CSE 511: Data Processing at Scale

Amey Bhilegaonkar
Arizona State University
abhilega@asu.edu

Introduction

This project involves creating a highly scalable and available data processing pipeline using Kubernetes and Kafka, integrated with Docker and Neo4j. The pipeline will ingest streaming data produced by Kafka stream and perform processing operations on it and then distribute it to Neo4j for near real-time processing and analytics. This project involved stepwise execution wherein the first step involved setting up the Kafka and Apache Zookeeper together using the orchestrator minikube, which is a lightweight Kubernetes implementation that runs locally on your machine. Kafka will be used to ingest data from the parquet file and distribute it to other components of the pipeline. In the second step, integration of Neo4j into the setup is performed, and the data is streamed into Neo4j for further analysis and processing. Once the data is ingested into Neo4j, the analysis and algorithms such as PageRank and BFS were applied to graph-based data to explore the data.

Methodology

The aim of this section is to propose a methodology for implementing Kafka and Neo4j as a service using Helm and Minikube in Kubernetes. The proposed methodology consists of the following steps:

1.1. Environment Setup

The first step in installing Kafka and Neo4j as a service is to create the proper environment. The installation of Minikube, Kubernetes, and Helm is the first step towards putting the Kafka and Neo4j services into practice. A local Kubernetes cluster called Minikube offers the essential conditions for setting up and running the services. Helm, on the other hand, is a Kubernetes package manager that streamlines the cluster's application administration and installation processes.

1.2 Kafka and Zookeeper Installation: This can be achieved by creating a Kafka cluster and Zookeeper on Kubernetes. The two YAML files kafka-setup.yaml and zookeeper-setup.yaml contain the necessary configurations required for a basic Kafka cluster to get up and running.

1.3 Neo4j Installation: This step involves storing the data in Neo4j. For this purpose, an official Neo4j Helm chart can be utilized to create a Neo4j cluster. The neo4j-values.yaml and neo4j-service.yaml will help us get the Neo4j service up and running to serve as a backend database.

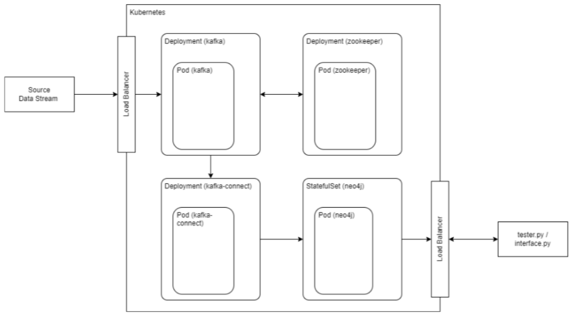
1.4 Data Ingestion: After the Kafka cluster is up and running, the next step is to ingest data into the Kafka cluster. For this step, we will run a data_producer.py file which will load the graph data from the parquet file to Kafka and then to Neo4j.

1.5 Testing and Deployment: In the end, thorough testing is required to guarantee the implementation's appropriate functionality. Simple tests like sending a message via Kafka-stream and receiving it on the other end should be included in the testing process. Also, when the data is loaded with data_producer.py into a graph database, we can run tester.py along with interface.py from phase 1 of this project to see if BFS and PageRank algorithms are working fine. When testing is successfully finished, the implementation can be reliably put into use.

1.6 Testing on limited resources: For testing purposes, since I had limited Memory and Cores for my personal laptop, it was really hard for me to run the original parquet file, it took almost a day for me just to run that file and still it was running the next day. So I decided to divide the whole file into chunks of 50 records and then test my code with 3 chunked files of 150 records. I modified the tester.py file, added my own test cases, and tested them. The part-parquet files and tester.py file is attached with the code. Also, screenshots from the results are attached to this report and the rest screenshots are pushed on GitHub. Also, the Python file with which I created chunk files was also pushed to GitHub.

With the steps followed as described above, and with automation test cases created on my own, I learned new things and explored various resources to debug the issue which gave me exposure to various other extensions that are used to manage minikube and help it automate it.

Results



Architecture of services

After the successful deployment of pods of Zookeeper, Kafka, Kafka-connector, and Neo4j we should see the following results with the commands to test our flow from point A to point B.

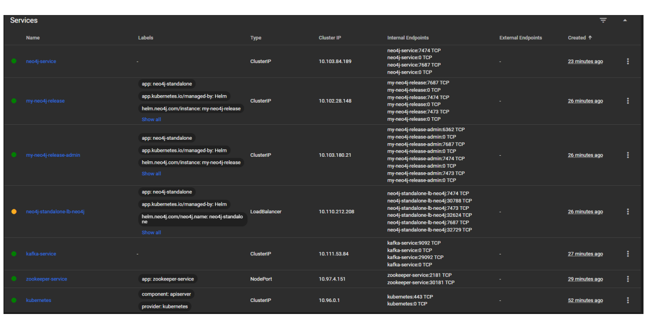
The following Commands were run to check if our deployments are correct:

1. kubectl apply -f ./zookeeper-setup.yaml
2. kubectl apply -f ./kafka-setup.yaml
3. helm install my-neo4j-release neo4j/neo4j -f neo4j-values.yaml
4. kubectl apply -f neo4j-service.yaml
5. kubectl apply -f Kafka-neo4j-connector.yaml
6. kubectl port-forward svc/neo4j-service 7474:7474 7687:7687
7. kubectl port-forward svc/Kafka-service 9092:9092

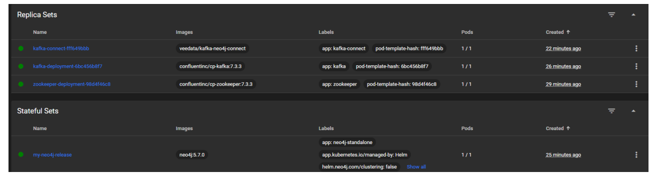
Testing Commands:

1. kubectl get pods
2. kubectl get services
3. python3 data_producer.py
4. python3 tester.py

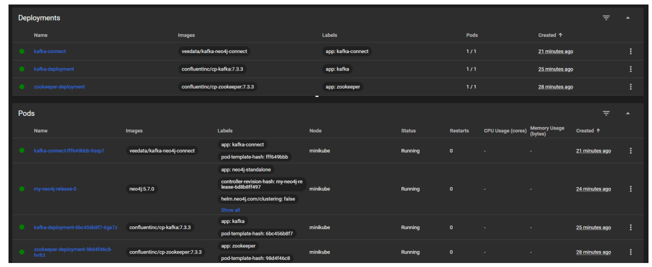
Kubernetes Deployments



Kubernetes Services



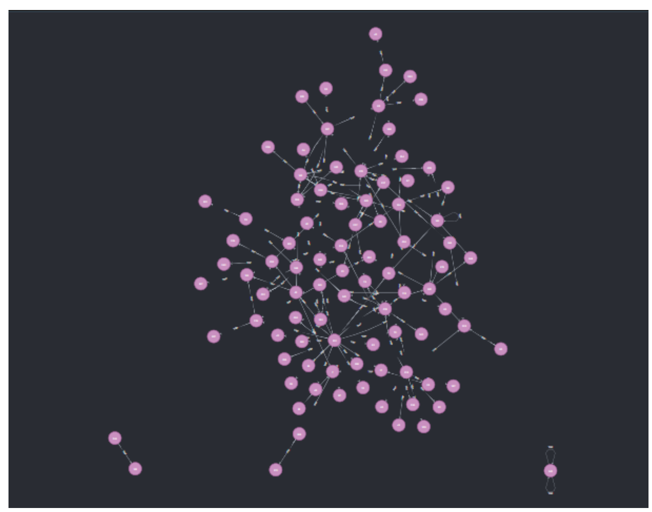
Kubernetes Replica Status



Kubernetes Pod and Deployment Status

The running pods show that our services are up and running. Now once we login into http://localhost:7474/browser/ we can see that our graph database Neo4j is up. We can run the following command to see if the database got updated with the right data or not:

1. match(n) return(n) - This will give us all the nodes present in the graph

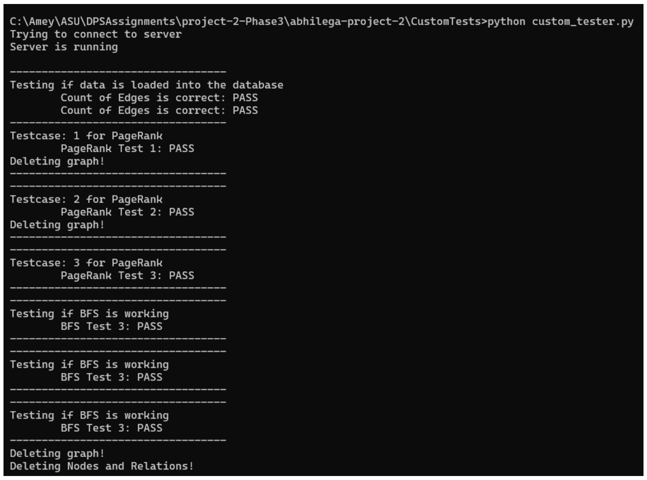


Graph Visualization of one of the Chunk files

Since I had limited hardware resources, I had to divide the big parquet file into small chunks and then test it. I ran the following commands to run custom_data_producer.py to load the data from the parquet file into the database and then ran custom_tester.py to verify the results.

1. cd CustomTests\
2. python custom_data_producer.py
3. python custom_tester.py

The below screenshot shows the output of the above command.



```
C:\Amey\ASU\DPSAssignments\project-2-Phase3\abhilega-project-2\CustomTests>python custom_tester.py
Trying to connect to server
Server is running

----------------------------------
Testing if data is loaded into the database
        Count of Edges is correct: PASS
        Count of Edges is correct: PASS
----------------------------------
Testcase: 1 for PageRank
        PageRank Test 1: PASS
Deleting graph!
----------------------------------
----------------------------------
Testcase: 2 for PageRank
        PageRank Test 2: PASS
Deleting graph!
----------------------------------
----------------------------------
Testcase: 3 for PageRank
        PageRank Test 3: PASS
----------------------------------
----------------------------------
Testing if BFS is working
        BFS Test 3: PASS
----------------------------------
----------------------------------
Testing if BFS is working
        BFS Test 3: PASS
----------------------------------
----------------------------------
Testing if BFS is working
        BFS Test 3: PASS
----------------------------------
Deleting graph!
Deleting Nodes and Relations!
```

Output of the Test Cases

This command will tell us if test cases for algorithms BFS and PageRank ran fine. Everything was successful.

For this particular project, I created small subsets of the original parquet file and tested it out with that file creating my own test cases, the corresponding tester.py file is submitted along with the other codes here.

Discussion

This project involved a lot of learning for me and the curve was definitely exponential. I got to learn about Kubernetes, and how the applications are scaled and managed by the orchestration tools such as Kubernetes. Kubernetes plays an important role in scaling applications, and it is crucial for managing containers in a production environment. Through this project, I got to learn about the different Kubernetes objects such as pods, deployments, services, and how they all work together to ensure that the application is running smoothly.

I also got to learn how streaming applications work together with services like Kafka, which is used to build real-time data pipelines and streaming applications. In simple terms, it is a tool that allows you to send and receive large volumes of data between different systems or applications in real time. Kafka is a distributed streaming platform that is designed to handle high volume and high throughput of data in real-time, making it a crucial tool in many industries, including finance, healthcare, and retail.

I liked one thing about this project: everything was pointed in a direction in which we could step and then gather information on our own, and apply that knowledge here to build scalable containerized applications, nothing was spoon-fed. Because of this, I got to explore technologies that I was not aware of, starting from Docker, Neo4j, Kafka, Kubernetes, Minikube, etc. This project helped me to develop my problem-solving skills and to be more comfortable with exploring new technologies on my own.

I think, for me, future work on this project will be significantly important, to take this project to AWS, a concept introduced in the class, and I would really like to explore that option as well. I will get to learn and explore AWS services, and how everything works in integration with one another in a cloud-based environment.

In conclusion, this project gave me insights into different technologies and services used for streaming jobs in real life, with a high learning curve. It helped me to develop a better understanding of how the different components of an application work together in a production environment. I also learned about the importance of message/streaming services such as Kafka, and how they can be used to build real-time data pipelines and streaming applications. Overall, this project was an excellent learning experience for me, and I am excited to continue exploring new technologies and services in the future.

Conclusion

Through this project, I gained a deeper understanding of how data flows in real-time from source systems to databases, and the crucial role that message/streaming services play in facilitating this flow. I learned about Kafka and SQS, both of which enable the reliable and scalable transmission of data between applications and services. This knowledge will be invaluable in any future work I do with data-intensive applications.

Furthermore, I was able to delve into Neo4j, a graph database management system, and understand how it differs from traditional relational databases. I explored its unique features and capabilities, and gained a greater appreciation for its usefulness in scenarios where relationships between data points are crucial. Overall, this project has given me a practical understanding of how complex applications are deployed to servers, and how containerized deployments enable multiple services to communicate and work together seamlessly.

References

1. How to do WSL config to Manage the Docker Memory and CPU cores: Microsoft Official Documentation
2. Official Kubernetes Documentation
3. Official Kubernetes Handbook
4. YouTube playlist DevOps BootCamp to understand the depth of Docker and Kubernetes

5. Neo4J Manual for Kubernetes
6. Neo4J documentation for Kafka
7. Canvas Discussion section for this project