UCB MIDS W205 Summer 2018 - Kevin Crook's agenda for Synchronous Session #14

Update docker images (before class)

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

```
docker pull confluentinc/cp-zookeeper:latest
docker pull confluentinc/cp-kafka:latest
docker pull midsw205/hadoop:0.0.2
docker pull midsw205/spark-python:0.0.6
docker pull midsw205/presto:0.0.1
docker pull midsw205/base:0.1.9
```

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

See instructions in previous synchronous sessions.

Activity - Building on what we did last week: Introduce Hue a web based gui tool for hadoop. Hue will listen on port 8888 which is what Jupyter Notebook usually listens on. Jupyter Notebook will be moved to 8889. Using hive, presto, hue, and hdfs, we will watch the table grow as we add files to hdfs in parquet format. We will remove files form hdfs and watch the table shrink.

Create directory, change to the directory, copy yml file, copy python files:

```
mkdir ~/w205/full-streaming-stack/
cd ~/w205/full-streaming-stack
cp ~/w205/course-content/14-Patterns-for-Data-Pipelines/docker-compose.yml .
cp ~/w205/course-content/14-Patterns-for-Data-Pipelines/*.py .
```

Review the docker-compose.yml file. As before, you may need to change directory mounts. The main difference is that we will be using hue the web based gui for hadoop on port 8888 and jupyter notebook will be moving to port 8889:

```
---
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 32181
      ZOOKEEPER_TICK_TIME: 2000
    expose:
      - "2181"
      - "2888"
      - "32181"
      - "3888"
    extra_hosts:
      - "moby:127.0.0.1"

  kafka:
```

```
  image: confluentinc/cp-kafka:latest
  depends_on:
    - zookeeper
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:32181
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:29092
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  expose:
    - "9092"
    - "29092"
  extra_hosts:
    - "moby:127.0.0.1"

cloudera:
  image: midsw205/hadoop:0.0.2
  hostname: cloudera
  expose:
    - "8020" # nn
    - "8888" # hue
    - "9083" # hive thrift
    - "10000" # hive jdbc
    - "50070" # nn http
  ports:
    - "8888:8888"
  extra_hosts:
    - "moby:127.0.0.1"

spark:
  image: midsw205/spark-python:0.0.6
  stdin_open: true
  tty: true
  volumes:
    - ~/w205:/w205
  expose:
    - "8888"
  ports:
    - "8889:8888" # 8888 conflicts with hue
  depends_on:
    - cloudera
  environment:
    HADOOP_NAMENODE: cloudera
    HIVE_THRIFTSERVER: cloudera:9083
  extra_hosts:
    - "moby:127.0.0.1"
  command: bash

presto:
  image: midsw205/presto:0.0.1
  hostname: presto
  volumes:
    - ~/w205:/w205
  expose:
```

```
      - "8080"
    environment:
      HIVE_THRIFTSERVER: cloudera:9083
    extra_hosts:
      - "moby:127.0.0.1"

  mids:
    image: midsw205/base:0.1.9
    stdin_open: true
    tty: true
    volumes:
      - ~/w205:/w205
    expose:
      - "5000"
    ports:
      - "5000:5000"
    extra_hosts:
      - "moby:127.0.0.1"
```

Start up our docker cluster (same as before):

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

Review our code for our web API server using python flask game_api.py (same as before):

```
#!/usr/bin/env python
import json
from kafka import KafkaProducer
from flask import Flask, request

app = Flask(__name__)
producer = KafkaProducer(bootstrap_servers='kafka:29092')


def log_to_kafka(topic, event):
    event.update(request.headers)
    producer.send(topic, json.dumps(event).encode())


@app.route("/")
def default_response():
    default_event = {'event_type': 'default'}
    log_to_kafka('events', default_event)
    return "This is the default response!\n"


@app.route("/purchase_a_sword")
def purchase_a_sword():
    purchase_sword_event = {'event_type': 'purchase_sword'}
    log_to_kafka('events', purchase_sword_event)
    return "Sword Purchased!\n"
```

Run our python flask web API server (same as before):

```
docker-compose exec mids \
  env FLASK_APP=/w205/full-streaming-stack/game_api.py \
```

```
  flask run --host 0.0.0.0
```

Same command on 1 line for convenience:

```
docker-compose exec mids env FLASK_APP=/w205/full-streaming-stack/game_api.py flask run --host 0.0.0.0
```

Remember that the kafka topic is now part of the image, so we don't have to create it (same as last week)

Run kafkacat in continuous mode (without the -e) so we can monitor kafka messages as they come through (same as before)

```
docker-compose exec mids \
  kafkacat -C -b kafka:29092 -t events -o beginning
```

Same command on 1 line for convenience:

```
docker-compose exec mids kafkacat -C -b kafka:29092 -t events -o beginning
```

Review our code for write_swords_stream.py which will use spark streaming to keep reading from the kafka topic until we stop it with a control-C and write them to parquet files in hadoop hdfs. Last time we ran 10 second batches. This time we will run 2 minute (120 second) batches, so we will have to wait a while when checking.

```
#!/usr/bin/env python
"""Extract events from kafka and write them to hdfs
"""
import json
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, from_json
from pyspark.sql.types import StructType, StructField, StringType


def purchase_sword_event_schema():
    """
    root
    |-- Accept: string (nullable = true)
    |-- Host: string (nullable = true)
    |-- User-Agent: string (nullable = true)
    |-- event_type: string (nullable = true)
    """
    return StructType([
        StructField("Accept", StringType(), True),
        StructField("Host", StringType(), True),
        StructField("User-Agent", StringType(), True),
        StructField("event_type", StringType(), True),
    ])


@udf('boolean')
def is_sword_purchase(event_as_json):
    """udf for filtering events
    """
    event = json.loads(event_as_json)
    if event['event_type'] == 'purchase_sword':
        return True
    return False
```

```
def main():
    """main
    """
    spark = SparkSession \
        .builder \
        .appName("ExtractEventsJob") \
        .getOrCreate()

    raw_events = spark \
        .readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "kafka:29092") \
        .option("subscribe", "events") \
        .load()

    sword_purchases = raw_events \
        .filter(is_sword_purchase(raw_events.value.cast('string'))) \
        .select(raw_events.value.cast('string').alias('raw_event'),
                raw_events.timestamp.cast('string'),
                from_json(raw_events.value.cast('string'),
                          purchase_sword_event_schema()).alias('json')) \
        .select('raw_event', 'timestamp', 'json.*')

    sink = sword_purchases \
        .writeStream \
        .format("parquet") \
        .option("checkpointLocation", "/tmp/checkpoints_for_sword_purchases") \
        .option("path", "/tmp/sword_purchases") \
        .trigger(processingTime="120 seconds") \
        .start()

    sink.awaitTermination()


if __name__ == "__main__":
    main()
```

Submit as a spark job using spark-submit (same as before):

```
docker-compose exec spark spark-submit /w205/full-streaming-stack/write_swords_stream.py
```

Verify that we wrote to hadoop (same as before). We may want to keep this command line window to check when the new files come through:

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

Start up a hive command line in the cloudera hadoop container (same as before):

```
docker-compose exec cloudera hive
```

Impose a schema on read on the parquet files in the /tmp/sword_purchases hadoop hdfs directory. Remember that schema on read in hive is imposed at the directory level. All parquet files in that directory will become part of the schema, essentially everytime we add a new file to this directory, it will become part of the table. Also, if we delete a file, it will be removed from the table. Remember we exit hive with `exit;` or a control-D (end of file):

```
create external table if not exists default.sword_purchases (
    Accept string,
    Host string,
    User_Agent string,
    event_type string,
    timestamp string,
    raw_event string
  )
  stored as parquet
  location '/tmp/sword_purchases'
  tblproperties ("parquet.compress"="SNAPPY");
```

Same command on 1 line for convenience:

```
create external table if not exists default.sword_purchases (Accept string, Host string, User_Agent stri
```

Introducing hue, a web based gui for hadoop. It uses port 8888 which is the same port used by jupyter notebook. Remember that our yml file moved jupyter notebook to 8889 so we can use hue on 8888. Also, note that we probably won't be able to run both hue and jupyter notebook at the same time, as droplets won't have enough memory.

Open up a web browser on your laptop and open an http page on your droplet's IP address on TCP port 8888. The login username is `cloudera` and the password is `cloudera`: `http://<droplet ip address>:8888/`

If you are running docker on your laptop: `http://localhost:8888/`

Let's explore the menu items of hue and see what each one does and how they are used. Note that we are now very similar to our query project. We have a similar architecture and now a web based gui to query.

Start a linux command line window and run presto in the presto container using hive's metadata for the catalog (same as last week):

```
docker-compose exec presto presto --server presto:8080 --catalog hive --schema default
```

See the tables in presto (same as last week):

```
presto:default> show tables;
```

See the schema for the table sword_purchases (same as last week):

```
presto:default> describe sword_purchases;
```

Query from the sword_purchases table (same as last week):

```
presto:default> select * from sword_purchases;
```

Query the number of rows from the sword_purchases table (same as last week):

```
presto:default> select count(*) from sword_purchases;
```

Use apache bench to make API calls to our web server (same as last week):

```
docker-compose exec mids \
  ab \
    -n 10 \
    -H "Host: user1.comcast.com" \
    http://localhost:5000/

docker-compose exec mids \
  ab \
    -n 10 \
    -H "Host: user1.comcast.com" \
```

```
    http://localhost:5000/purchase_a_sword

docker-compose exec mids \
  ab \
    -n 10 \
    -H "Host: user2.att.com" \
    http://localhost:5000/

docker-compose exec mids \
  ab \
    -n 10 \
    -H "Host: user2.att.com" \
    http://localhost:5000/purchase_a_sword
```

Same commands on 1 line for convenience:

```
docker-compose exec mids ab -n 10 -H "Host: user1.comcast.com" http://localhost:5000/
docker-compose exec mids ab -n 10 -H "Host: user1.comcast.com" http://localhost:5000/purchase_a_sword
docker-compose exec mids ab -n 10 -H "Host: user2.att.com" http://localhost:5000/
docker-compose exec mids ab -n 10 -H "Host: user2.att.com" http://localhost:5000/purchase_a_sword
```

Look at the flask window and see the web logs showing these transactions. Look at the kafkacat window and see the transacitons show up there. Look at the spark streaming job and see the transactions show up there (remember there is a 2 minute batch window, so we will need to wait at least 2 minutes for them to show up)

```
presto:default> select * from sword_purchases;
presto:default> select count(*) from sword_purchases;
```

Let's also look at the files in hdfs:

```
docker-compose exec cloudera hadoop fs -ls /tmp/sword_purchases
docker-compose exec cloudera hadoop fs -ls /tmp/sword_purchases | wc -l
```

Let's run a continuous loop of apache bench, every 10 seconds make 10 web API calls to our server. We will later use a control-C to stop this.

```
while true; do
  docker-compose exec mids \
    ab -n 10 -H "Host: user1.comcast.com" \
      http://localhost:5000/purchase_a_sword
  sleep 10
done
```

Same command on 1 line for convenience:

```
while true; do docker-compose exec mids ab -n 10 -H "Host: user1.comcast.com" http://localhost:5000/pur
```

Using presto watch our table grow:

```
presto:default> select count(*) from sword_purchases;
```

Watch files appear in hadoop hdfs:

```
docker-compose exec cloudera hadoop fs -ls /tmp/sword_purchases
docker-compose exec cloudera hadoop fs -ls /tmp/sword_purchases | wc -l
```

In addition to presto and hdfs command line, we can also use hue to run queries against hive and to see hdfs files.

Let's stop spark with a control-C, query the final count in sword_purchases, remove some of the hdfs files, and see the table count shrink. Use the following command to remove a file from hdfs:

```
docker-compose exec cloudera hadoop fs -rm /tmp/sword_purchases/<file name>
```

Tear down our docker cluster.

```
docker-compose down
```

Activity - Building Docker Images

Create a directory to build our image in and move to that directory:

```
mkdir -p ~/w205/docker/mytools
cd ~/w205/docker/mytools
```

Create a file `Dockerfile` in `~/w205/docker/mytools/` with the following contents:

```
FROM ubuntu:xenial
MAINTAINER Mark Mims <mark@digitalocean.com>

RUN apt-get -qq update \
  && apt-get -qq install -y jq apache2-utils
```

Generic format of the command to build a docker image from a Dockerfile:

```
docker build -t <tag> <path>
```

So, if we want to create an image called mytools using a Dockerfile located in our current directory, we would use the following command:

```
docker build -t mytools .
```

Also, tag can include namespace and versions... - `mytools` is implicitly `mytools:latest` - `mytools:0.0.1` or `mytools:some-string-here` - `markmims/mytools:0.0.1` or `midsw205/mytools:0.0.1`

Let's see the image we just created:

```
docker images
docker images | grep mytools
```

Let's now create and run a container based on our new image we just created and run a bash shell in that container:

```
docker run -it --rm mytools bash
which jq
exit
```

Let's run a couple of microservices to verify that our container is diffent from the base of ubuntu:xenial that we used. In the first command, pure ubuntu xenial we will not find the jq. In the second command, we will see jq because we installed it into our image (revisit the Dockerfile):

```
docker run -it --rm ubuntu:xenial which jq
docker run -it --rm mytools which jq
```

Here is an example of a more complicated Dockerfile. It was used to build the spark-minimal container. If we run it, it won't work because we don't have all of the dependent files. It's provided as an example of using some of the primitives: - `ENV` - `RUN` - `COPY` - `CMD`

```
FROM ubuntu:16.04
MAINTAINER Mark Mims <mark@digitalocean.com>

ENV SPARK_VERSION        2.2.0
ENV SPARK_HADOOP_VERSION 2.6
```

```
ENV SPARK_HOME /spark-$SPARK_VERSION-bin-hadoop$SPARK_HADOOP_VERSION
ENV JAVA_HOME  /usr/lib/jvm/java-8-oracle

ENV SPARK_TEMPLATE_PATH $SPARK_HOME/templates
ENV SPARK_CONF_PATH $SPARK_HOME/conf

ENV PATH $SPARK_HOME/bin:$PATH

RUN echo oracle-java8-installer shared/accepted-oracle-license-v1-1 select true | debconf-set-selections
  && apt-get update \
  && apt-get upgrade -y \
  && apt-get install -y software-properties-common \
  && add-apt-repository -y ppa:webupd8team/java \
  && apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF \
  && apt-get update \
  && apt-get install -y \
      curl \
      dnsutils \
      oracle-java8-installer \
  && apt-get purge -y software-properties-common \
  && apt-get autoremove -y \
  && curl -OL http://www-us.apache.org/dist/spark/spark-$SPARK_VERSION/spark-$SPARK_VERSION-bin-hadoop$
  && tar xf spark-$SPARK_VERSION-bin-hadoop$SPARK_HADOOP_VERSION.tgz \
  && rm spark-$SPARK_VERSION-bin-hadoop$SPARK_HADOOP_VERSION.tgz

COPY *-site.xml            $SPARK_TEMPLATE_PATH/
COPY *.properties          $SPARK_CONF_PATH/
COPY spark-defaults.conf   $SPARK_CONF_PATH
COPY spark-env.sh          $SPARK_CONF_PATH

COPY jars/* $SPARK_HOME/jars/

WORKDIR $SPARK_HOME

COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh entrypoint.sh
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["spark-shell"]
```

Examples of commonly used docker images on the internet and their Dockerfiles:

- nginx
- mongo
- mysql
- python
- etc...

Check out https://docs.docker.com/compose/gettingstarted/ for a good example of integrating a container you *build* into a cluster of containers.