Hortonworks Data Platform

Spark Guide

(Mar 1, 2016)

docs.hortonworks.com

Hortonworks Data Platform: Spark Guide

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.

Table of Contents

List of Tables

1. Introduction

Hortonworks Data Platform supports Apache Spark 1.6, 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 Core
- Spark on YARN
- Spark on YARN on Kerberos-enabled clusters
- Spark History Server
- Spark MLLib
- DataFrame API
- Optimized Row Columnar (ORC) files
- Spark SQL
- Spark SQL Thrift Server (JDBC/ODBC)
- Spark Streaming
- Support for Hive 1.2.1
- ML Pipeline API
- PySpark
- Dynamic Resource Allocation

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

- SparkR
- GraphX

- Zeppelin (link)

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

- Spark Standalone
- Spark on Mesos
- Jupyter/iPython Notebook
- Oozie Spark action is not supported, but there is a tech note available for HDP customers

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 runs in YARN-Client mode only.

Table 1.1. Spark - HDP Version Support

HDP	Ambari	Spark
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
Spark Core	Yes	Yes	Yes	Yes	Yes
Spark on YARN	Yes	Yes	Yes	Yes	Yes
Spark on YARN, Kerberos-enabled clusters	Yes	Yes	Yes	Yes	Yes
Spark History Server	Yes	Yes	Yes	Yes	Yes
Spark MLLib	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
ML Pipeline API			Yes	Yes	Yes
DataFrame API		TP	Yes	Yes	Yes
ORC Files		TP	Yes	Yes	Yes
PySpark		TP	Yes	Yes	Yes
Spark SQL	TP	TP	TP	Yes	Yes
Spark Thrift Server (JDBC/ODBC)		TP	TP	Yes	Yes

Feature	1.2.1	1.3.1	1.4.1	1.5.2	1.6.0
Spark Streaming	TP	TP	TP	Yes	Yes
Dynamic Resource Allocation		TP	TP	TP	Yes*
SparkR			TP	TP	TP
GraphX				TP	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	• 2.4.0 or later
(Optional) Ambari Version	• 2.2.1 or later
Software dependencies	• 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 to be installed on all nodes

Note

When you upgrade to HDP 2.4.0, Spark is automatically upgraded to 1.6.

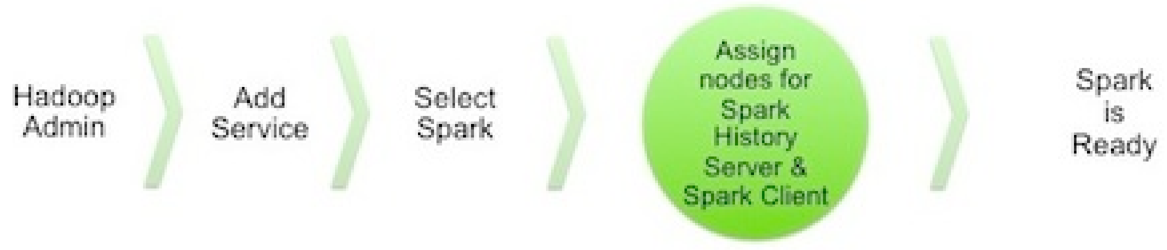
3. Installing and Configuring Spark

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

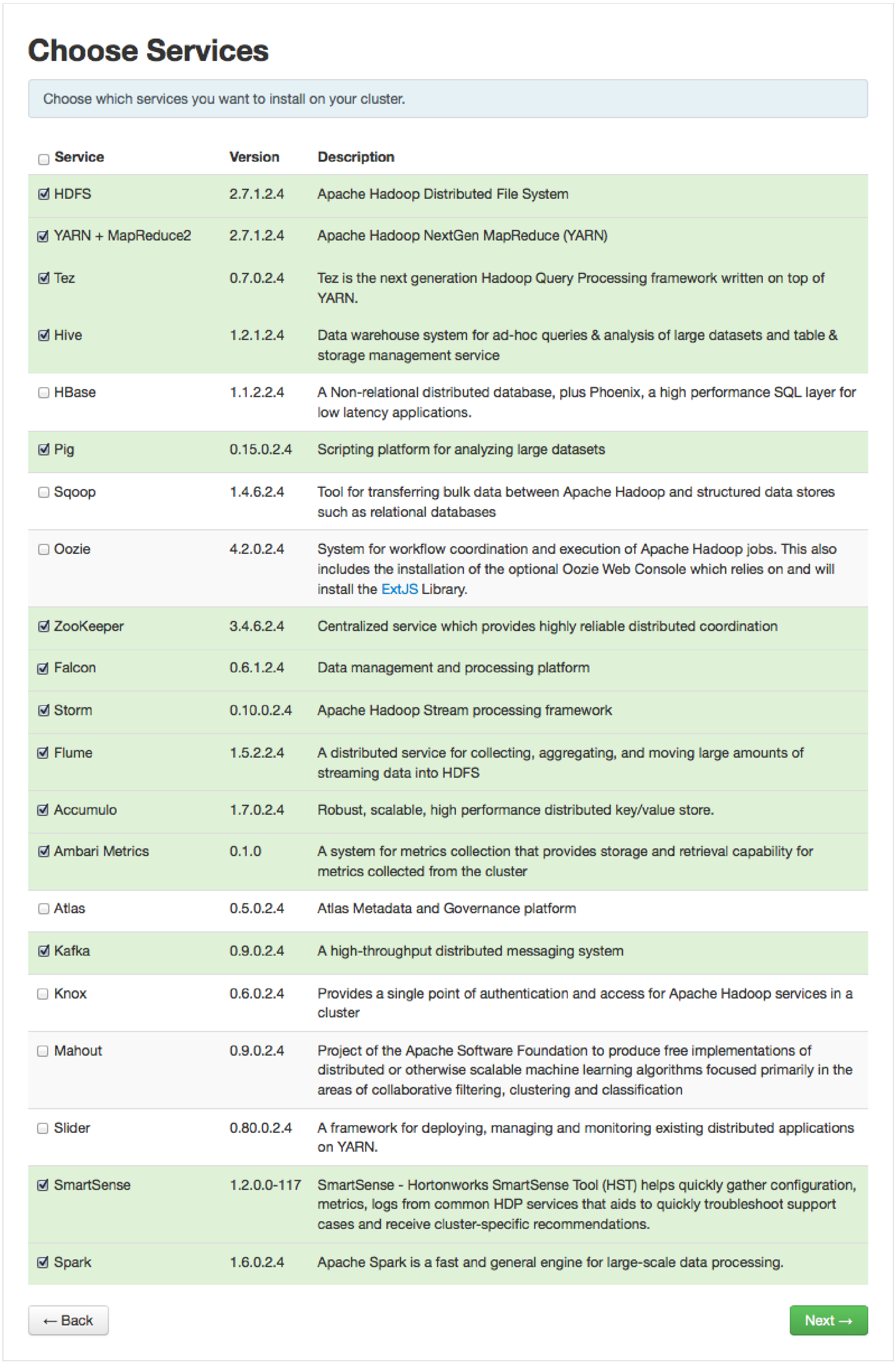
If you wish to install the Spark Thrift Server, you can install it during component installation (described in this subsection) or at any time after Spark has been installed and deployed. To install the Spark Thrift Server later, add the optional STS service to the specified host. For more information,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



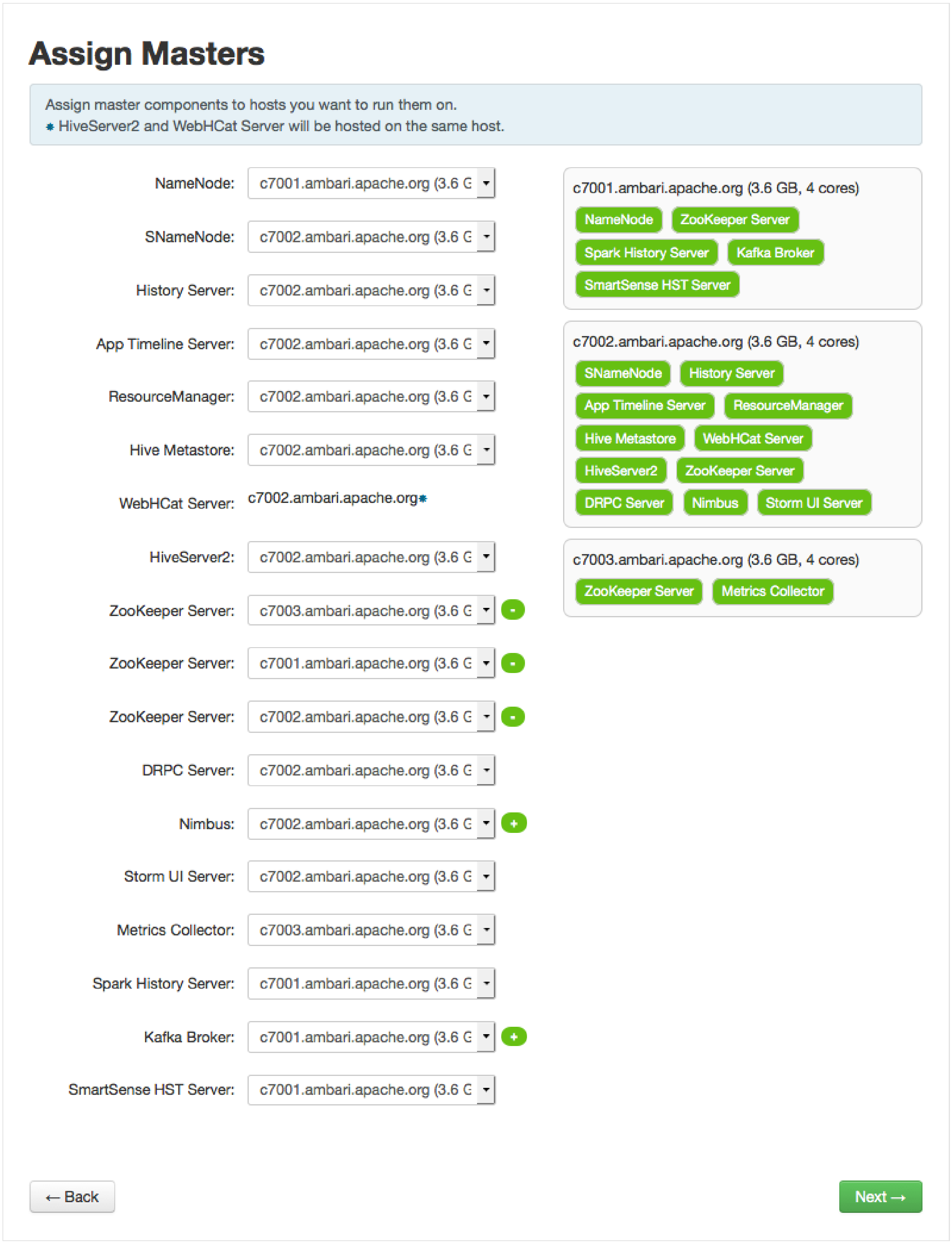
Choose Services

Choose which services you want to install on your cluster.

☐ Service	Version	Description
☑ HDFS	2.7.1.2.4	Apache Hadoop Distributed File System
☑ YARN + MapReduce2	2.7.1.2.4	Apache Hadoop NextGen MapReduce (YARN)
☑ Tez	0.7.0.2.4	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
☑ Hive	1.2.1.2.4	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
☐ HBase	1.1.2.2.4	A Non-relational distributed database, plus Phoenix, a high performance SQL layer for low latency applications.
☑ Pig	0.15.0.2.4	Scripting platform for analyzing large datasets
☐ Sqoop	1.4.6.2.4	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases
☐ 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.
☑ ZooKeeper	3.4.6.2.4	Centralized service which provides highly reliable distributed coordination
☑ Falcon	0.6.1.2.4	Data management and processing platform
☑ Storm	0.10.0.2.4	Apache Hadoop Stream processing framework
☑ Flume	1.5.2.2.4	A distributed service for collecting, aggregating, and moving large amounts of streaming data into HDFS
☑ Accumulo	1.7.0.2.4	Robust, scalable, high performance distributed key/value store.
☑ Ambari Metrics	0.1.0	A system for metrics collection that provides storage and retrieval capability for metrics collected from the cluster
☐ Atlas	0.5.0.2.4	Atlas Metadata and Governance platform
☑ Kafka	0.9.0.2.4	A high-throughput distributed messaging system
☐ Knox	0.6.0.2.4	Provides a single point of authentication and access for Apache Hadoop services in a cluster
☐ 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
☐ Slider	0.80.0.2.4	A framework for deploying, managing and monitoring existing distributed applications on YARN.
☑ 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.
☑ Spark	1.6.0.2.4	Apache Spark is a fast and general engine for large-scale data processing.

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 G
SNameNode:	c7002.ambari.apache.org (3.6 G
History Server:	c7002.ambari.apache.org (3.6 G
App Timeline Server:	c7002.ambari.apache.org (3.6 G
ResourceManager:	c7002.ambari.apache.org (3.6 G
Hive Metastore:	c7002.ambari.apache.org (3.6 G
WebHCat Server:	c7002.ambari.apache.org⁕
HiveServer2:	c7002.ambari.apache.org (3.6 G
ZooKeeper Server:	c7003.ambari.apache.org (3.6 G
ZooKeeper Server:	c7001.ambari.apache.org (3.6 G
ZooKeeper Server:	c7002.ambari.apache.org (3.6 G
DRPC Server:	c7002.ambari.apache.org (3.6 G
Nimbus:	c7002.ambari.apache.org (3.6 G
Storm UI Server:	c7002.ambari.apache.org (3.6 G
Metrics Collector:	c7003.ambari.apache.org (3.6 G
Spark History Server:	c7001.ambari.apache.org (3.6 G
Kafka Broker:	c7001.ambari.apache.org (3.6 G
SmartSense HST Server:	c7001.ambari.apache.org (3.6 G

c7001.ambari.apache.org (3.6 GB, 4 cores)
NameNode, ZooKeeper Server, Spark History Server, Kafka Broker, SmartSense HST Server

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

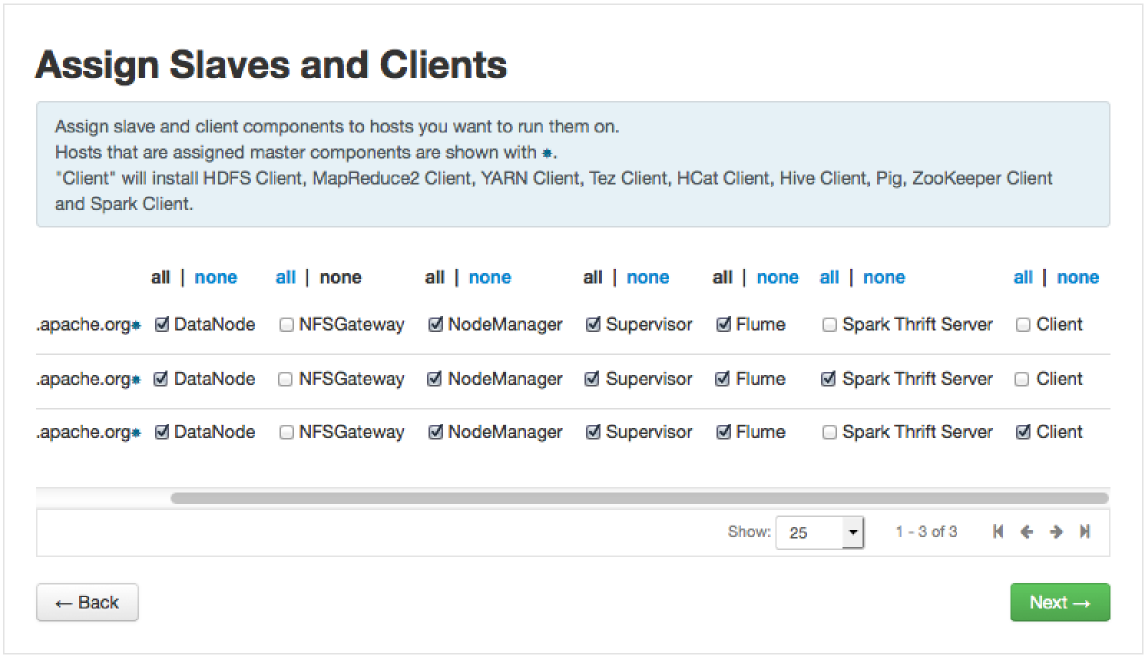
c7003.ambari.apache.org (3.6 GB, 4 cores)
ZooKeeper Server, Metrics Collector

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

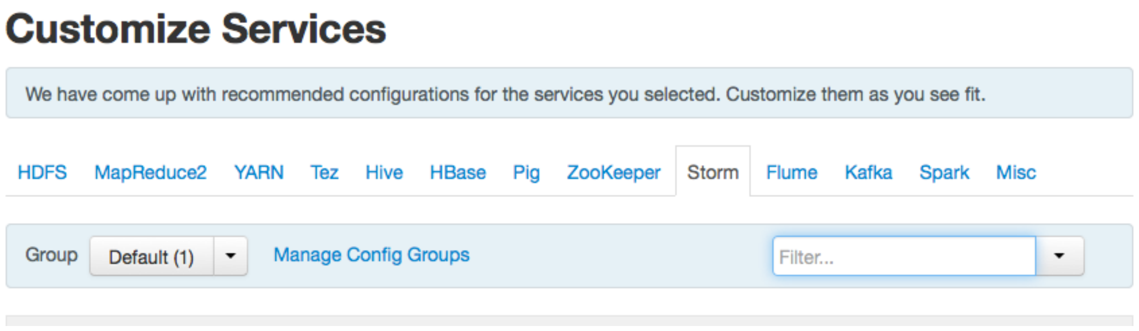
Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✱.
"Client" will install HDFS Client, MapReduce2 Client, YARN Client, Tez Client, HCat Client, Hive Client, Pig, ZooKeeper Client and Spark Client.

	all | none	all | none	all | none	all | none	all | none	all | none	all | none
.apache.org✱	✓ DataNode	☐ NFSGateway	✓ NodeManager	✓ Supervisor	✓ Flume	☐ Spark Thrift Server	☐ Client
.apache.org✱	✓ DataNode	☐ NFSGateway	✓ NodeManager	✓ Supervisor	✓ Flume	✓ Spark Thrift Server	☐ Client
.apache.org✱	✓ DataNode	☐ NFSGateway	✓ NodeManager	✓ Supervisor	✓ Flume	☐ Spark Thrift Server	✓ Client

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.

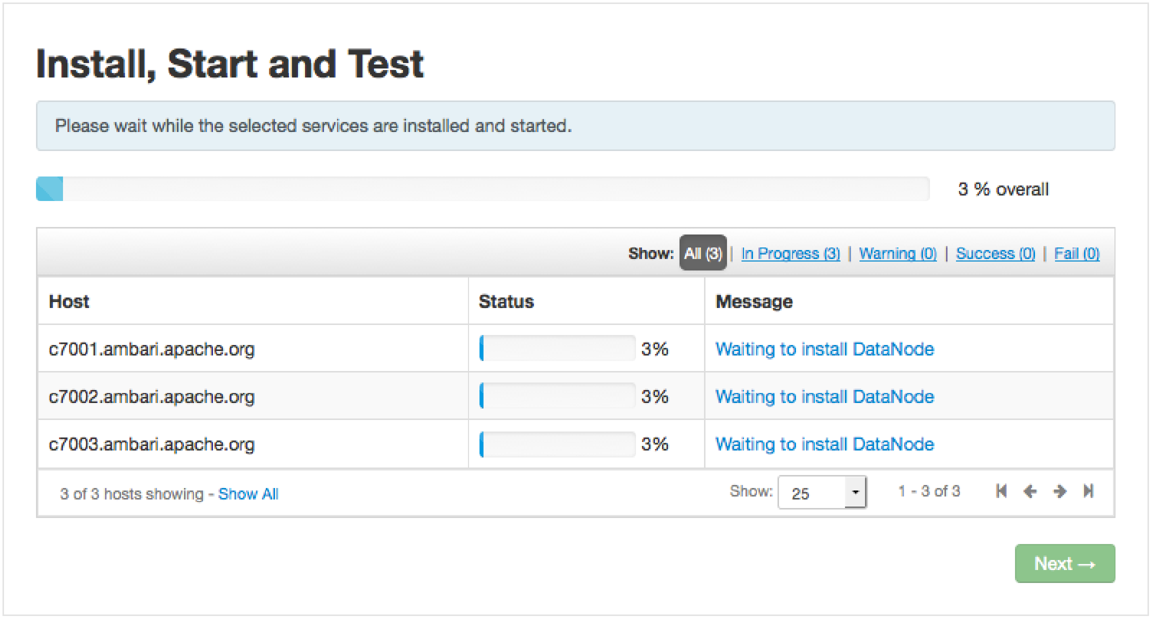
5. Ambari will display the Review screen.

   Important

   On the Review screen, make sure all HDP components correspond to HDP version 2.4.0 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

Show: All (3) | In Progress (3) | Warning (0) | Success (0) | Fail (0)

Host	Status	Message
c7001.ambari.apache.org	3%	Waiting to install DataNode
c7002.ambari.apache.org	3%	Waiting to install DataNode
c7003.ambari.apache.org	3%	Waiting to install DataNode

3 of 3 hosts showing - Show All   Show: 25   1 - 3 of 3

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.

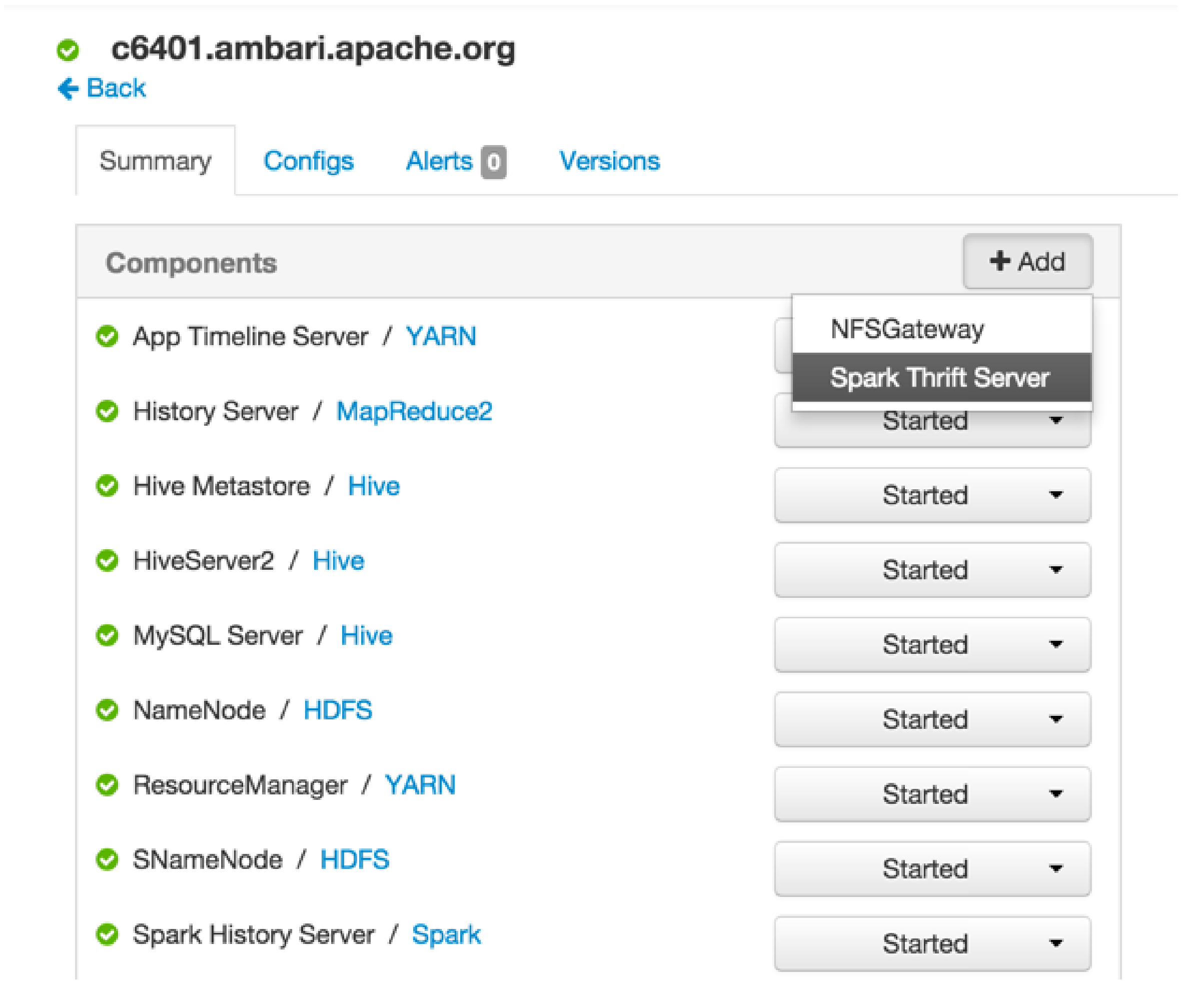
```
<configuration>
  <property>
  <name>hive.metastore.uris</name>
  <!-- hostname must point to the Hive Metastore URI in your cluster -->
    <value>thrift://hostname:9083</value>
    <description>URI for client to contact metastore server</description>
  </property>
</configuration>
```

3.1.2. 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

Components

App Timeline Server / YARN

History Server / MapReduce2 Started

Hive Metastore / Hive Started

HiveServer2 / Hive Started

MySQL Server / Hive Started

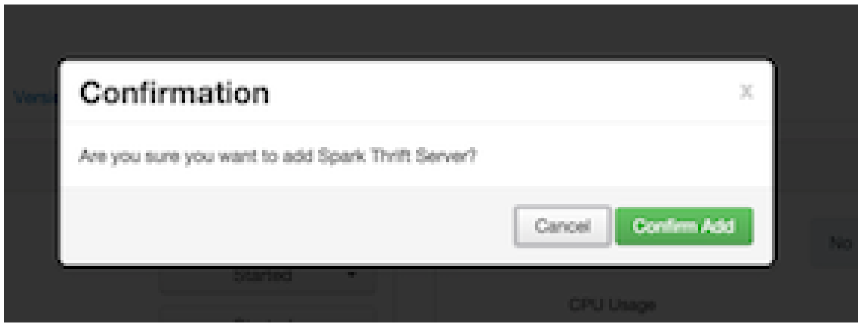
NameNode / HDFS Started

ResourceManager / YARN Started

SNameNode / HDFS Started

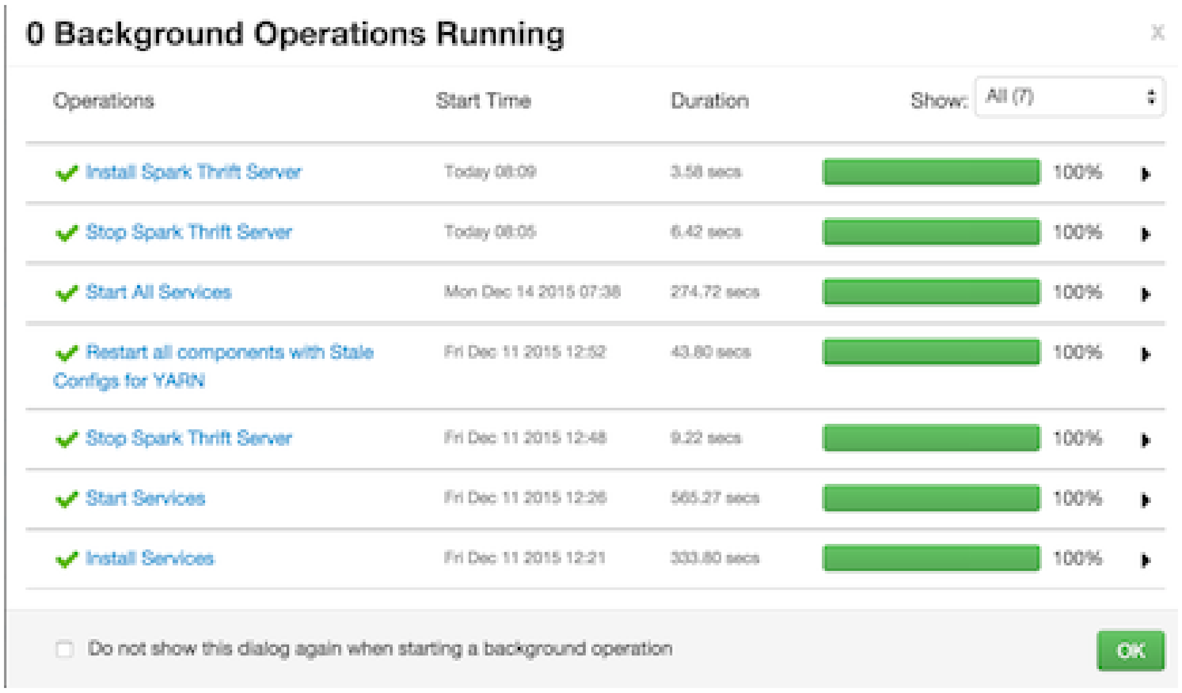
Spark History Server / Spark Started

2. Ambari will ask you to confirm the selection:


Confirmation

Are you sure you want to add Spark Thrift Server?

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.

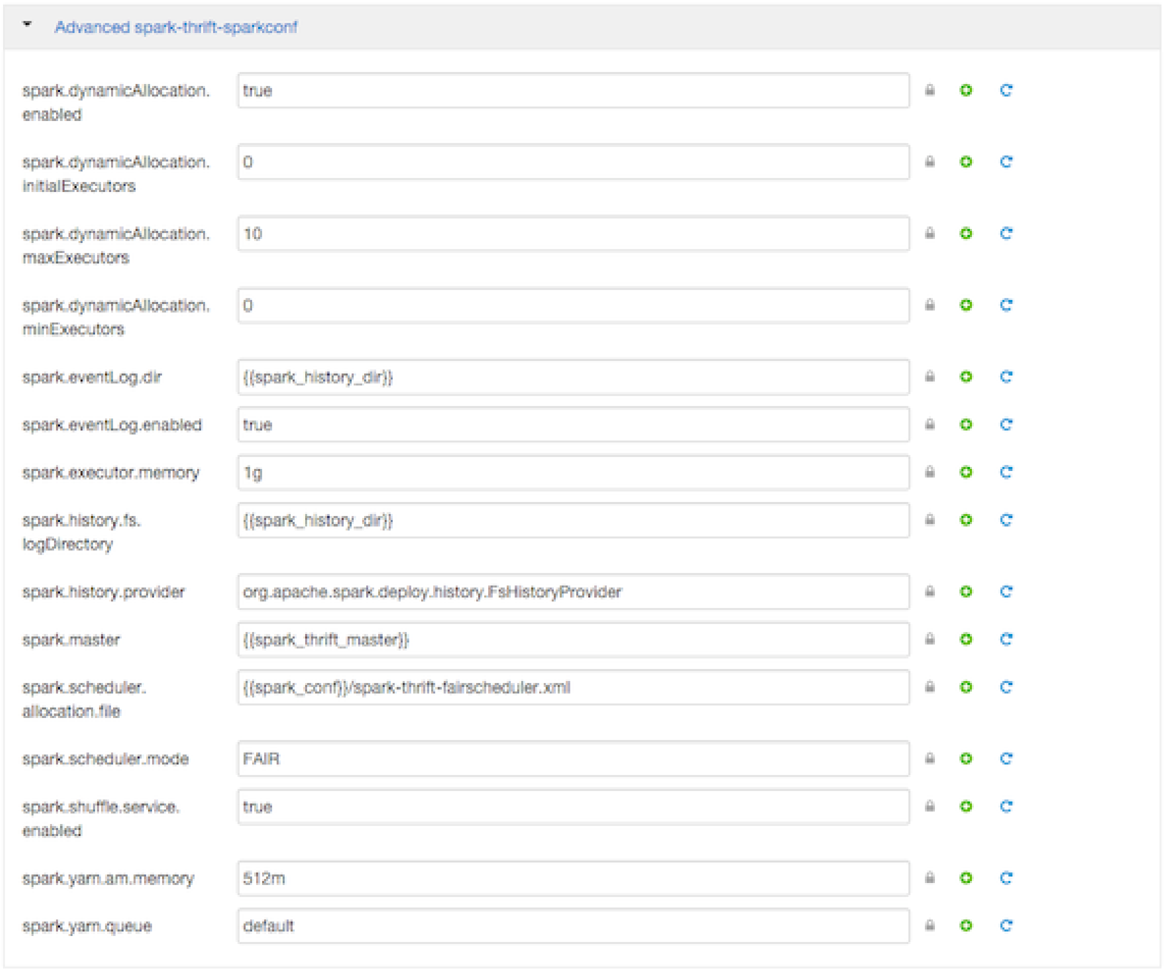
Cluster configuration is the default unless overridden by job configuration.

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	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

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
`spark.dynamicAllocation.enabled`	`true`	Whether to use dynamic resource allocation, which scales the number 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. DRA requires `spark.shuffle.service.enabled` to be set. The following configurations are also relevant: `spark.dynamicAllocation.minExecutors`, `spark.dynamicAllocation.maxExecutors`, and `spark.dynamicAllocation.initialExecutors`
`spark.shuffle.service.enabled`	`true`	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 `true` if `spark.dynamicAllocation.enabled` is `true`. 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.
`spark.dynamicAllocation.initialExecutors`	default is `spark.dynamicAllocation.minExecutors`	Initial number of executors to run if dynamic resource allocation is enabled. This value must be greater than or equal to the minExecutors value, and less than or equal to the maxExecutors value.
`spark.dynamicAllocation.maxExecutors`	Default is infinity	Upper bound for the number of executors if dynamic resource allocation is enabled.
`spark.dynamicAllocation.minExecutors`	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
`spark.dynamicAllocation.executorIdleTimeout`	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.
`spark.dynamicAllocation.cachedExecutorIdleTimeout`	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.
`spark.dynamicAllocation.schedulerBacklogTimeout`	1 second (`1s`)	If dynamic resource allocation is enabled and there have been pending tasks backlogged for more than this duration, new executors will be requested.
`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`	Default is `schedulerBacklogTimeout`	Same as `spark.dynamicAllocation.schedulerBacklogTimeout`, 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. 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 Configuring Ambari and Hadoop for Kerberos.
  - 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`).

3.4. (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 components such as Spark SQL and Spark Streaming are built on top of the core.

The fundamental programming abstraction of Spark is the resilient distributed dataset (RDD), a fault-tolerant collection of elements partitioned across the nodes of a cluster. These elements can be processed in parallel. There are two ways to create RDDs:

- *Parallelizing* an existing collection in your driver program
- Referencing a dataset in an external storage system, such as HDFS, HBase, or any data source offering a Hadoop InputFormat.

RDDs can contain Python, Java, or Scala objects, including user-defined classes.

The RDD API is a fundamental building block for Spark application development. On top of the RDD API there are several higher-level APIs, such as the DataFrame API and the Spark Streaming API. For additional information, see the Apache Spark Programming Guide.

The remainder of this chapter contains basic examples of Spark programs. Subsequent chapters describe API access to a range of data sources and analytic capabilities. Examples in this book use `SparkContext`, `SQLContext`, and `HiveContext` classes:

- A `SparkContext` is the main entry point for Spark functionality. It represents the connection to a Spark cluster, and provides access to Spark features and services.
- A `SQLContext` object is the entry point into Spark SQL functionality.
- `HiveContext` extends `SQLContext`, for applications that access Hive tables and features. Additional features include the ability to write queries using the more complete HiveQL parser, access to Hive UDFs, and the ability to read data from Hive tables. To use a HiveContext you do not need to have an existing Hive setup, and all of the data sources available to a SQLContext are still available.

Note

We do not currently support `HiveContext` in *yarn-cluster* mode in a Kerberos-enabled cluster. We do support `HiveContext` in *yarn-client* mode in a Kerberos-enabled cluster.

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 sample Python code for the Spark Pi program:

```
from __future__ import print_function

import sys
from random import random
from operator import add

from pyspark import SparkContext


if __name__ == "__main__":
    """
        Usage: pi [partitions]
    """
    sc = SparkContext(appName="PythonPi")
    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()<
```

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

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

   su spark
   ```

2. 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.

```
16/01/20 14:28:35 INFO scheduler.DAGScheduler: Job 0 finished: reduce at
 SparkPi.scala:36, took 1.721177 s
Pi is roughly 3.141296
16/01/20 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:

   ```
   bin/spark-shell

   Welcome to
         ____              __
        / __/__  ___ _____/ /__
       _\ \/ _ \/ _ `/ __/  '_/
      /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
         /_/

   Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.
   0_60)
   Type in expressions to have them evaluated.
   Type :help for more information.
   ```

```
16/01/20 16:28:09 INFO SparkContext: Running Spark version 1.6.0
16/01/20 16:28:09 INFO SecurityManager: Changing view acls to: root
...
...
16/01/20 16:28:14 INFO SparkILoop: Created sql context (with Hive support)..
SQL context available as sqlContext.

scala>
```

5. At the `scala>` prompt, submit the job: type 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. To view WordCount output in the scala shell:

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

To view the full output from within the scala shell:

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

To view the output using HDFS:

a. Exit the scala shell.

b. View WordCount job results:

```
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. 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")
   ```

   (For a description of SQLContext and related constructs, see Developing Spark Applications.)

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

   ```
   scala> df.show
   16/01/20 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:

```
16/01/20 14:36:49 INFO cluster.YarnScheduler: Removed TaskSet 13.0, whose
 tasks have all completed, from pool
16/01/20 14:36:49 INFO scheduler.DAGScheduler: ResultStage 13 (collect at :33)
 finished in 0.129 s
16/01/20 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)
```

(For a description of HiveContext and related constructs, see Developing Spark Applications.)

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. Accessing Hive Tables from Spark

This chapter describes how to access Hive data from Spark.

- Spark SQL is a Spark module for structured data processing. It supports Hive data formats, user-defined functions (UDFs), and the Hive metastore, and can act as a distributed SQL query engine. You can also use Spark SQL to incorporate Hive table data into DataFrames (see Using the Spark DataFrame API).
- "Hive on Spark" enables Hive to run on Spark; Spark operates as an execution backend for Hive queries.

7.1. Spark SQL

Spark SQL is a Spark module for structured data processing.

The recommended way to use SparkSQL is through programmatic APIs. For examples, see Accessing ORC Files from Spark.

An alternate way to access data, especially for a Beeline scenario, is through the Spark Thrift Server.

7.1.1. Using the Spark Thrift Server to Access Hive Tables

The Spark Thrift Server provides access to Hive tables through the use of JDBC.

The following example uses the Thrift Server over the HiveServer2 Beeline command-line interface.

1. Enable and start the Spark Thrift Server as specified in Starting the Spark Thrift Server (in the Non-Ambari Cluster Installation Guide), or by using the Ambari Web UI for an Ambari-managed cluster.
2. Connect to the Thrift Server over Beeline. Launch Beeline from `SPARK_HOME`.

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

3. Open Beeline and connect to the Spark SQL Thrift Server:

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

   Note

   This example does not have security enabled, so any username-password combination should work.

4. Issue a request. The following example issues a SHOW TABLES query on the HiveServer2 process:

```
0: jdbc:hive2://localhost:10015> show tables;

+------------+--------------+
| tableName | isTemporary   |
+------------+--------------+
| orc_table | false         |
| testtable | false         |
+------------+--------------+
2 rows selected (1.275 seconds)
```

5. Exit the Thrift Server:

```
0: jdbc:hive2://localhost:10015> exit
```

6. Stop the Thrift Server:

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

7.1.2. Using Spark SQL to Query JDBC Tables

More generally, Spark SQL uses the Thrift Server and JDBC federation to access JDBC databases. In addition, you can use scala or Python to create data frames from JDBC databases.

This feature works with a range of Spark interfaces, including spark-submit, spark-shell, Zeppelin (currently in tech preview), and the spark-sql client.

The first example uses HDP Sandbox version 2.3.2, Spark 1.5.1, and the Thrift Server, and is similar to the example in the previous section:

1. Start the Spark SQL Thrift Server and specify the MySQL JDBC driver:

```
sudo -u spark /usr/hdp/2.3.2.1-12/spark/sbin/start-thriftserver.sh --
hiveconf hive.server2.thrift.port=10010 --jars "/usr/share/java/mysql-
connector-java.jar"
```

2. Open Beeline and connect to the Spark SQL Thrift Server:

```
beeline -u "jdbc:hive2://localhost:10010/default" -n admin
```

3. Using Beeline, create a JDBC federated table that points to an existing MySQL database:

```
CREATE TABLE mysql_federated_sample
USING org.apache.spark.sql.jdbc
OPTIONS (
  driver "com.mysql.jdbc.Driver",
  url "jdbc:mysql://localhost/hive?user=hive&password=hive",
  dbtable "TBLS"
);
describe mysql_federated_sample;
select * from mysql_federated_sample;
select count(1) from mysql_federated_sample;
```

The next example uses the Spark shell, scala, and data frames:

1. Open spark-shell, specifying the MySQL JDBC driver:

```
spark-shell --jars "/usr/share/java/mysql-connector-java.jar"
```

2. Create a data frame pointing to the MySQL table:

```
val jdbcDF = sqlContext.read.format("jdbc").options(
  Map(
  "driver" -> "com.mysql.jdbc.Driver",
  "url" -> "jdbc:mysql://localhost/hive?user=hive&password=hive",
  "dbtable" -> "TBLS"
  )
).load()

jdbcDF.show
```

7.1.3. Using Hive UDF/UDAF/UDTF with Spark SQL

To use Hive UDF/UDAF/UDTF natively with Spark SQL:

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

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

2. From `spark-shell`, create functions:

```
sqlContext.sql("""create temporary function balance as 'com.github.
gbraccialli.hive.udf.BalanceFromRechargesAndOrders'""");
```

(For a description of SQLContext and related HiveContext, see Developing Spark Applications.)

3. From `spark-shell`, use your UDFs directly in SparkSQL:

```
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
""");
```

7.2. Hive on Spark

With "Hive on Spark," Spark operates as an execution backend for Hive queries.

The following example reads and writes to HDFS under Hive directories using the built-in UDF `collect_list(col)`, which 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
./bin/spark-shell --num-executors 2 --executor-memory 512m --master yarn-
client
```

2. Create a Hive table:

   ```
   scala> hiveContext.sql("CREATE TABLE IF NOT EXISTS TestTable (key INT, value
    STRING)")
   ```

   You should see output similar to the following:

   ```
   ...
   15/11/10 14:40:02 INFO log.PerfLogger: <PERFLOG method=Driver.run>
   start=1447184401403
   end=1447184402898
   duration=1495
   from=org.apache.hadoop.hive.ql.Driver
   res8: org.apache.spark.sql.DataFrame = [result: string]
   ```

3. Load sample data from `KV1.txt` into the table:

   ```
   scala> hiveContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/
   kv1.txt' INTO TABLE TestTable")
   ```

4. 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. 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

- Kafka Direct Receiver integration with Spark Streaming only works when the cluster is not Kerberos-enabled.
- 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.

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. 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`

```
2016-01-20 00:27:06,046|t1.machine|INFO|1580|140672245782272|MainThread|
Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.
PythonRDD.collectAndServe.
2016-01-20 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|: org.
apache.hadoop.security.AccessControlException: SIMPLE authentication is not
 enabled.  Available:[TOKEN, KERBEROS]
2016-01-20 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|at
 sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
2016-01-20 00:27:06,047|t1.machine|INFO|1580|140672245782272|
MainThread|at sun.reflect.NativeConstructorAccessorImpl.
newInstance(NativeConstructorAccessorImpl.java:57)
2016-01-20 00:27:06,048|t1.machine|INFO|1580|140672245782272|MainThread|at
{code}
```

11. Tuning and Troubleshooting 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.

General performance guidelines:

- 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.

Note

For information about known issues and workarounds related to Spark, see the "Known Issues" section of the HDP Release Notes.

11.1. Hardware Provisioning

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

When you run a Spark job, you will see a standard set of console messages.

If a job takes longer than expected or does not complete successfully, check the following to understand more about what the job was doing and where time was spent.

- To list running applications from the command line (including the application ID):

  ```
  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

If a job does not complete successfully, check the following resources to understand more about what the job was doing and where time was spent:

- The Spark History Server displays information about Spark jobs that have completed.

  - On an Ambari-managed cluster, in the Ambari Services tab, select Spark. Click on Quick Links and choose the Spark History Server UI. Ambari will display a list of jobs. Click "App ID" for job details.
  - You can access the Spark History Server Web UI directly, at `<host>:18080` (by default).

- The YARN Web UI displays 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/
  ```

- To list the contents of all log files from all containers associated with the specified application:

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

  You can also view container log files using the HDFS shell or API. For more information, see "Debugging your Application" in the Apache document Running Spark on YARN.

11.4. Configuring Spark JVM Memory Allocation

This section describes how to determine memory allocation for a JVM running the Spark executor.

To avoid memory issues, Spark uses 90% of the JVM heap by default. This percentage is controlled by `spark.storage.safetyFraction`.

Of this 90% of JVM allocation, Spark reserves memory for three purposes:

- Storing in-memory shuffle, 20% by default (controlled by `spark.shuffle.memoryFraction`)
- Unroll - used to serialize/deserialize Spark objects to disk when they don’t fit in memory, 20% is default (controlled by `spark.storage.unrollFraction`)
- Storing RDDs: 60% by default (controlled by `spark.storage.memoryFraction`)

Example

If the JVM heap is 4GB, the total memory available for RDD storage is calculated as:

4GB x 0.9 X 0. 6 = 2.16 GB

Therefore, with the default configuration approximately one half of the Executor JVM heap is used for storing RDDs.

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/2.3.0.0-$BUILD/hadoop/lib/hadoop-
lzo-0.6.0.2.3.0.0-$BUILD.jar test_read_write.py
```

For more information about the `--jar` option, see Adding Libraries to Spark.