UCB MIDS W205 Summer 2018 - Kevin Crook's agenda for Synchronous Session #8

Update docker images (before class)

Run these command in your droplet (but **NOT** in a docker container):

```
docker pull midsw205/base:latest
docker pull confluentinc/cp-zookeeper:latest
docker pull confluentinc/cp-kafka:latest
docker pull midsw205/spark-python:0.0.5
docker pull midsw205/cdh-minimal:latest
```

Update the course-content repo in your docker container in your droplet (before class)

See instructions in previous synchronous sessions.

Part 1 - Add an hadoop container to our cluster. We will download a json file of world cup player information, create a kafka topic, publish the world cup player information to the topic, use python code in pyspark to use spark to subscribe to the topic, load the data as json objects into spark, write the data out into parquet files in hadoop hdfs.

Create a directory for spark with kafka and hadoop hadfs

```
mkdir ~/w205/spark-with-kafka-and-hdfs
cd ~/w205/spark-with-kafka-and-hdfs
```

Copy the yml file:

```
cp ~/w205/course-content/08-Querying-Data/docker-compose.yml .
```

We will go through the yml file in class. You will need to edit the file and change the volume mapping as we have done before.

Start the docker cluser

```
docker-compose up -d
```

As we did before, you may want to use the following command to watch kafka come up. Multiple command line windows work best for this. Remember to use control-C to exit.

```
docker-compose logs -f kafka
```

The hadoop hdfs is a separate file system from our local linux file system. The directories are different and have different paths. You will need to use the following command to view a directory listing of the hdfs directory /tmp. If the cluster is first coming up, or if you have recently added a directory or file, it may take time to show up. Remember that big data architectures are eventually consistent and not immediately consistent.

```
docker-compose exec cloudera hadoop fs -ls /tmp/
```

The output should look similar to this. Remember it may take a while to show consistency:

```
drwxrwxrwt   - mapred mapred              0 2018-02-06 18:27 /tmp/hadoop-yarn
drwx-wx-wx   - root   supergroup          0 2018-02-20 22:31 /tmp/hive
```

Create a kafka topic called players. This is the same procedure we have used several times before with the foo topic:

```
docker-compose exec kafka \
  kafka-topics \
    --create \
    --topic players \
    --partitions 1 \
    --replication-factor 1 \
    --if-not-exists \
    --zookeeper zookeeper:32181
```

The same command on 1 line for convenience:

```
docker-compose exec kafka kafka-topics --create --topic players --partitions 1 --replication-factor 1 --
```

You should see the following or similar:

```
Created topic "players".
```

Download the dataset for world cup players in json format. Remember our downloads go into the ~/w205 directory:

```
cd ~/w205/spark-with-kafka-and-hdfs
curl -L -o players.json https://goo.gl/jSVrAe
```

Use kafkacat to publish the world cup players json data to the kafka topic players:

```
docker-compose exec mids \
  bash -c "cat /w205/spark-with-kafka-and-hdfs/players.json \
    | jq '.[]' -c \
    | kafkacat -P -b kafka:29092 -t players"
```

The same command on 1 line for convenience:

```
docker-compose exec mids bash -c "cat /w205/spark-with-kafka-and-hdfs/players.json | jq '.[]' -c | kafka
```

Start a spark pyspark shell in the spark container. Remember that pyspark is the python interface to spark.

```
docker-compose exec spark pyspark
```

Write python code in pyspark to consume from the kafka topic:

```
raw_players = spark \
  .read \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "kafka:29092") \
  .option("subscribe","players") \
  .option("startingOffsets", "earliest") \
  .option("endingOffsets", "latest") \
  .load()
```

The same command on 1 line for convenience:

```
raw_players = spark.read.format("kafka").option("kafka.bootstrap.servers", "kafka:29092").option("subsc
```

The following command will cache the spark data structure. Without it every time something isn't in memory it will generate a warning message. The warning messages don't hurt anything, but are very distracting. This is due to spark's use of "lazy evaluation" which is a hallmark of big data architecture. Add that to our big data architecture we have seen so far: immutable, eventually consistent, lazy evaluation.

```
raw_players.cache()
```

Before you print the schema, what do you expect to see? It's the same shema we have seen when consuming a kafka topic in the past:

```
raw_players.printSchema()
```

Remember that the value attribute of a kafka schema will be stored as raw bytes which is not easily human readable. So we convert it to strings so humans can read it:

```
players = raw_players.select(raw_players.value.cast('string'))
```

An alternative way to the previous command (you only need to do one):

```
players = raw_players.selectExpr("CAST(value AS STRING)")
```

Write the players data frame to a parquet file in hadoop hdfs. Note that this is writing to hadoop hdfs and not to the local linux file system - big difference. hdfs is intended to have a virtual presence on all nodes in the hadoop cluster. Parquet format is a binary format for storing data in binary format in a file in columnar format. Parquet files are immutable. Remember that immutable files are a hallmark of big data architecture. It allows them to be pushed out in a hadoop cluster, stored in object store (which we could use an elastic query resource against), delivered in content delivery networks, etc.

```
players.write.parquet("/tmp/players")
```

Using another command line window, (keep pyspark running), use the following hdfs command to see the directory and files we just created. Another hallmark of big data architecture is that when we write, we usually specify a directory rather than a file. In this case see the players is a directory and not a file. Note the unique naming convention of the data files. In our case we only see 1 file. But if we had multiple nodes, each node would have it's own file. This is so they can write in parallel. We can do this due the "shared nothing architecture" which is another hallmark of big data architecture. Also note that files will have a maximum size of 2 GiB. This is so they can be written using 32 bit pointers rather than 64 bit pointers.

```
docker-compose exec cloudera hadoop fs -ls /tmp/
docker-compose exec cloudera hadoop fs -ls /tmp/players/
```

Going back to our pyspark, let's look at the data we wrote. We may see some unicode characters. Most western alphabets need only 7 bits to store all characters, so we can store it in 1 byte (8 bits). (We used to call it ASCII). However, other languages of the world have their own alphabets and need 2 bytes to store a single character. Unicode is the standard for this, but unicode has the disadvantage of storing 2 bytes for every character whether it needs it or not. As a compromise, utf-8 format was invented. This format will use only 1 byte if the character needs only 1 byte, and 2 bytes if the character needs 2 bytes. It can do this by using the extra bit as a signal.

```
players.show()
```

The following code will set standard output to write data using utf-8 instead of unicode. In the modern era, it's almost always a good idea to always use utf-8:

```
import sys
sys.stdout = open(sys.stdout.fileno(), mode='w', encoding='utf8', buffering=1)
```

Take a look at the players data formatted for json:

```
import json
players.rdd.map(lambda x: json.loads(x.value)).toDF().show()
```

Create a new data frame to hold the data in json format:

```
extracted_players = players.rdd.map(lambda x: json.loads(x.value)).toDF()
```

As an alternative to the previous command, (don't do both), we could use the Row method. Notice the ** operator. Remember from your python class that this operator turns keyword arguments into a python dictionary. It works with any parameter (or parameter expression in our case), but traditionally uses the name kwargs.

```
from pyspark.sql import Row
extracted_players = players.rdd.map(lambda x: Row(**json.loads(x.value))).toDF()
```

See the results of the new data frame:

```
extracted_players.show()
```

Save the results to a parquet format file in hdfs:

```
extracted_players.write.parquet("/tmp/extracted_players")
```

Let's see the differences again between players and extracted_players:

```
players.show()
extracted_players.show()
```

Keep the cluster and pyspark running, no need to take them down and up again!

Breakout - streaming sensor data

Previously, we talked about using Kafka for:

- Message Queues in a corporate environment, such as IBM MQ Series or Tibco, where most production systems publish messages in real time to topic based message queues
- Internet Streaming Media, such as Twitter,

Some other interesting uses for Kafka include:

- Sensor data - sensors gather and transmit data in real time over networks (or cell phone networks). Examples might include a railroad having sensors on tracks and transmitting data in real time.
- Industrial robots and industrial machinery - factories - transmit data about the work they are doing in real time over networks.

Discuss these or other areas for Kafka. How would you implement a Lambda Architecture? What would be the speed layer? The Batch layer? What analytics could you do a the speed layer? at the batch layer?

Part 2 - adding to part 1 imposing a schema on the json objects in the spark dataframe and using spark SQL to query the data

Let's check out hdfs before we write anything to it (using a linux command line windows separate form our pyspark window):

```
docker-compose exec cloudera hadoop fs -ls /tmp/
```

Create a kafka topic called commits. Note that this topic can co-exist with our other players topic:

```
docker-compose exec kafka \
  kafka-topics \
    --create \
    --topic commits \
    --partitions 1 \
    --replication-factor 1 \
    --if-not-exists \
    --zookeeper zookeeper:32181
```

Same command on 1 line for convenience:

```
docker-compose exec kafka kafka-topics --create --topic commits --partitions 1 --replication-factor 1 --
```

You should see the following or similar output:

```
Created topic "commits".
```

Download the github commits dataset in json format into the ~/w205 directory

```
cd ~/w205/spark-with-kafka-and-hdfs
curl -L -o github-example-large.json https://goo.gl/Hr6erG
```

Using kafkacat publish the github json dataset to the topic commits

```
docker-compose exec mids \
  bash -c "cat /w205/spark-with-kafka-and-hdfs/github-example-large.json \
    | jq '.[]' -c \
    | kafkacat -P -b kafka:29092 -t commits"
```

Same command on 1 line for convenience:

```
docker-compose exec mids bash -c "cat /w205/spark-with-kafka-and-hdfs/github-example-large.json | jq '.
```

Using our pyspark command prompt, consume from the kafka topic commits into a kafka data frame:

```
raw_commits = spark \
  .read \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "kafka:29092") \
  .option("subscribe","commits") \
  .option("startingOffsets", "earliest") \
  .option("endingOffsets", "latest") \
  .load()
```

Same command on 1 line for convenience:

```
raw_commits = spark.read.format("kafka").option("kafka.bootstrap.servers", "kafka:29092").option("subsc
```

As before we will cache to supress warning messages which are distracting:

```
raw_commits.cache()
```

What should the schema for raw_commits look like? Remember it came from a kafka topic. Use the following command to find out:

```
raw_commits.printSchema()
```

Remember that the value from a kafka topic will be a raw byte string, which as before, we will convert into a string:

```
commits = raw_commits.select(raw_commits.value.cast('string'))
```

The following command will wthe commits data fram to a parquet file in hdfs:

```
commits.write.parquet("/tmp/commits")
```

As before, let's extract our json fields:

```
extracted_commits = commits.rdd.map(lambda x: json.loads(x.value)).toDF()
```

Show the data frame after the json extraction:

```
extracted_commits.show()
```

Notice that this time we have nested json data. Before our json data was flat.

Let's print the schema:

```
extracted_commits.printSchema()
```

We will now use spark sql to deal with the nested json.

First, cretae a spark temporary table called commits based on the data frame. registerTempTable() is a method of the spark class data frame.

```
extracted_commits.registerTempTable('commits')
```

Issue spark sql against the temporary table commits that we just registered:

```
spark.sql("select commit.committer.name from commits limit 10").show()
spark.sql("select commit.committer.name, commit.committer.date, sha from commits limit 10").show()
```

Save the results of the query into another data frame:

```
some_commit_info = spark.sql("select commit.committer.name, commit.committer.date, sha from commits lim
```

Write the data frame holding the results of our query to a parquet file in hdfs:

```
some_commit_info.write.parquet("/tmp/some_commit_info")
```

Go to our other linux command line window and use the following command to see the directory and files in hdfs:

```
docker-compose exec cloudera hadoop fs -ls /tmp/
docker-compose exec cloudera hadoop fs -ls /tmp/commits/
```

Go to the pyspark window and exit pyspark:

```
exit()
```

Tear down the docker cluster and make sure it's down:

```
docker-compose down
docker-compose ps
```