UCB MIDS W205 Summer 2018 - Kevin Crook's agenda for Synchronous Session #7

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
```

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

See instructions in previous synchronous sessions.

Discuss Project 2: Tracking User Activity

Assignment 6 - Get and Clean Data

Assignment 7 - Setup Pipeline

Assignment 8 - Build and Write-up Pipeline

Create a kafka topic, publish the number 1 to 42 as messages to that topic, use pyspark (python spark interface in the spark container) to subscribe to the topic and read the messages

Create a directory for spark with kafka and change to that directory:

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

Copy the yml file to the spark with kafka directory we should be in:

```
cp ~/w205/course-content/07-Sourcing-Data/docker-compose.yml .
```

Edit the yml file we just copied. During class we will talk through it. For volume mounts, you may need to change them to fully qualified path names. If you are running Windows desktop, you will need to change them to a fully qualified Windows path.

Start the docker cluster:

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

If you want to see the kafka logs live as it comes up use the following command. The -f tells it to keep checking the file for any new additions to the file and print them. To stop this command, use a control-C:

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

Create a topic called foo in the kafka container using the kafka-topics utility:

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

The same command on 1 line to make it easy to copy and paste:

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

You should see the following message to let us know the topic foo was created correctly:

```
Created topic "foo".
```

Check the topic in the kafka container using the kafka-topics utility:

```
docker-compose exec kafka \
  kafka-topics \
  --describe \
  --topic foo \
  --zookeeper zookeeper:32181
```

The same command on 1 line to make it easy to copy and paste:

```
docker-compose exec kafka kafka-topics --describe --topic foo --zookeeper zookeeper:32181
```

You should see the following or similar output from the previous command:

```
Topic:foo   PartitionCount:1    ReplicationFactor:1 Configs:
Topic: foo  Partition: 0    Leader: 1    Replicas: 1  Isr: 1
```

Now that we have the topic created in kafka, we want to publish the numbers from 1 to 42 to that topic. We use the kafka container with the kafka-console-producer utility:

```
docker-compose exec kafka \
  bash -c "seq 42 | kafka-console-producer \
    --request-required-acks 1 \
    --broker-list kafka:29092 \
    --topic foo && echo 'Produced 42 messages.'"
```

The same command on 1 line to make it easy to copy and paste:

```
docker-compose exec kafka bash -c "seq 42 | kafka-console-producer --request-required-acks 1 --broker-l
```

You should see the following or similar output from the previous command:

```
Produced 42 messages.
```

Previously, we used two methods to consume messages from the topic:

- the kafka-console-consumer utility
- the kafkacat utility

Instead of using those utilities, we are going to use spark to consume the messages.

In the spark container, run the python spark command line utility called pyspark:

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

Using pyspark, we will write some python spark code to consume from the kafka topic. numbers will be a spark data frame which is built on top of a spark RDD:

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

The same command on 1 line to make it easy to copy and paste:

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

Print the schema for the data frame. Note that the values are shown in binary, which we as humans have a hard time reading.:

```
numbers.printSchema()
```

Since we have a hard time reading binary for the value, let's translate the key and value into strings so we can read them. Create a new data frame which stores the numbers as strings. Note the data frames are immutable, so we cannot change them in place, we have to make a copy:

```
numbers_as_strings=numbers.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Use some of the methods of the data frame to display various things:

Display the entire data frame. Note that now we can read the key and value:

```
numbers_as_strings.show()
```

Display the schema for the data frame:

```
numbers_as_strings.printSchema()
```

Display the number of items in the data frame:

```
numbers_as_strings.count()
```

Exit pyspark using:

```
exit()
```

Tear down the docker cluster:

```
docker-compose down
```

Verify the docker cluster is down:

```
docker-compose ps -a
```

Breakout - discussing Massively Parallel Processing (MPP) concepts

Parallel Processing - running more than 1 process at the same time. We do this all the time with Windows, Mac, and Linux. We do this all the time in IT for reverse proxies, web servers, application servers, database servers, etc.

Issues with Parallel Processing

- Dependencies
- process A must wait for process B to finish before it can run
- more complex similar situations involving more than 2 processes
- solve using job scheduling that allow us to list dependencies
- Mutual Exclusion
- process A and process B cannot run at the same time

- more complex similar situations involving more than 2 processes
- solve using semaphores
- Deadlock
- process A is waiting on process B and process B is waiting on process A
- also circular waits involving more than 2 processes
- hard to solve - usually have to back out and restart processes
- Starvation
- a process doesn't get to run or doesn't get enough CPU time to complete
- priorities and aging - processes have priorities and as they get starved we raise their priority as they age

Massively Parallel Processing MPP

- Parallel processing on a massive scale, tens or hundreds or thousands etc. of nodes
- Common use for data science: algorithms that will run a long time with that can run in parallel usually involving gigantic data sets
- For algorithms to run MPP the most common model of the last 30 years:
- lists
  - data is typically stored in a gigantic list that is distributed among the nodes
  - if the list is 1 billion and we have 100 nodes, we distribute 10 million per node
- lambdas
  - lambda processes are processes that can be run on individual elements in the list at the same time
  - lambda processes have no dependencies
- DAGs
  - Directed Acyclic Graph
  - graph - set of verticies and edges
  - directed graph - edges have arrows for direction
  - acyclic - no cycles, can only visit each vertex once
  - Solves issues
  - Dependencies - clearly shows which processes a process must wait for to start
  - Mutual Exclusion - clearly shows which processes can run at the same time and which processes cannot
  - Deadlock - if each process is a lambda process, we will not have deadlock
  - Starvation - clearly shows that every process will get a chance to run if other processes complete

Generation of MPPs associated with running algorithms for Data Science

- 1st generation
  - 80's and 90's
  - artificial intelligence on the LISP machines using the LISP language
  - invented the model of lists, lambdas, and DAGs and coined the terms
  - problem - DAGs written by hand on drafting tables, DAGs were error prone

- 2nd generation
  - early 2000's
  - MapReduce on Hadoop
  - fixed DAG called MapReduce solved the problem of DAGs being error prone
  - problem - fixed DAG can't solve every algorithm - we can chain DAGs but extremely inefficient
  - https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
- 3rd generation
  - early 2010's
  - Spark
  - easy to use list called the RDD (and inherited children such as Spark Data Frames) that mimics the Python list
  - easy to use "lambda transforms" that can be run in parallel
  - generates an optimal DAG for you
  - https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html

Discuss as a group the above concepts, paying special attention to the Spark DAGs in the link above.

Create a kafka topic, publish json messages from a downloaded file to that topic, use pyspark (python spark interface in the spark container) to subscribe to the topic and read those json messages

Download our GitHub example data from the Internet using the curl utility. Note that since it's using HTTPS, you can paste the URL into a web browser to test if the download works or not. This is always highly recommended. It should produce a json file. However, if there are any errors, it will produce an XML file:

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

Continue in our same directory with the same yml file:

```
cd ~/w205/spark-with-kafka
```

Start the docker cluster (same as we did before):

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

Check the kafka logs (same as we did before) Remember to hit the control-C to exit:

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

Create a topic called foo (same as we did before):

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

The same command on 1 line to make it easy to copy and paste:

```
docker-compose exec kafka kafka-topics --create --topic foo --partitions 1 --replication-factor 1 --if-n
```

You should see the following message to let us know the topic foo was created correctly:

```
Created topic "foo".
```

Check the topic (same as we did before):

```
docker-compose exec kafka \
  kafka-topics \
    --describe \
    --topic foo \
    --zookeeper zookeeper:32181
```

The same command on 1 line to make it easy to copy and paste:

```
docker-compose exec kafka kafka-topics --describe --topic foo --zookeeper zookeeper:32181
```

You should see the following or similar output from the previous command:

```
Topic:foo   PartitionCount:1    ReplicationFactor:1 Configs:
Topic: foo  Partition: 0    Leader: 1    Replicas: 1  Isr: 1
```

We will execute a bash shell commands in the mids container as microservices. Remember with microservices, we start a container, run one command, and then exit the container when the command completes. In the first one, we just print out the json file as is. In the second one, we print out the json file and pipe it to the utility jq for formatting. Try running these in different windows and compare the results:

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

One thing I find strange about json files is that we are not allowed to just have a file of json objects. We have to place a wrapper around the json objects that consists of an open square bracket at the beginning of the file, comma separated json objects, and a close square bracket at the end of the file. The following command uses jq to remove this outer wrapper. It also uses a compact format which is good for computers, but hard for humans to read. Try running this command if a different window and compare it side by side with the previous two commands:

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

Execute a bash shell in the mids container to run a microservice. The cat piped into jq, we are already familiar with from the previous command. We add a step to pipe it into the kafkacat utility with the -P option. The -P option tells it to publish messages. The -t options gives it the topic name of foo. The kafka:29092 tells it the container name and the port number where kafka is running.

```
docker-compose exec mids \
  bash -c "cat /w205/spark-with-kafka/github-example-large.json \
    | jq '.[]' -c \
    | kafkacat -P -b kafka:29092 -t foo && echo 'Produced 100 messages.'"
```

The same command on 1 line to make it easier to copy and paste:

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

You should see the following or similar output from the previous command:

```
Produced 100 messages.
```

We are going to start pyspark to use spark via python (same as we did before):

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

At the pyspark prompt, we will write some python code to consume the messages (same as we did before):

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

The same command on 1 line to make it easy to copy and paste:

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

Print the schema for the data frame (same as we did before).

```
messages.printSchema()
```

Show the messages. Note that they values are in binary, which humans have a hard time reading:

```
messages.show()
```

Since we have a hard time reading binary for the value, let's translate the key and value into strings so we can read them. Create a new data fram which stores the numbers as strings. Note the data frames are immutable, so we cannot change them in place, we have to make a copy:

```
messages_as_strings=messages.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Use some of the methods of the data frame to display various things:

Display the entire data frame. Note that we can now read the key and value:

```
messages_as_strings.show()
```

Display the schema for the data frame:

```
messages_as_strings.printSchema()
```

Display the number of items in the data frame:

```
messages_as_strings.count()
```

Let's look at "unrolling" the json data.

Use the take method to extract individual data from json. Try the following and see what they do:

```
messages_as_strings.select('value').take(1)
messages_as_strings.select('value').take(1)[0].value
```

One mistake that people who are new to spark make is knowing when to use the spark parallel methods and when to use the python core and/or pandas methods. If we are going to manipulate an entire data frame (or RDD) we should use parallel methods. Using python core or pandas methods on large data frames defeats the purpose of using spark in the first place. If we are going to manipulate 1 record or a small number of records, we should use python core or pandas.

In the following example, we are extracting 1 json object from the spark data frame. In this case we want to use python core to manipulate it since it's only 1 record.

```
import json
```

```
first_message=json.loads(messages_as_strings.select('value').take(1)[0].value)
```

```
first_message
```

```
print(first_message['commit']['committer']['name'])
```

Should see the following output:

```
Nico Williams
```

Exit pyspark using the well formed method:

```
exit()
```

Tear down the docker cluster:

```
docker-compose down
```