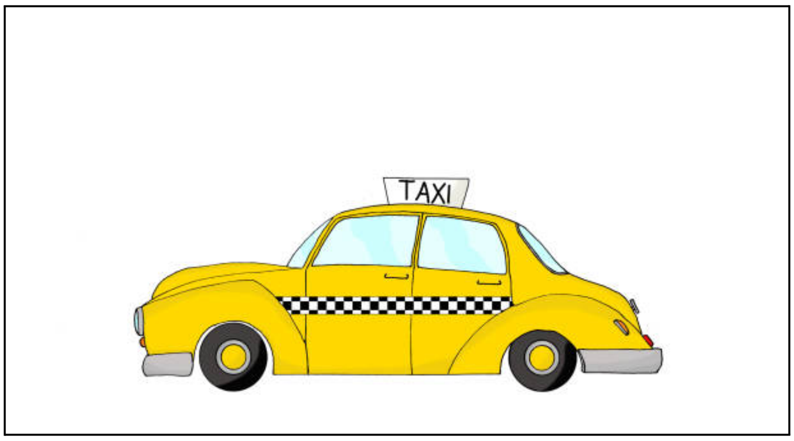
Project Report: Big Data Management

Project Name

NYC Yellow Taxi Ride Analysis



Project Members

Kartik Nautiyal <knautiyal@wpi.edu>
Nitaant Vyas<nvyas@wpi.edu>

Abstract

The volume of data being produced every second has been increasing exponentially recently. This calls for methods to look at data in a quick and effective way. Traditional techniques involved storing data into databases and running analytics jobs on them. This process takes so much time that eventually when insights are drawn from the data, it becomes irrelevant. Real-time analytics and monitoring is the way to go forward. This project is an attempt in the same direction and looks at NYC cab transactions in a real-time monitoring perspective and creates a dashboard for the same.

Problem Statement

In this project, we are looking to create an ETL pipeline by streaming data from AWS S3 using Kafka and Spark to populate a MongoDB database and create a real-time dashboard using MongoDB charts.

Technology Stack

The technologies that were chosen for this project are as shown in the data flow diagram:

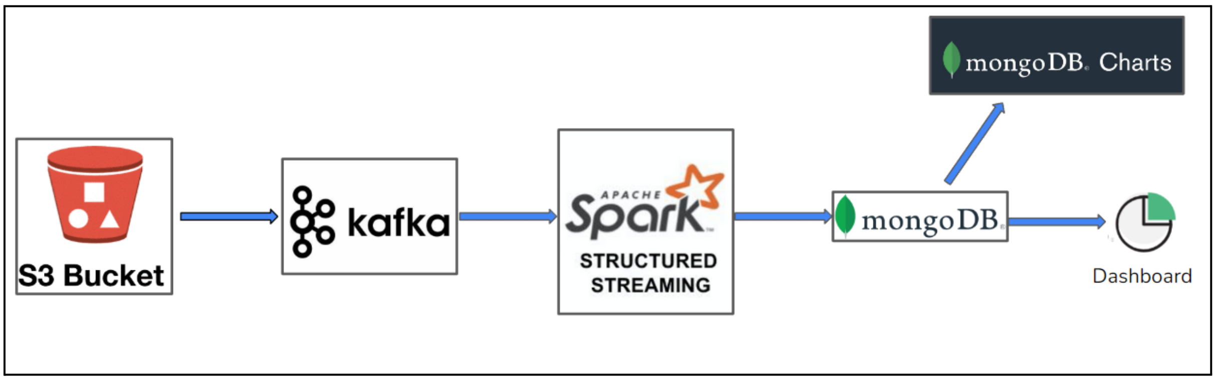


Figure 1: Data Flow Diagram

Docker:
Docker is a software framework for building, running, testing and deploying applications quickly. Docker can be quickly deployed into any environment and it can be said with certainty that the application inside will work..

Docker makes it easier to set up big data technologies like Hadoop, Spark and Kafka. Since the project involves running these technologies on a local environment, it makes more sense to isolate these technologies in different containers so as to closely resemble a real scenario.

Docker establishes the previously mentioned advantages by containerising applications. This gives it a standard way of running. It essentially is an operating system for the containers. These containers can be built by a standard dockerfile. Pre-built containers can also be used from docker hub which contains a number of previously built containers which can be chosen based on use-cases.

Amazon Web Service S3:
Amazon Simple Storage Service (S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. It gives the user the ability to store and retrieve any amount of data. It follows a pay-as-you-go and only requires the user to pay for the amount of data that the user is consuming on the bucket.

As cloud computing solutions are becoming more and more powerful, hosting large scale data is shifting to the cloud. Hence a web-based storage system was chosen.

Within the options in the cloud infrastructure options, AWS S3 was found to be one of the cheaper ones. Amazon is extremely reliable and companies across the world trust Amazon with their data.

Each S3 service can have many buckets and each bucket can store large amounts of data. These buckets are located in servers located across the world. There are many server locations but the one closest for this project is the one at Virginia called US-East-1. It also has access tokens and secret keys which allows the user to access the data via an API.

The S3 storage was used to stream in records into the producer file. This file rests inside the jupyter docker container. Boto3 library in Python was used to make callouts to the CSV files inside the bucket. To avoid the download of an entire dataset onto the disk or into the ram of the local machine, a reader from the codecs library is used to iterate over the CSV file in the bucket.

Kafka:

Kafka is an event streaming platform. It acts as a decoupling unit between data producing units and data consuming units. It can handle the different rates of data production and data consumption.

For the scope of this project, data from the bucket was to be streamed into spark structured streaming for analysis. Given that the bucket is in a web server and spark on a local container, a method of having data available in local somehow was needed. Kafka acts as the perfect decoupling agent between S3 and Spark for this use-case.

Kafka is able to perform this decoupling by storing data in the form of messages which are stored in Kafka topics. Each topic can be a multi-producer and a multi-subscriber. Unlike traditional queues, data is not deleted after consumption. Instead the data is retained by specifying the retention time of the Kafka topic. By default, the retention time is 1 week.

The bitnami image for Kafka was pulled from dockerhub. Given that zookeeper is a dependency for kafka, the zookeeper bitnami container was pulled as well.

Spark

Apache Spark is a data processing framework and is used for large scale data processing. Along with core spark, spark ships with three powerful libraries.

As it has python integration (PySpark), reading through the documentation and the implementation becomes more intuitive. It can also be used to write aggregation queries and the result can be saved in the mongoDB due to its easy to implement connection.

Spark image from bitnami is used for this project. Two containers are brought up using the same image. One is specified to be the master and the other to be the worker. The master container is accessed using the jupyter container via the Pyspark library in python.

MongoDB and MongoCharts

MongoDb is an open source NoSQL database management program and has distributed data management. The data in the MongoDB can be easily visualized in MongoCharts.

As in our project, our goal was to visualize real time data on a dashboard, we used MongoDB charts. MongoDB Charts is a tool to create visual representations of your MongoDB data. Data visualization is a key component to providing a clear understanding of your data and the Spark-Mongo Connector makes it extremely easy to connect spark to MongoDB

MongoDB and Mongo charts images were pulled and containerized. Mongo charts can be easily connected to MongoDB by providing the URI for MongoDB.

Dataset Description

The dataset we are using for this project is from kaggle[1]. The dataset consists of 19 rows and it is approximately around 8 GB in total. For the sake of this project, we are duplicating it to make it 16 GB in total. Following shows the structure of the dataset:

Field Name	Description
VendorID	A code indicating the TPEP provider that provided the record. 1. Creative Mobile Technologies 2. VeriFone Inc.
tpep_pickup_datetime	The date and time when the meter was engaged.
tpep_dropoff_datetime	The date and time when the meter was disengaged.
Passenger_count	The number of passengers in the vehicle. This is a driver-entered value.
Trip_distance	The elapsed trip distance in miles reported by the taximeter.
Pickup_longitude	Longitude where the meter was engaged.
Pickup_latitude	Latitude where the meter was engaged.
RateCodeID	The final rate code in effect at the end of the trip. 1. Standard rate 2. JFK 3. Newark 4. Nassau or Westchester 5. Negotiated fare 6. Group ride
Store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory

	before sending it to the vendor, aka “store and forward,” because the vehicle did not have a connection to the server. Y= store and forward trip N= not a store and forward trip
Dropoff_longitude	Longitude where the meter was disengaged.
Dropoff_ latitude	Latitude where the meter was disengaged.
Payment_type	A numeric code signifying how the passenger paid for the trip. 1. Credit card 2. Cash 3. No charge 4. Dispute 5. Unknown 6. Voided trip
Fare_amount	The time-and-distance fare calculated by the meter.
Extra	Miscellaneous extras and surcharges. Currently, this only includes. the $0.50 and $1 rush hour and overnight charges.
MTA_tax	0.50 MTA tax that is automatically triggered based on the metered rate in use.
Improvement_surcharg e	0.30 improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.
Tip_amount	Tip amount – This field is automatically populated for credit card tips.Cash tips are not included.
Tolls_amount	Total amount of all tolls paid on the trip.
Total_amount	The total amount charged to passengers. Does not include cash tips.

The dataset contains all the features that are useful to analyze taxi services in NYC and contains data of the months January 2015 and January to March of 2016 and is in the csv format. For the current use-case the timestamps are ignored and each record is treated as though it is being generated live from any transaction that is happening real-time

Challenges

While executing this project, some hurdles and obstacles were faced. For most of them we found a work around. Some of them are listed as follows:

Kafka Python Libraries:
Kafka has three different libraries in python. While all three implementations are easy to use, there are a few specific options which were found to be tough to implement on the Confluent-Kafka library. Particularly, the retention time of Kafka. Given the specifications of

the machine that the project was executed on, retention time could have been reduced to save on the much precious memory.

Spark Structured Streaming:
Although Spark Structured Streaming is extremely powerful, it was found to be a little difficult to debug. As soon as the write stream command was executed, the streaming job would start and it was not possible to make any changes post that. But without executing, it was not easy to understand the SQL operations.

For the work around, a subset of the data was kept in the local machine. This subset was imported as a Spark SQL dataframe. This Spark SQL dataframe was used for all operations and tested upon before running a batch streaming job. This would ensure that if there was any syntax error in operation, it would be caught before running a batch job.

Pipeline (Architecture)

Following figure shows the broad architecture of what was implemented. Each of the blue blocks refers to a docker container. The arrows mark the communication and data flow that is taking place between the containers and the corresponding port numbers are mentioned in brackets.

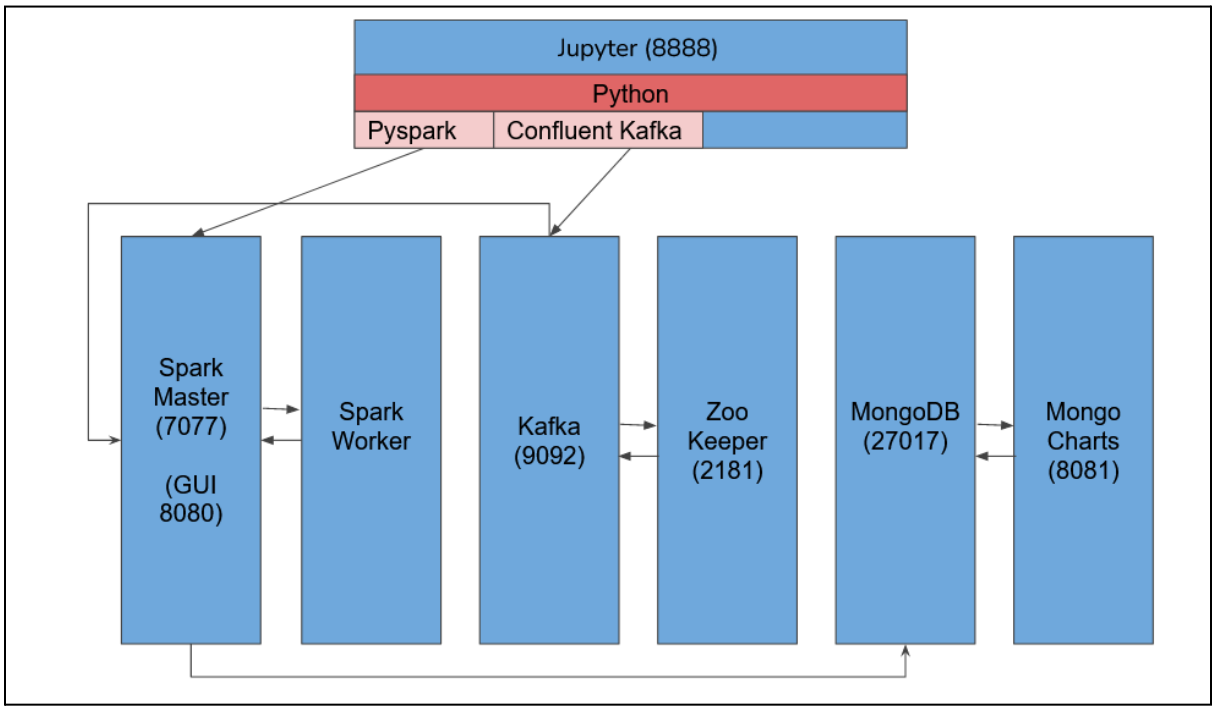


Figure 2: Architecture

Following are the components and their configurations:

Orchestrator: Jupyter Container

This is one of the most important components in the entire architecture. Jupyter notebooks are responsible for communication with the kafka producer and consumer. Further, Spark is used in consumer notebook. This notebook is constructed using the base Spark image on which boto3, jupyterlab, pyspark, pymongo, confluent_kafka python libraries are installed (refer dockerfile in root folder).

The container is connected to AWS via the Boto3 library in producer_demo notebook. The records are then fetched lazily by iterating over an S3 object. These records are pushed into the Kafka producer. Once the producer starts pushing the records, the

Queuing: Kafka
The kafka container is connected to the orchestrator container via the confluent kafka library in python. Further, it is connected to zookeeper which is a dependency for the Kafka container. It is also connected to the spark master container via the port of 9092. For the sake of the demo the records were pushed in every 1-10 seconds.

Processing: Spark
The spark master container along with the worker container are responsible for the processing aspect. The master is connected to the worker and the orchestrator via pyspark. The orchestrator is responsible for submitting jobs to the spark master which makes the worker process the job.

Spark is configured to accept 10 records for a batch and the trigger time for the query is set as 10 seconds.

Data Storage: MongoDB
MongoDB is mounted on the port of 27017. It acts as a sink for the spark container. The spark container processes data and posts it to mongodb.

Dashboard: MongoCharts
Mongocharts runs by communicating with the mongodb container. Data source can be specified by using the right URI for Mongodb. The dashboard is accessible at 8081 because the default 8080 is occupied by the spark container. No further configuration is required.

Execution guidelines - How to execute the pipeline

Docker-compose.yml is run using the command:

```
Docker-compose up -d
```

This brings up all the containers.

Log on to the localhost:8888 using admin@123 and run the producer_demo notebook.
In another tab run the consumer_demo notebook

This will start populating the mongoDB database.

Log on to localhost:8081 to see the dashboard on mongo charts using the credentials

admin@example.com
StrongPassw0rd

Results

One of the major bottleneck areas in big data related projects is the memory running out. In this project, care has been taken to not load the data in memory or download the entire dataset in the local memory. Few examples are as follows:

1. Csvreader for reading files from the S3 bucket.
2. Data generator using yield operation
3. Spark batch processing.

Given these ideas, it is certain that this application can be scaled up to process an even larger dataset.

Conclusion

It was a great experience working on this project. One of the major takeaways from the project is that the learning curve for these technologies is not linear. But once the technology and architecture is understood properly, it can get much easier.

A real-time dashboard was created using the technologies mentioned before. It can safely be said that this project has made us ready to work with these technologies. The entire project is containerised except for the S3 bucket which allows this application to be deployed on any machine. The streaming application works well and the dashboard updates on a real-time basis. The data flow is smooth because of the usage of Spark based Mongo connector which writes a Spark SQL dataframe into MongoDB. The dashboard can be manipulated easily with drag and drop features provided on Mongo charts. Last but not the least, one of the most important aspects of the application is its lazy evaluation of the entire data pipeline. Nowhere is the data being downloaded to the local system or more than a batch being taken into memory. This makes it very powerful to look at large size datasets.

That being said, there are a few issues with the current architecture. Spark is an extremely powerful technology for analytics. Currently, Spark is only creating a subset of the data that is being fed into it. It would have been much more interesting to see how aggregations could be done and then posted to MongoDB to create a monitoring dashboard which would report data from the past time interval.

References

1. https://www.kaggle.com/datasets/elemento/nyc-yellow-taxi-trip-data
2. https://github.com/kingspp/cs585_ds503_ss_assignment
3. https://github.com/CrusaderX/mongodb-charts
4. https://spark.apache.org/docs/latest/streaming-programming-guide.html
5. https://kafka.apache.org/documentation/