<input checked="" type="checkbox"/> YARN + MapReduce2	2.7.1.2.4	Apache Hadoop NextGen MapReduce (YARN)
<input checked="" type="checkbox"/> Tez	0.7.0.2.4	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
<input checked="" type="checkbox"/> Hive	1.2.1.2.4	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
<input type="checkbox"/> HBase	1.1.2.2.4	A Non-relational distributed database, plus Phoenix, a high performance SQL layer for low latency applications.
<input checked="" type="checkbox"/> Pig	0.15.0.2.4	Scripting platform for analyzing large datasets
<input type="checkbox"/> Sqoop	1.4.6.2.4	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases
<input type="checkbox"/> Oozie	4.2.0.2.4	System for workflow coordination and execution of Apache Hadoop jobs. This also includes the installation of the optional Oozie Web Console which relies on and will install the ExtJS Library.
<input checked="" type="checkbox"/> ZooKeeper	3.4.6.2.4	Centralized service which provides highly reliable distributed coordination
<input checked="" type="checkbox"/> Falcon	0.6.1.2.4	Data management and processing platform
<input checked="" type="checkbox"/> Storm	0.10.0.2.4	Apache Hadoop Stream processing framework
<input checked="" type="checkbox"/> Flume	1.5.2.2.4	A distributed service for collecting, aggregating, and moving large amounts of streaming data into HDFS
<input checked="" type="checkbox"/> Accumulo	1.7.0.2.4	Robust, scalable, high performance distributed key/value store.
<input checked="" type="checkbox"/> Ambari Metrics	0.1.0	A system for metrics collection that provides storage and retrieval capability for metrics collected from the cluster
<input type="checkbox"/> Atlas	0.5.0.2.4	Atlas Metadata and Governance platform
<input checked="" type="checkbox"/> Kafka	0.9.0.2.4	A high-throughput distributed messaging system
<input type="checkbox"/> Knox	0.6.0.2.4	Provides a single point of authentication and access for Apache Hadoop services in a cluster
<input type="checkbox"/> Mahout	0.9.0.2.4	Project of the Apache Software Foundation to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification
<input type="checkbox"/> Slider	0.80.0.2.4	A framework for deploying, managing and monitoring existing distributed applications on YARN.
<input checked="" type="checkbox"/> SmartSense	1.2.0.0-117	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly gather configuration, metrics, logs from common HDP services that aids to quickly troubleshoot support cases and receive cluster-specific recommendations.
<input checked="" type="checkbox"/> Spark	1.6.1.2.4	Apache Spark is a fast and general engine for large-scale data processing.

← Back

Next →

2. On the Assign Masters screen, review the node assignment for the Spark History Server. Modify the assignment if desired.

Click "Next" to continue.

Assign Masters

Assign master components to hosts you want to run them on.
 * HiveServer2 and WebHCat Server will be hosted on the same host.

NameNode:	c7001.ambari.apache.org (3.6 C)	c7001.ambari.apache.org (3.6 GB, 4 cores) NameNode ZooKeeper Server Spark History Server Kafka Broker SmartSense HST Server
SNameNode:	c7002.ambari.apache.org (3.6 C)	
History Server:	c7002.ambari.apache.org (3.6 C)	
App Timeline Server:	c7002.ambari.apache.org (3.6 C)	c7002.ambari.apache.org (3.6 GB, 4 cores) SNameNode History Server App Timeline Server ResourceManager Hive Metastore WebHCat Server HiveServer2 ZooKeeper Server DRPC Server Nimbus Storm UI Server
ResourceManager:	c7002.ambari.apache.org (3.6 C)	
Hive Metastore:	c7002.ambari.apache.org (3.6 C)	
WebHCat Server:	c7002.ambari.apache.org*	
HiveServer2:	c7002.ambari.apache.org (3.6 C)	c7003.ambari.apache.org (3.6 GB, 4 cores) ZooKeeper Server Metrics Collector
ZooKeeper Server:	c7003.ambari.apache.org (3.6 C) -	
ZooKeeper Server:	c7001.ambari.apache.org (3.6 C) -	
ZooKeeper Server:	c7002.ambari.apache.org (3.6 C) -	
DRPC Server:	c7002.ambari.apache.org (3.6 C)	
Nimbus:	c7002.ambari.apache.org (3.6 C) +	
Storm UI Server:	c7002.ambari.apache.org (3.6 C)	
Metrics Collector:	c7003.ambari.apache.org (3.6 C)	
Spark History Server:	c7001.ambari.apache.org (3.6 C)	
Kafka Broker:	c7001.ambari.apache.org (3.6 C) +	
SmartSense HST Server:	c7001.ambari.apache.org (3.6 C)	

← Back Next →

3. On the Assign Slaves and Clients screen:

- a. Specify the node(s) that will run Spark clients. These nodes will be the nodes from which Spark jobs can be submitted to YARN.
- b. (Optional) If you are installing the Spark Thrift Server at this time, review the node assignments for the Spark Thrift Server. Assign one or two nodes to the Spark Thrift Server, as needed.

Click "Next" to continue.

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 HDFS Client, MapReduce2 Client, YARN Client, Tez Client, HCat Client, Hive Client, Pig, ZooKeeper Client and Spark Client.

	all none	all none	all none	all none	all none	all none	all none
.apache.org*	<input checked="" type="checkbox"/> DataNode	<input type="checkbox"/> NFSGateway	<input checked="" type="checkbox"/> NodeManager	<input checked="" type="checkbox"/> Supervisor	<input checked="" type="checkbox"/> Flume	<input type="checkbox"/> Spark Thrift Server	<input type="checkbox"/> Client
.apache.org*	<input checked="" type="checkbox"/> DataNode	<input type="checkbox"/> NFSGateway	<input checked="" type="checkbox"/> NodeManager	<input checked="" type="checkbox"/> Supervisor	<input checked="" type="checkbox"/> Flume	<input checked="" type="checkbox"/> Spark Thrift Server	<input type="checkbox"/> Client
.apache.org*	<input checked="" type="checkbox"/> DataNode	<input type="checkbox"/> NFSGateway	<input checked="" type="checkbox"/> NodeManager	<input checked="" type="checkbox"/> Supervisor	<input checked="" type="checkbox"/> Flume	<input type="checkbox"/> Spark Thrift Server	<input checked="" type="checkbox"/> Client

Show: 25 1 - 3 of 3

← Back
Next →

4. (Optional) On the "Customize Services" screen: If you are installing the Spark Thrift Server at this time, choose the "Spark" tab and 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.

Aside from the YARN queue setting, we recommend that you use default values for your initial configuration. For additional information about configuring property values, see [Customizing Dynamic Resource Allocation and Spark Thrift Server Settings](#)

Click "Next" to continue.

Customize Services

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

HDFS
MapReduce2
YARN
Tez
Hive
HBase
Pig
ZooKeeper
Storm
Flume
Kafka
Spark
Misc

Group
Default (1) ▼
Manage Config Groups
Filter... ▼

5. Ambari will display the Review screen.



Important

On the Review screen, make sure all HDP components correspond to HDP version 2.4.2 or later.

Click "Deploy" to continue.

6. Ambari will display the Install, Start and Test screen. The status bar and messages will indicate progress.

Install, Start and Test

Please wait while the selected services are installed and started.

3 % overall

Host	Status	Message
c7001.ambari.apache.org	<div style="width: 3%; height: 10px; background-color: #0070c0; border: 1px solid #ccc; display: inline-block;"></div> 3%	Waiting to install DataNode
c7002.ambari.apache.org	<div style="width: 3%; height: 10px; background-color: #0070c0; border: 1px solid #ccc; display: inline-block;"></div> 3%	Waiting to install DataNode
c7003.ambari.apache.org	<div style="width: 3%; height: 10px; background-color: #0070c0; border: 1px solid #ccc; display: inline-block;"></div> 3%	Waiting to install DataNode

3 of 3 hosts showing - [Show All](#)
Show: 1 - 3 of 3 ⏪ ⏩ ⏴ ⏵

Next →

7. When finished, Ambari will present a summary of results. Click "Complete" to finish installing Spark.



Caution

Ambari will create and edit several configuration files. Do not edit these files directly if you configure and manage your cluster using Ambari.

3.1.1. (Optional) Configuring Spark for Hive Access

When you install Spark using Ambari, the `hive-site.xml` file is 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 `hostname` points to the URI where the Hive Metastore is running.



Important

`hive-site.xml` contains a number of properties that are not relevant to or supported by the Spark thrift server. Ensure that your Spark `hive-site.xml` file contains only the following configuration 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>
```

3.1.2. (Optional) Installing the Spark Thrift Server After Deploying Spark

The Spark Thrift Server can be installed during Spark installation or after Spark is deployed.

To install the Spark Thrift Server after deploying Spark, add the service to the specified host:

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](#)

NFSGateway

Spark Thrift Server

Started

Started

Started

Started

Started

Started

Started

Started

2. Ambari will ask you to confirm the selection:



3. The installation process will run in the background until it completes:

0 Background Operations Running			
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

3.2. Configuring Dynamic Resource Allocation and Thrift Server Settings

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

- On an Ambari-managed cluster, the Spark Thrift Server uses dynamic resource allocation by default. The Thrift Server will increase or decrease 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 will use resources from the YARN cluster.)

- 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.
- You can customize dynamic resource allocation settings on a per-job basis. Job settings will override cluster configuration settings.

Cluster configuration is the default unless overridden by job configuration.

The next three 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.

3.2.1. Customizing Cluster Dynamic Resource Allocation Settings (Ambari)

On an Ambari-managed cluster, dynamic resource allocation is configured for the Spark Thrift Server as part of the Spark installation process. Ambari starts the shuffle service for use by the Thrift Server and general Spark jobs.

To view or modify property values for the Spark Thrift Server, navigate to **Services > Spark**. Required settings are listed in the "Advanced spark-thrift-sparkconf" group; additional properties can be specified in the custom section. (For a complete list of DRA properties, see [Dynamic Resource Allocation Properties](#).)

Advanced spark-thrift-sparkconf

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

To enable and configure dynamic resource allocation for general Spark applications, navigate to **Services > Spark**. Review the list of properties in the Advanced spark-defaults group, and revise settings as needed.

Dynamic resource allocation requires an external shuffle service on each worker node. If you installed your cluster using Ambari, the service will be started automatically; no further steps are needed.

3.2.2. Configuring Cluster Dynamic Resource Allocation Manually

To configure a cluster to run Spark applications with dynamic resource allocation:

1. Add the following properties to the `spark-defaults.conf` file associated with your Spark installation. (For general Spark applications, this file typically resides at `$SPARK_HOME/conf/spark-defaults.conf`.)

- Set `spark.dynamicAllocation.enabled` to `true`
- Set `spark.shuffle.service.enabled` to `true`

(Optional) The following properties specify a starting point and range for the number of executors. Note that `initialExecutors` must be greater than or equal to `minExecutors`, and less than or equal to `maxExecutors`.

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

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

2. Start the shuffle service on each worker node in the cluster. (The shuffle service runs as an auxiliary service of the NodeManager.)
 - a. In the `yarn-site.xml` file on each node, add `spark_shuffle` to `yarn.nodemanager.aux-services`, 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.

3.2.3. 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 will load the default `spark-defaults.conf` file first, and then apply 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 will use the specified `properties-file`, without reading the default property file.

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

3.2.4. Dynamic Resource Allocation Properties

See the following tables for more information about dynamic resource allocation properties.

Table 3.1. Dynamic Resource Allocation Properties

Property Name	Value	Meaning
<code>spark.dynamicAllocation.enabled</code>	<code>true</code>	Whether to use dynamic resource allocation, which scales the number

Property Name	Value	Meaning
		<p>of executors registered with this application up and down based on the workload. Note that this is currently only available in YARN mode. For more information, see the Apache Dynamic Resource Allocation documentation.</p> <p>DRA requires <code>spark.shuffle.service.enabled</code> to be set. The following configurations are also relevant: <code>spark.dynamicAllocation.minExecutors</code>, <code>spark.dynamicAllocation.maxExecutors</code>, and <code>spark.dynamicAllocation.initialExecutors</code></p>
<code>spark.shuffle.service.enabled</code>	true	<p>Enables the external shuffle service, which preserves shuffle files written by executors so that the executors can be safely removed.</p> <p>This property must be set to true if <code>spark.dynamicAllocation.enabled</code> is true.</p> <p>The external shuffle service must be set up before enabling the property. For more information, see "Starting the Shuffle Service" at the end of this section.</p>
<code>spark.dynamicAllocation.initialExecutors</code>	default is <code>spark.dynamicAllocation.minExecutors</code>	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	Upper bound for the number of executors if dynamic resource allocation is enabled.
<code>spark.dynamicAllocation.minExecutors</code>	Default is 0	Lower bound for the number of executors if dynamic resource allocation is enabled.

Optional Settings: The following table lists several advanced settings for dynamic resource allocation.

Table 3.2. Dynamic Resource Allocation: Optional Settings

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 duration, the executor will be 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 duration, the executor will be 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

Property Name	Value	Meaning
		this duration, new executors will be requested.
<code>spark.dynamicAllocation.sustainedSchedulerBacklogTimeout</code>	Default is <code>schedulerBacklogTimeout</code>	Same as <code>spark.dynamicAllocation.schedulerBacklogTimeout</code> , but used only for subsequent executor requests.



Note

Dynamic resource allocation is enabled by default for the Spark Thrift Server.

3.2.5. Customizing the Spark Thrift Server Port

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

3.3. (Optional) Configuring Spark for a Kerberos-Enabled Cluster

Spark jobs are submitted to a Hadoop cluster as YARN jobs.

When a job is ready to run in a production environment, there are a few additional steps if the cluster is Kerberized:

- The Spark History Server daemon needs a Kerberos account and keytab to run in a Kerberized cluster.
- When you enable Kerberos for a Hadoop cluster with Ambari, Ambari configures Kerberos for the Spark History Server and automatically creates a Kerberos account and keytab for it. For more information, see [Enabling Kerberos Authentication Using Ambari](#).
- If you are not using Ambari, or if you plan to enable Kerberos manually for the Spark History Server, see [Creating Service Principals and Keytab Files for HDP](#) in the Hadoop Security Guide.
- To submit Spark jobs in a Kerberized cluster, the account (or person) submitting jobs needs a Kerberos account & keytab.
- When access is authenticated without human interaction – as happens for processes that submit job requests – the process would use a headless keytab. Security risk is mitigated by ensuring that only the service who should be using the headless keytab has the permissions to read it.
- An end user should use their own keytab when submitting a Spark job.

Setting Up Principals and Keytabs for End User Access to Spark

In the following example, user `$USERNAME` 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
```

Setting Up Service Principals and Keytabs for Processes Submitting Spark Jobs

The following example shows the creation and use of 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. Create a spark user and add it to the `hadoop` group. (Do this for every node of your cluster.)

```
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: make sure user `spark` is the only user with access to the keytab:

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

In the following steps, 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
```

3.3.1. Configuring the Spark Thrift Server on a Kerberos-Enabled Cluster

If you are installing the Spark Thrift Server on a Kerberos-secured 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.

- Edit permissions in `/var/run/spark` and `/var/log/spark` to specify read/write permissions to the Hive service account.
- Use the Hive service account to start the `thriftserver` process.



Note

We recommend that you run the Spark Thrift Server as user `hive` instead of user `spark` (this supersedes recommendations in previous releases). This ensures that the Spark Thrift Server can access Hive keytabs, the Hive metastore, and data in HDFS that is stored under user `hive`.



Important

When the Spark Thrift Server runs queries as user `hive`, all data accessible to user `hive` will be accessible to the user submitting the query. For a more secure configuration, use a different service account for the Spark Thrift Server. Provide appropriate access to the Hive keytabs and the Hive metastore.

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 (via `kinit`).

3.3.2. Configuring Spark Streaming with Kafka in a Kerberos-Enabled Environment

For information about using Spark Streaming with Kafka in a Kerberos-enabled environment, see [Using Spark Streaming](#).

3.4. (Optional) Configuring the Spark History Server

The Spark History Server is a monitoring tool that lists information about completed Spark applications. Applications write history data to a directory on HDFS (by default). The History Server pulls the data from HDFS and presents it in a Web UI at `<host>:18080` (by default).

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

For Kerberos considerations, see [Configuring Spark for a Kerberos-Enabled Cluster](#)

3.5. Validating the Spark Installation

To validate the Spark installation, run the following Spark jobs:

- [Spark Pi example](#)
- [WordCount example](#)

4. Developing Spark Applications

Apache Spark is designed for fast application development and fast 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.

To run Spark applications, use the `spark-submit` script in the Spark `bin` directory to launch applications on a cluster. Alternately, to use the API interactively you can launch an interactive shell for Scala (`spark-shell`), Python (`pyspark`), or SparkR. Note: Each interactive shell automatically creates `SparkContext` in a variable called `sc`.

For more information about getting started with Spark, see the Apache Spark [Quick Start](#). For more extensive information about application development, see the Apache [Spark Programming Guide](#) and [Submitting Applications](#).

The remainder of this chapter contains basic coding examples. Subsequent chapters describe how to access a range of data sources and analytic capabilities.

4.1. Spark Pi Program

To test compute-intensive tasks in Spark, the Pi example calculates pi by “throwing darts” at a circle — it 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.

Here is [Python code](#) for the Spark Pi program included with Spark.

To run the Spark Pi example:

1. Log on as a user with 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:

- `--class`: The entry point for your application (e.g., `org.apache.spark.examples.SparkPi`)
- `--master`: The master URL for the cluster (e.g., `spark://23.195.26.187:7077`)
- `--deploy-mode`: Whether to deploy your driver on the worker nodes (`cluster`) or locally as an external client (`client`) (default: `client`)

- `--conf`: Arbitrary Spark configuration property in `key=value` format. For values that contain spaces wrap `"key=value"` in quotes (as shown).
- `<application-jar>`: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an `hdfs:// path` or a `file:// path` that is present on all nodes.
- `<application-arguments>`: Arguments passed to the main method of your main class, if any.

The job should complete without errors.

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

```
16/04/10 14:28:35 INFO scheduler.DAGScheduler: Job 0 finished: reduce at
SparkPi.scala:36, took 1.721177 s
Pi is roughly 3.141296
16/04/10 14:28:35 INFO spark.ContextCleaner: Cleaned accumulator 1
```

To view job status in a browser, navigate to the YARN ResourceManager Web UI and view Job History Server information. (For more information about checking job status and history, see [Tuning and Troubleshooting Spark](#).)

4.2. WordCount Program

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:

```
16/04/06 19:33:26 INFO SecurityManager: Changing view acls to: spark
Welcome to
```


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

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

5. Using the Spark DataFrame API

The Spark DataFrame API provides table-like access to data from a variety of sources. Its purpose is similar to Python's `pandas` library and R's data frames: collect and organize data into a tabular format with named columns. DataFrames can be constructed from a wide array of sources, including structured data files, Hive tables, and existing Spark RDDs.

1. As user `spark`, upload the `people.txt` file to 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
16/04/10 11:24:10 INFO YarnScheduler: Removed TaskSet 2.0, whose tasks have
all completed, from pool

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

5.1. Additional DataFrame API Examples

Here are additional examples of Scala-based DataFrame access, using 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
df.groupBy("age").count().show()
```

5.2. Specify Schema Programmatically

The following example uses the DataFrame API to specify a schema for `people.txt`, and retrieve 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 will produce output similar to the following:

```
16/04/10 14:36:49 INFO cluster.YarnScheduler: Removed TaskSet 13.0, whose
  tasks have all completed, from pool
16/04/10 14:36:49 INFO scheduler.DAGScheduler: ResultStage 13 (collect at :33)
  finished in 0.129 s
16/04/10 14:36:49 INFO scheduler.DAGScheduler: Job 10 finished: collect
  at :33, took 0.162827 s
Name: Michael
Name: Andy
Name: Justin
```

6. Accessing ORC Files from Spark

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. The columnar format lets the reader read, decompress, and process only the columns that are required for the current query. ORC support in Spark SQL and DataFrame APIs provides fast access to ORC data contained in Hive tables. It supports ACID transactions, snapshot isolation, built-in indexes, and complex types.

6.1. Accessing ORC in Spark

Spark's ORC data source supports complex data types (such as array, map, and struct), and provides read and write access to ORC files. It leverages Spark SQL's Catalyst engine for common optimizations such as column pruning, predicate push-down, and partition pruning.

This chapter has several examples of Spark's ORC integration, showing how such optimizations are applied to user programs.

To start using ORC, define a HiveContext instance:

```
import org.apache.spark.sql._
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

The following examples use a few data structures to demonstrate working with complex types. The Person struct has name, age, and a sequence of Contacts, which are themselves defined by names and phone numbers. Define these structures as follows:

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

Next, create 100 records. In the physical file these records will be saved in columnar format, but users will see rows when accessing ORC files via the DataFrame API. Each row represents one Person record.

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

6.2. Reading and Writing with ORC

Spark's **DataFrameReader** and **DataFrameWriter** are used to access ORC files, in a similar manner to other data sources.

To write People objects as ORC files to directory "people", use the following command:

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

Read the objects back as follows:

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

For reuse in future operations, register it as temporary table "people":

```
people.registerTempTable("people")
```

6.3. Column Pruning

The previous step registered the table as a temporary table named “people”. The following SQL query references two columns from the underlying table.

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

At runtime, the physical table scan will only load columns **name** and **age**, without reading the **contacts** column from the file system. This improves read performance.

ORC reduces I/O overhead by only touching required columns. It requires significantly fewer seek operations because all columns within a single stripe are stored together on disk.

6.4. Predicate Push-down

The columnar nature of the ORC format helps avoid reading unnecessary columns, but it is still possible to read unnecessary rows. In our example, we read all rows where age was between 0 and 100, even though we requested rows where age was less than 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 and stripe level statistics are in the file footer, making it easy to determine if the rest of the file needs to be read.
- Row level indexes include column statistics for each row group and position, for seeking to the start of the row group.

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

This combination of indexed data and columnar storage reduces disk I/O significantly, especially for larger datasets where I/O bandwidth becomes the main bottleneck for performance.



Important

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:

```
sqlContext.setConf("spark.sql.orc.filterPushdown", "true")
```

6.5. Partition Pruning

When predicate pushdown 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(s) of partition column(s), and is stored as a sub-directory within the table's 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 out in a partitioned layout seamlessly, through the `partitionBy` method available during data source writes. To partition the people table by the "age" column, use the following command:

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

Records will be automatically partitioned by the age field, and then saved into different directories; for example, `peoplePartitioned/age=1/`, `peoplePartitioned/age=2/`, etc.

After partitioning the data, subsequent queries will be able to skip large amounts of I/O when the partition column is referenced in predicates. For example, the following query will automatically locate and load the file under `peoplePartitioned/age=20/`; it will skip all others.

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

6.6. DataFrame Support

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. ORC data can be loaded into DataFrames.

Here is the Scala API translation of the preceding SELECT query, using the DataFrame API:

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)  
sqlContext.setConf("spark.sql.orc.filterPushdown", "true")  
val people = sqlContext.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:

```
sqlContext = HiveContext(sc)  
sqlContext.setConf("spark.sql.orc.filterPushdown", "true")  
people = sqlContext.read.format("orc").load("peoplePartitioned")  
people.filter(people.age < 15).select("name").show()
```

6.7. Additional Resources

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

- ORC performance: <http://hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance/>
- Get Started with Spark: <http://hortonworks.com/hadoop/spark/get-started/>

7. Using Spark SQL

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

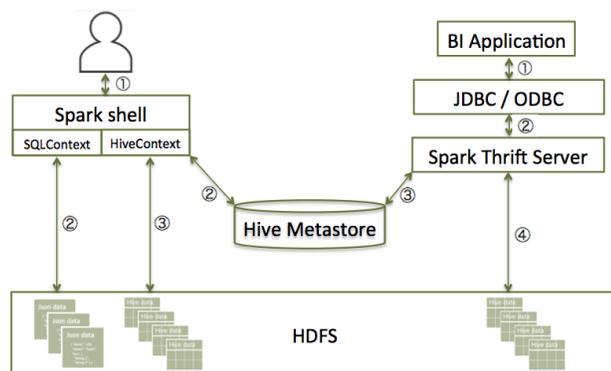
Spark SQL can also read data by interacting with the Hive MetaStore, when HiveContext is used. If you already use Hive, you should use HiveContext; it supports all Hive data formats and user-defined functions (UDFs), and allows full access to the HiveQL parser. HiveContext extends SQLContext, so HiveContext 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, via the Simba ODBC driver.

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

The following diagram outlines the access process, depending on type of interaction:



7.1. Accessing Spark SQL Through the Spark Shell

Here is a sample command that 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 filesystem, 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. (HiveContext extends SQLContext.) For an example of the use of HiveContext (instantiated as `val sqlContext`), see [Accessing ORC Files from Spark](#).

7.2. Accessing Spark SQL through JDBC and ODBC

With the use of the Spark Thrift Server, remote access to Spark SQL is possible over JDBC or ODBC. With this approach you can access data through JDBC (via the JDBC [Beeline](#) client) or ODBC (via the Simba driver).

The following considerations and prerequisites apply to JDBC and ODBC access.

Considerations:

- 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.
- When using JDBC or ODBC to access Spark SQL in a production environment, note that the Spark Thrift Server does not currently support the `doAs` authorization property, which propagates user identity. Workaround: use programmatic APIs or `spark-shell`, submitting the job under your identity.
- All client requests coming to Spark Thrift Server share a SparkContext.

Prerequisites:

- 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.
- If `SPARK_HOME` is not already defined, set it to your Spark directory. For example:

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

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

To manually stop the Spark Thrift Server:

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

7.2.1. Accessing Spark SQL through JDBC

1. Connect to the Thrift Server over the Beeline JDBC client.
 - a. Launch Beeline from `SPARK_HOME`:

```
su spark
```

```
./bin/beeline
```

- b. On the Beeline prompt, connect to the Spark SQL Thrift Server:

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

The host port must match the host port where 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 1.6.1)
Driver: Spark Project Core (version 1.6.1.2.4.0.0-169)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10015>
```

2. Once 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>
```

7.2.2. Accessing Spark SQL through ODBC

For documentation and driver download links, see "Hortonworks ODBC Driver for SparkSQL (1.5 & 1.6)" in [Hortonworks Data Platform Add-Ons](#).

7.3. Forming JDBC Connection Strings for Spark SQL

JDBC URL connection strings 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.

JDBC Parameter	Description
sessionConfs	Optional configuration parameters for the JDBC/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 last 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, including transport and security settings, see [Hive JDBC and ODBC Drivers](#) in the HDP Data Services Guide.

Accessing Spark via JDBC on a Kerberos-enabled Cluster

The following connection URL accesses Spark SQL via 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 URL accesses Spark SQL via 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
```

7.4. Calling Hive User-Defined Functions

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

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 would run 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:

```
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)
```

7.4.1. Using Custom UDFs

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

When using a custom UDF, make sure 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. Using Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant processing of real-time data streams. Data can be ingested from sources such as Kafka and Flume, and can be processed using complex algorithms expressed with high-level functions like `map`, `reduce`, `join`, and `window`. Processed data can be sent to file systems, databases, and live dashboards.



Important

Dynamic Resource Allocation does not work with Spark Streaming.

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

For additional examples, see the Apache GitHub example repositories for [Scala](#), [Java](#), and [Python](#).

8.1. Using Spark Streaming with Kafka on a Kerberos-Enabled Cluster

This section describes specific steps for developers using Spark Streaming with Kafka on a Kerberos-enabled cluster.

8.1.1. Adding the `spark-kafka-streaming` jar File to a Deployed Build

Before running Spark streaming jobs with Kafka in a Kerberos environment, you will need to add or retrieve the HDP `spark-streaming-kafka` jar file and associated jar files.



Note

The `spark-streaming-kafka` jar file is required for running a job that is not a Spark example job provided with HDP. If you are running a job that is part of the Spark examples package installed by HDP, you will not need the `spark-streaming-kafka` jar.

Instructions for Developing and Building Applications

If you are using `maven` as a compile tool:

1. Add the `hortonworks` repo 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>1.6.1.2.4.2.0-90</version>
</dependency>

<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming_2.10</artifactId>
  <version>1.6.1.2.4.2.0-90</version>
  <scope>provided</scope>
</dependency>
```

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

3. (Optional) The default scope of the `spark-streaming` jar is "provided", which means that the jar is provided by the environment; it will not pack into an uber jar. (An uber jar packages all dependencies into one jar.) If you prefer to pack a uber jar, 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 Spark Streaming Job

Instructions for submitting your job depend on whether you used an uber jar 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_2.10:1.6.1.2.4.2.0-90 \  
--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 into an uber jar, submit the jar like 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](#).

8.1.2. Running Spark Streaming - Kafka Jobs on a Kerberos-Enabled Cluster

The following instructions assume that Spark and Kafka are already deployed on a Kerberos-enabled cluster.

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 jaas configuration file (for example, `key.conf`), and add configuration settings to specify the user keytab.

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



Note

The keytab and configuration files will be distributed using YARN local resources. They will end up in the current directory of the Spark YARN container, thus the location should be specified as `./v.keytab`.

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

4. In your job submission instructions, pass the jaas configuration file and keytab as local resource files. Add the jaas configuration file options to the JVM options specified for the driver and executor:



Note

If you are running a job that is part of the Spark `examples` package installed by HDP, you do not need to add the `spark-streaming-kafka` jar. Otherwise, add the `spark-streaming-kafka` jar using the `--jars` command-line option.

```
--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"
```

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
```

9. Adding Libraries to Spark

To use a custom library with a Spark application (a library that is not available in Spark by default, such as a compression library or [Magellan](#)), use one of the following two `spark-submit` script options:

- The `--jars` option transfers associated jar files to the cluster.
- The `--packages` option pulls directly from Spark packages. This approach requires an internet connection.

For example, to add the LZO compression library to Spark using the `--jars` option:

```
spark-submit --driver-memory 1G --executor-memory 1G --master yarn-client
--jars /usr/hdp/2.3.0.0-2557/hadoop/lib/hadoop-lzo-0.6.0.2.3.0.0-2557.jar
test_read_write.py
```

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

10. Using Spark with HDFS

10.1. Specifying Compression

To add a compression library to Spark, use the `--jars` option. As mentioned in "Adding Libraries to Spark," the following example adds the LZO compression library:

```
spark-submit --driver-memory 1G --executor-memory 1G --master yarn-client
--jars /usr/hdp/2.3.0.0-2557/hadoop/lib/hadoop-lzo-0.6.0.2.3.0.0-2557.jar
test_read_write.py
```

To specify compression in Spark-shell when writing to HDFS, use code similar to:

```
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 Reference Guide.

10.2. Accessing HDFS from PySpark: Setting HADOOP_CONF_DIR

If PySpark is accessing an HDFS file, `HADOOP_CONF_DIR` needs to be set in an environment variable. For example:

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
[hrt_qa@ip-172-31-42-188 spark]$ pyspark
[hrt_qa@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

```
2016-04-10 00:27:06,046|t1.machine|INFO|1580|140672245782272|MainThread|
Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.
PythonRDD.collectAndServe.
2016-04-10 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|: org.
apache.hadoop.security.AccessControlException: SIMPLE authentication is not
enabled. Available:[TOKEN, KERBEROS]
2016-04-10 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|at
sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
2016-04-10 00:27:06,047|t1.machine|INFO|1580|140672245782272|
MainThread|at sun.reflect.NativeConstructorAccessorImpl.
newInstance(NativeConstructorAccessorImpl.java:57)
2016-04-10 00:27:06,048|t1.machine|INFO|1580|140672245782272|MainThread|at
{code}
```

11. Tuning Spark

When tuning 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.

11.1. 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.

11.2. 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 an RDD and its recursive dependencies, use `toDebugString()` on the RDD. This is useful for understanding how jobs will be executed.
- To check the query plan when using the DataFrame API, use `DataFrame#explain()`.

11.3. 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.
 - On an Ambari-managed cluster, in the Ambari Services tab, select Spark.
 - Click Quick Links.
 - Choose the Spark History Server UI. Ambari displays a list of jobs.
 - Click "App ID" for job details.
- Spark History Server Web UI: view information about 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](#).

11.4. 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.

11.5. Configuring YARN Memory Allocation for Spark

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

YARN takes into account all of the available compute resources on each machine in the 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 usage 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)

Configuring Spark for `yarn-cluster` Deployment Mode

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 command starts a YARN client in `yarn-cluster` mode. The client will start the default Application Master. SparkPi will run as a child thread of the Application Master. The client will periodically poll the Application Master for status updates, which will be displayed in the console. The client will exist when the application stops running.

```
./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
```

Configuring Spark for `yarn-client` Deployment Mode

In `yarn-client` mode, the driver runs in the client process. The application master is only used to request resources for YARN.

To launch a Spark application in `yarn-client` mode, replace `yarn-cluster` with `yarn-client`. For example:

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

Considerations

When configuring Spark on YARN, consider the following information:

- Executor processes will be not released if the job has not finished, even if they are no longer in use. Therefore, please do not overallocate executors above your estimated requirements.
- 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.

11.6. Specifying codec Files

If you try to use a codec library without specifying where the codec resides, you will see an error.

For example, if the `hadoop-lzo` codec file cannot be found during `spark-submit`, Spark will generate the following message:

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

SOLUTION: Specify the `hadoop-lzo` jar file with the `--jars` option in your job submit command.

For example:

```
spark-submit --driver-memory 1G --executor-memory 1G --master  
yarn-client --jars /usr/hdp/2.4.2.0-258/hadoop/lib/hadoop-  
lzo-0.6.0.2.4.2.0-258.jar test_read_write.py
```

For more information about the `--jar` option, see [Adding Libraries to Spark](#).

12. Appendix: 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>1.6.1.2.4.2.0-90</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-streaming_2.10</artifactId>
      <version>1.6.1.2.4.2.0-90</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>
```