Project 3 - w205 Fundamentals of Data Engineering - Kevin Martin

Pipeline and Analytics for Mid-EVIL III: Ghouls Just Wanna Have Fun

The forthcoming game Mid-EVIL III (ME3) is the highly anticipated conclusion of the Mid-Evil Saga. Agondell has fallen to the ghouls. Few will survive. If they want to be the fortunate ones, our players will have to claim the most powerful swords in the land. If they want to live their lives right, they will have to join together in guilds to resist the evil influence on the land. And when the working day is done they, like the ghouls, will ultimately have fun.

Pipeline Description

Here is a diagram of the proposed analytics pipeline.

Event Generation

Technologies:

- Python Requests Library
- Flask API for Python
- JSON encoding

Description:

Web requests are generated by a python script running the requests library. These get sent out to a flask API server. In the finished pipleline the API server will handle the business logic of the game. Here it is just used to augment the web requests and send them along down the analytics pipeline.

Event Queue

Technologies:

- Kafka

Description:

The web requests are written out to Kafka by the flask server. They are posted to a topic called "events" in kafka. No specific retention time is used, so events are maintained for the default period of 168 hours. In the next step, Spark will pick the topics out of the Kafka queue.

Ingestion

Technologies:

- Spark
- Parquet Format Files With Snappy compression

Description:

A spark structured streaming job takes the JSON encoded events from the "events" topic in Kafka and lands them into HDFS as parquet formatted files (with Snappy compression). They are put into either a sword purchases or a guild joins table depending on event.

Query

Technologies:

- Hadoop Distributed File Store (HDFS)
- Hadoop Hive Metastore Schema Registry
- Presto Query Engine
- Jupyter notebook

Description:

Presto is used to query the files stored in HDFS. It uses table schema that is defined in HIVE metastore. The presto engine is connected up to this very Jupyter notebook to aid in the exploration.

An alternate way to connect to presto in the terminal appears in the `KM_notes.md` file

Setting up the pipeline

Bringing up the pipeline will involve 3 separate terminal windows to run the various processes.

- Terminal 1: Input most commands. The workhorse terminal.
- Terminal 2: Runs the flask API.
- Terminal 3: Runs the spark streaming job.

Step 1: Bring up the cluster

The docker compose cluster contains the pipeline. The various components are housed in different docker containers.

Schematically our pipeline looks like so:... **IMAGE OF PIPELINE FIGURE THING**

The cluster is brought up using the docker compose command. The "-d" causes the cluster to run in detatched mode. If the cluster is not run in detatched mode, make sure that the cluster has its own dedicated terminal window.

```
# Terminal 1
docker-compose up -d
```

Step 2: Start up the flask app

The flask app simulates the game's API. The command below needs to run in its own terminal window. It takes terminal focus.

Note that the flask app will create a topic called 'events' in kafka when attempts to write out and finds that there is no such topic.

```
# Terminal 2
docker-compose exec mids env FLASK_APP=/external/game_api.py f
lask run --host 0.0.0.0
```

Step 3: Start up the spark streaming job

This job picks up things from the events topic and lands them in the database. It needs its own terminal window. It takes up focus.

```
# Terminal 3
docker-compose exec spark spark-submit ../external/spark_strea
m.py
```

Step 4: Hit the API with some python generated web requests

Note the docker build for mids container has w205 as the home directory, you can't use '~/external/event_gen.py' because that looks for a folder named 'external' in w205 similarly, you can't use 'external/event_gen.py' for the same reason.

```
# Terminal 1
docker-compose exec mids python /external/event_gen.py
```

Step 5: Register table schema in hive

note that passing commands directly to hive cli is depriciated. Commands should be sent to beeline. (see KM_notes for finding beeline)

```
# Terminal 1
docker-compose exec cloudera beeline
```

just hit enter when prompted for username and password. We don't have those things.

```
# Terminal 1
!connect jdbc:hive2://localhost:10000/default
```

Set up the purchases table

```
# Terminal 1
create external table if not exists default.purchases (
    raw_event string,
    timestamp string,
    Accept string,
    Host string,
    User_Agent string,
    event_type string,
    user_name string,
    sword_name string,
    event_id string
  )
  stored as parquet
  location '/tmp/sword_purchases'
  tblproperties ("parquet.compress"="SNAPPY");
```

Set up the gjoins table

```
# Terminal 1
create external table if not exists default.gjoins (
    timestamp string,
    raw_event string,
    Accept string,
    Host string,
    User_Agent string,
    event_type string,
    user_name string,
    guild_name string,
    event_id string
  )
  stored as parquet
  location '/tmp/guild_joins'
  tblproperties ("parquet.compress"="SNAPPY");
```

Step 6: Connect Presto to Jupyter Notebok Query with Presto

Get the ip address of the presto container

In [1]:
```
!docker-compose exec presto ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN g
roup default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noque
ue state UP group default
    link/ether 02:42:ac:12:00:05 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.5/16 brd 172.18.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

In [2]:
```
#Python Library installs if neccessary

#!pip install pyhive[presto] # DB driver library (NOT INCLUDED BY DEFA
#!pip install pandas
#!pip install sqlalchemy # ORM for databases
#!pip install ipython-sql # SQL magic function
```

In [3]:
```
#create connection object and point the sql magic at it. Do imports

#load up the sql magic
%reload_ext sql
# if it were the first time i would use # %load_ext sql

import pandas as pd
import numpy as np
```

In [4]:
```
# for engines that do not support autommit (do this or presto throws e
%config SqlMagic.autocommit=False

#looks for the 'default' schema in the 'hive' catalog.
%sql presto://172.18.0.5:8080/hive
```

Out[4]:
```
'Connected: @hive'
```

In [5]:
```
%sql SHOW tables
```

```
 * presto://172.18.0.5:8080/hive
Done.
```

Out[5]:

Table
gjoins
purchases

In [6]:
```
nPurch = %sql SELECT COUNT(*) FROM purchases
nPurch = pd.DataFrame(nPurch) #store it in a dataframe
nPurch = nPurch.iloc[0,0]

print("There are {:.0f} entries in the 'purchases' table".format(nPurc
```

```
 * presto://172.18.0.5:8080/hive
Done.
There are 718 entries in the 'purchases' table
```

In [7]:
```
nJoins = %sql SELECT COUNT(*) FROM gjoins
nJoins = pd.DataFrame(nJoins) #store it in a dataframe
nJoins = nJoins.iloc[0,0]

print("There are {:.0f} entries in the 'gjions' table".format(nJoins))
```

```
 * presto://172.18.0.5:8080/hive
Done.
There are 282 entries in the 'gjions' table
```

In [8]:
```
%sql SELECT * FROM purchases LIMIT 1
```

```
 * presto://172.18.0.5:8080/hive
Done.
```

Out[8]:

raw_event	timestamp	accept	host	user_agent	eve
{"Content-Length": "49", "event_type": "purchase_sword", "event_id": "a3e3f677db824077b3f2dd94b10b8a35", "Content-Type": "application/x-www-form-urlencoded", "Host": "localhost:5000", "Accept": "*/*", "User-Agent": "ME3_ios/1.1", "Connection": "keep-alive", "sword_name": "goatslayer", "user_name": "trevnorthistlebrush", "Accept-Encoding": "gzip, deflate"}	2020-08-09 18:31:10.868	*/*	localhost:5000	ME3_ios/1.1	purchas

Note that if we were to run the event generation again, we would continute to get updated results as the API dropped the events in the kafka topic and the spark streaming job continued to land them in HDFS.

Answer Business Questions

This section illustrates some business problems that could be analyzed with the pipeline. In our case of course, all of the data is synthetic.

Question 1

What is the most popular sword in the game?

In [9]:
```
%%sql result_set <<
SELECT count(*) AS purch_count, sword_name
FROM purchases
GROUP BY sword_name
ORDER BY purch_count DESC
```

```
 * presto://172.18.0.5:8080/hive
Done.
Returning data to local variable result_set
```

In [10]:
```
result_set = result_set.DataFrame()
result_set.head()
```

Out[10]:

	purch_count	sword_name
0	174	needle
1	138	swordofstabbing
2	137	goatslayer
3	136	dragonsbane
4	133	orphanmaker

In [11]:
```
swo_cnt = max(result_set['purch_count'])
pop_swo = result_set.loc[result_set['purch_count'] == swo_cnt,'sword_n
print("The most popular sword is {1}. It was purchased {0} times.".for
```

```
The most popular sword is needle. It was purchased 174 times.
```

Question 2

Which user has purchased the most swords?

In [12]:
```
%%sql result_set <<
SELECT count(*) AS purch_count, user_name
FROM purchases
GROUP BY user_name
ORDER BY purch_count DESC
```

```
 * presto://172.18.0.5:8080/hive
Done.
Returning data to local variable result_set
```

In [13]:
```
result_set = result_set.DataFrame()
result_set.head(15)
```

Out[13]:

	purch_count	user_name
0	79	kaladinstormblessed
1	72	twiddledee
2	70	trevnorthistlebrush
3	69	the_raven
4	67	CloudStrife
5	65	greg_of_albion
6	64	twiddledum
7	62	goatman_dan
8	59	mistresspain
9	59	ronconcama
10	52	Solid_Snake

In [14]:
```
prch_cnt = max(result_set['purch_count'])
usr_name = result_set.loc[result_set['purch_count'] == prch_cnt,'user_
print("The user who purchased most swords is {1}. They made {0} purcha
```

```
The user who purchased most swords is kaladinstormblessed. They made
79 purchases.
```

Question 3

How many access the app via android versus ios?

Pull out the user agents from the purchases table

In [15]:
```
%%sql p_result_set <<
SELECT count(*) AS agent_cnt, user_agent
FROM purchases
GROUP BY user_agent
ORDER BY user_agent DESC
```

```
 * presto://172.18.0.5:8080/hive
Done.
Returning data to local variable p_result_set
```

In [16]:
```
p_result_set = p_result_set.DataFrame()
p_result_set.head()
```

Out[16]:

	agent_cnt	user_agent
0	191	ME3_ios/1.1
1	173	ME3_ios/1.0
2	161	ME3_android/2.1
3	193	ME3_android/2.0

In [17]:
```
%%sql j_result_set <<
SELECT count(*) AS agent_cnt, user_agent
FROM gjoins
GROUP BY user_agent
ORDER BY user_agent DESC
```

```
 * presto://172.18.0.5:8080/hive
Done.
Returning data to local variable j_result_set
```

In [18]:
```
j_result_set = j_result_set.DataFrame()
j_result_set.head()
```

Out[18]:

	agent_cnt	user_agent
0	66	ME3_ios/1.1
1	77	ME3_ios/1.0
2	72	ME3_android/2.1
3	67	ME3_android/2.0

In [19]:
```
ttl_result_set = pd.DataFrame()
ttl_result_set['user_agent'] = j_result_set['user_agent']
ttl_result_set['agent_cnt'] = p_result_set['agent_cnt'] + j_result_set
ttl_result_set.head()
```

Out[19]:

	user_agent	agent_cnt
0	ME3_ios/1.1	257
1	ME3_ios/1.0	250
2	ME3_android/2.1	233
3	ME3_android/2.0	260

In [20]:
```
agent_cnt_dict = dict()

for ii in range(0,len(ttl_result_set)):
    os_type = ttl_result_set['user_agent'][ii].split("/")[0].split("_"
    if not os_type in agent_cnt_dict:
        agent_cnt_dict[os_type] = 0
    agent_cnt_dict[os_type] = agent_cnt_dict[os_type] + ttl_result_set

for ii in agent_cnt_dict:
    print("{0} users use {1}".format(agent_cnt_dict[ii],ii))
```

```
507 users use ios
493 users use android
```

End Material

Acknowledgements

- The Spark Documentation
  - https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html)
- The Flask Documentation
  - https://flask.palletsprojects.com/en/1.1.x/quickstart/ (https://flask.palletsprojects.com/en/1.1.x/quickstart/)
- The python-kafka Documentation
  - https://github.com/dpkp/kafka-python (https://github.com/dpkp/kafka-python)
- The json library Documentation
  - https://docs.python.org/3/library/json.html (https://docs.python.org/3/library/json.html)
- This post on the request library
  - https://www.datacamp.com/community/tutorials/making-http-requests-in-python (https://www.datacamp.com/community/tutorials/making-http-requests-in-python)
- The requests library Documentation
  - https://requests.readthedocs.io/en/latest/ (https://requests.readthedocs.io/en/latest/)
  - https://requests.readthedocs.io/en/master/user/quickstart/#custom-headers (https://requests.readthedocs.io/en/master/user/quickstart/#custom-headers)
- Hive documentation describing external tables
  - https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManagedandExternalTables (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManagedandExternalTables)
- This post on connecting Presto to jupyter
  - https://towardsdatascience.com/jupyter-magics-with-sql-921370099589 (https://towardsdatascience.com/jupyter-magics-with-sql-921370099589)

Document Backup

In [21]:
```
#Create a backup of the jupyter notebook in a format for where changes
!jupyter nbconvert Project_3.ipynb --to="python" --output="backups/Pro
!jupyter nbconvert Project_3.ipynb --to markdown --output="backups/Pro

# Also archiving this bad boy
!jupyter nbconvert Project_3.ipynb --to html --output="backups/Project
```

```
[NbConvertApp] Converting notebook Project_3.ipynb to python
[NbConvertApp] Writing 11707 bytes to backups/Project_3.py
[NbConvertApp] Converting notebook Project_3.ipynb to markdown
[NbConvertApp] Writing 18521 bytes to backups/Project_3.md
[NbConvertApp] Converting notebook Project_3.ipynb to html
[NbConvertApp] Writing 316477 bytes to backups/Project_3.html
```

In [22]:
```
#fixes the broken image in the .md file
!sed -i -e 's|(\./images/Pipe|(\.\./images/Pipe|g' backups/Project_3.m
```