Apache Spark 3

Developing Apache Spark Applications

Date of Publish: 2018-04-01

http://docs.hortonworks.com

Contents

Introduction

Apache Spark enables you to quickly develop applications and process jobs.

Apache Spark is designed for fast application development and 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.

Depending on your use case, you can extend your use of Spark into several domains, including the following described in this chapter:

- Spark DataFrames
- Spark SQL
- Calling Hive user-defined functions from Spark SQL
- Spark Streaming
- Accessing HBase tables, HDFS files, and ORC data (Hive)
- Using custom libraries

For more information about using Livy to submit Spark jobs, see "Submitting Spark Applications Through Livy" in this guide.

Related Information

Apache Spark Quick Start

Apache Spark Overview

Apache Spark Programming Guide

Using the Spark DataFrame API

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

This section provides examples of DataFrame API use.

To list JSON file contents as a DataFrame:

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

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

2. Launch the Spark shell:

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

3. At the Spark shell, type the following:

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

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

```
scala> df.show
17/03/12 11:24:10 INFO YarnScheduler: Removed TaskSet 2.0, whose tasks
 have all completed, from pool

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

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

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

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

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

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

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

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

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

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

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

peopleDataFrame.registerTempTable("people")

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

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

This produces output similar to the following:

```
17/03/12 14:36:49 INFO cluster.YarnScheduler: Removed TaskSet 13.0, whose
 tasks have all completed, from pool
17/03/12 14:36:49 INFO scheduler.DAGScheduler: ResultStage 13 (collect
 at :33) finished in 0.129 s
17/03/12 14:36:49 INFO scheduler.DAGScheduler: Job 10 finished: collect
 at :33, took 0.162827 s
Name: Michael
Name: Andy
Name: Justin
```

Using Spark SQL

This section provides information about using Spark SQL.

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

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

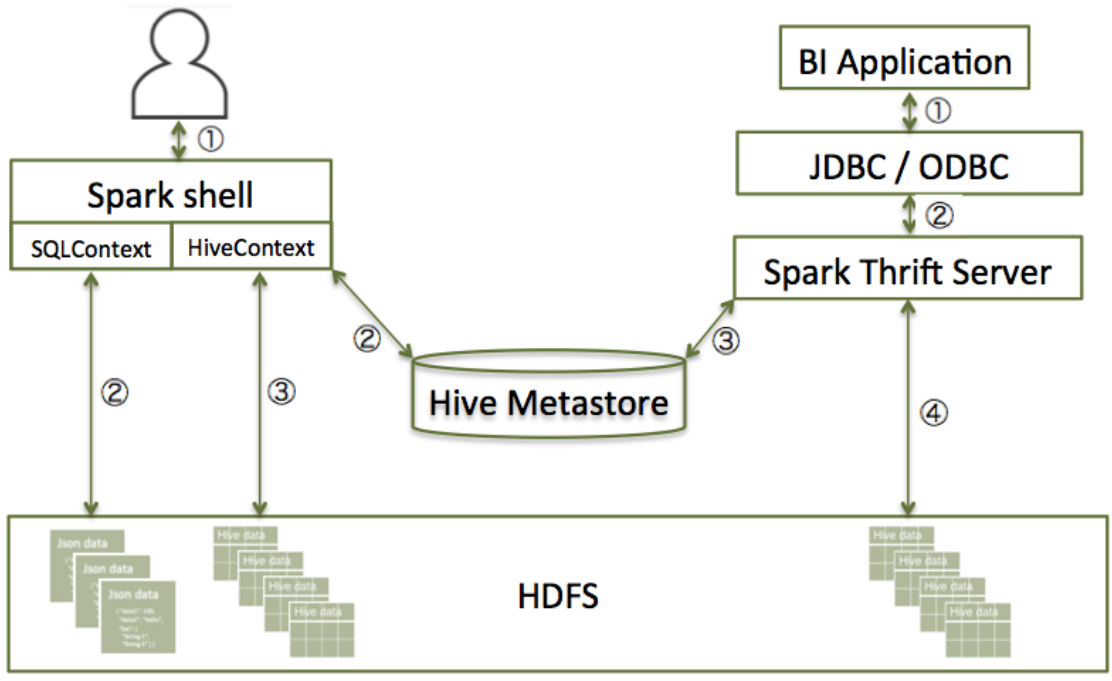
There are two ways to interact with Spark SQL:

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

  For more information about JDBC and ODBC access, see "Accessing Spark SQL through JDBC: Prerequisites" and "Accessing Spark SQL through JDBC and ODBC" in this guide.

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

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



The following subsections describe how to access Spark SQL through the Spark shell, and through JDBC and ODBC.

Related Information

Beeline Command Line Shell

Access Spark SQL through the Spark shell

Use the following steps to acess Spark SQL using the Spark shell.

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

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

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

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

Access Spark SQL through JDBC or ODBC: prerequisites

This section describes prerequisites for accessing Spark SQL through JDBC or ODBC.

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

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

- The Spark Thrift server must be deployed on the cluster. See "Installing and Configuring Spark Over Ambari" in this guide for more information).
- Ensure that SPARK_HOME is defined as your Spark directory:

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

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

- The Spark Thrift server works in YARN client mode only.
- ODBC and JDBC client configurations must match Spark Thrift server configuration parameters. For example, if the Thrift server is configured to listen in binary mode, the client should send binary requests and use HTTP mode when the Thrift server is configured over HTTP.
- All client requests coming to the Spark Thrift server share a SparkContext.

Additional Spark Thrift Server Commands

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

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

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

Related Information

Beeline Command Line Shell

Access Spark SQL through JDBC

Use the following steps to access Spark SQL through JDBC.

To access Spark SQL through JDBC, you need a JDBC URL connection string to supply connection information to the JDBC data source. Connection strings for the Spark SQL JDBC driver have the following format:

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

JDBC Parameter	Description
host	The node hosting the Thrift server

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

Note:

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

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

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

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

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

To access Spark SQL, complete the following steps:

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

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

   b. At the Beeline prompt, connect to the Spark SQL Thrift server with the JDBC connection string:

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

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

   You should see output similar to the following:

   ```
   beeline> !connect jdbc:hive2://localhost:10015
   Connecting to jdbc:hive2://localhost:10015
   Enter username for jdbc:hive2://localhost:10015:
   Enter password for jdbc:hive2://localhost:10015:
   ...
   Connected to: Spark SQL (version 2.0.0)
   Driver: Spark Project Core (version 2.0.0.2.4.0.0-169)
   Transaction isolation: TRANSACTION_REPEATABLE_READ
   0: jdbc:hive2://localhost:10015>
   ```

2. When connected, issue a Spark SQL statement.

The following example executes a SHOW TABLES query:

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

Accessing Spark SQL through ODBC

Use the following steps to access Spark SQL through ODBC.

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

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

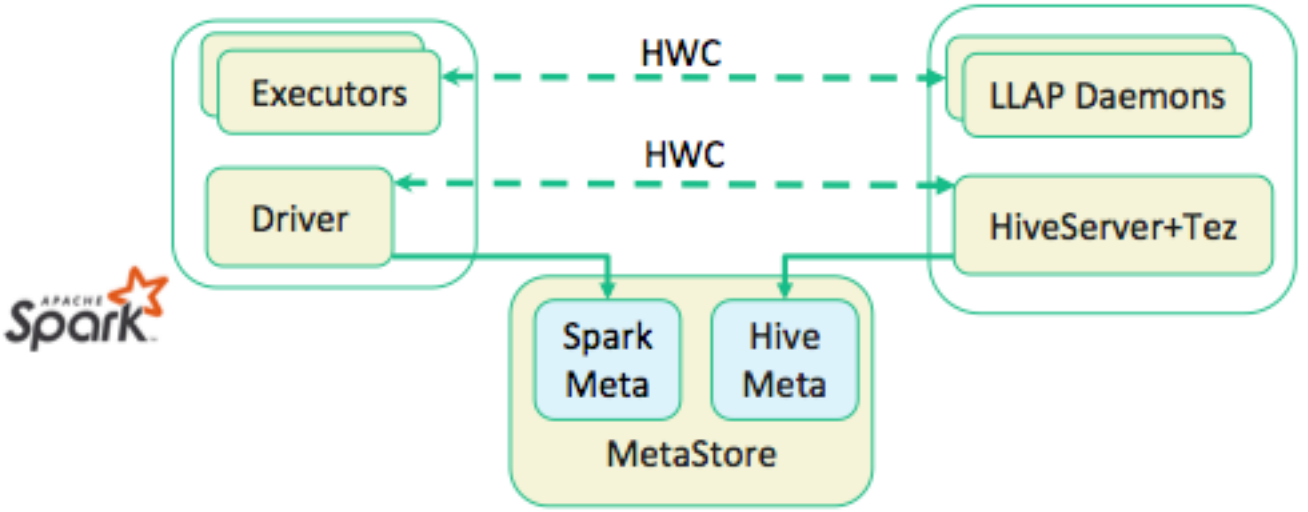
Related Information

Hortonworks Downloads

Using the Hive Warehouse Connector with Spark

This section provides information about using the Hive Warehouse Connector with Apache Spark.

Significant improvements were introduced for Hive in HDP-3.0, including performance and security improvements, as well as ACID compliance. Spark cannot read from or write to ACID tables, so Hive catalogs and the Hive Warehouse Connector (HWC) have been introduced in order to accommodate these improvements.



Updates for HDP-3.0:

- Hive uses the "hive" catalog, and Spark uses the "spark" catalog. No extra configuration steps are required – these catalogs are created automatically when you install or upgrade to HDP-3.0 (in Ambari the Spark metastore.catalog.default property is set to spark in "Advanced spark2-hive-site-override").
- You can use the Hive Warehouse Connector to read and write Spark DataFrames and Streaming DataFrames to and from Apache Hive using low-latency, analytical processing (LLAP). Apache Ranger and the Hive Warehouse Connector now provide fine-grained row and column access control to Spark data stored in Hive.

Related Information

Hive Warehouse Connector for accessing Apache Spark data
Hive Schema Tool

Calling Hive User-Defined Functions

Use the following steps to call Hive user-defined functions.

About this task

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

Using Built-in UDFs

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

1. Launch the Spark Shell on a YARN cluster:

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

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

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

3. Invoke the Hive collect_list UDF:

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

Using Custom UDFs

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

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

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

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

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

2. From spark-shell, define a function:

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

3. From spark-shell, invoke your UDF:

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

Using Spark Streaming

This section provides information on using Spark streaming.

Before you begin

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

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

About this task

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

with high-level functions like map, reduce, join, and window; and send results to file systems, databases, and live dashboards.

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

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

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

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

Important:

Dynamic Resource Allocation does not work with Spark Streaming.

Related Information

Apache Streaming Programming Guide
spark-kafka-0-10-connector
Apache GitHub Scala Streaming Examples
Apache GitHub Java Streaming Examples
Apache GitHub Python Streaming Examples

Building and Running a Secure Spark Streaming Job

Use the following steps to build and run a secure Spark streaming job.

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

- If you are using maven as a compile tool:

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

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

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

```
<dependency>
    <groupId>org.apache.spark</groupId>
```

```
    <artifactId>spark-streaming-kafka_2.10</artifactId>
    <version>2.0.0.2.4.2.0-90</version>
</dependency>

<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>2.0.0.2.4.2.0-90</version>
    <scope>provided</scope>
</dependency>
```

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

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

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

- Instructions for submitting your job depend on whether you used an uber .jar file or not:
  - If you kept the default .jar scope and you can access an external network, use --packages to download dependencies in the runtime library:

```
spark-submit --master yarn-client \
    --num-executors 1 \
    --packages org.apache.spark:spark-streaming-
kafka_2.10:2.0.0.2.4.2.0-90 \
    --repositories http://repo.hortonworks.com/content/repositories/
releases/ \
    --class <user-main-class> \
    <user-application.jar> \
    <user arg lists>
```

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

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

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

For a sample pom.xml file, see "Sample pom.xml file for Spark Streaming with Kafka" in this guide.

Running Spark Streaming Jobs on a Kerberos-Enabled Cluster

Use the following steps to run a Spark Streaming job on a Kerberos-enabled cluster.

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

   This should not be the kafka or spark service account.
2. Generate a keytab for the user.
3. Create a Java Authentication and Authorization Service (JAAS) login configuration file: for example, key.conf.
4. Add configuration settings that specify the user keytab.

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

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

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

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

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

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

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

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

Sample pom.xml File for Spark Streaming with Kafka

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

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

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

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka_2.10</artifactId>
            <version>2.0.0.2.4.2.0-90</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.10</artifactId>
            <version>2.0.0.2.4.2.0-90</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <build>
        <defaultGoal>package</defaultGoal>
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
            </resource>
            <resource>
                <directory>src/test/resources</directory>
                <filtering>true</filtering>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <configuration>
                    <encoding>UTF-8</encoding>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>copy-resources</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
```

```
                <recompileMode>incremental</recompileMode>
                <args>
                    <arg>-target:jvm-1.7</arg>
                </args>
                <javacArgs>
                    <javacArg>-source</javacArg>
                    <javacArg>1.7</javacArg>
                    <javacArg>-target</javacArg>
                    <javacArg>1.7</javacArg>
                </javacArgs>
            </configuration>
            <executions>
                <execution>
                    <id>scala-compile</id>
                    <phase>process-resources</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>scala-test-compile</id>
                    <phase>process-test-resources</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>

            <executions>
                <execution>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>


        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.3</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <filters>
                    <filter>
                        <artifact>*:*</artifact>
                        <excludes>
```

```
                                <exclude>META-INF/*.SF</exclude>
                                <exclude>META-INF/*.DSA</exclude>
                                <exclude>META-INF/*.RSA</exclude>
                            </excludes>
                        </filter>
                    </filters>
                    <finalName>uber-${project.artifactId}-
${project.version}</finalName>
                </configuration>
            </plugin>

        </plugins>

    </build>
</project>
```

HBase Data on Spark with Connectors

This section provides information on streaming HBase data into Spark using connectors.

Software connectors are architectural elements in the cluster that facilitate interaction between different Hadoop components. For real-time and near-real-time data analytics, there are connectors that bridge the gap between the HBase key-value store and complex relational SQL queries that Spark supports. Developers can enrich applications and interactive tools with connectors because connectors allow operations such as complex SQL queries on top of an HBase table inside Spark and table JOINs against data frames.

Important:

The HDP bundle includes two different connectors that extract datasets out of HBase and streams them into Spark:

- Hortonworks Spark-HBase Connector
- RDD-Based Spark-HBase Connector: a connector from Apache HBase that uses resilient distributed datasets (RDDs)

Selecting a Connector

Use the following information to select an HBase connector for Spark.

The two connectors are designed to meet the needs of different workloads. In general, use the Hortonworks Spark-HBase Connector for SparkSQL, DataFrame, and other fixed schema workloads. Use the RDD-Based Spark-HBase Connector for RDDs and other flexible schema workloads.

Hortonworks Spark-HBase Connector

When using the connector developed by Hortonworks, the underlying context is data frame, with support for optimizations such as partition pruning, predicate pushdowns, and scanning. The connector is highly optimized to push down filters into the HBase level, speeding up workload. The tradeoff is limited flexibility because you must specify your schema upfront. The connector leverages the standard Spark DataSource API for query optimization.

The connector is open-sourced for the community. The Hortonworks Spark-HBase Connectorlibrary is available as a downloadable Spark package at https://github.com/hortonworks-spark/shc. The repository readme file contains information about how to use the package with Spark applications.

For more information about the connector, see A Year in Review blog.

RDD-Based Spark-HBase Connector

The RDD-based connector is developed by the Apache community. The connector is designed with full flexibility in mind: you can define schema on read and therefore it is suitable for workloads where schema is undefined at ingestion time. However, the architecture has some tradeoffs when it comes to performance.

Refer to the following table for other factors that might affect your choice of connector, source repos, and code examples.

Table 8.1. Comparison of the Spark-HBase Connectors

	Hortonworks Spark-HBase Connector Connector	**RDD-Based Spark-HBase Connector**
Source	Hortonworks	Apache HBase community
Apache Open Source?	Yes	Yes
Requires a Schema?	Yes: Fixed schema	No: Flexible schema
Suitable Data for Connector	SparkSQL or DataFrame	RDD
Main Repo	shc git repo	Apache hbase-spark git repo
Sample Code for Java	Not available	Apache hbase.git repo
Sample Code for Scala	shc git repo	Apache hbase.git repo

Using the Connector with Apache Phoenix

If you use a Spark-HBase connector in an environment that uses Apache Phoenix as a SQL skin, be aware that both connectors use only HBase .jar files by default. If you want to submit jobs on an HBase cluster with Phoenix enabled, you must include --jars phoenix-server.jar in your spark-submit command. For example:

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

Accessing HDFS Files from Spark

This section contains information on running Spark jobs over HDFS data.

Specifying Compression

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

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

```
rdd.saveAsHadoopFile("/tmp/spark_compressed",
                     "org.apache.hadoop.mapred.TextOutputFormat",

 compressionCodecClass="org.apache.hadoop.io.compress.GzipCodec")
```

For more information about supported compression algorithms, see "Configuring HDFS Compression" in the HDP Data Storage guide.

Accessing HDFS from PySpark

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

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

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

```
                    Error from secure cluster

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

Accessing ORC Data in Hive Tables

Apache Spark on HDP supports the Optimized Row Columnar (ORC) file format, a self-describing, type-aware, column-based file format that is one of the primary file formats supported in Apache Hive.

ORC reduces I/O overhead by accessing only the columns that are required for the current query. It requires significantly fewer seek operations because all columns within a single group of row data (known as a "stripe") are stored together on disk.

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

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

Related Information

ORC File Format

Apache Hive ACID Transactions

Access ORC files from Spark

Use the following steps to access ORC files from Apache Spark.

About this task

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

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()
import spark.implicits._
```

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

Procedure

1. Define Contact and Person data structures:

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

2. Create 100 Person records:

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

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

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

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

4. Read the objects back:

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

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

   ```
   people.createOrReplaceTempView("people")
   ```

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

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

Results

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

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

Related Information

Apache Spark DataFrameReader Methods

Apache Spark DataFrameWriter Methods

Predicate Push-Down Optimization

The columnar nature of the ORC format helps avoid reading unnecessary columns, but it is still possible to read unnecessary rows. The example in this subsection reads all rows in which the age value is between 0 and 100, even

though the query requested rows in which the age value is less than 15 ("...WHERE age < 15"). Such full table scanning is an expensive operation.

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

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

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

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

ORC predicate push-down is enabled by default in Spark SQL.

Load ORC Data into DataFrames Using Predicate Push-Down

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

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

```
val spark = SparkSession.builder().getOrCreate()
val people = spark.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:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
people = spark.read.format("orc").load("peoplePartitioned")
people.filter(people.age < 15).select("name").show()
```

Optimize Queries Using Partition Pruning

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

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

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

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

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

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

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

Enable Vectorized Query Execution

About this task

Vectorized query execution is a feature that greatly reduces the CPU usage for typical query operations such as scans, filters, aggregates, and joins. Vectorization is also implemented for the ORC format. Spark also uses Whole Stage Codegen and this vectorization (for Parquet) since Spark 2.0 (released on July 26, 2016).

Use the following steps to implement the new ORC format and enable vectorization for ORC files with SparkSQL.

Procedure

1. On the Ambari Dashboard, select Spark2 > Configs. For Spark shells and applications, click Custom spark2-defaults, then add the following properties. For the Spark Thrift Server, add these properties under Custom spark2-thrift-sparkconf.
   - spark.sql.orc.enabled=true – Enables the new ORC format to read/write Spark data source tables and files.
   - spark.sql.hive.convertMetastoreOrc=true – Enables the new ORC format to read/write Hive tables.
   - spark.sql.orc.char.enabled=true – Enables the new ORC format to use CHAR types to read Hive tables. By default, STRING types are used for performance reasons. This is an optional configuration for Hive compatibility.
2. Click Save, then restart Spark and any other components that require a restart.

Read Hive ORC Tables

For existing Hive tables, Spark can read them without createOrReplaceTempView. If the table is stored as ORC format (the default), predicate Push-down, partition pruning, and vectorized query execution are also applied according to the configuration.

```
spark.sql("SELECT * FROM hiveTable WHERE age = 20")
```

Additional Resources

- Apache ORC website: https://orc.apache.org/
- ORC performance:
  - http://hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance/
  - https://www.slideshare.net/Hadoop_Summit/performance-update-when-apache-orc-met-apache-spark-81023199
- Get Started with Spark: http://hortonworks.com/hadoop/spark/get-started/

Using Custom Libraries with Spark

Spark comes equipped with a selection of libraries, including Spark SQL, Spark Streaming, and MLlib.

If you want to use a custom library, such as a compression library or Magellan, you can use one of the following two spark-submit script options:

- The --jars option, which transfers associated .jar files to the cluster. Specify a list of comma-separated .jar files.
- The --packages option, which pulls files directly from Spark packages. This approach requires an internet connection.

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

```
spark-submit --driver-memory 1G \
    --executor-memory 1G \
    --master yarn-client \
    --jars /usr/hdp/2.6.0.3-8/hadoop/lib/hadoop-lzo-0.6.0.2.6.0.3-8.jar \
    test_read_write.py
```

For more information about the two options, see Advanced Dependency Management on the Apache Spark "Submitting Applications" web page.

Note:

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

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

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

Related Information

Magellan: Geospatial Analytics on Spark

Submitting Applications: Advanced Dependency Management

Using Spark from R: SparkR

About this task

SparkR is an R package that provides a lightweight front end for using Apache Spark from R, supporting large-scale analytics on Hortonworks Data Platform (HDP) from the R language and environment.

SparkR provides a distributed data frame implementation that supports operations like selection, filtering, and aggregation on large datasets. In addition, SparkR supports distributed machine learning through MLlib.

This section lists prerequisites, followed by a SparkR example. Here are several links to additional information:

Before you begin

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

- R must be installed on all nodes. Commands for installing R are specific to the operating system. For example, for CentOS you would log on as root and run the following command:

```
yum  install R
```

- JAVA_HOME must be set on all nodes.

SparkR Example

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

1. Launch SparkR:

```
su spark
cd /usr/hdp/2.6.0.0-598/spark/bin
./sparkR
```

Output similar to the following displays:

```
Welcome to
    ____              __
   / __/__  ___ _____/ /__
  _\ \/ _ \/ _ `/ __/  '_/
 /___/ .__/\_,_/_/ /_/\_\   version  2.0.0
    /_/

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

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

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

You should see results similar to the following:

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

3. Read the people DataFrame:

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

You should see results similar to the following:

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

Related Information

Integrate SparkR and R for Better Data Science Workflow
Using R Packages with SparkR
Apache SparkR Documentation