Hortonworks Data Platform

Spark Guide

(September 30, 2015)

docs.cloudera.com

Hortonworks Data Platform: Spark Guide

Copyright © 2012-2015 Hortonworks, Inc. Some rights reserved.

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

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

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

Except where otherwise noted, this document is licensed under
Creative Commons Attribution ShareAlike 4.0 License.
http://creativecommons.org/licenses/by-sa/4.0/legalcode

Table of Contents

List of Tables

1. Introduction

Hortonworks Data Platform supports Apache Spark 1.4.1, 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
- DataFrame API
- Spark MLLib
- Optimized Row Columnar (ORC) files
- Support for Hive 0.13.1, including the `collect_list` UDF
- The ML Pipeline API in PySpark

The following features are available as technical previews:

- Spark SQL
- Spark Streaming
- Spark Thrift Server
- Dynamic Executor Allocation
- SparkR

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

- Spark Standalone

- GraphX
- Apache Zeppelin
- iPython

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.

Table 1.1. Spark - HDP Version Support

HDP	Ambari	Spark
2.3.2	2.1.2	1.4.1
2.3.0	2.1.1	1.3.1
2.2.9	2.1.1	1.3.1
2.2.8	2.1.1	1.3.1
2.2.6	2.1.1	1.2.1
2.2.4	2.0.1	1.2.1

Table 1.2. Spark Feature Support by Version

Feature	1.2.1	1.3.1	1.4.1
Spark Core	Yes	Yes	Yes
Spark on YARN	Yes	Yes	Yes
Spark on YARN, Kerberos-enabled clusters	Yes	Yes	Yes
Spark History Server	Yes	Yes	Yes
Spark MLLib	Yes	Yes	Yes
Hive 0.1.3, including `collect_list` UDF		Yes	Yes
ML Pipeline API (PySpark)			Yes
DataFrame API		TP	Yes
ORC Files		TP	Yes
Spark SQL	TP	TP	TP
Spark Streaming	TP	TP	TP
Spark Thrift Server		TP	TP
Dynamic Executor Allocation		TP	TP
SparkR			TP
Spark Standalone			
GraphX			

TP: Tech Preview

2. Prerequisites

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

Table 2.1. Prerequisites for Running Spark 1.4.1

Prerequisite	Description
HDP Cluster Stack Version	• 2.3.2 or later
(Optional) Ambari Version	• 2.1.0.0 or later
Software dependencies	• Spark requires HDFS and YARN • PySpark requires Python to be installed on all nodes • SparkR (tech preview) requires R binaries to be installed on all nodes

Note

HDP 2.3.2 supports Spark 1.3.1 and Spark 1.4.1. When you upgrade your cluster to HDP 2.3.2, Spark is automatically upgraded to 1.4.1. If you wish to return to Spark 1.3.1, follow the Spark Manual Downgrade procedure in the HDP 2.3.2 release notes.

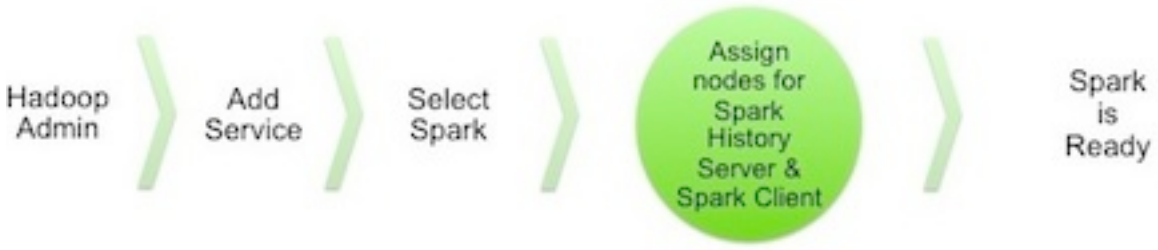
3. Installing and Configuring Spark

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

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

3.1. Installing 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.

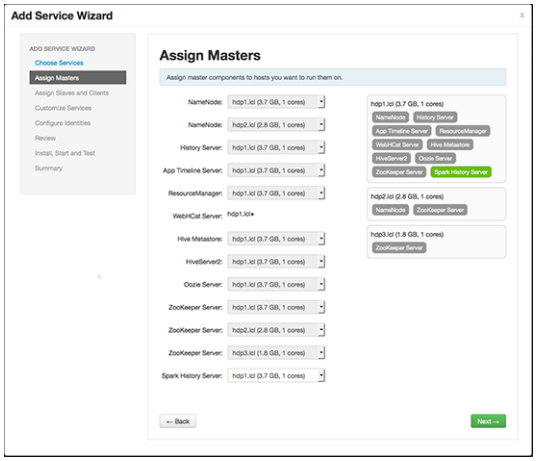
Choose Services

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	1.2.0.2.3	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	1.4.1.2.3	Apache Spark is a fast and general engine for large-scale data processing.

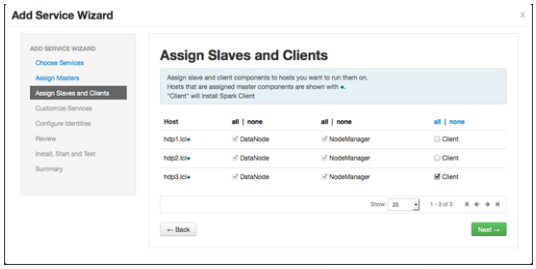
2. On the Assign Masters screen, choose a node for the Spark History Server.

   Click "Next" to continue.



3. On the Assign Slaves and Clients screen, specify the node(s) that will run Spark clients. These nodes will be the nodes from which Spark jobs can be submitted to YARN.

   Click "Next" to continue.



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.

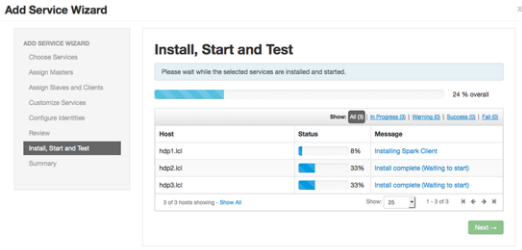
5. Ambari will display the Review screen.

   Important

   On the Review screen, make sure all HDP components are version 2.3.2 or later.

   Click "Deploy" to continue.

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



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

Caution

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

3.2. (Optional) Configuring Spark for Hive Access

If Spark was installed manually (without using Ambari), create and edit a `hive-site.xml` file in the Spark configuration directory:

1. Create the file SPARK_HOME/`conf/hive-site.xml`.

2. Edit the file so that it contains only the `hive.metastore.uris` property. Make sure that `hostname` points to the URI where the Hive Metastore is running.

```
<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>
```

If you installed Spark using Ambari, the `hive-site.xml` file is automatically populated with the correct 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.

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

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
```

3.3.1. 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.)

3.4. Validating the Spark Installation

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

- Spark Pi example
- WordCount example

4. Developing Spark Applications

Apache Spark is designed for fast application development and fast processing. Spark Core is the underlying execution engine; other services such as Spark SQL, MLlib, and Spark Streaming are built on top of the Spark Core.

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

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

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

4.1. Spark Pi Program

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

Here is Python code for the Spark Pi program included with Spark.

To run the Spark Pi example:

1. Log on as a user with HDFS access–for example, your `spark` user, if you defined one, or `hdfs`. (When the job runs, the library is uploaded into HDFS, so the user running the job needs permission to write to HDFS.)

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

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

   su spark
   ```

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

   ```
   ./bin/spark-submit --class org.apache.spark.examples.SparkPi --master yarn-client --num-executors 3 --driver-memory 512m --executor-memory 512m --executor-cores 1 lib/spark-examples*.jar 10
   ```

   Commonly-used options include:

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

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

The job should complete without errors.

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

```
15/08/20 17:33:38 INFO YarnScheduler: Removed TaskSet 0.0, whose tasks have
 all completed, from pool
15/08/20 17:33:38 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.
scala:35, took 10.581715 s
Pi is roughly 3.141104
15/08/20 17:33:38 INFO ContextHandler: stopped o.s.j.s.
ServletContextHandler{/metrics/json,null}
```

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

4.2. WordCount Program

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

The following example submits WordCount code to the scala shell:

1. Select an input file for the Spark WordCount example. You can use any text file as input.
2. Log on as a user with HDFS access–for example, your `spark` user (if you defined one) or `hdfs`. Upload the input file to 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:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.
0_40)
Type in expressions to have them evaluated.
Type :help for more information.
…
15/08/20 13:14:43 INFO metastore: Connected to metastore.
15/08/20 13:14:43 INFO SessionState: No Tez session required at this point.
 hive.execution.engine=mr.
15/08/20 13:14:43 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*
```

5. Using the Spark DataFrame API

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

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

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

2. Launch the Spark shell:

```
cd /usr/hdp/current/spark-client
su spark
./bin/spark-shell --num-executors 2 --executor-memory 512m --master yarn-
client
```

3. At the Spark shell, type the following:

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

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

```
scala> df.show
15/08/20 13:24:10 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:

```
15/08/20 13:29:49 INFO DAGScheduler: ResultStage 11 (collect at <console>:33)
 finished in 0.235 s
15/08/20 13:29:49 INFO YarnScheduler: Removed TaskSet 11.0, whose tasks have
 all completed, from pool
15/08/20 13:29:49 INFO DAGScheduler: Job 9 finished: collect at <console>:33,
 took 0.244865 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")
```

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

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

6.3. Column Pruning

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

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

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

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

6.4. Predicate Push-down

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

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

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

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

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

Important

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

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

6.5. Partition Pruning

When predicate pushdown is not applicable–for example, if all stripes contain records that match the predicate condition–a query with a *WHERE* clause might need to read the

entire data set. This becomes a bottleneck over a large table. Partition pruning is another optimization method; it exploits query semantics to avoid reading large amounts of data unnecessarily.

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

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

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

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

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

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

6.6. DataFrame Support

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

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

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

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

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

6.7. Additional Resources

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

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

7. Adding Libraries to Spark

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

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

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

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

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

8. Using Spark with HDFS

8.1. Specifying Compression

To add a compression library to Spark, use the `--jars` option. The following example adds the LZO compression library:

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

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

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

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

8.2. Accessing HDFS from PySpark: Setting `HADOOP_CONF_DIR`

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

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
[hrt_qa@ip-172-31-42-188 spark]$ pyspark
[hrt_qa@ip-172-31-42-188 spark]$ >>>lines = sc.textFile("hdfs://
ip-172-31-42-188.ec2.internal:8020/tmp/PySparkTest/file-01")
.......
```

If `HADOOP_CONF_DIR` is not set properly, you might see the following error:

Error from secure cluster

```
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. Accessing Hive Tables from Spark

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.

Note

If Spark was installed manually (without using Ambari), see Configuring Spark for Hive Access before accessing Hive data from Spark.

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

1. Launch the Spark Shell on a YARN cluster:

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

2. Create Hive Context:

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

   You should see output similar to the following:

```
…
hiveContext: org.apache.spark.sql.hive.HiveContext = org.apache.spark.sql.
hive.HiveContext@7d9b2e8d
```

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:

```
…
from=org.apache.hadoop.hive.ql.Driver>
15/08/20 13:39:18 INFO PerfLogger: </PERFLOG method=Driver.run
start=1440092357218 end=1440092358126 duration=908
from=org.apache.hadoop.hive.ql.Driver>
res0: 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)
```

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.

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

  ```
  yarn application –list
  ```

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

10.3. Checking Job History

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

- The Spark History Server displays information about Spark jobs that have completed.
  - On an Ambari-managed cluster, in the Ambari Services tab, select Spark. Click on Quick Links and choose the Spark History Server UI. Ambari will display a list of jobs. Click "App ID" for job details.
  - You can access the Spark History Server Web UI directly, at `<host>:18080` (by default).
- The YARN Web UI displays job history and time spent in various stages of the job:

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

  `http://<host>:8088/proxy/<app_id>/stages/`
- To list the contents of all log files from all containers associated with the specified application, check the application log from the command line:

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

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

10.4. Configuring Spark JVM Memory Allocation

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

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

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

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

Example

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

4GB x 0.9 X 0. 6 = 2.16 GB

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

10.5. Configuring YARN Memory Allocation for Spark

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

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

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

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

- RAM (Amount of memory)
- CORES (Number of CPU cores)

Configuring Spark for `yarn-cluster` Deployment Mode

In `yarn-cluster` mode, the Spark driver runs inside an application master process that is managed by YARN on the cluster. The client can stop after initiating the application.

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

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

Configuring Spark for `yarn-client` Deployment Mode

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

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

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

Considerations

When configuring Spark on YARN, consider the following information:

- Executor processes will be not released if the job has not finished, even if they are no longer in use. Therefore, please do not overallocate executors above your estimated requirements.
- Driver memory does not need to be large if the job does not aggregate much data (as with a `collect()` action).
- There are tradeoffs between `num-executors` and `executor-memory`. Large executor memory does not imply better performance, due to JVM garbage collection. Sometimes it is better to configur a larger number of small JVMs than a small number of large JVMs.

10.6. Specifying codec Files

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

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

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

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

For example:

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

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