ISIT312 Big Data Management

Session 2, 2021

Exercise 9

Advanced Operations in Spark

In this exercise, you will learn some advanced operations in Spark, including integration of Spark with Hive and HBase as well as Spark Streaming.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

In this lab, we use *Terminal* but not Zeppelin.

(1) How to start Hadoop, Hive, HBase and Spark services?

Start the five essential Hadoop services: resourcemanager, nodemanager, namenode, datanode and historyserver in a Terminal window (see a previous lab).

First, start the Hive's metastore service. In a Terminal window, process:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running.

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

Next, start hiveserver2. Open a ***new*** Terminal window and process:

```
$HIVE_HOME/bin/hiveserver2
```

Still next, start the HBase service. Open another ***new*** Terminal window and process:

```
$HBASE_HOME/bin/start-hbase.sh
```

[Troubleshooting] Some HBase services many exit due to idle time. Use the JPS command to check. If they exit, re-start the HBase service by repeating the above step.

Finally, configurate and start Spark. Process the following in your terminal window:

```
export SPARK_CONF_DIR=$SPARK_HOME/remote-hive-metastore-conf
```

The above shell command tells Spark to use the configuration file in the folder `remote-hive-metastore-conf` in `$SPARK_HOME` instead of using the default one. Then, **in the same terminal window**, process:

```
$SPARK_HOME/bin/spark-shell --master local[*] --packages
com.hortonworks:shc-core:1.1.0-2.1-s_2.11 --repositories
http://repo.hortonworks.com/content/groups/public
```

or

```
$SPARK_HOME/bin/spark-shell --master local[*] --packages
com.hortonworks:shc-core:1.1.0-2.1-s_2.11 --repositories
https://repo.hortonworks.com/content/repositories/releases
```

The last two options in both the above commands indicate that we use a third-party connector, namely the *Hortonworks Spark-HBase-Connector (shc)* to connect Spark and HBase.

If you don't use HBase in your operations, just process:

```
$SPARK_HOME/bin/spark-shell --master local[*]
```

(2) Spark DataFrame and Dataset operations

Download a flight-dataset `2015-summary.json` and load it to HDFS.

```
$HADOOP_HOME/bin/hadoop fs -put 2015-summary.json 2015-summary.json
$HADOOP_HOME/bin/hadoop fs -ls
```

Input the following DataFrame/Dataset operations.

```
> val flightData = "<your hdfs path>/2015-summary.json"
> val df = spark.read.format("json").load(flightData)
// Or, equivalently,
> val df = spark.read.json(flightData)

> df.createOrReplaceTempView("dfTable")
> df.printSchema()

> df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

> import org.apache.spark.sql.functions.{expr, col}
> df.select(col("DEST_COUNTRY_NAME"),
    expr("ORIGIN_COUNTRY_NAME")).show(2)

> df.select(col("DEST_COUNTRY_NAME").alias("Destination"),
  expr("ORIGIN_COUNTRY_NAME").alias("Origin")).show(2)
> df.filter(col("count") < 2).show(2)

> import org.apache.spark.sql.functions.{desc, asc}
> df.orderBy(desc("count"), asc("DEST_COUNTRY_NAME")).show(2)
> df.limit(5).count()
```

```
> import org.apache.spark.sql.functions.{count,sum}

> val df1 = df.groupBy(col("DEST_COUNTRY_NAME"))
.agg(
      count(col("ORIGIN_COUNTRY_NAME")).
      alias("#CountriesWithFlightsTo"),
  sum(col("count")).alias("#AllFlightsTo")
).orderBy(desc("#AllFlightsTo"))
> df1.show(2)

> import spark.implicits._

> val person = Seq(
  (0, "Bill Chambers", 0, Seq(100)),
  (1, "Matei Zaharia", 1, Seq(500, 250, 100)),
  (2, "Michael Armbrust", 1, Seq(250, 100)))
  .toDF("id", "name", "graduate_program", "spark_status")

> val graduateProgram = Seq(
  (0, "Masters", "School of Information", "UC Berkeley"),
  (2, "Masters", "EECS", "UC Berkeley"),
  (1, "Ph.D.", "EECS", "UC Berkeley"))
  .toDF("id", "degree", "department", "school")

> val joinExpression = person
  .col("graduate_program") === graduateProgram.col("id")

> person.join(graduateProgram, joinExpression, "inner").show()
```

You can save DataFrames as different data formats (e.g., JSON, Parquet, CSV, JDBC, etc.). For example:

```
> df.write.json("<your_file_name>")
```

(3) The `:paste` command and Scala script execution

The `:paste` command allows you to paste multiple lines of Scala codes into the Spark shell. Process:

```
> :paste
// Entering paste mode (ctrl-D to finish)
```

With the `:paste` command, you can also run a script of Scala code. For example, open a plain document in Text Editor (`gedit`) and copy some lines to it, such as:

```
import spark.implicits._
val person = Seq(
  (0, "Bill Chambers", 0, Seq(100)),
  (1, "Matei Zaharia", 1, Seq(500, 250, 100)),
  (2, "Michael Armbrust", 1, Seq(250, 100)))
  .toDF("id", "name", "graduate_program", "spark_status")
```

Save the document in `Desktop` with a name `myscript.sc`. Then, process in the Spark shell:

```
> :paste /home/bigdata/Desktop/myscript.sc
```

(4) Spark and Hive

Spark has native support to Hive. In particular, Spark can use Hive's metastore service to manage data shared with multiple users currently.

We can retrieve the Hive tables (if some tables are saved in Hive):

```
> import spark.sql
> sql("show tables").show()
 // you will also see all of your Hive tables here.
```

We can also save a table to the database of Hive.

```
> import org.apache.spark.sql.{SaveMode}

> val df = spark.read.format("json").load(flightData).limit(20)
> df.write
    .mode(SaveMode.Overwrite)
    .saveAsTable("flight_data")
> sql("select * from flight_data").show()
```

You should be able to query the same table in the Hive's Beeline shell:

```
$HIVE_HOME/bin/beeline
> !connect jdbc:hive2://localhost:10000
> show tables;
// You should see the table you created before.
```

You can also develop a self-contained application to connect to Hive. See the lecture note for details.

(5) Spark and HBase

Start the HBase shell:

```
$HBASE_HOME/bin/hbase shell
```

Create a `Contacts` table with the column families `Personal` and `Office` in HBase shell:

```
> create 'Contacts', 'Personal', 'Office'
```

Load a few rows of data into HBase:

```
> put 'Contacts', '1000', 'Personal:Name', 'John Dole'
> put 'Contacts', '1000', 'Personal:Phone', '1-425-000-0001'
> put 'Contacts', '1000', 'Office:Phone', '1-425-000-0002'
> put 'Contacts', '1000', 'Office:Address', '1111 San Gabriel Dr.'
```

Then in Spark-shell, process:

```
> def catalog = s"""{
    |"table":{"namespace":"default", "name":"Contacts"},
    |"rowkey":"key",
    |"columns":{
    |"recordID":{"cf":"rowkey", "col":"key", "type":"string"},
```

```
    |"officeAddress":{"cf":"Office", "col":"Address", "type":"string"},
    |"officePhone":{"cf":"Office", "col":"Phone", "type":"string"},
    |"personalName":{"cf":"Personal", "col":"Name", "type":"string"},
    |"personalPhone":{"cf":"Personal", "col":"Phone", "type":"string"}
    |}
    |}""".stripMargin
```

The `catalog` function defined above corresponds to the structure of the HBbase table `Contacts` we have created. We can make HBase queries by Spark's Structured API:

```
> import org.apache.spark.sql.execution.datasources.hbase._
> def withCatalog(cat: String) = {
  spark.sqlContext
    .read
    .options(Map(HBaseTableCatalog.tableCatalog->cat))
    .format(
    "org.apache.spark.sql.execution.datasources.hbase"
    )
    .load()
  }

// Query
> def dfHbase = withCatalog(catalog)
> dfHbase.show()
```

Besides reading, we also can write data into HBase tables. To this end, we first create a Dataset:

```
> case class ContactRecord(
    recordID: String, // rowkey
    officeAddress: String,
    officePhone: String,
    personalName: String,
    personalPhone: String )

> import spark.implicits._
  // Insert records
> val newContact1 = ContactRecord("16891", "40 Ellis St.",
     "674-555-0110", "John Jackson","230-555-0194")
> val newContact2 = ContactRecord("2234", "8 Church St.",
     "234-325-23", "Ale Kole","")
> val newDataDS = Seq(newContact1, newContact2).toDS()
```

We can save the Dataset object to HBase:

```
> newDataDS.write
  .options(Map(
   HBaseTableCatalog.tableCatalog -> catalog,
    HBaseTableCatalog.newTable -> "5" //this param must be set
  )).format(
   "org.apache.spark.sql.execution.datasources.hbase"
   ).save()
> dfHbase.show()
```

Stop the HBase service when you finish with it:

```
$HBASE_HOME/bin/stop-hbase.sh
```

(5) Spark Structured Streaming

In the following, we stimulate a streaming version of the wordcount application. We use the Unix NetCat utility to generate the input strings. Open a new terminal window and process:

```
nc -lk 9999
```

Input the following code in Spark-shell, which tells Spark to listen to Port 9999 in the localhost.

```
> val lines = spark.readStream
  .format("socket") // socket source
  .option("host", "localhost") // listen to the localhost
  .option("port", 9999) // and port 9999
  .load()
```

Then, develop the wordcount application and start the streaming.

```
> import spark.implicits._
> val words = lines.as[String].flatMap(_.split(" "))
> val wordCounts = words.groupBy("value").count()
> val query = wordCounts.writeStream
  .outputMode("complete") // accumulate counting result of the
stream
  .format("console") // use the console as the sink
  .start()
```

There is no output in Spark-shell yet, because there is no input generated on Port 9999. Input some words in your NetCat terminal window, such as:

```
apache spark
apache hadoop
spark streaming
...
```

You should see return batches in Spark-shell.

End of exercise 9