Hands-On Exercise: Process Multiple Batches with Apache Spark Streaming

Files and Data Used in This Exercise

Exercise directory: `$DEVSH/exercises/spark-streaming-multi`

Python stub: `stubs-python/StreamingLogsMB.py`

Python solution: `solution-python/StreamingLogsMB.py`

Scala project:

- **Project directory:** `streaminglogsMB_project`
- **Stub class:** `stubs.StreamingLogsMB`
- **Solution class:** `solution.StreamingLogsMB`

Data (local): `$DEVDATA/weblogs/*`

In this exercise, you will write a Spark Streaming application to count web page requests over time.

Simulating Streaming Web Logs

To simulate a streaming data source, you will use the provided `streamtest.py` Python script, which waits for a connection on the host and port specified and, once it receives a connection, sends the contents of the file(s) specified to the client (which will be your Spark Streaming application). You can specify the speed (in lines per second) at which the data should be sent.

1. Change to the exercise directory.

```
$ cd $DEVSH/exercises/spark-streaming-multi
```

2. Stream the Loudacre Web log files at a rate of 20 lines per second using the provided test script.

```
$ python streamtest.py localhost 1234 20 \
  $DEVDATA/weblogs/*
```

This script exits after the client disconnects, so you will need to restart the script when you restart your Spark application.

Displaying the Total Request Count

3. A stub file for this exercise has been provided for you in the exercise directory. The stub code creates a Streaming context for you, and creates a DStream called `logs` based on web log request messages received on a network socket.

   For Python, start with the stub file `StreamingLogsMB.py` in the `stubs-python` directory.

   For Scala, a Maven project directory called `streaminglogsMB_project` has been provided in the exercise directory. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogsMB.scala`.

4. Enable checkpointing to a directory called `logcheckpt`.

5. Count the number of page requests over a window of five seconds. Print out the updated five-second total every two seconds.

   - Hint: Use the `countByWindow` function.

Building and Running Your Application

6. In a different terminal window than the one in which you started the `streamtest.py` script, change to the correct directory for the language you are using for your application.

For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/spark-streaming-multi
```

For Scala, change to the project directory for the exercise:

```
$ cd \
$DEVSH/exercises/spark-streaming-multi/streaminglogsMB_project
```

7. If you are using Scala, build your application JAR file using the `mvn package` command.

8. Use `spark-submit` to run your application locally and be sure to specify two threads; at least two threads or nodes are required to running a streaming application, while the VM cluster has only one. Your application takes two parameters: the host name and the port number to connect the DStream to. Specify the same host and port at which the test script you started earlier is listening.

```
$ spark-submit --master 'local[2]' \
stubs-python/StreamingLogsMB.py localhost 1234
```

   - **Note:** Use **`solution-python`**`/StreamingLogsMB.py` to run the solution application instead.

```
$ spark-submit --master 'local[2]' \
--class stubs.StreamingLogsMB \
target/streamlogmb-1.0.jar localhost 1234
```

   - **Note:** Use `--class` **`solution`**`.StreamingLogsMB` to run the solution class instead.

9. After a few moments, the application should connect to the test script's simulated stream of web server log output. Confirm that for every batch of data received (every second), the application displays the first few Knowledge Base requests and the count of requests in the batch. Review the files.

10. Return to the terminal window in which you started the `streamtest.py` test script earlier. Stop the test script by typing `Ctrl+C`. You do not need to wait until all the web log data has been sent.

> **Warning: Stopping Your Application**
>
> You *must* stop the test script before stopping your Spark Streaming application. If you attempt to stop the application while the test script is still running, you may find that the application appears to hang while it takes several minutes to complete. (It will make repeated attempts to reconnect with the data source, which the test script does not support.)

11. After the test script has stopped, stop your application by typing `Ctrl+C` in the terminal window the application is running in.

Bonus Exercise

Extend the application you wrote above to also count the total number of page requests by user from the start of the application, and then display the top ten users with the highest number of requests.

Follow the steps below to implement a solution for this bonus exercise:

1. Use map-reduce to count the number of times each user made a page request in each batch (a hit-count).
    - Hint: Remember that the User ID is the 3rd field in each line.
2. Define a function called `updateCount` that takes an array (in Python) or sequence (in Scala) of hit-counts and an existing hit-count for a user. The function should return the sum of the new hit-counts plus the existing count.
    - Hint: An example of an `updateCount` function is in the course material and the code can be found in `$DEVSH/examples/spark/spark-streaming`.
3. Use `updateStateByKey` with your `updateCount` function to create a new DStream of users and their hit-counts over time.

4. Use `transform` to call the `sortByKey` transformation to sort by hit-count.

   - Hint: You will have to swap the key (user ID) with the value (hit-count) to sort.

Note: The solution files for this exercise include code for this bonus exercise.

This is the end of the exercise

Hands-On Exercise: Process Apache Kafka Messages with Apache Spark Streaming

Files and Data Used in This Exercise

Exercise directory: `$DEVSH/exercises/spark-streaming-kafka`

Python stub: `stubs-python/StreamingLogsKafka.py`

Python solution: `stubs-solution/StreamingLogsKafka.py`

Scala project:

- **Project directory:** `streaminglogskafka_project`
- **Stub class:** `stubs.StreamingLogsKafka`
- **Solution class:** `solution.StreamingLogsKafka`

Data (local): `$DEVDATA/weblogs/*`

In this exercise, you will write an Apache Spark Streaming application to handle web logs received as messages on a Kafka topic.

In a prior exercise, you started a Flume agent that collects web log files from a local spool directory and passes them to a Kafka sink. In this exercise, you will use the same Flume agent to produce data and publish it to Kafka. The Kafka topic will be the data source for your Spark Streaming application.

Important: This exercise depends on a prior exercise: "Send Web Server Log Messages from Flume to Kafka." If you were unable to complete that exercise, run the catch- up script and advance to the current exercise:

```
$ $DEVSH/scripts/catchup.sh
```

Consuming Messages from a Kafka Direct DStream

1. For Python, start with the stub file `StreamingLogsKafka.py` in the `stubs-python` directory, which imports the necessary classes for the application.

   For Scala, a Maven project directory calling `streaminglogs` has been provided in the exercise directory. To complete the exercise, start with the stub code in `src/main/scala/stubs/StreamingLogs.scala`, which imports the necessary classes for the application.

2. Create a DStream using `KafkaUtils.createDirectStream`.

   - The broker list consists of a single broker: `localhost:9092`.
   - The topic list consists of a single topic: the argument to the main function passed in by the user when the application is submitted.

   Refer to the course materials for the details of creating a Kafka stream.

3. Kafka messages are in (key, value) form, but for this application, the key is null and only the value is needed. (The value is the web log line.) Map the DStream to remove the key and use only the value.
4. To verify that the DStream is correctly receiving messages, display the first 10 elements in each batch.
5. For each RDD in the DStream, display the number of items—that is, the number of requests.

   Tip: Python does not allow calling `print` within a lambda function, so define a named function to print.

6. Save the filtered logs to text files in HDFS. Use the base directory name `/loudacre/streamlog/kafkalogs`.

Building and Running Your Application

7. Change to the correct directory for the language you are using for your application.

For Python, change to the exercise directory:

```
$ cd $DEVSH/exercises/spark-streaming-kafka
```

For Scala, change to the project directory for the exercise:

```
$ cd \
$DEVSH/exercises/spark-streaming-kafka/streaminglogskafka_project
```

8. If you are using Scala, you will need to build your application JAR file using the `mvn package` command.

9. Use `spark-submit` to run your application locally and be sure to specify two threads; at least two threads or nodes are required to run a streaming application, while the VM cluster has only one. Your application takes one parameter: the name of the Kafka topic from which the DStream will read messages, `weblogs`.

```
$ spark-submit --master 'local[2]' \
stubs-python/StreamingLogsKafka.py weblogs
```

   - **Note:** Use **`solution-python`**`/StreamingLogsKafka.py` to run the solution application instead.

```
$ spark-submit --master 'local[2]' \
--class stubs.StreamingLogsKafka \
target/streamlogkafka-1.0.jar weblogs
```

   - **Note:** Use `--class` **`solution`**`.StreamingLogsKafka` to run the solution class instead.

Producing Messages for Spark Streaming

10. In a separate terminal window, start a Flume agent using the configuration file from the Flume/Kafka exercise:

```
$ flume-ng agent --conf /etc/flume-ng/conf \
 --conf-file \
 $DEVSH/exercises/flafka/spooldir_kafka.conf \
 --name agent1 -Dflume.root.logger=INFO,console
```

11. Wait a few moments for the Flume agent to start up. You will see a message like:
    `Component type: SINK, name: kafka-sink started`
12. In a separate new terminal window, run the script to place the web log files

    the `/flume/weblogs_spooldir` in directory.

If you completed the Flume exercises or ran `catchup.sh` previously, the script will prompt whether you want to clear out the `spooldir` directory. Be sure to enter `y` when prompted.

```
$ $DEVSH/exercises/flafka/copy-move-weblogs.sh\
 /flume/weblogs_spooldir
```

- **Note:** You can rerun the `copy-move-weblogs.sh` script to send the web log data to Spark Streaming again if needed to test your application.

13. Return to the terminal window where your Spark application is running to verify the count output. Also review the contents of the saved files in HDFS directories `/loudacre/streamlog/kafkalogs-<time-stamp>`. These directories hold `part` files containing the page requests.

Cleaning Up

14. Stop the Flume agent by pressing `Ctrl+C`. You do not need to wait until all the web log data has been sent.
15. Stop the Spark application by pressing `Ctrl+C`. (You may see several error messages resulting from the interruption of the job in Spark; you may disregard these.)

This is the end of the exercise