Cloudera Runtime 7.2.17

Indexing Data Using Spark-Solr Connector

Date published: 2020-09-16
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

Batch indexing to Solr using SparkApp framework.....4
Create indexer Maven project.....5
Run the spark-submit job.....8

Batch indexing to Solr using SparkApp framework

The Maven project presented here is provided as an example on using the Spark-Solr connector to batch-index data from a CSV file in HDFS into a Solr collection

The Spark-Solr connector framework comes bundled with Cloudera Search. It enables Extraction, Transformation, and Loading (ETL) of large datasets to Solr. You can use spark-submit with a Spark job to batch index HDFS files into Solr. For this you need to create a class which implements the SparkApp.RDDProcesor interface.

To use the SparkApp framework, you must create a Maven project with the spark-solr dependency.

```
<dependencies>
        <dependency>
            <groupId>com.lucidworks.spark</groupId>
            <artifactId>spark-solr</artifactId>
            <version>{latest_version}</version>
        </dependency>
    </dependencies>
```

This project needs to have at a minimum one class, which implements the SparkApp.RDDProcessor interface. This class can be written either in Java or Scala. This documentation uses a Java class to demonstrate how to use the framework.

The SparkApp.RDDProcessor interface has three functions which need to be overwritten:

- getName()
- getOptions()
- run

getName()

The getName() function returns the short name of the application as a string. When running your spark-submit job, this is the name you pass as a parameter allowing the job to find your class.

public String getName() { return "csv"; }

getOptions()

In the getOptions() function you may specify parameters that are specific to your application. Certain parameters, for example zkHost, collection, or batchSize are present by default. You do not need to specify those here.

```
public Option[] getOptions() {
     return new Option[]{
          OptionBuilder
               .withArgName("PATH").hasArgs()
               .isRequired(true)
               .withDescription("Path to the CSV file to index")
               .create("csvPath")
     };
}
```

run

The run function is the core of the application. This returns an integer, and has two parameters, SparkConf and Comm andLine.

You can create a JavaSparkContext class with the use of the SparkConf parameter, and use this to open the CSV file as a JavaRDD<String> class:

```
JavaSparkContext jsc = new JavaSparkContext(conf);
JavaRDD<String> textFile = jsc.textFile(cli.getOptionValue("csvPath"));
```

You now have to convert these String values to SolrInputDocument, and create a JavaRDD class . To achieve this the script uses a custom-made map function which splits the CSV file upon commas and adds the records to the SolrInpu tDocument document. You must specify the schema used in the CSV file in advance.

```
JavaRDD<SolrInputDocument> jrdd = textFile.map(new Function<String, SolrInpu
tDocument>() {
            @Override
            public SolrInputDocument call(String line) throws Exception {
                 SolrInputDocument doc = new SolrInputDocument();
                 String[] row = line.split(",");

                 if (row.length != schema.length)
                       return null;
                 for (int i=0;i<schema.length;i++){
                       doc.setField(schema[i], row[i]);
                 }
                 return doc;
           }
});
```

After this, the script requires the CommandLine instance options to perform indexing:

```
String zkhost = cli.getOptionValue("zkHost", "localhost:9983");
String collection = cli.getOptionValue("collection", "collection1");
int batchSize = Integer.parseInt(cli.getOptionValue("batchSize", "100"));
```

Finally, the job indexes data into the Solr cluster:

```
SolrSupport.indexDocs(zkhost, collection, batchSize, jrdd.rdd());
```

If the function is succesfully called, 0 is returned.

Create indexer Maven project

As a prerequisite to using the SparkApp framework, you need to create a Maven project with the Spark-Solr dependency and at least one class, implementing the SparkApp.RDDProcessor interface.

About this task

You can either write a Java or a Scala class implementation. The examples show implementation with a Java class.

Procedure

1. Create the indexer Maven project.
2. Edit the .pom file, add the following spark-solr-dependency:

```
<dependencies>
        <dependency>
            <groupId>com.lucidworks.spark</groupId>
            <artifactId>spark-solr</artifactId>
            <version>[***latest version***]</version>
            <classifier>shaded</classifier>
```

```
        </dependency>
    </dependencies>
```

Replace *[***latest version***]* with an actual version number.

For example:

```
<?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://m
aven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>indexer</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
    <repositories>
      <repository>
        <id>cdh.repo</id>
        <url>https://repository.cloudera.com/artifactory/cloudera-repos/</
url>
        <name>Cloudera Repositories</name>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>com.lucidworks.spark</groupId>
            <artifactId>spark-solr</artifactId>
            <version>3.9.0.7.2.2.0-244</version>
            <classifier>shaded</classifier>
        </dependency>
    </dependencies>

</project>
```

3. Create a CSVIndexer.java file that implements the SparkApp.RDDProcessor interface in the main/java/com/lucidworks/spark directory.
   For example:

```
package com.lucidworks.spark;

import com.lucidworks.spark.SparkApp;
import com.lucidworks.spark.util.SolrSupport;
import shaded.apache.commons.cli.CommandLine;
import shaded.apache.commons.cli.Option;
import shaded.apache.commons.cli.OptionBuilder;
import org.apache.solr.common.SolrInputDocument;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;

public class CSVIndexer implements SparkApp.RDDProcessor {
```

```
    @Override
    public String getName() {
        return "csv";
    }

    @Override
    public Option[] getOptions() {
        return new Option[]{
                OptionBuilder
                        .withArgName("PATH").hasArgs()
                        .isRequired(true)
                        .withDescription("Path to the CSV file to index")
                        .create("csvPath")
        };
    }

    private String[] schema = "vendor_id,pickup_datetime,dropoff_datetim
e,passenger_count,trip_distance,pickup_longitude,pickup_latitude,rate_co
de_id,store_and_fwd_flag,dropoff_longitude,dropoff_latitude,payment_type
,fare_amount,extra,mta_tax,tip_amount,tolls_amount,improvement_surcharge
,total_amount".split(",");

    @Override
    public int run(SparkConf conf, CommandLine cli) throws Exception {
        JavaSparkContext jsc = new JavaSparkContext(conf);
        JavaRDD<String> textFile = jsc.textFile(cli.getOptionValue("csvP
ath"));
        JavaRDD<SolrInputDocument> jrdd = textFile.map(new Function<Stri
ng, SolrInputDocument>() {
            @Override
            public SolrInputDocument call(String line) throws Exception {
                SolrInputDocument doc = new SolrInputDocument();
                String[] row = line.split(",");
                if (row.length != schema.length)
                    return null;
                for (int i=0;i<schema.length;i++){
                    doc.setField(schema[i], row[i]);
                }
                return doc;
            }
        });

        String zkhost = cli.getOptionValue("zkHost", "localhost:9983");
        String collection = cli.getOptionValue("collection", "collection
1");
        int batchSize = Integer.parseInt(cli.getOptionValue("batchSize",
 "100"));

        SolrSupport.indexDocs(zkhost, collection, batchSize, jrdd.rdd());

        return 0;
    }
}
```

4. Create a JAR file:

```
mvn clean install
```

The indexer.jar file is created.

Run the spark-submit job

After you create an indexer.jar file, you need to run a spark-submit job on a Solr worker node to index your input file.

Before you begin

- You have prepared the indexer.jar file and it is available on your local machine.
- A DDE Data Hub cluster is up and running.
- You have sufficient rights to SSH into one of the cluster nodes.
- Your user has a role assigned that provides 'write' rights on S3.
- You have retrieved the keytab for your environment.

Procedure

1. SSH to one of the worker nodes in your Data Hub cluster.
2. Copy your keytab file to the working directory:

   ```
   scp <keytab> <user>@<IP_OF_WORKER_NODE>:/<PATH/TO/WORKING/DIRECTORY>
   ```

   For example:

   ```
   scp sampleuser.keytab sampleuser@1.1.1.1:/tmp
   ```

3. Create a JAAS file with the following content:

   ```
   Client {
     com.sun.security.auth.module.Krb5LoginModule required
     useKeyTab=true
     useTicketCache=false
     doNotPrompt=true
     debug=true
     keyTab="sampleuser.keytab"
     principal="sampleuser@EXAMPLE.COM";
   };
   ```

   Replace *sampleuser@EXAMPLE.COM* with your user principal.
4. Copy the indexer JAR file to the working directory:

   ```
   scp <indexer>.jar <user>@<IP_OF_WORKER_NODE>:/<PATH/TO/WORKING/DIRECTORY>
   ```

   For example:

   ```
   scp indexer-1.0-SNAPSHOT.jar sampleuser@1.1.1.1:/tmp
   ```

5. Copy the input CSV file to the working directory:

   ```
   scp <INPUT_FILE> <user>@<IP_OF_WORKER_NODE>:/<PATH/TO/WORKING/DIRECTORY>
   ```

   For example:

   ```
   scp nyc_yellow_taxi_sample_1k.csv sampleuser@1.1.1.1:/tmp
   ```

6. Add the input file to HDFS:

```
hdfs dfs -put <INPUT_FILE>
```

For example:

```
hdfs dfs -put nyc_yellow_taxi_sample_1k.csv
```

7. Create a Solr collection:

```
solrctl config --create <configName> <baseConfige> -p immutable=false
```

```
solrctl collection --create <collectionName> -s <numShards> -
c <collectionConfName>
```

For example:

```
solrctl config --create testConfig managedTemplate -p immutable=false
solrctl collection --create testcollection -s 2 -c testConfig
```

8. Submit your spark job:

```
spark-submit --jars /opt/cloudera/parcels/CDH/jars/spark-solr-*-shaded.jar
 --files <KEYTAB>,<JAAS_CONF_FILE> --name <SPARK_JOB_NAME> --driver-java-
options="-Djavax.net.ssl.trustStoreType=<TRUSTSTORE_TYPE> -Djavax.net.ssl
.trustStore=<ABSOLUTE/PATH/TO/TRUSTSTORE/FILE> -Djavax.net.ssl.trustSt
orePassword=" --class com.lucidworks.spark.SparkApp <INDEXER_JAR> csv -
```

```
zkHost <ZOOKEEPER_ENSEMBLE> -collection <TARGET_SOLR_COLLECTION> -csvPat
h <INPUT_CSV_FILE> -solrJaasAuthConfig=<JAAS_CONF_FILE>
```

Note:

To use Spark-Solr with an SSL-enabled ZooKeeper, you need to add extra Java options. Add the following JVM parameters as both driver and executor Java options:

```
-Dzookeeper.client.secure=true \
-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocket
Netty \
-Dzookeeper.ssl.trustStore.location=[***TRUSTSTORE_LOCATION***] \
-Dzookeeper.ssl.trustStore.password=[***TRUSTSTORE_PASSWORD***] \
```

Replace *[***TRUSTSTORE_LOCATION***]* with the Java truststore location and *[***TRUSTSTORE_PASSWORD***]* with the Java truststore password.

Replace
spark-solr-*-shaded.jar

with the name of the shaded.jar file under /opt/cloudera/parcels/CDH/jars/

<KEYTAB>

with the keytab file of your user

<JAAS_CONF_FILE>

with the JAAS file you created

<SPARK_JOB_NAME>

with the name of the job you want to run

<TRUSTSTORE_TYPE>

with the type of the truststore used. If you use the default jks type, you do not need to specify -Dja vax.net.ssl.trustStoreType. In every other case it is mandatory.

<ABSOLUTE/PATH/TO/TRUSTSTORE/FILE>

with the absolute path to the truststore file

<INDEXER_JAR>

with the indexer.jar file you created

<ZOOKEEPER_ENSEMBLE>

with the address of the ZooKeeper ensemble used by the Solr cluster.

<TARGET_SOLR_COLLECTION>

with the name of the Solr collection you created

<INPUT_CSV_FILE>

with the name of the file that you want to index into the *<TARGET_SOLR_COLLECTION>*

For example:

```
spark-submit --jars /opt/cloudera/parcels/CDH/jars/spark-solr-3.9.0.7.2.
2.0-218-shaded.jar --files sampleuser.keytab,jaas-client.conf --name spa
rk-solr --driver-java-options="-Djavax.net.ssl.trustStoreType=bcfks -Dja
vax.net.ssl.trustStore=/var/lib/cloudera-scm-agent/agent-cert/cm-auto-gl
obal_truststore.jks -Djavax.net.ssl.trustStorePassword=" --class com.luc
idworks.spark.SparkApp indexer-1.0-SNAPSHOT.jar csv -zkHost sampleuser-l
eader2.sampleuser.work:2181,sampleuser.work:2181,sampleuser-master7.work
:2181/solr-dde -collection testcollection -csvPath nyc_yellow_taxi_sampl
e_1k.csv -solrJaasAuthConfig=jaas-client.conf
```

Related Information

Retrieving keytabs for workload users
Generating collection configuration using configs
Creating a Solr collection
Enabling ZooKeeper SSL/TLS for Solr