Hortonworks Data Platform

Spark Guide

(Sep 30, 2015)

docs.hortonworks.com

Hortonworks Data Platform: Spark Guide

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

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

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

Table of Contents

List of Tables

1. Introduction

Hortonworks Data Platform supports Apache Spark 1.5, a fast, large-scale data processing engine.

Deep integration of Spark with YARN allows Spark to operate as a cluster tenant alongside other engines such as Hive, Storm, and HBase, all running simultaneously on a single data platform. YARN allows flexibility: you can choose the right processing tool for the job. Instead of creating and managing a set of dedicated clusters for Spark applications, you can store data in a single location, access and analyze it with multiple processing engines, and leverage your resources. In a modern data architecture with multiple processing engines using YARN and accessing data in HDFS, Spark on YARN is the leading Spark deployment mode.

Spark Features

Spark on HDP supports the following features:

- Spark Core
- Spark on YARN
- Spark on YARN on Kerberos-enabled clusters
- Spark History Server
- Spark MLLib
- DataFrame API
- Optimized Row Columnar (ORC) files
- Spark SQL
- Spark SQL Thrift Server (JDBC/ODBC)
- Spark Streaming
- Support for Hive 1.2
- ML Pipeline API
- PySpark

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

- Dynamic Executor Allocation
- SparkR
- GraphX

- Zeppelin (link)

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

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

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

- YARN-Cluster mode, optimized for long-running production jobs.
- YARN-Client mode, best for interactive use such as prototyping, testing, and debugging. Spark Shell runs in YARN-Client mode only.

Table 1.1. Spark - HDP Version Support

HDP	Ambari	Spark
2.2.4	2.0.1	1.2.1
2.2.6	2.1.1	1.2.1
2.2.8	2.1.1	1.3.1
2.3.0	2.1.1	1.3.1
2.3.2	2.1.2	1.4.1
2.3.4	2.2	1.5.2

Table 1.2. Spark Feature Support by Version

Feature	1.2.1	1.3.1	1.4.1	1.5.2
Spark Core	Yes	Yes	Yes	Yes
Spark on YARN	Yes	Yes	Yes	Yes
Spark on YARN, Kerberos-enabled clusters	Yes	Yes	Yes	Yes
Spark History Server	Yes	Yes	Yes	Yes
Spark MLLib	Yes	Yes	Yes	Yes
Hive 0.13.1, including `collect_list` UDF		Yes	Yes	Yes
ML Pipeline API			Yes	Yes
DataFrame API		TP	Yes	Yes
ORC Files		TP	Yes	Yes
PySpark		TP	Yes	Yes
Spark SQL	TP	TP	TP	Yes
Spark Thrift Server (JDBC/ODBC)		TP	TP	Yes
Spark Streaming	TP	TP	TP	Yes
Dynamic Executor Allocation		TP	TP	TP
SparkR			TP	TP

Feature	**1.2.1**	**1.3.1**	**1.4.1**	**1.5.2**
GraphX				TP

TP: Tech Preview

2. Prerequisites

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

Table 2.1. Prerequisites for Running Spark 1.5.2

Prerequisite	Description
HDP Cluster Stack Version	• 2.3.4 or later
(Optional) Ambari Version	• 2.2 or later
Software dependencies	• Spark requires HDFS and YARN • PySpark requires Python to be installed on all nodes • SparkR (tech preview) requires R to be installed on all nodes • (Optional) For optimal performance with MLlib, consider installing the `netlib-java` library.

Note

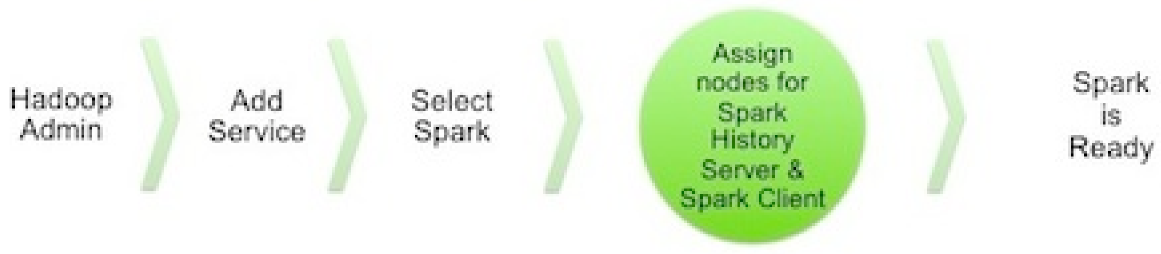
When you upgrade to HDP 2.3.4, Spark is automatically upgraded to 1.5.2. If you wish to use a previous version of Spark, follow the Spark Manual Downgrade procedure in the Release Notes.

3. Installing and Configuring Spark

To install Spark manually, see Installing and Configuring Apache Spark in the *Non-Ambari Cluster Installation Guide*. The remainder of this section describes how to install and configure Spark using Ambari.

3.1. Installing Spark Over Ambari

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

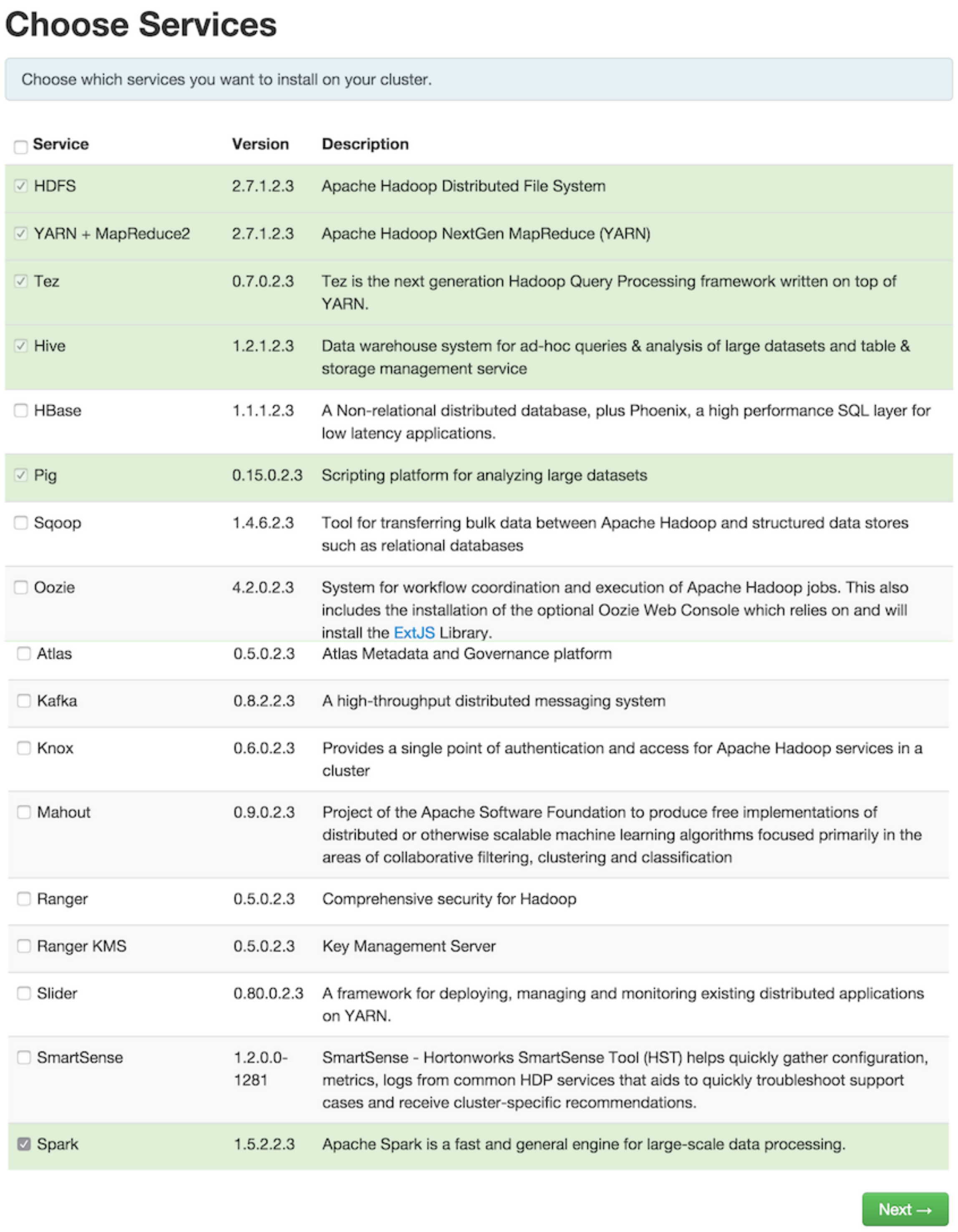


To install Spark using Ambari, complete the following steps:

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.



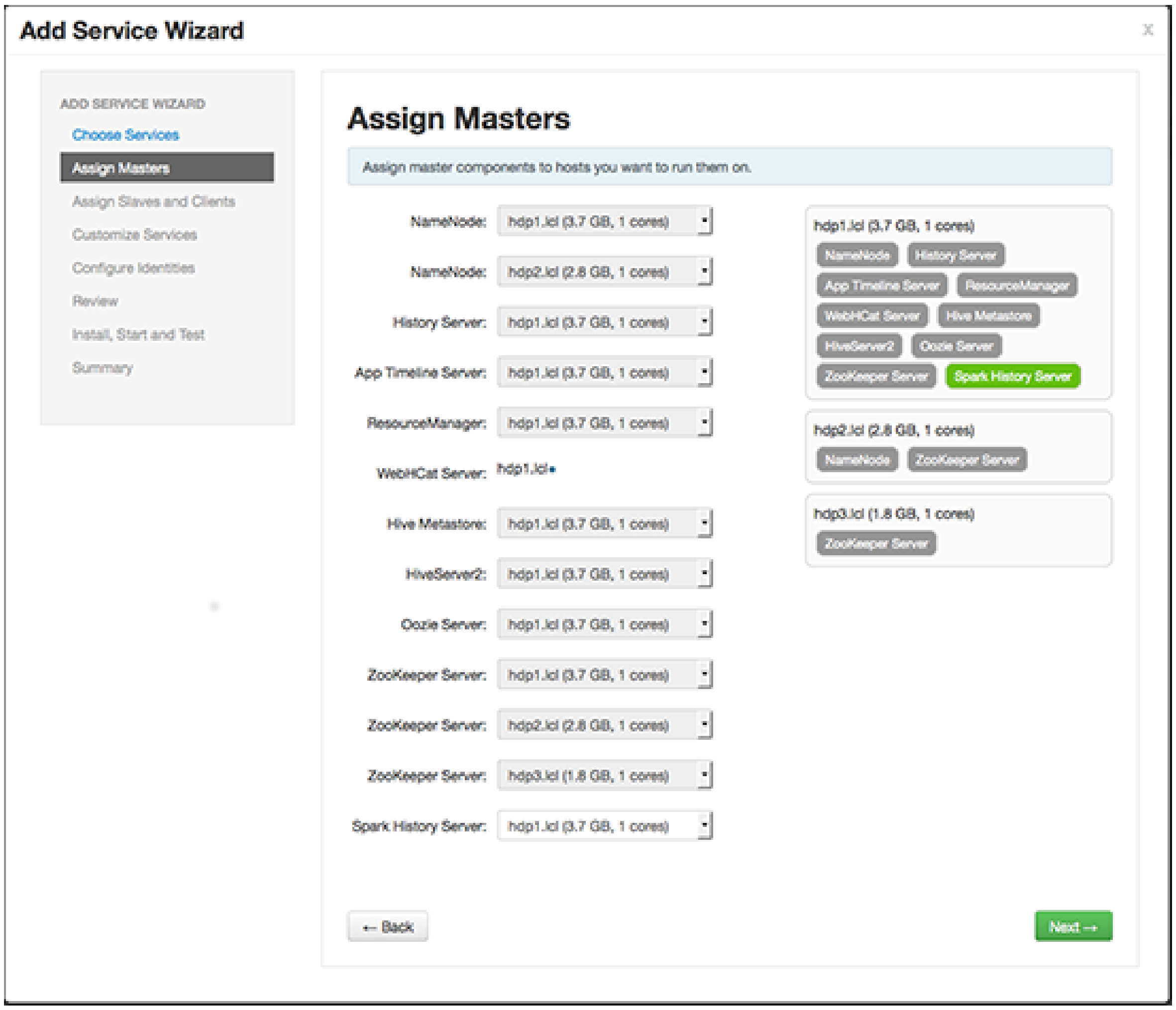
Choose Services

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	1.2.1.2.3	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
HBase	1.1.1.2.3	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.
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	0.9.0.2.3	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
Ranger	0.5.0.2.3	Comprehensive security for Hadoop
Ranger KMS	0.5.0.2.3	Key Management Server
Slider	0.80.0.2.3	A framework for deploying, managing and monitoring existing distributed applications on YARN.
SmartSense	1.2.0.0-1281	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly gather configuration, metrics, logs from common HDP services that aids to quickly troubleshoot support cases and receive cluster-specific recommendations.
✓ Spark	1.5.2.2.3	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 node(s) that will run Spark clients. These nodes – where the Spark client is deployed – will be the nodes from which Spark jobs can be submitted to YARN.

4. Optionally, install the Spark Thrift Server on specific nodes in the cluster.

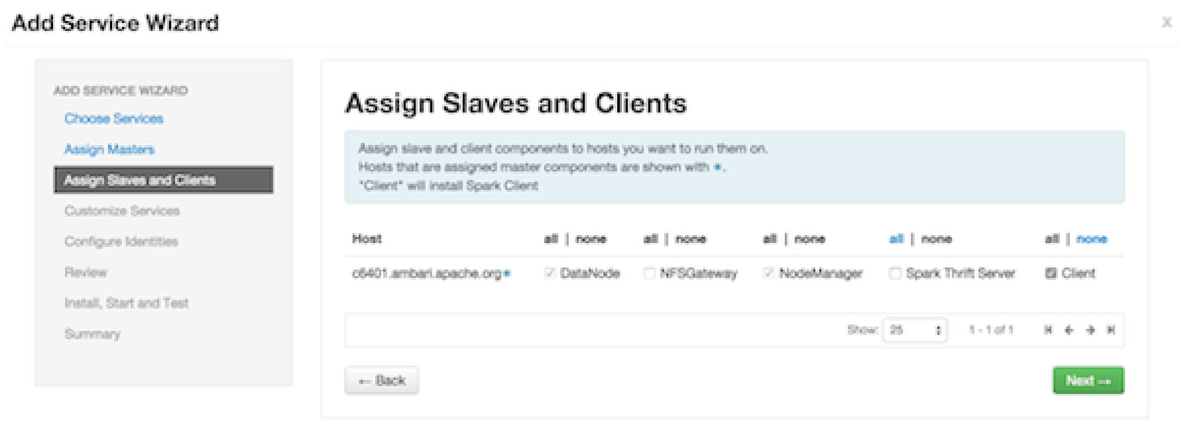
Note

There are two ways to add the Spark Thrift Server to your cluster: during component installation (described in this subsection), or at any time after Spark has been installed and deployed. To install the Spark Thrift Server later, add the optional STS service to the specified host. For more information, see "Installing the Spark Thrift Server after Installing Spark" (later in this chapter).

Important

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

Click "Next" to continue.



Assign Slaves and Clients

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

Host	all | none	all | none	all | none	all | none	all | none
c6401.ambari.apache.org✱	✓ DataNode	☐ NFSGateway	✓ NodeManager	☐ Spark Thrift Server	☑ Client

Show: 25 1 - 1 of 1

5. If you are installing the Spark Thrift Server at this time, navigate to the "Advanced spark-thrift-sparkconf" area and set the `spark.yarn.queue` value to the queue that you want to use. Other than that there are no properties that must be set using the Customize Services screen. We recommend that you use default values for your initial configuration.

   Click "Next" to continue.

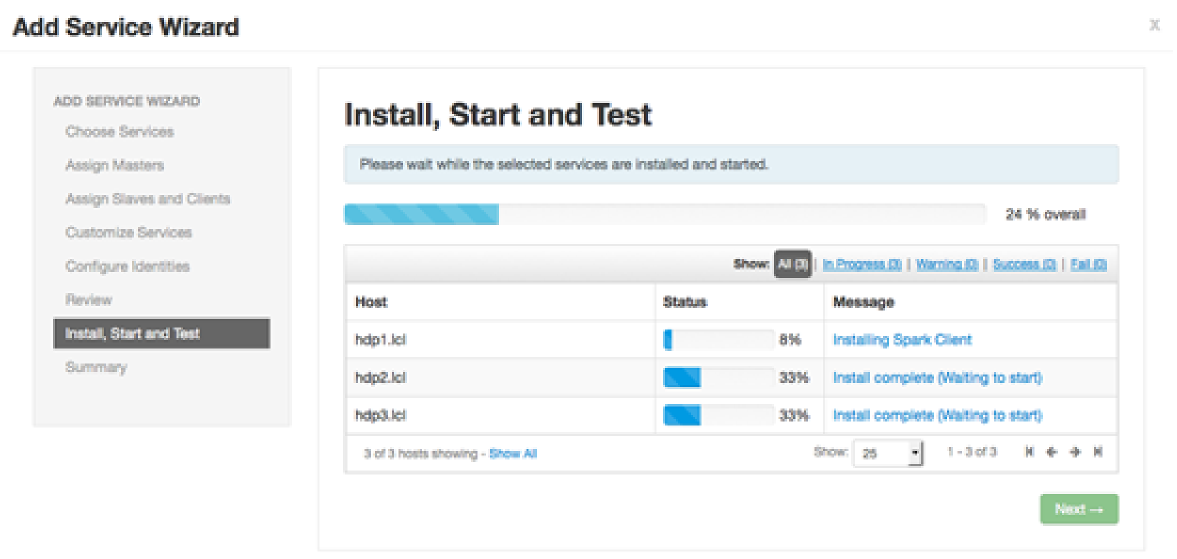
6. Ambari will display the Review screen.

   Important

   On the Review screen, make sure all HDP components are version 2.3.4 or later.

   Click "Deploy" to continue.

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



Install, Start and Test

Please wait while the selected services are installed and started.

24 % overall

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

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

Caution

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

3.1.1. (Optional) Configuring Spark for Hive Access

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

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

Important

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

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

3.1.2. Validating the Spark Installation

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

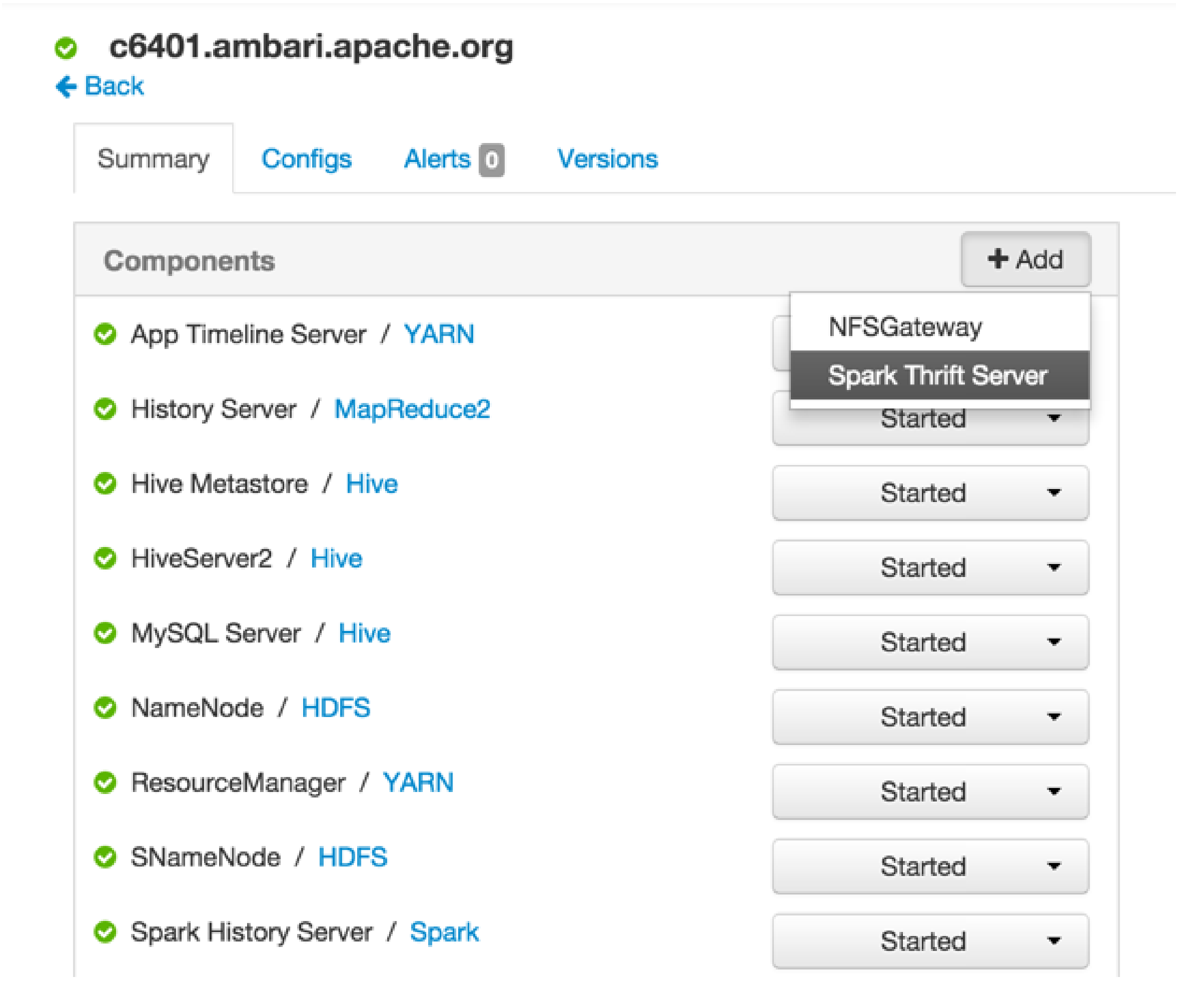
- Spark Pi example
- WordCount example

3.2. Installing the Spark Thrift Server After Deploying Spark

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

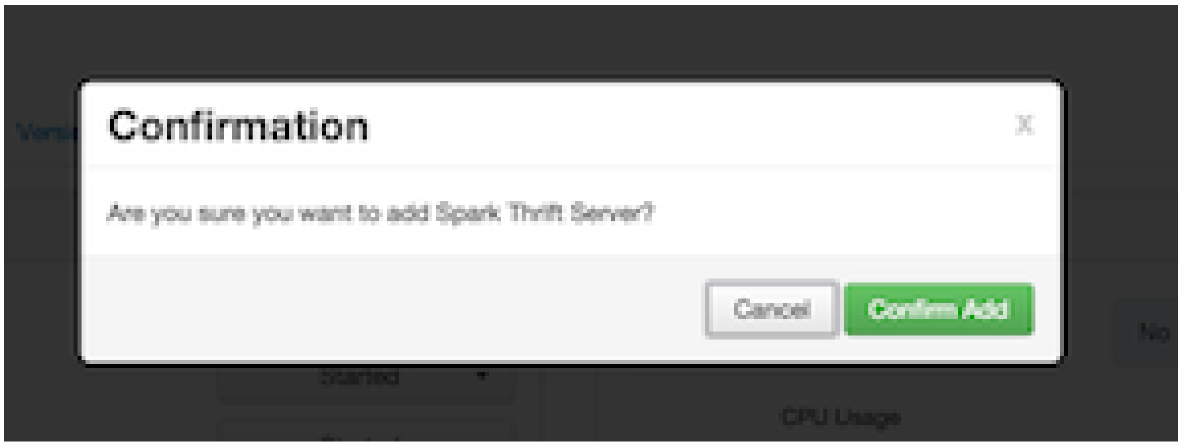
To install the Spark Thrift Server after deploying Spark, add the service to the specified host:

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



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

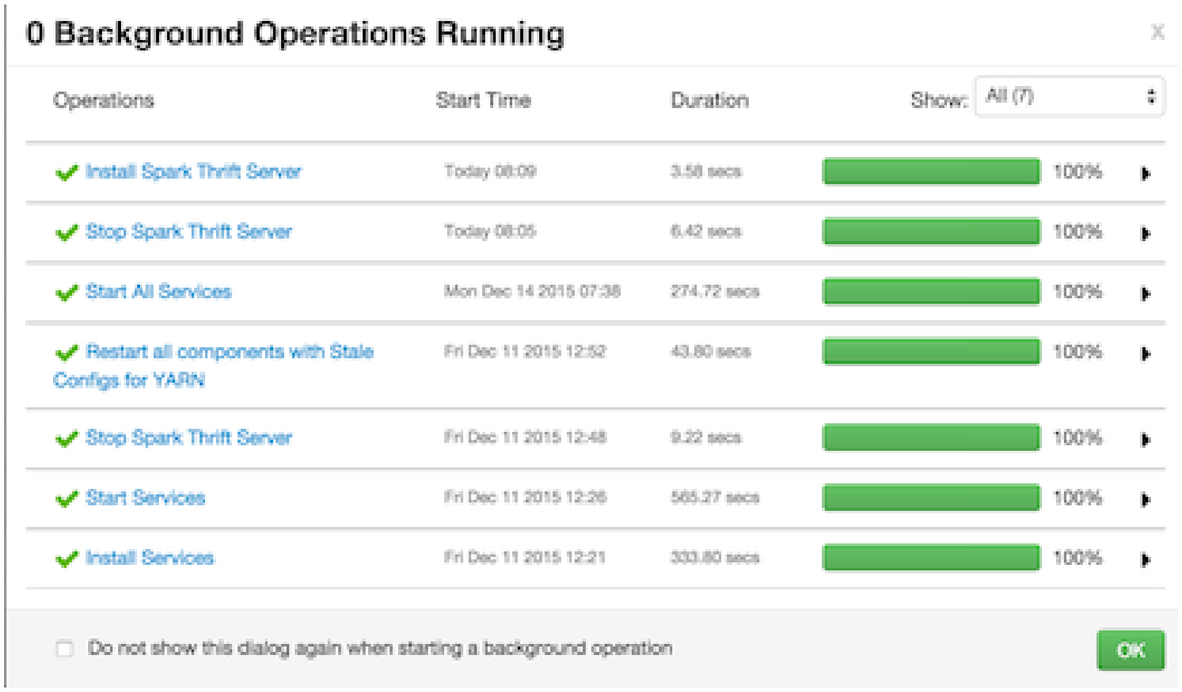
2. Ambari will ask you to confirm the selection:



Confirmation

Are you sure you want to add Spark Thrift Server?

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



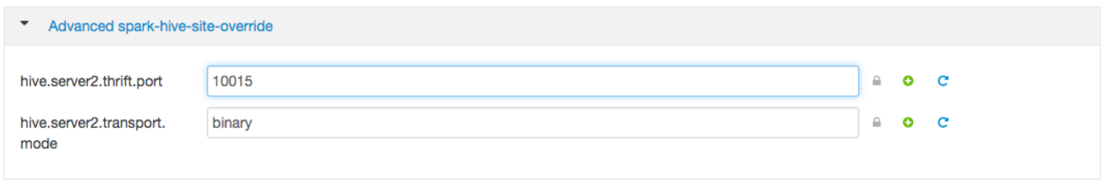
0 Background Operations Running

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

Do not show this dialog again when starting a background operation

3.3. Customizing the Spark Thrift Server Port

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



Advanced spark-hive-site-override

hive.server2.thrift.port	10015
hive.server2.transport.mode	binary

3.4. Configuring Spark for a Kerberos-Enabled Cluster

Spark jobs are submitted to a Hadoop cluster as YARN jobs. When a job is ready to run in a production environment, there are a few additional steps if the cluster is Kerberized:

- The Spark History Server daemon needs a Kerberos account and keytab to run in a Kerberized cluster.
- To submit Spark jobs in a Kerberized cluster, the account (or person) submitting jobs needs a Kerberos account & keytab.

When you enable Kerberos for a Hadoop cluster with Ambari, Ambari sets up 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, refer to "Creating Service Principals and Keytab Files for HDP" in the "Setting up Security for Manual Installs" section of the Non-Ambari Cluster Installation Guide.

Here is an example showing how to create a `spark` principal and keytab file for node `blue1@example.com`:

1. Create a Kerberos service principal:

   ```
   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
   ```

The following example shows user `spark` running the Spark Pi example in a Kerberos-enabled environment:

```
su spark

kinit -kt /etc/security/keytabs/spark.keytab spark/blue1@EXAMPLE.COM

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

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

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

If you are installing the Spark Thrift Server on a Kerberos-secured cluster, the following instructions apply:

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

Note

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

Important

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

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

4. Spark Examples

4.1. Spark Pi Program

To test compute-intensive tasks in Spark, the Pi example calculates pi by “throwing darts” at a circle — it generates points in the unit square ((0,0) to (1,1)) and counts how many points fall within the unit circle within the square. The result approximates pi.

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:

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

   ```
   su spark
   ```

2. Run the Spark Pi job in yarn-client mode:

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

   Commonly-used options include:

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

   The job should complete without errors.

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

   ```
   15/11/10 14:28:35 INFO scheduler.DAGScheduler: Job 0 finished: reduce at
    SparkPi.scala:36, took 1.721177 s
   Pi is roughly 3.141296
   15/11/10 14:28:35 INFO spark.ContextCleaner: Cleaned accumulator 1
   ```

   To view job status in a browser, navigate to the YARN ResourceManager Web UI and view Job History Server information.

4.2. WordCount Program

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

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

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

   ```
   su spark
   ```
3. Upload the input file to HDFS:

   ```
   hadoop fs -copyFromLocal /etc/hadoop/conf/log4j.properties /tmp/
   data
   ```
4. Run the Spark shell:

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

   You should see output similar to the following:

   ```
   bin/spark-shell

   Welcome to
         ____              __
        / __/__  ___ _____/ /__
       _\ \/ _ \/ _ `/ __/  '_/
      /___/ .__/\_,_/_/ /_/\_\   version 1.5.2
         /_/

   Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.
   0_60)
   Type in expressions to have them evaluated.
   Type :help for more information.
   15/12/15 16:28:09 INFO SparkContext: Running Spark version 1.5.2
   15/12/15 16:28:09 INFO SecurityManager: Changing view acls to: root
   ...
   ...
   15/12/15 16:28:14 INFO SparkILoop: Created sql context (with Hive support)..
   SQL context available as sqlContext.

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

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

6. To view `WordCount` output in the scala shell:

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

   To view the full output from within the scala shell:

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

   To view the output using HDFS:

   a. Exit the scala shell.

   b. View WordCount job results:

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

      You should see output similar to the following:

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

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

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

5. Using the Spark DataFrame API

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

1. As user `spark`, upload the `people.txt` file to HDFS:

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

2. Launch the Spark shell:

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

3. At the Spark shell, type the following:

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

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

   ```
   scala> df.show
   15/11/10 11:24:10 INFO YarnScheduler: Removed TaskSet 2.0, whose tasks have
    all completed, from pool

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

5.1. Additional DataFrame API Examples

Here are additional examples of Scala-based DataFrame access, using DataFrame `df` defined in the previous subsection:

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

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

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

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

5.2. Specify Schema Programmatically

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

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

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

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

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

peopleDataFrame.registerTempTable("people")

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

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

This will produce output similar to the following:

```
15/11/10 14:36:49 INFO cluster.YarnScheduler: Removed TaskSet 13.0, whose
 tasks have all completed, from pool
15/11/10 14:36:49 INFO scheduler.DAGScheduler: ResultStage 13 (collect at :33)
 finished in 0.129 s
15/11/10 14:36:49 INFO scheduler.DAGScheduler: Job 10 finished: collect
 at :33, took 0.162827 s
Name: Michael
Name: Andy
Name: Justin
```

6. Accessing ORC Files from Spark

Spark on HDP supports the Optimized Row Columnar ("ORC") file format, a self-describing, type-aware column-based file format that is one of the primary file formats supported in Apache Hive. The columnar format lets the reader read, decompress, and process only the columns that are required for the current query. ORC support in Spark SQL and DataFrame APIs provides fast access to ORC data contained in Hive tables. It supports ACID transactions, snapshot isolation, built-in indexes, and complex types.

6.1. Accessing ORC in Spark

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

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

To start using ORC, define a HiveContext instance:

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

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

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

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

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

6.2. Reading and Writing with ORC

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

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

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

Read the objects back as follows:

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

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

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

6.3. Column Pruning

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

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

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

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

6.4. Predicate Push-down

The columnar nature of the ORC format helps avoid reading unnecessary columns, but it is still possible to read unnecessary rows. In our example, we read all rows where age was between 0 and 100, even though we requested rows where age was less than 15. Such full table scanning is an expensive operation.

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

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

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

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

Important

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

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

6.5. Partition Pruning

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

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

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

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

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

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

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

6.6. DataFrame Support

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

Here's the Scala API translation of the SELECT query above using the DataFrame API

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

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

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

6.7. Additional Resources

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

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

7. Accessing Hive Tables from Spark

This chapter describes how to access Hive data from Spark.

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

7.1. Spark SQL

Spark SQL is a Spark module for structured data processing.

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

An alternate way to access SparkSQL, especially for a Beeline scenario, is through the Spark Thrift Server. For more information about Accessing Spark SQL through the Spark Thrift Server, see "Accessing Spark SQL through the Spark Thrift Server."

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

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

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

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

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

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

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

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

7.1.2. Accessing Spark SQL through the Spark Thrift Server

The Spark Thrift Server provides JDBC access to Spark SQL.

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

1. Enable and start the Spark Thrift Server as specified in "Starting the Spark Thrift Server" (in the Non-Ambari Cluster Installation Guide).

2. Connect to the Thrift Server over Beeline. Launch Beeline from `SPARK_HOME`.

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

3. Issue SQL commands on the Beeline prompt:

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

   Note

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

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

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

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

5. Exit the Thrift Server:

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

6. Stop the Thrift Server:

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

7.2. Hive on Spark

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

The following example reads and writes to HDFS under Hive directories using the built-in UDF `collect_list(col)`, which returns a list of objects with duplicates.

In a production environment this type of operation would run under an account with appropriate HDFS permissions; the following example uses `hdfs` user.

1. Launch the Spark Shell on a YARN cluster:

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

2. As of Spark 1.5, `hiveContext` is created automatically and is named `sqlContext`. If you have existing `hiveContext` code, you can optionally change it to `sqlContext` and remove the context creation code.

3. Create a Hive table:

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

   You should see output similar to the following:

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

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

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

5. Invoke the Hive `collect_list` UDF:

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

8. Using Spark Streaming

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

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

For additional examples, see the Apache GitHub example repositories for Scala, Java, and Python.

9. 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}
```

10. Tuning and Troubleshooting Spark

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

General performance guidelines:

- Minimize shuffle operations where possible.
- Match join strategy (ShuffledHashJoin vs. BroadcastHashJoin) to the table. This requires manual configuration.
- Consider switching from the default serializer to the Kryo serializer to improve performance. This requires manual configuration and class registration.

Note

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

10.1. Hardware Provisioning

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

10.2. Checking Job Status

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

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

- Using Ambari: In the Ambari Services tab, select Spark (in the left column). Click on Quick Links and choose the Spark History Server UI. Ambari will display a list of jobs. Click "App ID" for job details. (By default, the Spark History Server is at `<host>:18080`.)
- Using the YARN Web UI at `http://<host>:8088/proxy/<job_id>/environment/`, view job history and time spent in various stages of the job:

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

- From the command line, list running applications (including the application ID):

  ```
  yarn application –list
  ```

- From the command line, check the application log:

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

The `yarn logs` command prints the contents of all log files from all containers associated with the specified application. You can also view container log files using the HDFS shell or API. For more information, see "Debugging your Application" in the Apache document Running Spark on YARN.

- Use `toDebugString()` on RDD to see a list of RDD's that will be executed. This is useful for understanding how jobs will be executed.
- Use `DataFrame#explain()` to check the query plan if you are using the DataFrame API.

10.3. Configuring Spark JVM Memory Allocation

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

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

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

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

Example

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

4GB x 0.9 X 0. 6 = 2.16 GB

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

10.4. 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.