Day 2 – Lab3:

Word Count Example

Introductions

In this exercise we'll implement the obligatory streaming example used by almost every streaming tool.

The example simply takes a string of words and count them.

All the below directory references are relative to the root directory for this lab `labs/06-Streaming`.
To ensure that you can build a `java` project from scratch, we decided to make these instructions start from an empty directory.
If you don't feel surefooted using Java, you may want to simply run the solution to this exercise instead of creating it from scratch. To do so, skip the steps below and start with the step "create topic".

Create a new directory

Create a new directory (anywhere you'd prefer) and cd into the new directory. E.g.:

```
$ mkdir wordcount
$ cd wordcount
```

Create a Maven `pom.xml` file

Create the `pom.xml` file required by Maven. You may do this in your IDE or with a simple text editor.
Copy the content shown below:

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

    <groupId>com.example</groupId>
    <artifactId>wordcount</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>1.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>1.1.0</version>
        </dependency>
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>19.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
                <version>3.5.1</version>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>com.example.StreamExample</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.skife.maven</groupId>
                <artifactId>really-executable-jar-maven-plugin</artifactId>
```

```
                <version>1.1.0</version>
                <configuration>
                    <!-- value of flags will be interpolated into the java invocation
-->
                    <!-- as "java $flags -jar ..." -->
                    <!--<flags></flags>-->

                    <!-- (optional) name for binary executable, if not set will just
-->
                    <!-- make the regular jar artifact executable -->
                    <programFile>wordcounter</programFile>
                </configuration>

                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>really-executable-jar</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

The main thing to notice in the above `pom.xml` file is the dependency section where we declare a dependency on the client and stream library of Kafka.
Also notice the section where we build an executable jar with a program wrapper creating a program called `wordcounter`.

Create the directory structure required

Maven requires a particular directory structure.

On a Mac or Linux machine, you can simply perform the following task:

```
$ mkdir -p src/main/java/com/example src/main/resources
```

In windows you can do a similar thing if you quote the directory definitions.

```
> mkdir "src/main/java/com/example"
> mkdir "src/main/resources"
```

Create the processing topology implementation

In the directory `src/main/java/com/example`, create a file `StreamExample.java` and copy the code below:

```java
package com.example;

import java.util.Arrays;
import java.util.Locale;
import java.util.Properties;

/**
 * This example has been reworked from one of the sample test applications
 * provided by Kafka in their test suite.
 *
 */

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KStreamBuilder;

public class StreamExample {

    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        // This is the consumer ID. Kafka keeps track of where you are in the stream
(in Zoomaker) for each consumer group
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "sample-stream-count");
        // Where is zookeeper?
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        // Default key serializer
        props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
        // Default value serializer
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

        // setting offset reset to earliest so that we can re-run the the example
code
        // with the same pre-loaded data
        // Note: To re-run the example, you need to use the offset reset tool:
        //
https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Too
l
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // First we create a stream builder
        KStreamBuilder bld = new KStreamBuilder();

        // Next, lets specify which stream we consume from
        KStream<String, String> source = bld.stream("stream-input");

        // This starts the processing topology
        source
            // convert each message list of words
```

```
            .flatMapValues(value ->
Arrays.asList(value.toLowerCase(Locale.getDefault()).split(" ")))
            // We're not really interested in the key in the incoming messages, we
only want the values
            .map( (key,value) -> new KeyValue<>(value, value))
            // now we need to group them by key
            .groupByKey()
            // let's keep a table count called Counts"
            .count("Counts")
            // next we map the counts into strings to make serialization work
            .mapValues(value -> value.toString())
            // and finally we pipe the output into the stream-output topic
            .to("stream-output");

        // let's hook up to Kafka using the builder and the properties
        KafkaStreams streams = new KafkaStreams(bld, props);
        // and then... we can start the stream processing
        streams.start();

        // this is a bit of a hack... most typical would be for the stream processor
        // to run forever or until some condition are met, but for now we run
        // until someone hits enter...
        System.out.println("Press enter to quit the stream processor");
        System.in.read();
        // finally, let's close the stream
        streams.close();
    }
}
```

In the previous exercises, we used Java 7 to be as backwards compatible as possible. In the above example, we switched to Java 8 as the new lambda syntax makes the code much more readable.

Take some time to study the processing topology and predict what will happen.

Build using maven

If you have maven installed

```
$ mvn package
```

If you don't have maven installed

```
$ docker run -it --rm --name lesson -v "$PWD":/usr/src/lesson -w /usr/src/lesson
maven:3-jdk-8 bash

root@aa2c30126c78:/usr/src/lesson# mvn package
```

Make sure your docker images are running

If your docker images are not already running, you'll need to start them now. We have created a separate `docker-compose` file for this exercise. If you do not intend to use Spark, you can simply use your old docker setup.

If you are going to run the new docker image, you should go to the `docker` directory and edit the file `docker-compose.yml`. Replace the string `[INSERT IP ADDRESS HERE]`
On OSX, you can use the following command:

```
$ ifconfig | grep inet
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet6 fe80::1cf1:f9f4:6104:f2a3%en0 prefixlen 64 secured scopeid 0x4
    inet 10.0.1.4 netmask 0xffffff00 broadcast 10.0.1.255
    inet6 2605:6000:1025:407f:101b:9cd8:e973:b9dd prefixlen 64 autoconf secured
    inet6 2605:6000:1025:407f:9077:8802:8c88:e63d prefixlen 64 autoconf temporary
    inet6 fe80::cc8a:c5ff:fe43:b670%awdl0 prefixlen 64 scopeid 0x8
    inet6 fe80::7df6:ec93:ffea:367a%utun0 prefixlen 64 scopeid 0xa
```

In this case, the IP address to use is `10.0.1.4`. Make sure you *do not* use `127.0.0.1` because that will not work correctly.
On Windows, you can use the following command:

```
$ ipconfig
Ethernet adapter Local Area Connection:
Connection-specific DNS Suffix . : hsd1.ut.comcast.net.
IP Address. . . . . . . . . . . . : 192.168.201.245
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Default Gateway . . . . . . . . . : 192.168.201.1
```

Here the IP address to use is `192.168.201.245`.

After you replace the line in `docker-compose.yml`, it should look like this:

```
KAFKA_ADVERTISED_HOST_NAME: 192.168.201.245
```

Save the `docker-compose.yml` file after making this modification.
We have added an instance of Spark to the `docker-compose.yml`in the directory `labs/06-Streaming/docker` directory for those that want to play with streaming through Spark.
If your images are not up, go to to the `docker` directory and run:

```
$ docker-compose up
```

Create the topic

The program we created (or copied :)) requires two topics:

- `stream-input`
- `stream-output`

The `stream-input` is used to push content words to be counted.
The `stream-output` is used to read the result from the word count.
Let's create the topics. To do this, make sure you have a shell/terminal opened in the directory where you ran `docker-compose up`.
In this shell, run the following commands:

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-topics.sh --create --zookeeper
zookeeper:2181 --replication-factor 1 --partitions 1 --topic stream-input
$ docker-compose exec kafka /opt/kafka/bin/kafka-topics.sh --create --zookeeper
zookeeper:2181 --replication-factor 1 --partitions 1 --topic stream-output
```

Start a stream listener

Next let's read the result of the final stream (the one produced by our little stream example):

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-
server localhost:9092 --topic stream-output --from-beginning --property
print.key=true
```

The observant may notice that we've added `--property print.key=true` to this startup (compared with previous runs). This is important because we want to see the word that is being counted (the messages have the word as its `key` and the count as its `value`).

Start the stream producer

In another terminal, start the stream producer.

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-console-producer.sh --broker-list
kafka:9092 --topic stream-input
```

This will be the window where we produce the text where from which we'll perform a word count.

Think about this for a second....

Let's count the processes and discuss what roles they play:

- Process 1: The `docker-compose up`
    - This is the process that runs Kafka and Zookeeper. These would normally be running on different machines or in the case of setups like Docker Swarm or Kubernetes, perhaps in different docker containers.
- Process 2: The Kafka producer
    - This is a processed started using the command line tools where you can type in text and the text is published to the topic `stream-input`
- Process 3: The Kafka consumer
    - This is another process started with the command line tools. This process will consume any message produced on the topic `stream-output`
- [*MISSING*] Process 4: Our stream processor
    - This is the process we wrote in Java that consumes messages from the topic `stream-input` and produces a word count on the topic `stream-output`

The Java program we wrote will make up the final piece of the puzzle (Process 4).

Build your `Java` program

In the development directory, run (either with your native install or with the docker tool):

```
$ mvn package
```

Run the processor

Run the stream processor by running the command `target/wordcounter` from the same directory where you ran the `maven` command.

Produce some messages

In the process that runs the `Kafka producer`, type in some messages on the terminal. When you press enter, you should see:

- Some messages in the process running Kafka (Process 1 above)
- A message in the final consumer (Process 3 above) printing out what you sent from your stream processor

There may be a significant delay between when you type the messages and when you see the message being consumed by the final consumer. You may want to add some log messages to the processing topologies in the stream process to help you visualize what is happening.

Here is an example of what you may see:

Here is what I typed into the producer

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-console-producer.sh --broker-list
kafka:9092 --topic stream-input
this is a test message
```

Here is what I'll see in the final consumer (eventually):

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-
server localhost:9092 --topic stream-output --from-beginning --property
print.key=true
this    1
is      1
a       1
test    1
message 1
```

Done

Congratulations, this lab is done!