Hortonworks Data Platform

Spark QuickStart Guide

(July 21, 2015)

docs.cloudera.com

Hortonworks Data Platform: Spark QuickStart 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 1.3.1, 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
- Support for Hive 0.13.1, including the `collect_list` UDF

The following features are available as technical previews:

- Spark DataFrame API
- ORC file support
- Spark SQL
- Spark Streaming
- Spark SQL Thrift Server
- Dynamic Executor Allocation

The following features and tools are not officially supported in this release:

- ML Pipeline API
- SparkR
- Spark Standalone

- GraphX
- iPython
- Zeppelin

Spark on YARN uses YARN services for resource allocation, running 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.

The following tables summarize Spark versions and feature support across HDP and Ambari versions.

Table 1.1. Spark Support in HDP, Ambari

HDP	Ambari	Spark
2.2.4	2.0.1	1.2.1
2.2.6	2.1.1	1.2.1
2.2.8	2.1.1	1.3.1
2.2.9	2.1.1	1.3.1
2.3.0	2.1.1	1.3.1

Table 1.2. Spark Feature Support by Version

Feature	1.2.1	1.3.1
Spark Core	Yes	Yes
Spark on YARN	Yes	Yes
Spark on YARN, Kerberos-enabled clusters	Yes	Yes
Spark History Server	Yes	Yes
Spark MLLib	Yes	Yes
Hive 0.1.3, including `collect_list UDF`		Yes
ML Pipeline API (PySpark)		
DataFrame API		TP
ORC Files		TP
Spark SQL	TP	TP
Spark Streaming	TP	TP
Spark SQL Thrift Server		TP
Dynamic Executor Allocation		TP
SparkR		
Spark Standalone		
GraphX		

TP: Tech Preview

If you are evaluating custom Spark builds or builds from Apache, please see the Troubleshooting Spark section.

2. Prerequisites

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

Table 2.1. Prerequisites for running Spark 1.3.1

Prerequisite	**Description**
Cluster Stack Version	• HDP 2.2.6 or later
(Optional) Ambari	• Version 2.1 or later
Software dependencies	• Spark requires HDFS and YARN • PySpark requires Python to be installed on all nodes

Note

If you installed the tech preview, save any configuration changes you made to the tech preview environment. Install Spark, and then update the configuration with your changes.

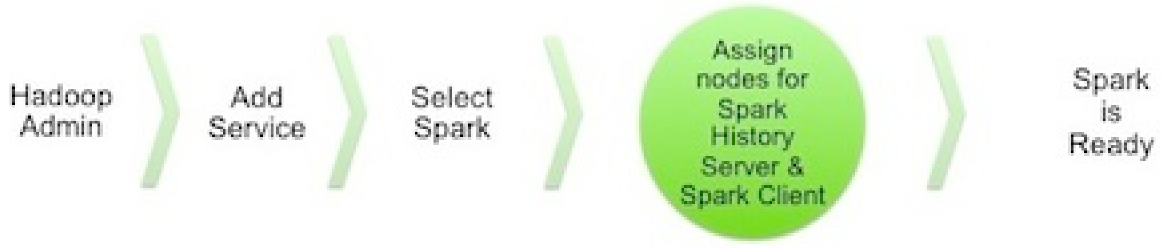
3. Installing Spark

To install Spark manually, see "Installing and Configuring Apache Spark" in the Manual Install Guide.

To install Spark on a Kerberized cluster, first read Installing Spark with Kerberos (the next topic in this Quick Start Guide).

The remainder of this section describes how to install Spark using Ambari. (For general information about installing HDP components using Ambari, see Adding a Service in the Ambari Documentation Suite.)

The following diagram shows the Spark installation process using Ambari.

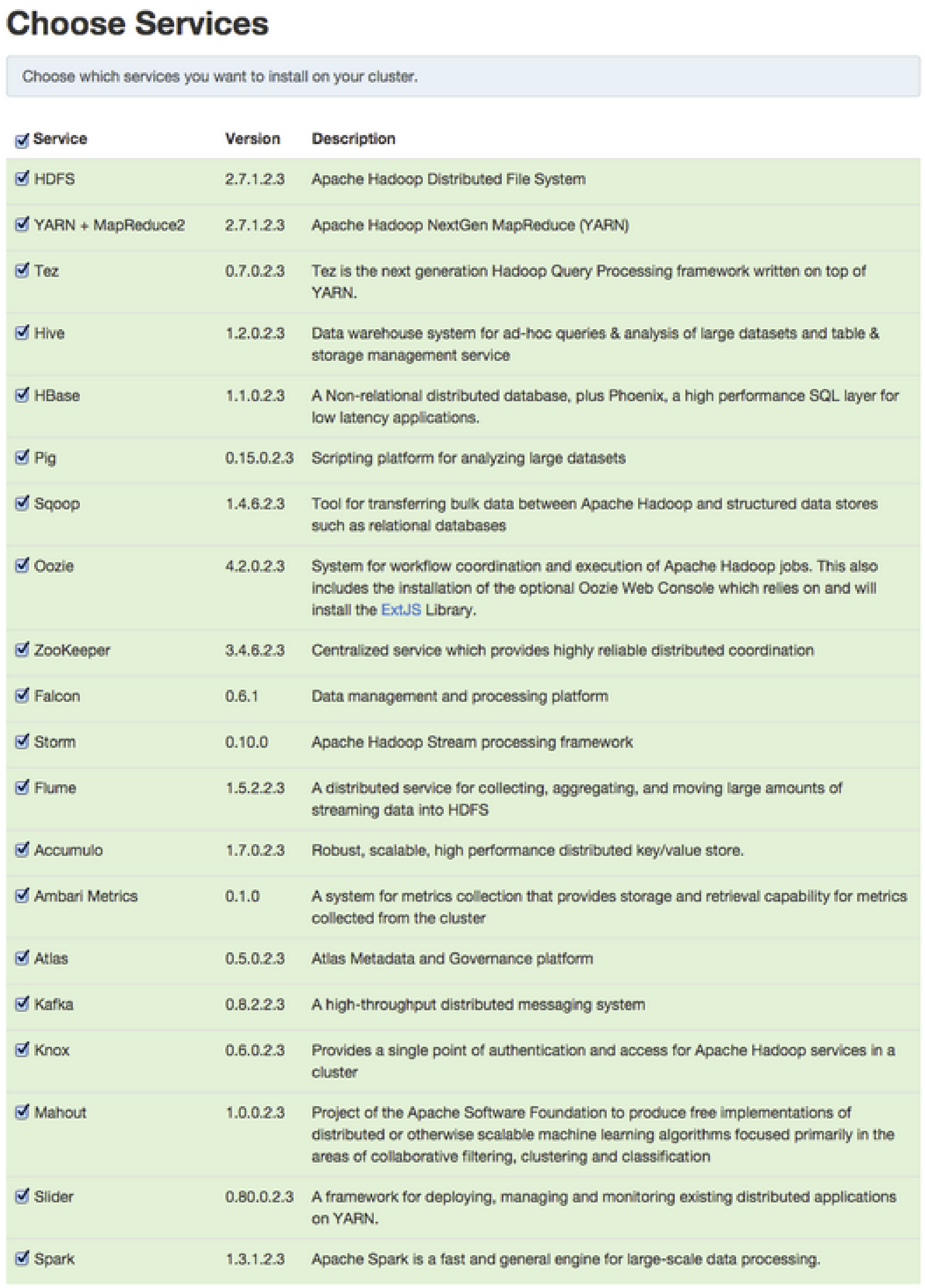


To install Spark using Ambari, complete the following steps:

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.



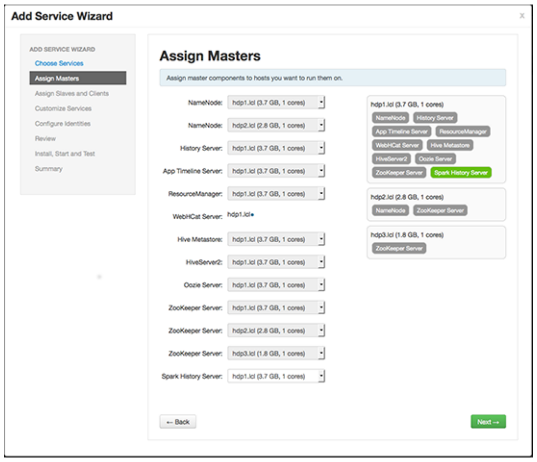
Choose Services

Choose which services you want to install on your cluster.

✓ Service	Version	Description
✓ HDFS	2.7.1.2.3	Apache Hadoop Distributed File System
✓ YARN + MapReduce2	2.7.1.2.3	Apache Hadoop NextGen MapReduce (YARN)
✓ Tez	0.7.0.2.3	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
✓ Hive	1.2.0.2.3	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
✓ HBase	1.1.0.2.3	A Non-relational distributed database, plus Phoenix, a high performance SQL layer for low latency applications.
✓ Pig	0.15.0.2.3	Scripting platform for analyzing large datasets
✓ Sqoop	1.4.6.2.3	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases
✓ Oozie	4.2.0.2.3	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	3.4.6.2.3	Centralized service which provides highly reliable distributed coordination
✓ Falcon	0.6.1	Data management and processing platform
✓ Storm	0.10.0	Apache Hadoop Stream processing framework
✓ Flume	1.5.2.2.3	A distributed service for collecting, aggregating, and moving large amounts of streaming data into HDFS
✓ Accumulo	1.7.0.2.3	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	0.5.0.2.3	Atlas Metadata and Governance platform
✓ Kafka	0.8.2.2.3	A high-throughput distributed messaging system
✓ Knox	0.6.0.2.3	Provides a single point of authentication and access for Apache Hadoop services in a cluster
✓ Mahout	1.0.0.2.3	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	0.80.0.2.3	A framework for deploying, managing and monitoring existing distributed applications on YARN.
✓ Spark	1.3.1.2.3	Apache Spark is a fast and general engine for large-scale data processing.

2. On the Assign Masters screen, choose a node for the Spark History Server.

   Click "Next" to continue.



Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Assign Masters

Assign master components to hosts you want to run them on.

NameNode:	hdp1.lcl (3.7 GB, 1 cores)
NameNode:	hdp2.lcl (2.8 GB, 1 cores)
History Server:	hdp1.lcl (3.7 GB, 1 cores)
App Timeline Server:	hdp1.lcl (3.7 GB, 1 cores)
ResourceManager:	hdp1.lcl (3.7 GB, 1 cores)
WebHCat Server:	hdp1.lcl●
Hive Metastore:	hdp1.lcl (3.7 GB, 1 cores)
HiveServer2:	hdp1.lcl (3.7 GB, 1 cores)
Oozie Server:	hdp1.lcl (3.7 GB, 1 cores)
ZooKeeper Server:	hdp1.lcl (3.7 GB, 1 cores)
ZooKeeper Server:	hdp2.lcl (2.8 GB, 1 cores)
ZooKeeper Server:	hdp3.lcl (1.8 GB, 1 cores)
Spark History Server:	hdp1.lcl (3.7 GB, 1 cores)

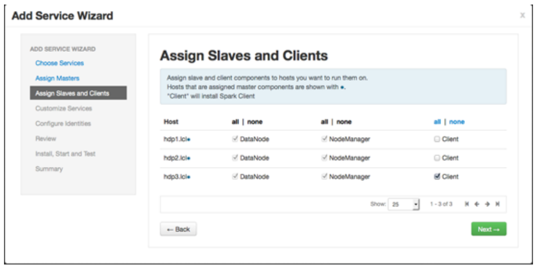
hdp1.lcl (3.7 GB, 1 cores): NameNode, History Server, App Timeline Server, ResourceManager, WebHCat Server, Hive Metastore, HiveServer2, Oozie Server, ZooKeeper Server, Spark History Server

hdp2.lcl (2.8 GB, 1 cores): NameNode, ZooKeeper Server

hdp3.lcl (1.8 GB, 1 cores): ZooKeeper Server

3. On the Assign Slaves and Clients screen, specify the machine(s) that will run Spark clients.

   Click "Next" to continue.



Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Assign Slaves and Clients

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

Host	all | none	all | none	all | none
hdp1.lcl●	✓ DataNode	✓ NodeManager	☐ Client
hdp2.lcl●	✓ DataNode	✓ NodeManager	☐ Client
hdp3.lcl●	✓ DataNode	✓ NodeManager	✓ Client

Show: 25 1 - 3 of 3

4. On the Customize Services screen there are no properties that must be specified. We recommend that you use default values for your initial configuration. Click "Next" to continue.

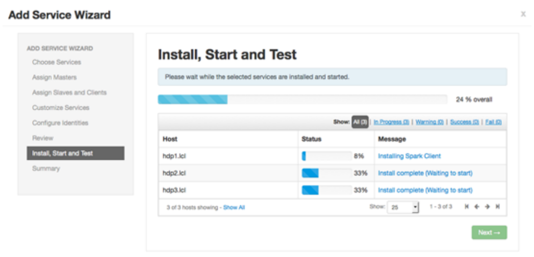
5. Ambari will display the Review screen.

   ## Important

   On the Review screen, make sure all HDP components are version 2.2.6 or later.

   Click "Deploy" to continue.

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



Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Install, Start and Test

Please wait while the selected services are installed and started.

24 % overall

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

Host	Status	Message
hdp1.lcl	8%	Installing Spark Client
hdp2.lcl	33%	Install complete (Waiting to start)
hdp3.lcl	33%	Install complete (Waiting to start)

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.

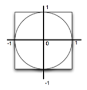
4. Validating Spark

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

- Spark Pi example
- WordCount example

4.1. Run the Spark Pi example

The Pi program tests compute-intensive tasks by calculating pi using an approximation method. The program "throws darts" at a circle – it generates points in the unit square ((0,0) to (1,1)) and sees how many fall within the unit circle. The result approximates pi.



To run Spark Pi:

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:

   ```
   su hdfs
   ```

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

2. Submit the Spark Pi job:

   ```
   ./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
   ```

   The job should complete without errors. It should produce output similar to the following:

   ```
   15/06/10 17:29:35 INFO Client:
           client token: N/A
           diagnostics: N/A
           ApplicationMaster host: N/A
           ApplicationMaster RPC port: 0
           queue: default
           start time: 1428686924325
           final status: SUCCEEDED
           tracking URL: http://blue1:8088/proxy/
   application_1428670545834_0009/
           user: hdfs
   ```

   To view job status in a browser, copy the URL tracking from the job output and go to the associated URL.

3. Job output should list the estimated value of pi. In the following example, output was directed to stdout:

```
Log Type: stdout
Log Upload Time: 10-Jun-2015 17:13:33
Log Length: 23
Pi is roughly 3.142532
```

4.2. Run the WordCount Example

WordCount is a simple program that counts how often a word occurs in a text file.

1. Select an input file for the Spark WordCount example. You can use any text file as input.
2. Upload the input file to HDFS. The following example uses `log4j.properties` as the input file:

   ```
   su hdfs
   ```

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

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

3. Run the Spark shell:

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

   You should see output similar to the following:

   ```
   Spark assembly has been built with Hive, including Datanucleus jars on
    classpath
   15/06/30 17:42:41 INFO SecurityManager: Changing view acls to: root
   15/06/30 17:42:41 INFO SecurityManager: Changing modify acls to: root
   15/06/30 17:42:41 INFO SecurityManager: SecurityManager: authentication
    disabled; ui acls disabled; users with view permissions: Set(root); users
    with modify permissions: Set(root)
   15/06/30 17:42:41 INFO HttpServer: Starting HTTP Server
   15/06/30 17:42:41 INFO Utils: Successfully started service 'HTTP class
    server' on port 55958.
   Welcome to
         ____              __
        / __/__  ___ _____/ /__
       _\ \/ _ \/ _ `/ __/  '_/
      /___/ .__/\_,_/_/ /_/\_\   version 1.3.1
         /_/

   Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.
   0_67)
   Type in expressions to have them evaluated.
   Type :help for more information.
   15/06/30 17:42:47 INFO SecurityManager: Changing view acls to: root
   15/06/30 17:42:47 INFO SecurityManager: Changing modify acls to: root
   15/06/30 17:42:47 INFO SecurityManager: SecurityManager: authentication
    disabled; ui acls disabled; users with view permissions: Set(root); users
    with modify permissions: Set(root)
   15/06/30 17:42:48 INFO Slf4jLogger: Slf4jLogger started
   15/06/30 17:42:48 INFO Remoting: Starting remoting
   ```

```
15/06/30 17:42:48 INFO Remoting: Remoting started; listening on addresses :
[akka.tcp://sparkDriver@green4:33452]
15/06/30 17:42:48 INFO Utils: Successfully started service 'sparkDriver' on
 port 33452.
15/06/30 17:42:48 INFO SparkEnv: Registering MapOutputTracker
15/06/30 17:42:48 INFO SparkEnv: Registering BlockManagerMaster
15/06/30 17:42:48 INFO DiskBlockManager: Created local directory at /
tmp/spark-a0fdb1ce-d395-497d-bf6f-1cf00ae253b7/spark-52dfe754-7f19-4b5b-
bd73-0745a1f6d158
15/06/30 17:42:48 INFO MemoryStore: MemoryStore started with capacity 265.4
 MB
15/06/30 17:42:48 WARN NativeCodeLoader: Unable to load native-hadoop
 library for your platform... using builtin-java classes where applicable
15/06/30 17:42:49 INFO HttpFileServer: HTTP File server directory
 is /tmp/spark-817944df-07d2-4205-972c-e1b877ca4869/spark-280ea9dd-
e40d-4ec0-8ecf-8c4b159dafaf
15/06/30 17:42:49 INFO HttpServer: Starting HTTP Server
15/06/30 17:42:49 INFO Utils: Successfully started service 'HTTP file
 server' on port 56174.
15/06/30 17:42:49 INFO Utils: Successfully started service 'SparkUI' on port
 4040.
15/06/30 17:42:49 INFO SparkUI: Started SparkUI at http://green4:4040
15/06/30 17:42:49 INFO Executor: Starting executor ID <driver> on host
 localhost
15/06/30 17:42:49 INFO Executor: Using REPL class URI: http://172.23.160.
52:55958
15/06/30 17:42:49 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.
tcp://sparkDriver@green4:33452/user/HeartbeatReceiver
15/06/30 17:42:49 INFO NettyBlockTransferService: Server created on 47704
15/06/30 17:42:49 INFO BlockManagerMaster: Trying to register BlockManager
15/06/30 17:42:49 INFO BlockManagerMasterActor: Registering block manager
 localhost:47704 with 265.4 MB RAM, BlockManagerId(<driver>, localhost,
 47704)
15/06/30 17:42:49 INFO BlockManagerMaster: Registered BlockManager
15/06/30 17:42:49 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala>
```

4. Submit the job. At the scala prompt, 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")
   ```

5. To view the output from within the scala shell:

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

   To view the output using HDFS:

   a. Exit the scala shell (`control-d`).

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*
```

5. Installing Spark with Kerberos

Spark jobs are submitted to a Hadoop cluster as YARN jobs. The developer creates a Spark application in a local environment, and tests it in a single-node Spark Standalone cluster on their developer workstation.

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 Manual Install 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
```

5.1. Accessing the Hive Metastore in Secure Mode

Requirements for accessing the Hive Metastore in secure mode (with Kerberos):

- The Spark thrift server must be co-located with the Hive thrift server.
- The `spark` user must be able to access the Hive keytab.
- In yarn-client mode on a secure cluster you can use HiveContext to access the Hive Metastore. (HiveContext is not supported for yarn-cluster mode on a secure cluster.)

6. Best Practices

This section contains recommendations and best practices for using Spark with HDP 2.3.

6.1. Using SQLContext and HiveContext

There are two ways to create context in Spark SQL:

- The `SQLContext` class is the entry point into all Spark SQL functionality.
- The `HiveContext` class inherits from `SQLContext` and implements a superset of the functionality provided by `SQLContext`. Additional features include the ability to write queries using HiveQL, and the ability to read data from Hive tables.

Recommendation: use HiveContext (instead of SQLContext) whenever possible.

Note

In yarn-client mode on a secure cluster you can use HiveContext to access the Hive Metastore. HiveContext is not supported for yarn-cluster mode on a secure cluster.

Examples

The following functions work with both `HiveContext` & `SQLContext`:

`Avg()`

`Sum()`

The following functions work only with `HiveContext`:

`variance(col)`

`var_pop(col)`

`stddev_pop(col)`

`stddev_samp(col)`

`covar_samp(col1, col2)`

For more information, see the Spark Programming Guide.

6.2. Guidelines for Determining Spark 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.

For additional information about Spark memory use, see the Apache Spark Hardware Provisioning recommendations.

6.3. 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 configur a larger number of small JVMs than a small number of large JVMs.

7. Accessing ORC Files from Spark

Spark on HDP provides full support for Optimized Row Columnar ("ORC") files. ORC is a column-based file format that offers efficient storage of Hive data.

The following example shows how to access an ORC file programmatically as a table.

The example uses a text file called `people.txt`, which is included in the Apache Spark distribution. The file contains three lines:

```
Michael, 29
Andy, 30
Justin, 19
```

1. Download or create the `people.txt` file.

2. Copy `people.txt` into HDFS:

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

   hadoop dfs -put examples/src/main/resources/people.txt
   people.txt
   ```

3. Create and populate a Hive table:

   ```
   **# Import ORC support and Spark SQL libraries
   import org.apache.spark.sql.hive.orc._
   import org.apache.spark.sql._

   **# Prepare the Spark table**

   **# Create a hiveContext
   **# sc is an existing SparkContext
   val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

   **# Create an RDD of "people" objects
   val people = sc.textFile("people.txt")

   **# Specify the schema as a string
   val schemaString = "name age"

   **# Create the schema based on the schemaString
   val schema =
    StructType(
    schemaString.split(" ").map(fieldName =>
    {if(fieldName == "name")
    StructField(fieldName, StringType, true) else
    StructField(fieldName, IntegerType, true)}))

   **# Convert records in people to rows
   val rowRDD = people.map(_.split(",")).map(p => Row(p(0), new Integer(p(1).
   trim)))

   **# Apply the schema to the RDD
   val peopleSchemaRDD = hiveContext.applySchema(rowRDD, schema)

   **# Register the people SchemaRdd as a table
   ```

```
peopleSchemaRDD.registerTempTable("people")
val results = hiveContext.sql("SELECT * FROM people")

**# List query results
results.map(t => "Name: " + t.toString).collect().foreach(println)
```

4. Create and populate an ORC table from `people`:

```
**# ORC-specific section **

**# Save people as an ORC-format file
peopleSchemaRDD.saveAsOrcFile("people.orc")

**# Import "people.orc" into a Spark SQL table called "morePeople"
val morePeople = hiveContext.orcFile("people.orc")

**# Register morePeople as a table
**# This allows you to run standard SQL queries on morePeople
morePeople.registerTempTable("morePeople")

**# Display all rows
hiveContext.sql("SELECT * from morePeople").collect.foreach(println)
```

8. Using Spark with HDFS

Specifying Compression

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")`

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

```
2015-09-04 00:27:06,046|t1.machine|INFO|1580|140672245782272|MainThread|
Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.
PythonRDD.collectAndServe.
2015-09-04 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|: org.
apache.hadoop.security.AccessControlException: SIMPLE authentication is not
 enabled.  Available:[TOKEN, KERBEROS]
2015-09-04 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|at
 sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
2015-09-04 00:27:06,047|t1.machine|INFO|1580|140672245782272|
MainThread|at sun.reflect.NativeConstructorAccessorImpl.
newInstance(NativeConstructorAccessorImpl.java:57)
2015-09-04 00:27:06,048|t1.machine|INFO|1580|140672245782272|MainThread|at
{code}
```

9. Troubleshooting Spark

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

In addition, the following information is available:

- A list of running applications, where you can retrieve the application ID and check the application log:

  ```
  yarn application –list
  ```

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

- For information about a specific job, check the Spark web UI:

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

The following paragraphs describe specific issues and possible solutions.

Issue: Spark YARN jobs don’t seem to start. YARN Resource Manager logs show an application with "bad substitution” errors in its logs.

Solution: Make sure that your `$SPARK_HOME/config/spark-defaults.conf` file includes your HDP version. For example:

```
spark.driver.extraJavaOptions
-Dhdp.version=2.3.0.0-2557
spark.yarn.am.extraJavaOptions
-Dhdp.version=2.3.0.0-2557
```

To check the HDP version for an Ambari-managed cluster, navigate to `http://$AMBARI_SERVER:8080/#/main/admin/stack/versions`, where $AMBARI_SERVER is your Ambari Web URL.

To check the version via bash, run the following command:

```
   > bash-4.1# hdp-select status hadoop-client | sed 's/hadoop-
client - \(.*\)/\1/'

   2.3.0.0-2557
```

Issue: Job stays in "accepted" state; it doesn't run. This can happen when a job requests more memory or cores than available.

Solution: Assess workload to see if any resources can be released. You might need to stop unresponsive jobs to make room for the job.

Issue: Insufficient HDFS access. This can lead to errors such as the following:

```
“Loading data to table default.testtable
Failed with exception
Unable to move sourcehdfs://blue1:8020/tmp/hive-spark/hive_2015-06-04_
12-45-42_404_3643812080461575333-1/-ext-10000/kv1.txt to destination
hdfs://blue1:8020/apps/hive/warehouse/testtable/kv1.txt”
```

Solution: Make sure the user or group running the job has sufficient HDFS privileges to the location.

Issue: Wrong host in Beeline, shows error as invalid URL:

```
Error: Invalid URL: jdbc:hive2://localhost:10001 (state=08S01,code=0)
```

Solution: Specify the correct Beeline host assignment.

Issue: Error: closed SQLContext.

Solution: Restart the Thrift server.

10. Appendix A: Upgrading from the Spark Tech Preview

When moving from the Spark Tech Preview to the full HDP version of Spark, make sure that the `hive.metastore.uris` property in your `hive-site.xml` file is set to the Hive Metastore URI in your cluster. The `hive-site.xml` file typically resides in `/usr/hdp/current/spark-client/conf/`.

Example:

```
<configuration>
<property>
    <name>hive.metastore.uris</name>
    <value>thrift://blue1:9083</value>
    <description>URI for client to contact metastore server</description>
</property>
</configuration>
```

When you install Spark 1.3.1 using Ambari or the manual installation process, Spark creates and populates the `hive-site.xml` file – you no longer need to create `hive-site.xml`.