UCB MIDS W205 Summer 2018 - Kevin Crook's agenda for Synchronous Session #13

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: We will introduce Hive wich is an SQL based data warehouse platform that runs on top of hadoop hdfs. We will use hive to create a schema on read for our parquet files stores in hdfs. We will run some queries from hive against our parquet files stored in hdfs. We will use python code to see another way to interact with the hive metastore. We will introduce another tool, Presto, to do ad hoc queries of our parquet files in hadoop hdfs. (At this point, we have basically built the architecture that we used for the bike share data in the Google BigQuery) We will introduce Spark Streaming. So far we have just read the entire topic on kafka and written everything to hdfs. Now we want to define batches of 10 second intervals to write parquet to hdfs. We will see how immutability plays into this.

Create the full-stack2 directory, copy the yml and python files:

```
git pull in ~/w205/course-content
mkdir ~/w205/full-stack2/
cd ~/w205/full-stack2
cp ~/w205/course-content/13-Understanding-Data/docker-compose.yml .
docker-compose pull
cp ~/w205/course-content/13-Understanding-Data/*.py .
```

Review our docker-compose.yml file and update directories as needed. Note that we have added a new container called presto. Presto is the basis of querying similar to what we did earlier in the semester with Google Big Query.

```
---
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:
    #  - "8888:8888"
    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"
```

Starup the docker cluster:

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

Review our python flask web API code (same as 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 web API server:

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

Same command as above on 1 line for convenience:

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

As we introduced last week, we will run kafkacat in continuous mode in a separate window:

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

Same command as above on 1 line for convenience:

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

As we introduced last week, we will use Apache Bench to stress test our web API server. Remember we will spoof host name for debugging purposes.

```
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 as above, but on 1 line each 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
```

As we did last week, we have python spark code we will use spark-submit to run which will read and process our kafka topic. Note that it will write all data at once to the hadoop hdfs. (Later today, we will use Spark Streaming to write it out in 10 second batches continuously)

```
#!/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()
```

Run using spark-submit (same as before):

```
docker-compose exec spark spark-submit /w205/full-stack2/filtered_writes.py
```

Check and make sure the file is created in hadoop hdfs:

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

We will now introduce Hive. Hive is an SQL based data warehousing platform that runs on top of hadoop hdfs. It's scale out SQL with both horizontal and vertical partitioning. Hive would be comparable to commercial products such as Teradata, Netezza, Amazon Redshift, etc.

Start a hive command prompt in our hadoop container:

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

We will run some SQL to impose a schema-on-read on top of our parquet files. Most traditional databases use schema-on-write. Schema-on-read allows us to impose schema just in time for querying and to impose multiple schemas on the same data, and works hand and hand with our big data architecture concepts such as immutability, content delivery networks, etc. Hive calls its repository of schema definitions (also called metadata) the Hive Metastore. The hive metastore can be used by most big data architecture tools to pull data from hive, including used by spark sql.

```
create external table if not exists default.purchases2 (
    Accept string,
    Host string,
    User_Agent string,
    event_type string,
    timestamp string
  )
  stored as parquet
  location '/tmp/purchases'
  tblproperties ("parquet.compress"="SNAPPY");
```

Same command on 1 line for convenience:

```
create external table if not exists default.purchases2 (Accept string, Host string, User_Agent string,
```

Another way is to use python spark to impose schema-on-read on parquet tables stored in hdfs.

Starup a pyspark shell:

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

The following code will impose schema-on-read on the parquet files for spark sql:

```
df = spark.read.parquet('/tmp/purchases')
df.registerTempTable('purchases')
query = """
create external table purchase_events
  stored as parquet
  location '/tmp/purchase_events'
  as
  select * from purchases
"""
spark.sql(query)
```

Same command on 1 line for convenience:

```
spark.sql("create external table purchase_events stored as parquet location '/tmp/purchase_events' as s
```

In addition to pyspark, we can do the same thing in our python code we will submit using spark-submit:

```
#!/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") \
        .enableHiveSupport() \
        .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.registerTempTable("extracted_purchase_events")

    spark.sql("""
```

```
        create external table purchases
        stored as parquet
        location '/tmp/purchases'
        as
        select * from extracted_purchase_events
    """)


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

Command to run our python spark code using pyspark:

```
docker-compose exec spark spark-submit /w205/full-stack2/write_hive_table.py
```

See if the files are present in hadoop hdfs (same as before):

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

We now introduce Presto. Presto is a query engine which can query data from numerous different types of sources, including hive. We will now use it to query our hive data we just defined. It will connect to the hive metastore and pull our metadata (schemas). Note that at this point we have basically built out an architecture similar to the architecture in our query project where we queried using Google Big Query against the bike share dataset.

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

Play around with presto to see a table list, the schema of a table, query a table, etc. “‘ presto:default> show tables; Table
——— purchases (1 row)

Query 20180404_224746_00009_zsma3, FINISHED, 1 node Splits: 2 total, 1 done (50.00%) 0:00 [1 rows, 34B] [10 rows/s, 342B/s]

presto:default> describe purchases; Column | Type | Comment ————+———+——— accept | varchar |
host | varchar |
user-agent | varchar |
event_type | varchar |
timestamp | varchar |
(5 rows)

Query 20180404_224828_00010_zsma3, FINISHED, 1 node Splits: 2 total, 1 done (50.00%) 0:00 [5 rows, 344B] [34 rows/s, 2.31KB/s]

presto:default> select * from purchases; accept | host | user-agent | event_type | timestamp
———+————-+————+————-+————————- / | user1.comcast.com | ApacheBench/2.3
| purchase_sword | 2018-04-04 22:36:13.124 / | user1.comcast.com | ApacheBench/2.3 | purchase_sword |
2018-04-04 22:36:13.128 / | user1.comcast.com | ApacheBench/2.3 | purchase_sword | 2018-04-04 22:36:13.131
/ | user1.comcast.com | ApacheBench/2.3 | purchase_sword | 2018-04-04 22:36:13.135 / | user1.comcast.com
| ApacheBench/2.3 | purchase_sword | 2018-04-04 22:36:13.138 . . . “‘

Up until now, we have been inferring the schema from kafka. Suppose we want to make that a bit more formal and define a schema to be imposed on the kafka data? The code below has been enhanced to do this. This will get us ready for spark streaming which needs to have a defined schema.

```
#!/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)
    |-- timestamp: 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 \
        .read \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "kafka:29092") \
        .option("subscribe", "events") \
        .option("startingOffsets", "earliest") \
        .option("endingOffsets", "latest") \
        .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.*')

    sword_purchases.printSchema()
    sword_purchases.show(100)


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

Run our python spark code using spark-submit (as before):

```
docker-compose exec spark spark-submit /w205/full-stack2/filter_swords_batch.py
```

Now we introduce Spark Streaming. Spark Streaming will run continuously and as we specified in the code below, every 10 seconds it will read kafka, process our data, and write it to the console.

```
#!/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)
    |-- timestamp: 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.*')

    query = sword_purchases \
        .writeStream \
        .format("console") \
        .start()

    query.awaitTermination()


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

Run our new spark streaming code using spark-submit. Note that it will run indefinitely, so we will need to run it in a separate linux command line, and we will need to eventually stop it using control-C. We will keep it running for now.

```
docker-compose exec spark spark-submit /w205/full-stack2/filter_swords_stream.py
```

We will see it write our previous kafka data to the console and wait for more data. We will test adding more data using apache bench as we did before:

```
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
```

We will now enhance our python spark code to write to hadoop hdfs in parquet format instead of just writing to the console. Note that with schema-on-read and immutability, we can keep putting files in the hdfs directory and they will become part of the data.

```
#!/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)
    |-- timestamp: 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="10 seconds") \
        .start()

    sink.awaitTermination()


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

Run our spark python code using spark-submit (as we did before):

```
docker-compose exec spark spark-submit /w205/full-stack2/write_swords_stream.py
```

We will see it process the events already on the kafka topic. We need to verify that new data on the kafka topic will be written out to our parquet files in the same directory in 10 second batches. Using Apache Bench, we will create more test data. We should see the test data go through flash, kafka, and now spark streaming. Note that immutability, a hallmark of big data architecture, and schema-on-read both play into making this possible. We will use a bash shell while loop that will run indefinitely until we stop it with control-C.

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

Verify that it wrote to the hadoop hdfs:

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

We can also use hive to impose a schema and then test querying from both hive and presto. You may need to stop and restart presto as presto reads the hive metastore when started:

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

Tear down our cluster:

```
docker-compose down
```