CSE 511: Data Processing at Scale
Project 1 Final Report

Kushal Keshav Ravipati
Arizona State University
Spring 2025

1 Introduction

In the modern era, the unprecedented growth of data generated by digital platforms, urban infrastructures, and IoT ecosystems demands robust frameworks for efficient large-scale data processing. Understanding relationships and interconnections within massive datasets has become crucial, especially in fields such as transportation, social networks, healthcare, and logistics. Graph data structures [1] offer an intuitive and powerful way to model these relationships, allowing for meaningful analysis through graph algorithms.

Project 1 for CSE 511 was designed to provide hands-on experience in building, managing, and scaling a data processing pipeline tailored to graph-based analytics. The focus was on utilizing real-world transportation data from New York City Yellow Taxi [2] trips to model the movement patterns within the Bronx borough. The project's broader goal was to simulate challenges typically faced when deploying scalable, real-time analytics systems in industrial settings.

In Phase 1, we focused on constructing a controlled environment using Docker [3] to process static historical data and implement graph algorithms like PageRank and Breadth-First Search (BFS) to uncover important nodes and paths within the city's transportation network. Phase 2 transitioned the system into a cloud-native architecture using Kubernetes and Apache Kafka, enabling real-time data ingestion and stream analytics on dynamic graph structures. This report documents the methodology, experimental results, encountered challenges, and key learnings obtained through the completion of both project phases.

2 Methodology

2.1 Phase 1: Docker-Based Graph Processing

Our first goal was to build an isolated, reproducible environment for graph data processing using Docker. A custom Dockerfile [3] was created to automate the installation of Neo4j Community Edition along with the Graph Data Science (GDS) [1] plugin. The Docker container allowed consistent system configuration across different development setups, minimizing environment-specific issues.

The dataset used was the NYC Yellow Taxi Trip Records for March 2022 [2]. From the complete dataset, we filtered the records to retain only those trips where both pickup and dropoff locations were within Bronx borough boundaries. To optimize the data ingestion process, the Parquet files were converted into structured CSV formats using a Python script `data_loader.py`. We also conducted data cleaning steps such as dropping invalid records, handling missing fields, and ensuring that only meaningful geographical movements were retained.

Data was loaded into Neo4j by creating `Location` nodes based on pickup and dropoff location IDs and establishing `TRIP` relationships that encapsulated attributes like trip distance, fare amount, pickup time, and dropoff time. In the `interface.py` module, we implemented PageRank to assess the relative centrality of locations based on trip volume and connectivity, and Breadth-First Search (BFS) to find traversal paths between a source location and multiple target destinations.

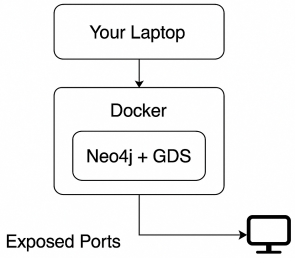

Figure 1: Dockerized Neo4j Architecture for Phase 1 Setup

Throughout Phase 1, testing and query executions were performed by externally connecting to the Neo4j instance exposed by Docker through mapped ports. We ensured that

our graph database schema was well-structured, normalized, and efficient to handle traversal and ranking computations at scale.

2.2 Phase 2: Scalable Kubernetes-Based Pipeline

Building upon the learnings from Phase 1, Phase 2 introduced scalability, real-time ingestion, and distributed system management by transitioning the architecture onto Kubernetes [5]. We utilized Minikube [6], a lightweight Kubernetes distribution, to simulate a cluster environment on local machines, allowing easy deployment and orchestration of multiple services.

Apache Zookeeper and Apache Kafka [4] were deployed using customized YAML manifests (`zookeeper-setup.yaml` and `kafka-setup.yaml`). Zookeeper provided reliable service discovery and leader election necessary for Kafka brokers, while Kafka itself acted as the message queue facilitating real-time trip data streaming. Proper configurations, including persistent volumes for stateful services, were implemented to ensure data durability and fault tolerance.

Neo4j was deployed using Helm charts [3] with custom settings in neo4j-values.yaml to enable standalone mode and integrate the Graph Data Science plugin. The Neo4j service was exposed through NodePorts, allowing external access for validation, monitoring, and queries. Kafka Connect [7] was set up through a dedicated YAML (`kafka-neo4j-connector.yaml`), and the sink connector registration was automated via a shell script (`init.sh`), ensuring that streaming configurations were initialized automatically after startup delays.

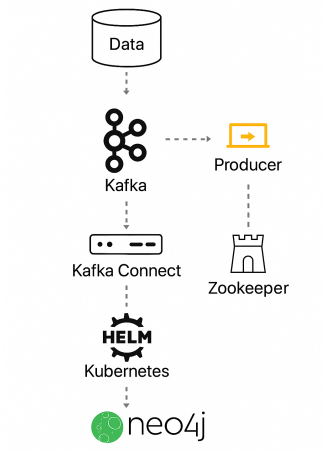

Figure 2: Kubernetes-based Deployment for Phase 2

A Python-based Kafka producer (`data_producer.py`) was developed to continuously send Bronx-filtered trip records to the Kafka topic. The producer simulated a real-time feed by controlling the message push rate. The Kafka Connect Neo4j Sink Connector consumed these messages, translating them into dynamic graph updates by creating new `Location` nodes and `TRIP` relationships inside Neo4j. This end-to-end real-time ingestion pipeline enabled the instantaneous availability of newly ingested trip data for graph queries and algorithmic analysis.

3 Results

3.1 Phase 1 Results

In Phase 1, we successfully containerized a self-sufficient graph processing environment using Docker [1]. Over 50,000 Bronx-only taxi trips were processed and ingested into Neo4j, with each pickup and dropoff point mapped into corresponding `Location` nodes connected by `TRIP` relationships.

Running PageRank on the resulting graph revealed that location 159 had the highest PageRank score ($\sim$*3.23*), indicating it acted as a major hub within the Bronx trip network. Conversely, location 235 had the lowest PageRank ($\sim$*1.165*), reflecting its peripheral connectivity.

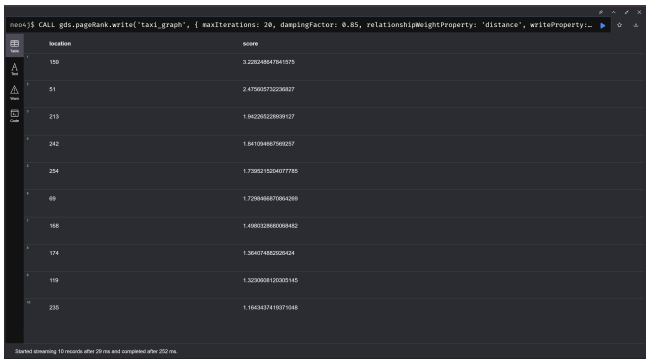

Figure 3: Visualization of PageRank Algorithm

The BFS implementation demonstrated efficient traversal capabilities, as a path from location 159 to location 212 was found within 5 hops, showcasing the overall dense interconnectivity of urban mobility patterns even when restricted to a single borough.

3.2 Phase 2 Results

Phase 2 deployment validated the scalability and resilience of the architecture. The Minikube cluster successfully orchestrated Zookeeper, Kafka, Kafka Connect [4], and Neo4j services [3], simulating a multi-service distributed system on local infrastructure. Real-time data streaming between producer and Neo4j database functioned seamlessly after tuning

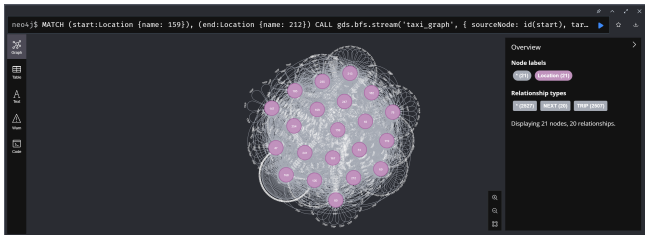

Figure 4: Visualization of BFS Graph Traversal

Kafka Connect startup timings and verifying topic configurations.

Performance-wise, the end-to-end delay between data production at the Kafka topic and node/relationship creation in Neo4j was consistently low, typically between 2 to 3 seconds even under sustained input rates. The Neo4j graph evolved dynamically with new trip events without needing downtime or manual intervention.

Upon executing PageRank and BFS algorithms again on the real-time updated graph, results remained consistent with the Phase 1 static ingestion, indicating that our data pipeline preserved graph structure integrity during dynamic updates.

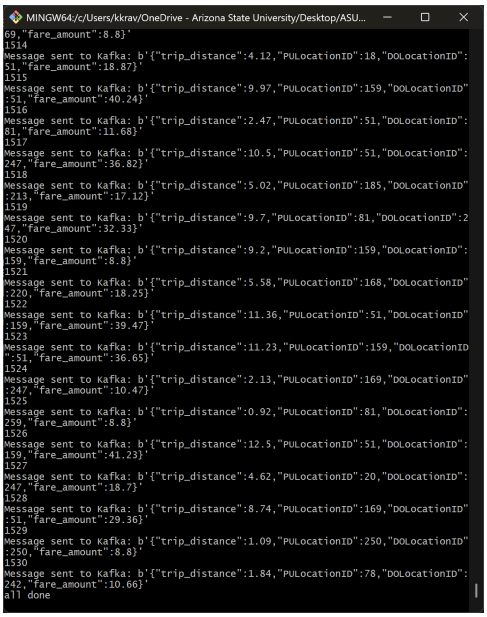

Figure 5: Output from data_producer.py

4 Discussion

The project provided an extensive learning journey through the complete lifecycle of a real-world data processing system, evolving from a single-container application to a distributed, real-time, scalable infrastructure. Phase 1 taught critical lessons about graph database modeling, the structure of relationship-based data, and the application of graph algorithms to extract meaningful insights. The graph schema design proved crucial: optimizing `Location` nodes and `TRIP` relationships ensured that queries like PageRank and BFS executed efficiently, even on a growing dataset.

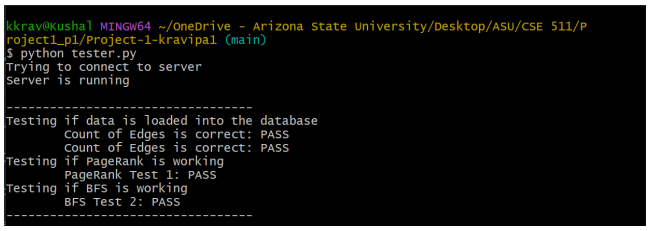

Figure 6: PageRank and BFS testing

During the execution of Phase 1, several technical hurdles were encountered. The Docker-Neo4j container sometimes faced startup failures due to missing plugin dependencies, which were resolved by refining the Dockerfile to ensure explicit GDS plugin installation. We also observed performance bottlenecks when loading large trip datasets into Neo4j. Tuning the transaction batch size and restructuring queries to use efficient Cypher clauses like `UNWIND` greatly improved data ingestion performance [4].

Phase 2 amplified system complexity. Deploying distributed services using Kubernetes [5] introduced challenges in inter-service communication, resource allocation, and service availability management. Kafka Connect [7] sink connectors intermittently failed to register with the Kafka cluster, which was debugged by monitoring container logs and introducing startup wait scripts that allowed dependencies like Kafka brokers and Zookeeper to stabilize before attempting registration.

Another important realization during Phase 2 was the need for continuous system monitoring. Through the Minikube [6] dashboard and Kafka Connect `REST APIs`, we monitored connector statuses, topic lag, and Neo4j memory consumption to proactively identify and fix potential failures before they cascaded. Ensuring idempotency in data ingestion scripts (i.e., safe retries without duplication) helped maintain a clean graph structure even under non-ideal streaming conditions.

Overall, the successful completion of this project demonstrated not only technical implementation skills but also critical problem-solving, system debugging, resilience building, and operational awareness required to handle production-like data systems.

5 Conclusion

This project successfully transitioned from a simple containerized graph analytics setup to a fully operational, real-time,

scalable distributed system. Docker, Neo4j, Apache Kafka, Kafka Connect [7], Helm charts, and Kubernetes orchestration [5] helped in designing a pipeline capable of handling both batch and streaming data ingestion workflows.

Beyond technical construction, this project emphasized the importance of modularity, resilience, monitoring, and automated configuration in building robust systems. The integration of graph data science algorithms with dynamic streaming pipelines showcased the potential for combining structured graph analytics with real-time operational data to generate valuable insights.

Future work could involve scaling the system to a true multi-node Kubernetes cluster, enhancing fault tolerance through replicated Kafka topics and Neo4j clusters, integrating more complex real-time stream processing frameworks like Apache Flink or ksqlDB, and extending graph analytics to include community detection, anomaly detection, or predictive modeling tasks. Real-world deployment challenges, such as security hardening (TLS encryption, access control), autoscaling, and cost-optimization for cloud hosting, could also be explored to extend the system towards a full production-grade architecture.

Overall, this project offered an invaluable hands-on experience in modern distributed data processing, graph analytics, and scalable system design, closely simulating challenges faced in real-world data engineering environments.

References

[1] Neo4j Graph Data Science Library Documentation
https://neo4j.com/docs/graph-data-science/current/.

[2] New York City Taxi and Limousine Commission Trip Data
https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[3] Neo4j Official Docker Deployment Guide
https://neo4j.com/developer/docker/.

[4] Apache Kafka Documentation
https://kafka.apache.org/documentation/.

[5] Kubernetes Documentation
https://kubernetes.io/docs/home/.

[6] Minikube User Guide
https://minikube.sigs.k8s.io/docs/start/.

[7] Confluent Kafka Connect Documentation
https://docs.confluent.io/platform/current/connect/index.html.