A Data Processing Pipeline with Kafka, Neo4j, and Kubernetes for Distributed Graph Algorithm Processing

Eric Waters
Arizona State University

1 Introduction

Per the trend of exponentially increasing volumes of data available, it is becoming increasingly important to implement ways to efficiently store, manage, and process said data. Furthermore, streaming data is becoming more common as the world continues to collect data at regular intervals through means such as IoT devices or smartphones. For this case study, the task at hand is to create a data processing pipeline that will support a taxi business. The data will come from a large geospatial dataset of taxi trips from the NYC Yellow Cap Trip dataset [9]. The taxi trip data will be streamed into the platform. At a later time, PageRank [13] and Breadth-First Search [14] algorithms will be run on the data.

The PageRank algorithm was developed by Google in order to assess the importance of various pages on the web. A web page is more important if many other important pages contain links to that page. In the case of the taxi cab dataset, this will find pickup and drop-off locations that are crucial to the business.

The Breath-First Search algorithm is a simple graph-traversal algorithm that explores all nodes at the current level before proceeding to the next. For the taxi cab dataset, this algorithm will explore all other pickup and drop-off locations starting from a designated location.

The ultimate goal of this project is to allow the taxi trip data to be streamed into a distributed data pipeline such that the taxi trips are automatically inserted into a database. Then, the graph processing algorithms can be run in the distributed environment.

2 Methodology

2.1 Technology Used

The data will be stored in Neo4j, a popular native graph database that supports ACID-compliant transactions [4]. The dataset is mainly composed of locations and taxi trips to and from those locations. The locations are stored as nodes in Neo4j and the trips are stored as relationships between those nodes. Each trip has a distance, fare amount, pickup time, and dropoff time.

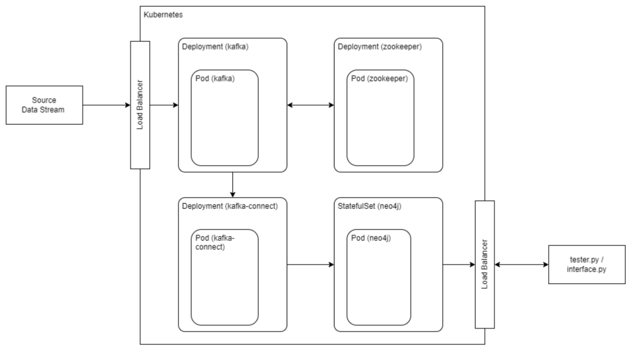


Figure 1: The Architecture of the Data Processing Pipeline

Apache Kafka is a distributed messaging queue system that has the ability to handle massive volumes of data streams [11]. It was originally developed by LinkedIn and later open-sourced and released to the Apache Software Foundation. In this context, Kafka will be used to feed the graph data from the Yellow Cab Trip dataset into other components of the system.

Kubernetes is a tool for managing, running, and scaling containerized applications [6]. In the system to be developed in this paper, Kubernetes will be used to orchestrate the different processes involved in the pipeline. The smallest execution unit in Kubernetes, a Pod, is a group of one or several containerized applications. There will be Pods for Kafka, Zookeeper (the master coordinator for Kafka), Kafka-Connect (a tool for connecting Kafka with other systems), and Neo4j. Kubernetes ensures that these Pods remain healthy and has the ability to dynamically adjust the amount of replicas in order to react to increased traffic or even heal Pods if they stop functioning properly. The architecture for the system can be seen in Figure 1.

Lastly, since Kubernetes typically runs on multiple systems and requires a significant amount of resources, Minikube will be used. Minikube is a tool to run a Kubernetes environment on a single machine [12]. Although the use of Minikube defeats the purpose of Kubernetes, which is to distribute and manage containers across multiple machines, it is highly useful for development. Developers can test their deployments using Minikube before it is deployed into production. And, in this case, students can learn to deploy their distributed systems without requiring access to a great deal of computational resources.

2.2 Deploying Kafka and Zookeeper

In order to deploy Kafka into a Kubernetes cluster, one must understand the concept of Deployments and Services within Kubernetes. Deployments are the means through which the developer specifies the desired state of the Kubernetes Pod. In this case, two deployments were specified: one for Kafka and another for Zookeeper [10]. The reason for grouping together Kafka and Zookeeper is that Zookeeper is a coordination service that enables Kafka to be distributed. It keeps track of the Kafka broker nodes, elects certain brokers to be leaders, stores metadata about the partitioning, and more. Confluent, a company that provides tools for real-time data processing [1], has Docker images (easy-to-use pre-packaged environments) for both Kafka and Zookeeper. These docker images do all the work for specifying the environments for the deployments. Services, on the other hand, are a means of accessing the deployed Pods. As the Pods can be removed, added, or updated, their IP addresses often change. Services handle the work of identifying the current addresses of the Pods and make communicating with the Pods simple. A Kafka service is specified that speaks with the Kafka Deployment Pod over the TCP protocol. Similarly, a Zookeeper Service is delegated to handle communication with the ZooKeeper Deployment Pod. These deployments and services are declared using yaml configuration files.

2.3 Deploying Neo4j

As mentioned before, the graph database being used to store the taxi trip data will be Neo4j. For this step, Helm [2], a package manager for Kubernetes, will be used to install the Neo4j database into a containerized environment so that it can then be easily deployed as a Pod. Additionally, the Neo4j Graph Data Science (GDS) [5] plugin will be installed. The GDS plugin provides pre-packaged algorithms for graphs. In this case, GDS will be used for the PageRank and Breadth-First Search algorithms. Helm deploys Neo4j as a StatefulSet, a special type of Kubernetes resource that supports persistent storage and more stable network identities. Then, a Neo4j service is specified that enables communication with the Neo4j database.

2.4 Connecting Kafka and Neo4j

Kafka Connect [3] is a tool for integrating Kafka with other systems. In this case, Kafka Connect will be used to convert messages within Kafka topics into Neo4j insert statements.

An example of a message sent to the Kafka topic is shown in Listing 1.

Listing 1: An Example Kafka Message

```
{"trip_distance":1.84, "PULocationID":78,
"DOLocationID":242, "fare_amount":10.66}
```

In the Kafka Connector, a Cypher (the query language for Neo4j) statement is defined as a template. When the Kafka messages are consumed, the various parameters are placed into this query template so that the data is properly inserted into the database. The template used is shown in Listing 2.

Listing 2: The Neo4j Cypher Insert Statement

```
MERGE
(p:Location {name: toInteger(event.PULocationID)})
MERGE
(d:Location {name: toInteger(event.DOLocationID)})
MERGE
(p)-
[:TRIP {distance: toFloat(event.trip_distance),
fare: toFloat(event.fare_amount)}]
->(d)
```

With all of these pieces configured, the trip data can be streamed into the Kafka topics which are then automatically inserted into the database.

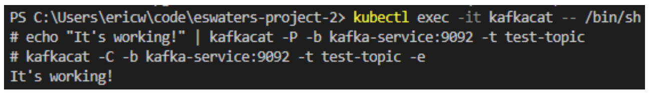
3 Results

3.1 Testing Kafka

In order to test if the Kafka Pods were working properly, Kafkacat was used. Kafkacat [8] is a tool for debugging and testing Kafka deployments. The steps to test the deployment were as follows:

1. Create and deploy a Kafkacat Pod into the Kubernetes cluster
2. Connect to the Kafka Test Pod
3. Produce a test message
4. Consume the test message

The output of the test can be seen in Figure 2. The message *It's working!* was produced into the Kafka service under the topic *test-topic*. Then, upon subscribing to the *test-topic*, the message can be seen.



```
PS C:\Users\ericw\code\eswaters-project-2> kubectl exec -it kafkacat -- /bin/sh
# echo "It's working!" | kafkacat -P -b kafka-service:9092 -t test-topic
# kafkacat -C -b kafka-service:9092 -t test-topic -e
It's working!
```

Figure 2: A Message Being Produced and Consumed Using Kafkacat

3.2 Testing Neo4j

Testing to see if the Neo4j Deployment was running properly was simple. Neo4j provides an interface called Neo4j Browser that can be opened locally. It provides an interface to query the data and visualize the results. Ensuring that Neo4j was working properly was as simple as opening Neo4j Browser at *localhost:7474/browser* (7474 was specified as the Neo4j-http port in the Kubernetes setup) and ensuring that the GDS plugin is properly installed with a simple GDS query as shown in Listing 3.

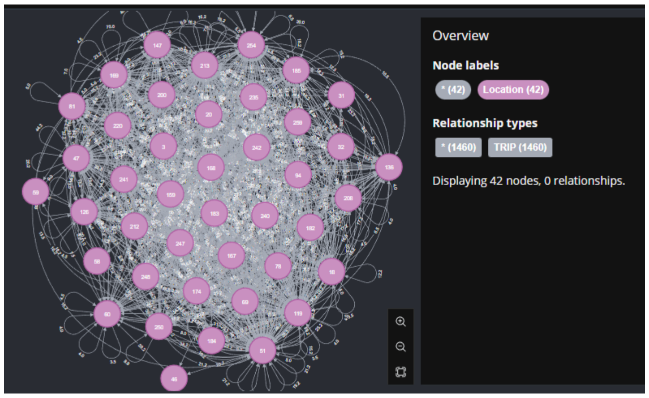
Listing 3: A Neo4j GDS Test Query

```
CALL gds.graph.exists('gds_graph')
YIELD exists RETURN exists
```

This query is convenient to test the database because if GDS is not properly installed, it will throw an error saying that there is no method *gds.graph.exists*.

3.3 Testing the Entire Pipeline

In order to test the whole pipeline, the following process was followed. First, the taxi data was loaded into a Python script. In order to make the size of the dataset manageable, the data was filtered to only include trips in the Bronx. Then, each row of the data set was converted into a JSON object and sent to Kafka. Next, the Neo4j database was queried with Cypher to show that the data had been successfully inserted into the data (see Figure 3). Lastly, the PageRank and Breadth-First Search algorithms were run and verified to have the correct values.



Displaying 42 nodes, 0 relationships.

Figure 3: The Taxi Data Successfully Loaded into Neo4j

4 Discussion

The results of the project indicate that the use of Kubernetes, Kafka, and Neo4j is a powerful combination of tools to create a streaming data processing pipeline. The pipeline is able to keep the database current in real-time and allows for distributed processing of the PageRank and Breath-First Search algorithms. The use of these distributed components will allow the system to handle moments of higher-than-usual demand during busy times such as holidays.

Future avenues for this project should include the real-time processing of the graph algorithms used. In this project, the only events happening in real time are the insertion of the taxi data into the database. The graph algorithms are processed manually at the developer's time of choosing. Doing this processing in real-time could provide more immediate and useful results.

In addition, future work could include implementing the functionality for different types of data, such as document-based data or time-series data. It could also involve different algorithms or streaming data platforms such as Apache Flink [7] to carry out the computation. Furthermore, instead of using a private cloud approach, the processing could be offloaded to the public cloud in order to reduce the computational resources required.

5 Conclusion

This work explored an approach to providing real-time data support for a taxi cab company. Apache Kafka was used as the messaging queue service to hold the data and enable greater flexibility in times of high demand. Neo4j was used as the database to store the graph data and provide out-of-the-box PageRank and Breadth-First Search functionality through its popular Graph Data Science plugin. Kafka Connect was used to translate the Kafka messages into Cypher queries for the Neo4j database to insert the data in real time. And finally, Kubernetes was used to orchestrate these various containers, providing the ability to scale up and down according to demand. Although this project was developed in Minibase for demonstrative purposes, the system serves as a proof of concept for a solution to be implemented in practice. Demand for streaming data processing pipelines is becoming commonplace in the industry and the search for optimal solutions to similar problems is crucial for meeting business needs.

References

[1] Confluent. https://www.confluent.io/. Accessed: 2023-04-24.

[2] Helm. https://helm.sh/. Accessed: 2023-04-24.

[3] Kafka connect. https://docs.confluent.io/platform/current/connect/index.html. Accessed: 2023-04-24.

[4] Neo4j graph data platform | graph database management system. https://neo4j.com/. Accessed: 2023-04-24.

[5] The neo4j graph data science library manual v2.3 - neo4j graph data science. https://neo4j.com/docs/graph-data-science/current/.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[8] Edenhill. Edenhill/kcat: Generic command line non-jvm apache kafka producer and consumer. https://github.com/edenhill/kcat.

[9] Elemento. Nyc yellow taxi trip data. https://www.kaggle.com/datasets/elemento/nyc-yellow-taxi-trip-data.

[10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[11] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.

[12] Ruchika Muddinagiri, Shubham Ambavane, and Simran Bayas. Self-hosted kubernetes: Deploying docker containers locally with minikube. In *2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, pages 239–243, 2019.

[13] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking : Bringing order to the web. In *The Web Conference*, 1999.

[14] Konrad Zuse. Der plankalkül. In *Konrad Zuse Internet Archive*, pages 96–105, 1972.