Cloudera Runtime 7.2.2

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 in spark-submit

You may use spark-submit with your spark job to batch index HDFS files into Solr. For this you need to create a class which implements the SparkApp.RDDProcesor interface. This allows ETL of large datasets to Solr, exploiting Spark's robust data processing capabillities.

To use the SparkApp framework, you need to 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 least one class, which implements the SparkApp.RDDProcessor. This class can either be a Java or a Scala class. This documentation uses a Java class to demonstrate how to use the framework.

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

- getName()
- getOptions()
- run

getName()

The getName() function returns a string, the short name of the application. When running your spark-submit job, this is the name you pass as a parameter to make the job 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, a SparkConf instance and CommandLine instance.

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

```
JavaSparkContext jsc = new JavaSparkContext(conf);
```

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

You now have to convert these Strings to SolrInputDocument, and create a JavaRDD of them. To achieve this the script uses a custom made map function which splits the CSV file upon commas and adds the records to the SolrInputDocument. For this step to work, you have to 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 asks the CommandLine instance for the options it needs 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, it indexes data into the Solr cluster:

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

If the function was 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 - 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>
```

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>http://nexus-private.hortonworks.com/nexus/content/groups/
public/</url>
        <name>Cloudera Repositories</name>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>com.lucidworks.spark</groupId>
            <artifactId>spark-solr</artifactId>
            <version>3.9.0.7.2.2.0-218</version>
            <classifier>shaded</classifier>
        </dependency>
    </dependencies>
</project>
```

3. Create a CSVIndexer.java file that implements the SparkApp.RDDProcessor interface.
   For example:

```
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_dateti
me,passenger_count,trip_distance,pickup_longitude,pickup_latitude,rate_c
ode_id,store_and_fwd_flag,dropoff_longitude,dropoff_latitude,payment_typ
e,fare_amount,extra,mta_tax,tip_amount,tolls_amount,improvement_surcharg
e,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("csv
Path"));
        JavaRDD<SolrInputDocument> jrdd = textFile.map(new Function<Str
ing, 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", "collectio
n1");
        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

Once you have created 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 (Tech Preview) 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 /<PATH/TO/WORKING/DIRECTORY>/SPARK-SOLR-*-SHADED.JAR
 --files <KEYTAB>,<JAAS_CONF_FILE> --name <SPARK_JOB_NAME> --conf "spa
rk.executor.extraJavaOptions=-Djavax.net.ssl.trustStore=<ABSOLUT/PATH/
TO/TRUSTSTORE/FILE> -Djavax.net.ssl.trustStorePassword=" --driver-j
ava-options="-Djavax.net.ssl.trustStore=<ABSOLUT/PATH/TO/TRUSTSTORE/
FILE> -Djavax.net.ssl.trustStorePassword=" --class com.lucidworks.spa
rk.SparkApp <INDEXER_JAR> csv -zkHost <ZOOKEEPER_ENSEMBLE> -collec
tion <TARGET_SOLR_COLLECTION> -csvPath <INPUT_CSV_FILE> -solrJaasAuthCo
nfig=<JAAS_CONF_FILE>
```

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 --conf "spark.executor.extraJavaOptions=-Djavax.net.ssl.trustSto
re=/var/lib/cloudera-scm-agent/agent-cert/cm-auto-global_truststore.jks
-Djavax.net.ssl.trustStorePassword=" --driver-java-options="-Djavax.net.
ssl.trustStore=/var/lib/cloudera-scm-agent/agent-cert/cm-auto-global_tru
ststore.jks -Djavax.net.ssl.trustStorePassword=" --class com.lucidworks.
spark.SparkApp indexer-1.0-SNAPSHOT.jar csv -zkHost sampleuser-leader2.s
ampleuser.work:2181,sampleuser.work:2181,sampleuser-master7.work:2181/so
lr-dde -collection testcollection -csvPath nyc_yellow_taxi_sample_1k.csv -
solrJaasAuthConfig=jaas-client.conf
```