UCB MIDS W205 Summer 2018 - Kevin Crook's agenda for Synchronous Session #12

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/cdh-minimal:latest
docker pull midsw205/spark-python:0.0.5
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 - Continuing with what we did last week, we will add an hadoop container to our cluster, and our python spark code will subscribe to the kafka topic and write the results into parquet format in hdfs. We will also introduce the batch python spark interface called spark-submit to submit our python files instead of using pyspark. We will run a couple of variations. One will transform the events. Another will separate the events.

Create the full stack directory in your droplet. Copy the yml file. Copy the python files we will be using.

```
mkdir ~/w205/full-stack/
cd ~/w205/full-stack
cp ~/w205/course-content/12-Querying-Data-II/docker-compose.yml .
cp ~/w205/course-content/12-Querying-Data-II/*.py .
```

Review the docker compose file (same as before). Like before, we may need to vi the file and change the directory mounts.

```
---
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/cdh-minimal:latest
  expose:
    - "8020" # nn
    - "50070" # nn http
    - "8888" # hue
  #ports:
  #- "8888:8888"
  extra_hosts:
    - "moby:127.0.0.1"

spark:
  image: midsw205/spark-python:0.0.5
  stdin_open: true
  tty: true
  volumes:
    - ~/w205:/w205
  expose:
    - "8888"
  ports:
    - "8888:8888"
  depends_on:
    - cloudera
  environment:
    HADOOP_NAMENODE: cloudera
  extra_hosts:
    - "moby:127.0.0.1"
  command: bash

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"
```

Startup the cluster (same as before):

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

Create a kafka topic called events (same as before):

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

We should see (same as before):

```
Created topic "events".
```

Review our python code for are web API server (same as before): - Take our instrumented web-app from before `~/w205/full-stack/game_api.py`

```
#!/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 code for our web API server (same as before):

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

For convenience, the command above on 1 line:

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

Something new: we will run kafkacat in continuous mode this time in a separate window so we can see events as they come through. We do this by leaving off the -e to give the endpoint:

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

For convenience, the command above on 1 line:

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

Apache Bench is a utility designed to stress test web servers using a high volume of data in a short amount of time. We will use apache bench as shown below to generate multiple requests of the same thing. The -n option is used below to specify 10 of each:

```
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
```

For convenience, the commands above on 1 line:

```
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
```

Last time we wrote the following spark code using python and submitted it using spark-submit. Let's review it before we go to this week's code. Note that it can only handle 1 schema for events and would break if we gave it 2 different schemas for events.:

~/w205/spark-from-files/separate_events.py

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


@udf('string')
def munge_event(event_as_json):
    event = json.loads(event_as_json)
    event['Host'] = "moe"
    event['Cache-Control'] = "no-cache"
```

```
    return json.dumps(event)


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

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

    munged_events = raw_events \
        .select(raw_events.value.cast('string').alias('raw'),
                raw_events.timestamp.cast('string')) \
        .withColumn('munged', munge_event('raw'))

    extracted_events = munged_events \
        .rdd \
        .map(lambda r: Row(timestamp=r.timestamp, **json.loads(r.munged))) \
        .toDF()

    sword_purchases = extracted_events \
        .filter(extracted_events.event_type == 'purchase_sword')
    sword_purchases.show()
    # sword_purchases \
        # .write \
        # .mode("overwrite") \
        # .parquet("/tmp/sword_purchases")

    default_hits = extracted_events \
        .filter(extracted_events.event_type == 'default')
    default_hits.show()
    # default_hits \
        # .write \
        # .mode("overwrite") \
        # .parquet("/tmp/default_hits")


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

We used this command to run it:

```
docker-compose exec spark \
  spark-submit /w205/spark-from-files/separate_events.py
```

For convenience, the command above on 1 line:

```
docker-compose exec spark spark-submit /w205/spark-from-files/separate_events.py
```

Let's change our previous code to handle multiple schemas for events:

~/w205/full-stack/just_filtering.py

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


@udf('boolean')
def is_purchase(event_as_json):
    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 \
        .read \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "kafka:29092") \
        .option("subscribe", "events") \
        .option("startingOffsets", "earliest") \
        .option("endingOffsets", "latest") \
        .load()

    purchase_events = raw_events \
        .select(raw_events.value.cast('string').alias('raw'),
                raw_events.timestamp.cast('string')) \
        .filter(is_purchase('raw'))

    extracted_purchase_events = purchase_events \
        .rdd \
        .map(lambda r: Row(timestamp=r.timestamp, **json.loads(r.raw))) \
        .toDF()
    extracted_purchase_events.printSchema()
    extracted_purchase_events.show()


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

Use the following code to run it using spark-submit (similar to last time):

```
docker-compose exec spark \
  spark-submit /w205/full-stack/just_filtering.py
```

For convenience, the command above on 1 line:

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

Let's play around with our flask web API server. Stop the flask web API server. Add a new event type of purchase_a_knife. Restart the flask web API server. Modify our spark code to handle it.

```
@app.route("/purchase_a_knife")
def purchase_a_knife():
    purchase_knife_event = {'event_type': 'purchase_knife',
                            'description': 'very sharp knife'}
    log_to_kafka('events', purchase_knife_event)
    return "Knife Purchased!\n"
```

Let's modify our spark code to write out using massively parallel processing to hadoop hdfs in parquet format. Last week, we got an error if the directory already existed and had to delete it or pick a new name for the directory. This week we will use the overwrite option. Remember that we want to do it this way so we can read it back in quickly if it's a large data set. `full-stack/filtered_writes.py`

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


@udf('boolean')
def is_purchase(event_as_json):
    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 \
        .read \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "kafka:29092") \
        .option("subscribe", "events") \
        .option("startingOffsets", "earliest") \
        .option("endingOffsets", "latest") \
        .load()
```

```
    purchase_events = raw_events \
        .select(raw_events.value.cast('string').alias('raw'),
                raw_events.timestamp.cast('string')) \
        .filter(is_purchase('raw'))

    extracted_purchase_events = purchase_events \
        .rdd \
        .map(lambda r: Row(timestamp=r.timestamp, **json.loads(r.raw))) \
        .toDF()
    extracted_purchase_events.printSchema()
    extracted_purchase_events.show()

    extracted_purchase_events \
        .write \
        .mode('overwrite') \
        .parquet('/tmp/purchases')


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

Submit it to spark using spark-submit (same as before)

```
docker-compose exec spark \
  spark-submit /w205/full-stack/filtered_writes.py
```

For convenience, the command above on 1 line:

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

Check our hadoop hdfs to make sure it's there

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

Startup a jupyter notebook. Remember that to access it from our laptop web browser, we will need to change the IP address to the IP address of our droplet.

```
docker-compose exec spark \
  env \
    PYSPARK_DRIVER_PYTHON=jupyter \
    PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser --port 8888 --ip 0.0.0.0 --allow-root' \
  pyspark
```

For convenience, the command above on 1 line:

```
docker-compose exec spark env PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-b
```

In our jupyter notebook, run each of the following in a separate cell.

```
purchases = spark.read.parquet('/tmp/purchases')
purchases.show()
purchases.registerTempTable('purchases')
purchases_by_example2 = spark.sql("select * from purchases where Host = 'user1.comcast.com'")
purchases_by_example2.show()
df = purchases_by_example2.toPandas()
df.describe()
```

Let's discuss the "easy" spark workflow using the "netflix architecture": We usually receive files in csv or json format, which in a cloud environment, we may want to put in object store (such as AWS S3). We load

the file (sequentially - not parallel) into spark. We filter and process the data until we get it into spark tables like we need for analytics. We use SQL as much as we can, go to lambda transforms for things that we cannot, and using special purpose libraries, such as MLLib for machine learning. We save the file out using massively parallel processing to object store. We may also write our results out to object store. At this point our cluster can die and our data in object store will outlive the cluster (similar concept to our docker volume mount outliving the docker container). Next time we need our data, we can read it back in using massively parallel processing, which will be much faster than the original sequential read.

Note: netflix architecture is actually a very specific architecture using object store (AWS S3) and an elastic form of hadoop cluster (AWS EMR - Elastic MapReduce). However, industry slang tends to call any use of object store => load and process in a temporary cluster => save results to object store as "netflix architecture".

Tear down our cluster (as before):

```
docker-compose down
```