ISIT312/ISIT912 Big Data Management

Spring 2023

Spark Practice II

In this practice, you will learn some advanced operations on Spark, including the Integration of Spark with Hive and HBase as well as Spark's Structured Streaming.

Important: In this lab, we use Terminal.

In the following specification, if the **prompt symbol ">"** is presented, the command is processed in the **Spark shell or HBase shell**; without the prompt symbol ">", the command is processed a standard Terminal window session.

Laboratory Instructions.

(0) Start the Hadoop, Hive, HBase and Spark services

Start the 5 essential Hadoop services: resourcemanager, nodemanager, namenode, datanode and historyserver in a Terminal window.

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

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

The following 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 them. If some of them exit, re-start them by repeating the above step.*

Finally, configure and start Spark. Process the following command 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[*] --jars
/home/bigdata/hbase-spark-connector/*,/usr/share/hbase-2.2.2/lib/* -
-conf spark.sql.hive.metastore.version=2.1.1 --conf
spark.sql.hive.metastore.jars=/usr/share/hive/lib/*
```

Note. The `--jars` option loads the necessary HBase dependencies in to the spark-shell classpath. The two `--conf` options indicates the correct Hive version installed in the VM.

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

```
$SPARK_HOME/bin/spark-shell --master local[*] --conf
spark.sql.hive.metastore.version=2.1.1 --conf
spark.sql.hive.metastore.jars=/usr/share/hive/lib/*
```

(1) Spark DataFrame and Dataset operations

Use `Resources` link available on Moodle to download a flight-dataset `2015-summary.json`. Next, upload the file to HDFS. Do it in a different Terminal window.

Process 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("no", "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 the following saves a data frame `df` as a file in HDFS in JSON format:

```
> df.write.json("<path + file name>")
```

(2) 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` command:

```
> :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` in a file with a name `myscript.sc`. Then, process the following `:paste` command in the Spark shell:

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

(3) 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.

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 in the following way:

```
> 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 notes for details.

(4) Spark and HBase

Start the HBase shell:

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

Create HBase table `Contacts` 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:

```
> val hbdf = spark.
    read.
    format("org.apache.hadoop.hbase.spark").
    option("catalog",catalog).
    option("hbase.spark.use.hbasecontext", false).
    option("hbase.spark.pushdown.columnfilter", false).
    load()

// Query
> hbdf.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.
    format("org.apache.hadoop.hbase.spark").
    option("catalog",catalog).
    option("hbase.spark.use.hbasecontext", false).
    option("hbase.spark.pushdown.columnfilter", false).
    save()
> hbdf.show()
```

Stop the HBase service when you finish with it:

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

(5) Spark Structured Streaming

In the following example, we simulate 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 a NetCat command that sends data over a network connection:

```
nc -lk 9999
```

Process the following code in Spark-shell to tells Spark to listen to port 9999 at the localhost.

```
> spark.sparkContext.setLogLevel("ERROR")
> 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 in the following way:

```
> 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 in 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.
