Hortonworks Data Platform

Apache Spark Component Guide

(November 30, 2016)

docs.cloudera.com

Hortonworks Data Platform: Apache Spark Component 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. Analyzing Data with Apache Spark

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

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

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

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

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

Table 1.1. Spark Feature Support by Version

Spark Version	**1.6.2**	**1.6.2**	**1.6.1**	**1.6.0**	**1.5.2**	**1.4.1**	**1.3.1**	**1.2.1**
HDP Version(s)	**2.5.0** **2.5.3**	**2.4.3**	**2.4.2**	**2.4.0**	**2.3.4** **2.3.4.7** **2.3.6**	**2.3.2**	**2.2.8** **2.2.9** **2.3.0**	**2.2.4** **2.2.6**
Feature:								
Spark Core	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Spark on YARN	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Spark on YARN, Kerberos-enabled clusters	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Spark history server	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hive support	1.2.1	1.2.1	1.2.1	1.2.1	1.2.1	0.13.1	0.13.1	
Spark MLlib	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ML Pipeline API	Yes	Yes	Yes	Yes	Yes	Yes		
DataFrame API	Yes	Yes	Yes	Yes	Yes	Yes	TP	
ORC Files	Yes	Yes	Yes	Yes	Yes	Yes	TP	
PySpark	Yes	Yes	Yes	Yes	Yes	Yes	TP	
SparkR	Yes	TP	TP	TP	TP	TP		
Spark SQL	Yes	Yes	Yes	Yes	Yes	TP	TP	TP
Spark Thrift server (JDBC, ODBC)	Yes	Yes	Yes	Yes	Yes	TP	TP	
Spark Streaming	Yes	Yes	Yes	Yes	Yes	TP	TP	TP
Dynamic resource allocation	Yes*	Yes*	Yes*	Yes*	Yes*	TP	TP	
HBase connector	Yes	TP	TP					

TP: Tech Preview

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

The following features are available as technical previews, and are considered under development. Do not use these features in your production systems. If you have questions regarding these features, contact Support by logging a case on the Hortonworks Support Portal at https://support.hortonworks.com.

- Spark 2.0, including side-by-side installation with Spark 1.6.2 (see Installing Spark)
- GraphX
- DataSet API

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

- Direct use of the Livy server and REST API. (The Livy server is accessible through the %livy interpreter, within Zeppelin.)
- Spark Standalone
- Spark on Mesos
- Jupyter (formerly IPython) Notebook

2. Installing Spark

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

- HDP cluster stack version 2.5.0 or later
- (Optional) Ambari version 2.4.0 or later
- HDFS and YARN deployed on the cluster

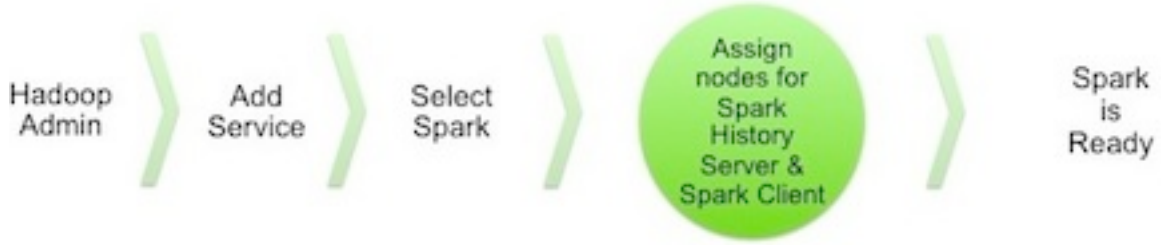
Additionally, note the following requirements and recommendations for optional features:

- Spark Thrift server requires Hive deployed on the cluster.
- PySpark requires Python installed on all nodes.
- SparkR requires R binaries installed on all nodes.
- SparkR is not currently supported on SLES.
- Spark access from Zeppelin using Livy requires the Livy server installed on the cluster (described in Installing Spark Using Ambari).
- For optimal performance with MLlib, consider installing the `netlib-java` library.

Although you can install Spark on a cluster not managed by Ambari (see Installing and Configuring Apache Spark in the *Non-Ambari Cluster Installation Guide*), this chapter describes how to install Spark on an Ambari-managed cluster.

2.1. Installing Spark Using Ambari

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



Caution

During the installation process, Ambari creates and edits several configuration files. If you configure and manage your cluster using Ambari, do not edit these files during or after installation. Instead, use the Ambari web UI to revise configuration settings.

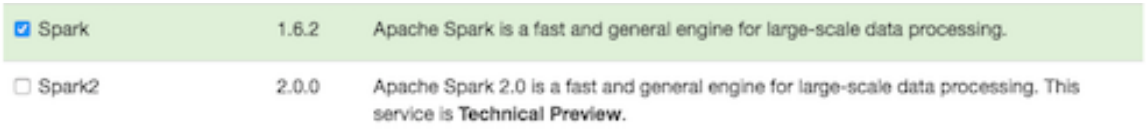
To install Spark using Ambari, complete the following steps.

1. Click the Ambari "Services" tab.

2. In the Ambari "Actions" menu, select "Add Service."

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

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



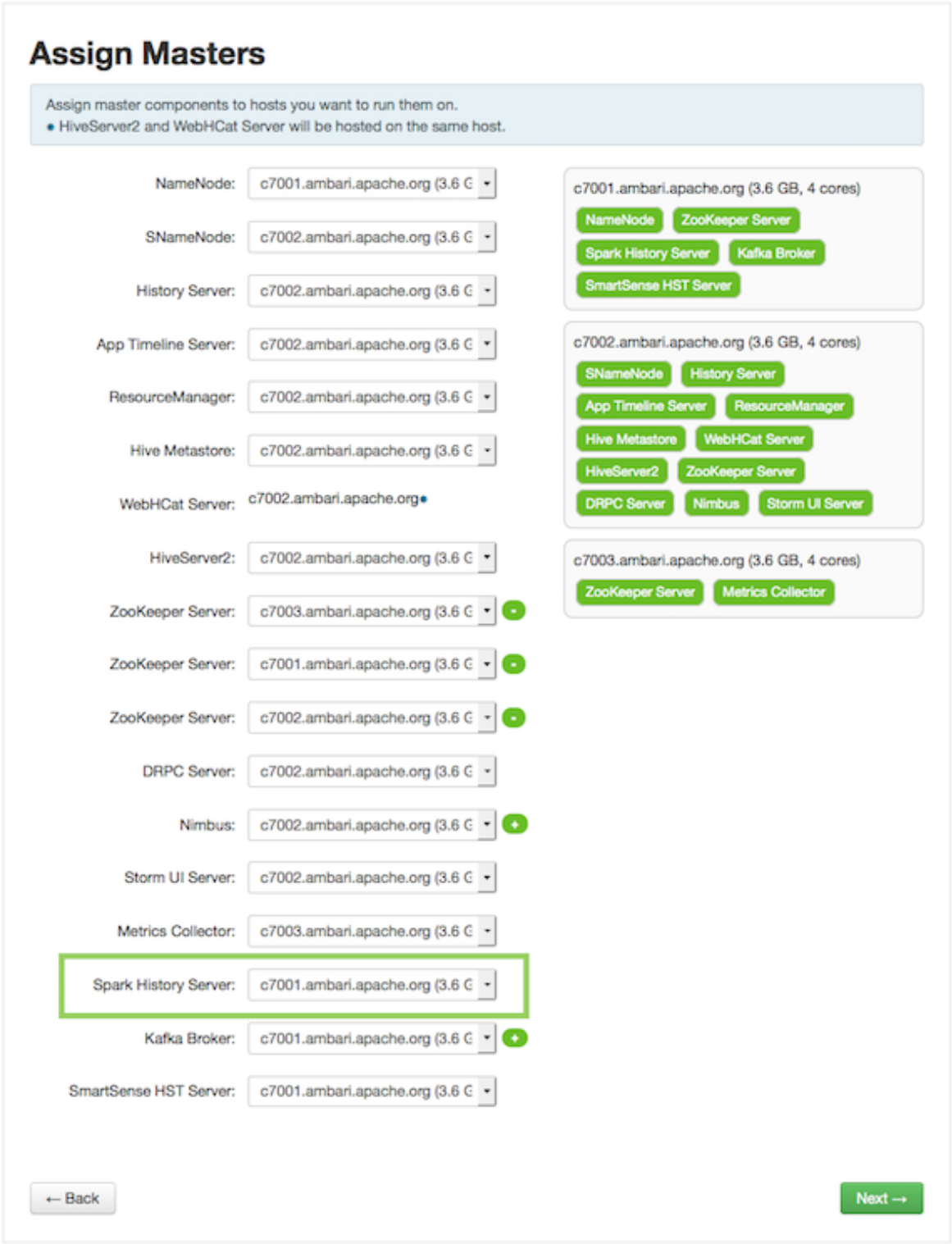
Note

If you want to install the Spark2 technical preview (not for production use), you can also select Spark2. Ambari installs it alongside Spark 1.6.

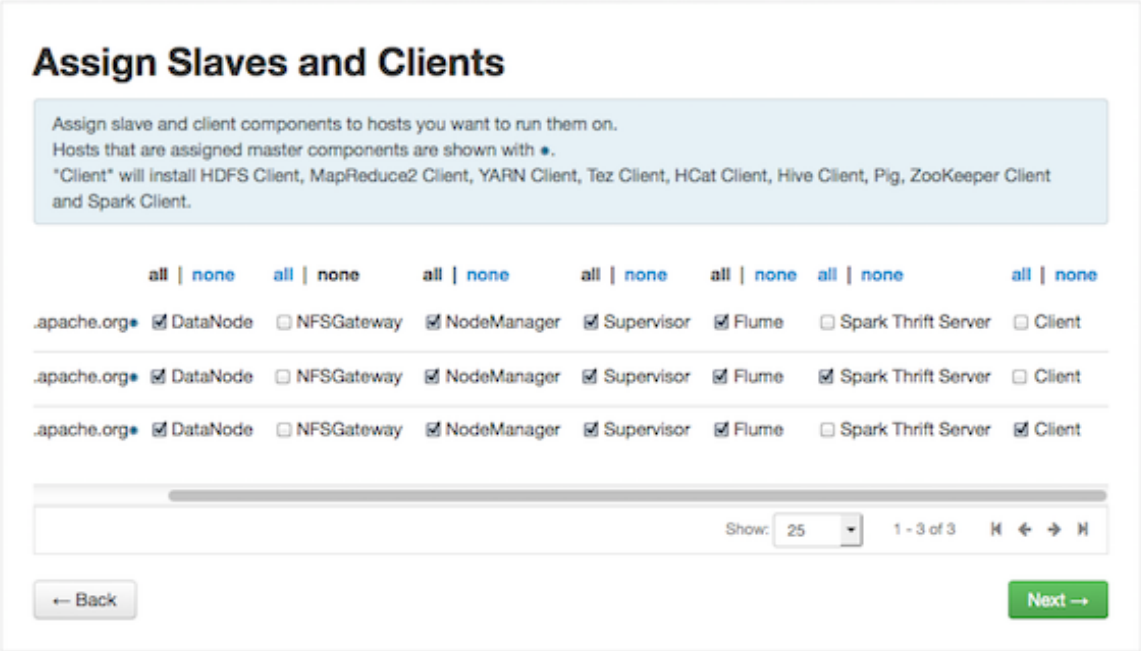
When running two Spark versions side by side, see Specifying Which Version of Spark to Use for information about choosing which Spark version runs a job.

4. Click "Next" to continue.

5. On the Assign Masters page, review the node assignment for the Spark history server, modify the assignment if desired, and click "Next":



6. On the Assign Slaves and Clients page, choose the nodes that you want to run Spark clients; these are the nodes from which Spark jobs can be submitted to YARN:



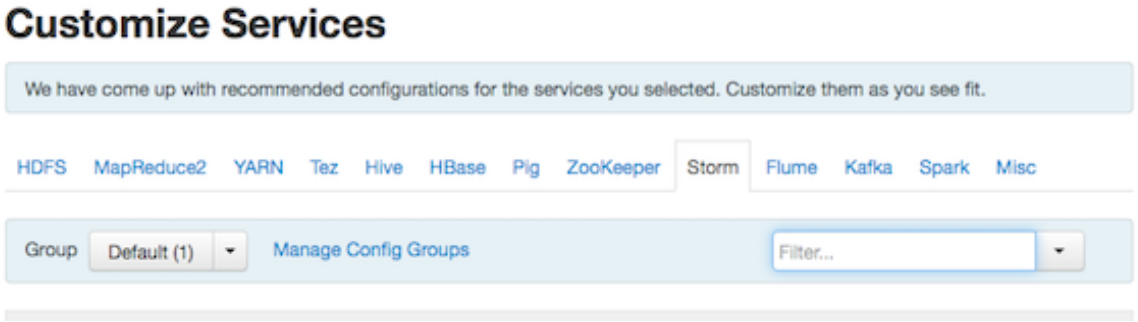
7. To install the optional Spark Thrift server for ODBC or JDBC access now, review node assignments on the Assign Slaves and Clients page and assign one or two nodes to it, as needed. Deploying the Thrift server on multiple hosts increases scalability of the Thrift server; the number of hosts should take into consideration the cluster capacity allocated to Spark.

   To install the Thrift server after Ambari finishes installing Spark, see Installing the Spark Thrift Server after Deploying Spark.

8. To install the optional Livy server (for features such as user identity propagation from Zeppelin to Spark), check the "Livy Server" box for the desired node assignment on the Assign Slaves and Clients page.

9. Click "Next" to continue.

10. Unless you are installing the Spark Thrift server now, use the default values displayed on the Customize Services page:



11. If you are installing the Spark Thrift server at this time, complete the following steps:

    a. Click the "Spark" tab on the Customize Services page.

b. Navigate to the "Advanced spark-thrift-sparkconf" group.

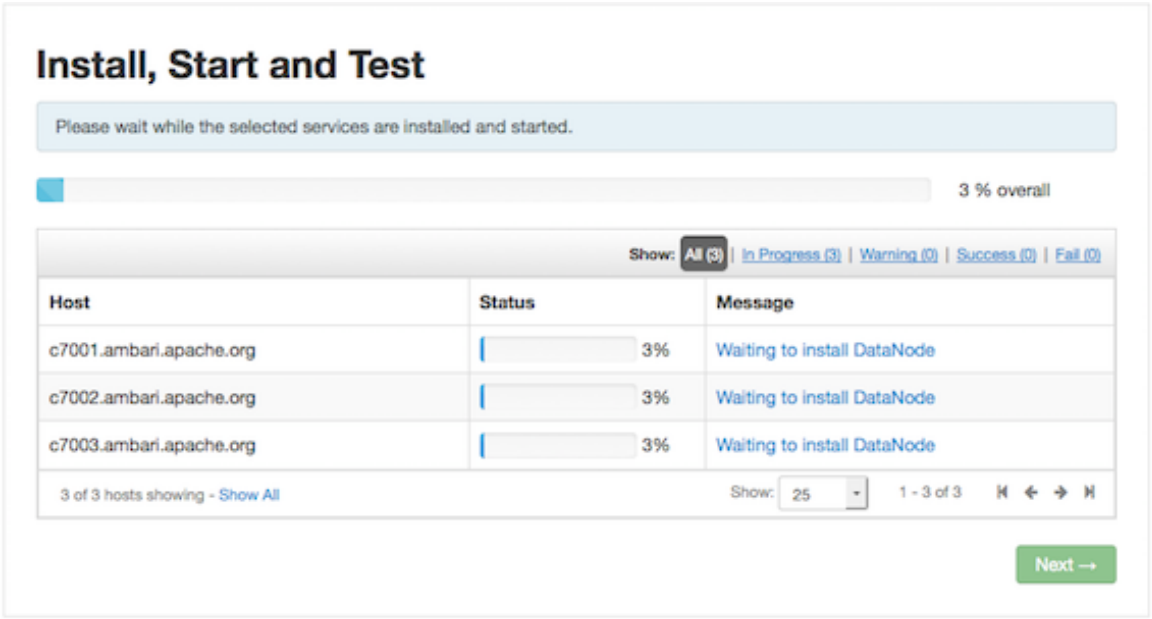
c. Set the `spark.yarn.queue` value to the name of the YARN queue that you want to use.

12. Click "Next" to continue.

13. When the wizard displays the Review page, ensure that all HDP components shown correspond to HDP 2.5.3 or later.

14. Click "Deploy" to begin installation.

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



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

2.2. Verify Spark Configuration for Hive Access

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

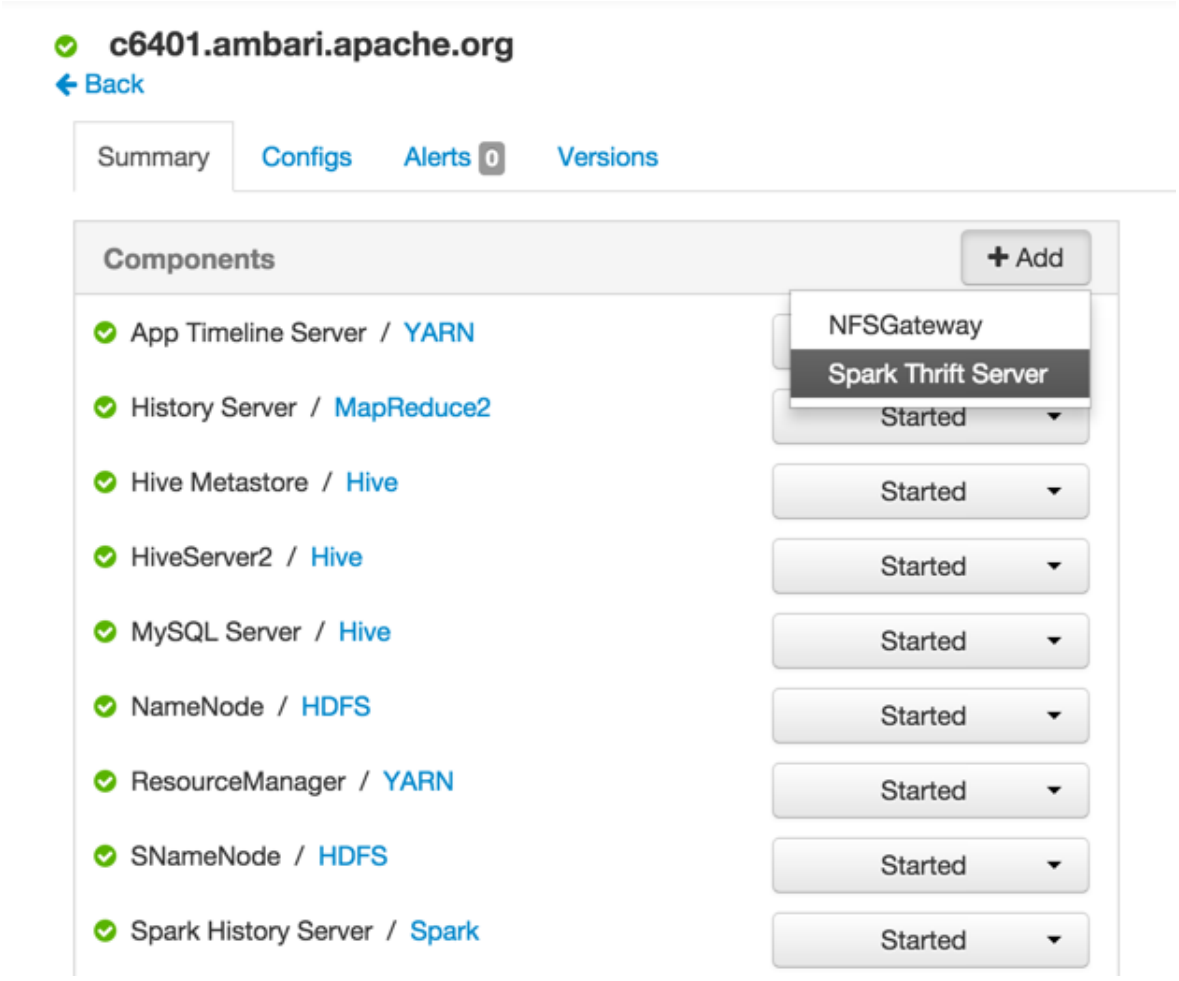
If you move Hive to a different server, edit the SPARK_HOME/`conf/hive-site.xml` file so that it contains only the hive.metastore.uris property. Make sure that the host name points to the URI where the Hive metastore is running, and that the Spark copy of `hive-site.xml` contains only the hive.metastore.uris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

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

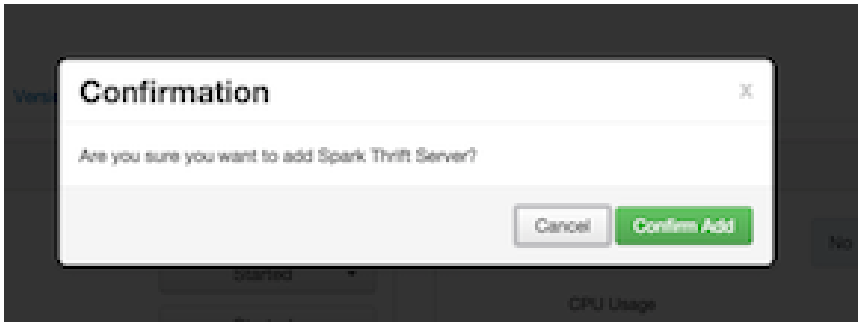
2.3. Installing the Spark Thrift Server After Deploying Spark

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

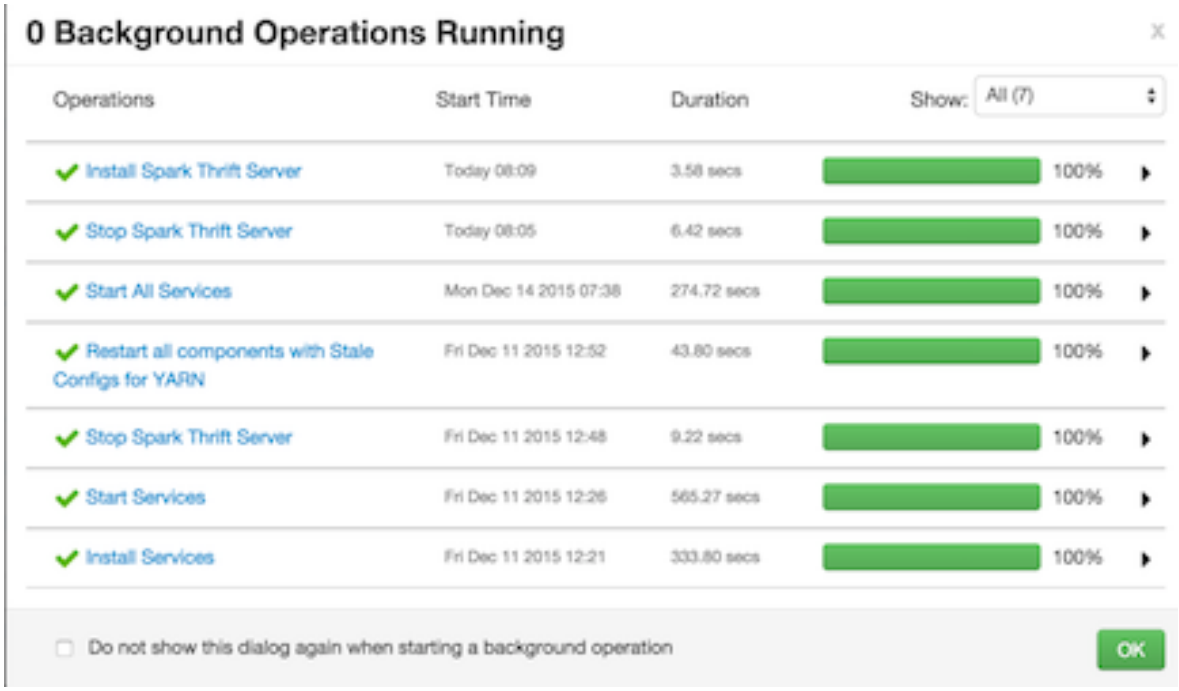
1. On the Summary tab, click "**+ Add**" and choose the Spark Thrift server:



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



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



2.4. Validating the Spark Installation

To validate the Spark installation, run the Spark Pi and WordCount jobs, supplied with the Spark package. For more information, see Running Spark Applications.

3. Configuring Spark

This chapter describes how to configure server ports, the Apache Spark history server, the Livy server, and dynamic resource allocation, as well as how to configure Spark for a Kerberos-enabled cluster.

3.1. Customizing the Spark Thrift Server Port

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

3.2. Configuring the Livy Server

To configure the optional Livy server (for features such as user identity propagation from Zeppelin to Spark), on an Ambari-managed cluster navigate to Spark > Configs, Custom livy-conf category. Add `livy.superusers` as a property, and set it to the Zeppelin service account.

Note

HDP 2.5 does not support direct use of Livy REST APIs. Livy can be accessed through the `%livy` interpreter in Zeppelin notebooks.

3.3. Configuring the Spark History Server

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

For information about configuring optional history server properties, see the Apache Monitoring and Instrumentation document.

3.4. Configuring Dynamic Resource Allocation

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

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

Note

Dynamic Resource Allocation does not work with Spark Streaming.

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

- Cluster level:
  - On an Ambari-managed cluster, the Spark Thrift server uses dynamic resource allocation by default. The Thrift Server increases or decreases the number of running executors based on a specified range, depending on load. (In addition, the Thrift Server runs in YARN mode by default, so the Thrift Server uses resources from the YARN cluster.) The associated shuffle service starts automatically, for use by the Thrift Server and general Spark jobs.
  - On a manually installed cluster, dynamic resource allocation is not enabled by default for the Thrift Server or for other Spark applications. You can enable and configure dynamic resource allocation and start the shuffle service during the Spark manual installation or upgrade process.
- Job level: You can customize dynamic resource allocation settings on a per-job basis. Job settings override cluster configuration settings.

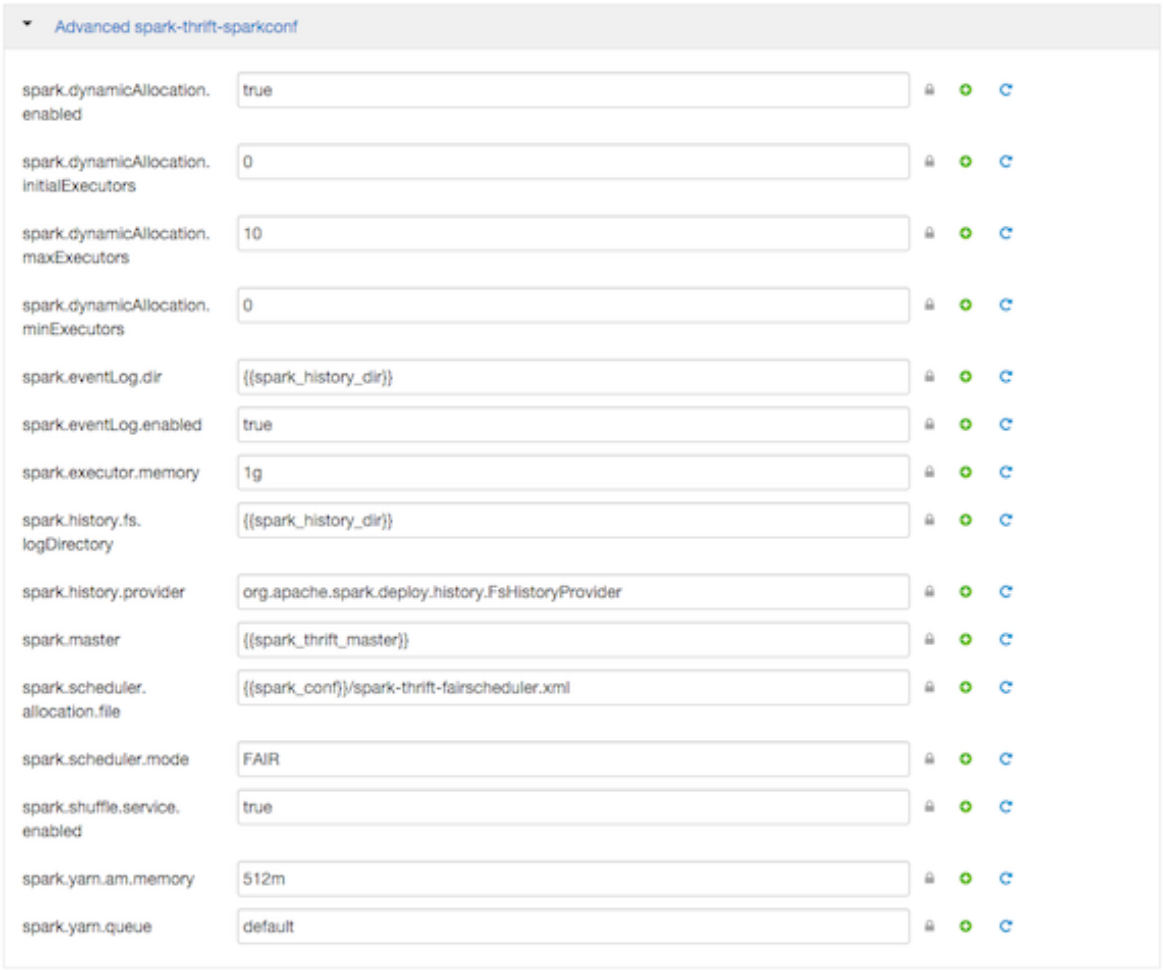
Cluster configuration is the default, unless overridden by job configuration.

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

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

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

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



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

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

3.4.2. Configuring Cluster Dynamic Resource Allocation Manually

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

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

2. (Optional) To specify a starting point and range for the number of executors, use the following properties:

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

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

   For a description of each property, see Dynamic Resource Allocation Properties.

3. Start the shuffle service on each worker node in the cluster:

   a. In the `yarn-site.xml` file on each node, add `spark_shuffle` to `yarn.nodemanager.aux-services`, and then set `yarn.nodemanager.aux-services.spark_shuffle.class` to `org.apache.spark.network.yarn.YarnShuffleService`.

   b. Review and, if necessary, edit `spark.shuffle.service.*` configuration settings.

      For more information, see the Apache Spark Shuffle Behavior documentation.

   c. Restart all NodeManagers in your cluster.

3.4.3. Configuring a Job for Dynamic Resource Allocation

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

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

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

  ```
  spark-submit –conf “property_name=property_value”
  ```

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

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

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

3.4.4. Dynamic Resource Allocation Properties

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

Table 3.1. Dynamic Resource Allocation Properties

Property Name	Value	Meaning
`spark.dynamicAllocation.enabled`	Default is `true` for the Spark Thrift server, and `false` for Spark jobs.	Specifies whether to use dynamic resource allocation, which scales the number of executors registered for an application up and down based on workload. Note that this feature is currently only available in YARN mode.
`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`.
`spark.dynamicAllocation.initialExecutors`	Default is `spark.dynamicAllocation.minExecutors`	The 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	Specifies the upper bound for the number of executors if dynamic resource allocation is enabled.
`spark.dynamicAllocation.minExecutors`	Default is 0	Specifies the lower bound for the number of executors if dynamic resource allocation is enabled.

Table 3.2. Optional Dynamic Resource Allocation Properties

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

3.5. Configuring Spark for Wire Encryption

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

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

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

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

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

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

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

Note

The following instructions enable SSL for Spark. Starting with Spark 2.0 (currently in technical preview), you can also enable HTTPS on the History Server UI, for browsing job history data.

Configuration Instructions,

Use the following commands to configure Spark for wire encryption:

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

   a. Create a keystore file:

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

   b. Create a certificate:

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

   c. Create a truststore file:

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

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

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

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

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

3. Enable Spark authentication.

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

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

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

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

4. Enable Spark SSL.

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

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

3.6. Configuring Spark for a Kerberos-Enabled Cluster

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

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

Each of these scenarios is described in the following subsections.

3.6.1. Configuring the Spark History Server

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

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

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

3.6.2. Configuring the Spark Thrift Server

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

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

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

Important

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

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

3.6.3. Setting Up Access for Submitting Jobs

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

Setting Up Access for an Account

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

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

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

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

2. Create the keytab:

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

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

   ```
   useradd spark -g hadoop
   ```

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

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

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

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

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

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

Setting Up Access for an End User

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

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

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

4. Developing and Submitting Spark Applications

Apache Spark enables you to quickly develop applications and process jobs. It 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 launch Spark applications on a cluster, you typically use the `spark-submit` script in the Spark `bin` directory. You can also use the API interactively by launching an interactive shell for Scala (`spark-shell`), Python (`pyspark`), or SparkR. Note that each interactive shell automatically creates `SparkContext` in a variable called `sc`.

For more information about 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.

This chapter describes how to run two sample programs, followed by guidelines for applications that use the Spark DataFrame API, external libraries, Spark SQL, and Hive user-defined functions.

4.1. Running Spark Applications

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

4.1.1. Spark Pi

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

Follow these steps to run the Spark Pi example:

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

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

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

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

   su spark
   ```

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

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

Commonly used options include the following:

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

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

```
16/08/22 14:28:35 INFO scheduler.DAGScheduler: Job 0 finished: reduce at
 SparkPi.scala:36, took 1.721177 s
Pi is roughly 3.141296
16/08/22 14:28:35 INFO spark.ContextCleaner: Cleaned accumulator 1
```

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

4.1.2. WordCount

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

The following example submits WordCount code to the Scala shell:

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

   You can use any text file as input.

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

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

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

   su spark
   ```

3. Upload the input file to HDFS:

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

4. Run the Spark shell:

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

   You should see output similar to the following:

   ```
   16/08/22 19:33:26 INFO SecurityManager: Changing view acls to: spark

   Welcome to
         ____              __
        / __/__  ___ _____/ /__
       _\ \/ _ \/ _ `/ __/  '_/
      /___/ .__/\_,_/_/ /_/\_\   version 1.6.2
         /_/

   Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.
   0_60)
   Type in expressions to have them evaluated.
   Type :help for more information.
   16/08/22 19:33:30 INFO SparkContext: Running Spark version 1.6.2
   16/08/22 19:33:38 INFO EventLoggingListener: Logging events to hdfs:///
   spark-history/application_1459984611485_0009
   ...
   Spark context available as sc.
   ...
   16/08/22 19:33:42 INFO HiveContext: Initializing execution hive, version 1.
   2.1
   16/08/22 19:34:00 INFO HiveContext: Initializing HiveMetastoreConnection
    version 1.2.1 using Spark classes.
   16/08/22 19:34:00 INFO ClientWrapper: Inspected Hadoop version: 2.7.1.2.5.0.
   0-130
   16/08/22 19:34:00 INFO ClientWrapper: Loaded org.apache.hadoop.hive.shims.
   Hadoop23Shims for Hadoop version 2.7.1.2.5.0.0-130
   16/08/22 19:34:01 INFO metastore: Trying to connect to metastore with URI
    thrift://green2:9083
   16/08/22 19:34:01 INFO metastore: Connected to metastore.
   SQL context available as sqlContext.

   scala>
   ```

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

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

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

   - View output in the Scala shell:

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

   - View the full output from within the Scala shell:

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

   - View the output using HDFS:

     a. Exit the Scala shell.

     b. View WordCount job status:

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

     You should see output similar to the following:

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

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

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

4.2. Specifying Which Version of Spark to Use

You can install more than one version of Spark on a node. Here are the guidelines for determining which version runs your job:

- By default, if only one version of Spark is installed on a node, your job runs with the installed version.
- By default, if more than one version of Spark is installed on a node, your job runs with the default version for your HDP package.

  The default version for HDP 2.5.3 is Spark 1.6.2.

- If more than one version of Spark is installed on a node, you can select which version of Spark runs your job.

  To do this, set the `SPARK_MAJOR_VERSION` environment variable to the desired version before you launch the job.

  For example, if Spark 1.6.2 and the Spark 2.0 technical preview are both installed on a node, and you want to run your job with Spark 2.0, set `SPARK_MAJOR_VERSION` to `2.0`.

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

Here is an example for a user who submits jobs using `spark-submit` under `/usr/bin`:

1. Navigate to a host where Spark 2.0 is installed.
2. Change to the Spark2 client directory:

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

3. Set the SPARK_MAJOR_VERSION environment variable to 2:

   ```
   export SPARK_MAJOR_VERSION=2
   ```

4. Run the Spark Pi example:

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

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

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

4.3. Using the Spark DataFrame API

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

This subsection contains several examples of DataFrame API use.

To list JSON file contents as a DataFrame:

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

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

2. Launch the Spark shell:

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

3. At the Spark shell, type the following:

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

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

```
scala> df.show
16/08/22 11:24:10 INFO YarnScheduler: Removed TaskSet 2.0, whose tasks have
 all completed, from pool

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

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

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

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

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

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

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

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

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

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

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

peopleDataFrame.registerTempTable("people")

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

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

This produces output similar to the following:

```
16/08/22 14:36:49 INFO cluster.YarnScheduler: Removed TaskSet 13.0, whose
 tasks have all completed, from pool
16/08/22 14:36:49 INFO scheduler.DAGScheduler: ResultStage 13 (collect at :33)
 finished in 0.129 s
16/08/22 14:36:49 INFO scheduler.DAGScheduler: Job 10 finished: collect
 at :33, took 0.162827 s
Name: Michael
Name: Andy
Name: Justin
```

4.4. Adding Libraries to Spark

Spark comes equipped with a selection of libraries, including Spark SQL, Spark Streaming, and MLlib. However, if you want to use a custom library with a Spark application (such as a compression library or Magellan), you can use one of the following two `spark-submit` script options:

- The `--jars` option transfers associated .jar files to the cluster.

  Specify a list of comma-separated .jar files.

- The `--packages` option pulls files directly from Spark packages.

  This approach requires an internet connection.

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

```
spark-submit --driver-memory 1G --executor-memory 1G --master yarn-client
 --jars /usr/hdp/2.3.0.0-2557/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 on the Apache Spark "Submitting Applications" web page.

Note

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

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

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

4.5. Using Spark SQL

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

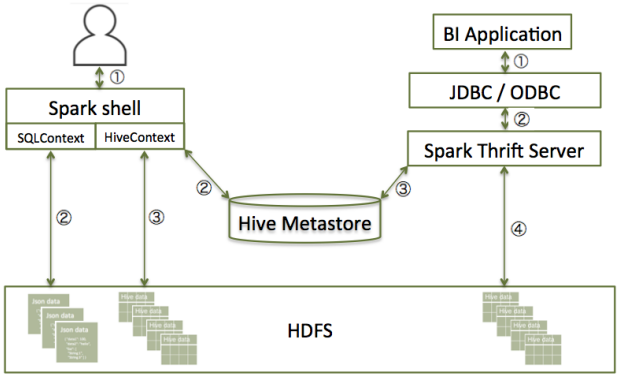
Using HiveContext, Spark SQL can also read data by interacting with the Hive MetaStore. If you already use Hive, you should use HiveContext; it supports all Hive data formats and user-defined functions (UDFs), and it enables you to have full access to the HiveQL parser. HiveContext extends SQLContext,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

There are two ways to interact with Spark SQL:

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

  For more information, see Accessing Spark SQL through JDBC and ODBC.

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



This subsection describes how to access Spark SQL through the Spark shell, and through JDBC and ODBC. The third section describes how to form JDBC connection strings for Spark SQL, and the final section shows how to call Hive user-defined functions from Spark shell applications.

4.5.1. Accessing Spark SQL Through the Spark Shell

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

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

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

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

4.5.2. Accessing Spark SQL through JDBC or ODBC

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

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

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

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

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

- The Spark Thrift server works in YARN client mode only.
- ODBC and JDBC client configurations must match Spark Thrift server configuration parameters.

  For example, if the Thrift Server is configured to listen in binary mode, the client should send binary requests and use HTTP mode when the Thrift Server is configured over HTTP.
- 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.

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

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

```
su spark

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

4.5.2.1. Accessing Spark SQL through JDBC

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

```
su spark
```

```
./bin/beeline
```

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

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

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

You should see output similar to the following:

```
beeline> !connect jdbc:hive2://localhost:10015
Connecting to jdbc:hive2://localhost:10015
Enter username for jdbc:hive2://localhost:10015:
Enter password for jdbc:hive2://localhost:10015:
...
Connected to: Spark SQL (version 1.6.2)
Driver: Spark Project Core (version 1.6.2.2.4.0.0-169)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10015>
```

2. When connected, issue a Spark SQL statement.

The following example executes a SHOW TABLES query:

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

4.5.2.2. Accessing Spark SQL through ODBC

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

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

4.5.3. Forming JDBC Connection Strings for Spark SQL

A JDBC URL connection string supplies connection information to the JDBC data source. Connection strings for the Spark SQL JDBC driver have the following format:

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

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

Note

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

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

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

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

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

4.5.4. Calling Hive User-Defined Functions

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

4.5.4.1. Using Built-in UDFs

The following interactive example reads and writes to HDFS under Hive directories, using `hiveContext` and the built-in `collect_list(col)` UDF. The `collect_list(col)` UDF returns a list of objects with duplicates. In a production environment, this type of operation runs under an account with appropriate HDFS permissions; the following example uses `hdfs` user.

1. Launch the Spark Shell on a YARN cluster:

   ```
   su hdfs
   ```

```
cd $SPARK_HOME

./bin/spark-shell --num-executors 2 --executor-memory 512m --
master yarn-client
```

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

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

3. Invoke the Hive `collect_list` UDF:

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

4.5.4.2. Using Custom UDFs

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

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

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

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

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

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

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

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

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

4.6. Using Spark Streaming

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

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

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

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

Apache Spark 1.6 has built-in support for the Apache Kafka 08 API. If you want to access a Kafka 0.10 cluster using new Kafka 0.10 APIs (such as wire encryption support) from Spark 1.6 streaming jobs, the spark-kafka-0-10-connector package supports a Kafka 0.10 connector for Spark 1.x streaming. See the package readme file for additional documentation.

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

Important

Dynamic Resource Allocation does not work with Spark Streaming.

4.6.1. Prerequisites

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

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

4.6.2. Building and Running a Secure Spark Streaming Job

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

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

  1. Add the Hortonworks repository to your `pom.xml` file:

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

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

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka_2.10</artifactId>
    <version>1.6.2.2.4.2.0-90</version>
</dependency>

<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.6.2.2.4.2.0-90</version>
    <scope>provided</scope>
</dependency>
```

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

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

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

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

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

```
spark-submit --master yarn-client --num-executors 1 \
--packages org.apache.spark:spark-streaming-kafka_2.10:1.6.2.2.4.2.0-90 \
--repositories http://repo.hortonworks.com/content/repositories/releases/
 \
--class <user-main-class> \
<user-application.jar> \
<user arg lists>
```

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

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

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

For a sample `pom.xml` file, see Sample pom.xml file for Spark Streaming with Kafka.

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

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

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

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

2. Generate a keytab for the user.

3. Create a Java Authentication and Authorization Service (JAAS) login configuration file: for example, `key.conf`.

4. Add configuration settings that specify the user keytab.

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

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

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

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

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

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

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

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

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

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

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

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

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka_2.10</artifactId>
            <version>1.6.2.2.4.2.0-90</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.10</artifactId>
            <version>1.6.2.2.4.2.0-90</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>
```

4.7. Spark on HBase: Using the HBase Connector

The Spark-HBase connector (`shc`) is a Spark library that supports access to HBase tables as external sources or sinks. Application access is through Spark SQL at the data frame level, with support for optimizations such as partition pruning, predicate pushdown, and scanning.

The connector bridges the gap between the HBase key-value store and complex relational SQL queries. It is useful for Spark applications and interactive tools, as it allows operations

such as complex SQL queries on top of an HBase table inside Spark, and table joins against data frames. The connector leverages the standard Spark DataSource API for query optimization.

Note

The Spark HBase connector uses HBase jar files by default. If you want to submit jobs on an HBase cluster with Phoenix enabled, you must include `--jars phoenix-server.jar` in your `spark-submit` command; for example:

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

The HBase connector library is available as a Spark package; you can download it from https://github.com/hortonworks-spark/shc. The repository `readme` file contains information about how to use the package with Spark applications.

4.8. Accessing ORC Data in Hive Tables

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

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

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

4.8.1. Accessing ORC Files from Spark

To start using ORC, you can define a HiveContext instance:

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

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

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

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

2. Create 100 `Person` records:

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

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

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

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

4. Read the objects back:

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

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

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

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

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

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

You can also use Spark `DataFrameReader` and `DataFrameWriter` methods to access ORC files.

4.8.2. Enabling Predicate Push-Down Optimization

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

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

- File-level and stripe-level statistics are in the file footer, making it easy to determine if the rest of the file must be read.
- Row-level indexes include column statistics for each row group and position, for finding the start of the row group.

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

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

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

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

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

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

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

```
val 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()
```

4.8.4. Optimizing Queries Through Partition Pruning

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

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

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

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

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

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

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

4.8.5. 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/

4.9. Accessing HDFS Files from Spark

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

4.9.1. Specifying Compression

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

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

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

For more information about supported compression algorithms, see Configuring HDFS Compression in the *HDFS Administration Guide*.

4.9.2. Accessing HDFS from PySpark

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

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
[hrt_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

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

5. Using Spark from R: SparkR

SparkR is an R package that provides a lightweight front end for using Apache Spark from R, thus supporting large-scale analytics on Hortonworks Data Platform (HDP) from the R language and environment. As of Spark 1.6.2, SparkR provides a distributed data frame implementation that supports operations like selection, filtering, and aggregation on large datasets. In addition, SparkR supports distributed machine learning through MLlib.

5.1. Prerequisites

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

- R must be installed on all nodes.
- JAVA_HOME must be set on all nodes.

Note: SparkR is not currently supported on SLES.

5.2. SparkR Example

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

1. Launch SparkR:

```
su spark
cd /usr/hdp/2.5.0.0-3485/spark/bin
./sparkR
```

Output similar to the following displays:

```
Welcome to
    ____              __
   / __/__  ___ _____/ /__
  _\ \/ _ \/ _ `/ __/  '_/
 /___/ .__/\_,_/_/ /_/\_\   version  1.6.2
    /_/

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

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

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

Output similar to the following displays:

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

3. Read the `people` DataFrame:

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

Output similar to the following displays:

```
 age    name
1  NA Michael
2  30    Andy
3  19  Justin
```

5.3. Additional Resources

For additional SparkR examples, see the Apache SparkR documentation.

6. Tuning Spark

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

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

6.1. 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.

6.2. Checking Job Status

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

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

6.3. Checking Job History

You can use the following resources to view job history:

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

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

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

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

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

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

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

  `yarn logs -applicationId <app_id>`.

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

  For more information, see "Debugging your Application" in the Apache document Running Spark on YARN.

6.4. 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.

6.4.1. Configuring YARN Memory Allocation for Spark

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

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

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

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

- RAM (amount of memory)

- CORES (number of CPU cores)

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

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

  Large executor memory does not imply better performance, due to JVM garbage collection. Sometimes it is better to configure a larger number of small JVMs than a small number of large JVMs.
- Executor processes are not released if the job has not finished, even if they are no longer in use.

  Therefore, do not overallocate executors above your estimated requirements.

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

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

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

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