Scalable Graph Data Processing and Real-Time Analytics with Neo4j, Docker, Kubernetes, and Kafka

Narottaman Gangadaran
MS CS Ira A. Fulton Schools of Engineering ASU

Abstract

This project focuses on building a scalable graph data processing system in two stages. Initially, a Docker-based Neo4j environment was set up to load and analyze the NYC Yellow Cab Trip dataset using PageRank and Breadth-First Search algorithms. In the second stage, the system was extended using Kubernetes and Kafka to support real-time data ingestion and distributed graph analytics. The complete pipeline demonstrates a transition from standalone graph processing to a highly available, scalable architecture for near real-time analysis.

1 Introduction

In today's data-driven world, graph-based data models have become essential for understanding complex relationships across domains such as transportation, social networks, and bioinformatics. This project focuses on developing a scalable and highly available system for graph data processing and real-time analytics.

In the first stage, a Neo4j graph database [1] was deployed within a Docker environment to process and analyze the NYC Yellow Cab Trip dataset. Key graph algorithms such as PageRank and Breadth-First Search were implemented to extract meaningful insights from the data.

Building upon this foundation, the system was extended in the second stage to enhance scalability, availability, and real-time performance. Using Kubernetes for orchestration and Kafka [2] for streaming, a distributed data pipeline was created to ingest document streams and integrate them dynamically into a Neo4j database.

The completed system demonstrates the evolution from a standalone graph-processing setup to a distributed, cloud-native architecture capable of handling real-time data ingestion and analysis.

2 Methodology

The project was executed in two phases, each targeting different aspects of graph data processing, scalability, and real-time analytics.

2.1 Phase 1: Graph Processing with Docker and Neo4j

In Phase 1, the goal was to create a containerized graph processing environment using Docker [6] to deploy Neo4j, load trip data, and run basic graph algorithms.

- **Docker Image Building:** A custom Dockerfile was written to set up the environment, install Python dependencies (neo4j, pandas, pyarrow [4]), and prepare Neo4j for remote access.
- **The Docker image was built using the command:** docker build –no-cache -t neo4jdataproject.
- **Container Execution:**A Neo4j container was started with mapped ports to allow web browser access to the Neo4j UI: docker run -p 7474:7474 -p 7687:7687 –name neo4j-container neo4jdataproject.
- **Data Loading:** The NYC Yellow Cab Trip dataset [5] was loaded and filtered for trips within the Bronx area. The dataset was transformed from Parquet to CSV and saved inside the container's Neo4j import directory. A Python script (data_loader.py) connected to Neo4j and created: Location nodes (pickup and dropoff locations). TRIP relationships (containing trip distance, fare, pickup, and dropoff times).

```
MERGE (a:Location {name: $pickup})
MERGE (b:Location {name: $dropoff})
CREATE (a)-[:TRIP {distance: $distance,
fare: $fare, pickup_dt:
datetime($pickup_dt),
dropoff_dt: datetime($dropoff_dt)}]->(b)
```

- **Graph Algorithms Implementation:** In interface.py interface, two algorithms were implemented. **PageRank:** To rank the importance of locations based on trip flow. **Breadth-First Search (BFS):** To find shortest paths between locations.

- **Data Verification:** After container deployment, access to the Neo4j Browser (http://localhost:7474) allowed manual verification with Cypher queries:

```
MATCH (n) RETURN count(n);
MATCH ()-[r]->() RETURN count(r);
```

  Automated verification was performed by running python tester.py, checking if nodes, edges, PageRank scores, and BFS paths matched expected values.

2.2 Phase 2: Scalable Real-Time Pipeline with Kubernetes, Kafka, and Neo4j

Phase 2 focused on scaling the solution to enable real-time data ingestion and dynamic graph updates within a Kubernetes environment.

- **Cluster Setup:** A local Kubernetes cluster was set up using Minikube. Zookeeper and Kafka were deployed using zookeeper-setup.yaml zookeeper-setup and kafka-setup.yaml kafka-setup for managing and distributing data streams.

- **Neo4j Deployment on Kubernetes:** Neo4j was deployed inside Kubernetes using Helm with a custom configuration (neo4j-values.yaml) neo4j-values, enabling: Password setup (project1phase2). Installation of the Graph Data Science (GDS) library. neo4j-service.yaml neo4j-service exposed Neo4j ports for internal cluster access.

- **Kafka to Neo4j Data Streaming:** A Kafka Connect deployment (kafka-neo4j-connector.yaml) kafka-neo4j-connector was configured to automatically consume Kafka topic messages and ingest them into Neo4j in real-time.

- **Data Producer:** A Python script (data_producer.py) data_producer simulated real-world document streaming by: Loading and cleaning the trip dataset. Publishing trip records into Kafka (nyc_taxicab_data topic) message-by-message.

- **Real-Time Graph Analytics:** After streaming, PageRank and BFS algorithms were re-executed through interface.py to validate the dynamic graph.

- **Validation:** Streaming integrity was checked manually via Neo4j Browser and automatically using tester.py, confirming that the graph structure, PageRank scores, and BFS results were consistent.

3 Results

The developed system was validated by successfully ingesting and processing the NYC Yellow Cab Trip dataset [5] into a graph structure across both Phase 1 and Phase 2.

In both phases, the Neo4j database correctly represented the trip data as a network of Location nodes and TRIP relationships. Verification using Cypher queries confirmed the expected structure: Total number of nodes: 42 Total number of relationships: 1530

The PageRank algorithm identified location ID 159 as the node with the highest rank and location ID 59 as the node with the lowest rank, consistent with the expected connectivity of the dataset. Similarly, the Breadth-First Search (BFS) algorithm successfully discovered valid paths between designated pickup and dropoff locations. Graph visualizations from the Neo4j Browser further confirmed the correct structure and connectivity of the graph, showing a well-formed network of trip relationships between Bronx locations.

Across both a Docker-based environment (Phase 1) and a Kubernetes-based distributed setup (Phase 2), the system consistently produced accurate and reliable outputs, demonstrating the robustness of the designed pipeline for both static and real-time data processing scenarios.

4 Discussion

This project demonstrated the successful design and deployment of a scalable graph data processing system, evolving from a static containerized setup to a dynamic, distributed architecture.

In Phase 1, the use of Docker to build a standalone Neo4j environment allowed for controlled experimentation with graph data. Loading the NYC Yellow Cab Trip dataset into a graph structure provided hands-on experience with modeling real-world data as nodes and relationships. Implementing PageRank and Breadth-First Search (BFS) using the Neo4j Graph Data Science (GDS) library highlighted the power of graph algorithms in revealing important structures and connections within the dataset. Phase 1 also emphasized the importance of verifying data ingestion and ensuring correct environment configuration, which proved critical for seamless transition to more complex setups.

Phase 2 expanded the project into a distributed system using Kubernetes [3], Kafka, and Kafka Connect. Deploying services such as Zookeeper, Kafka brokers, and Neo4j within a Kubernetes cluster introduced challenges related to container orchestration, service communication, and dynamic

data handling. Integrating Kafka Connect to stream real-time trip data into Neo4j allowed the graph to grow continuously, simulating a live data environment. Despite the added complexity, the system maintained high availability and consistent analytics performance, as validated through re-execution of PageRank and BFS algorithms.

Across both phases, one of the key insights was the critical role of automation and containerization in managing complexity. Automating the data ingestion and processing pipelines made scaling straightforward, while careful configuration of services and connectors ensured system stability. Challenges included handling service startup dependencies, ensuring correct port mappings, and tuning Kubernetes resource allocations to accommodate heavier workloads.

Overall, the project successfully demonstrated core principles of modern data engineering, including containerized deployment, real-time data streaming, graph-based modeling, and scalable analytics. Future extensions could explore further enhancements such as deploying Neo4j in cluster mode for fault tolerance, optimizing Kafka for higher throughput, or integrating monitoring solutions to visualize system health and performance over time.

5 Conclusion

This project successfully demonstrated the construction of a scalable graph data processing and analytics system, progressing from a standalone Docker-based Neo4j deployment to a distributed, real-time streaming architecture orchestrated with Kubernetes and Kafka.

In Phase 1, a controlled environment was established to ingest and model static trip data into a Neo4j graph database, enabling effective execution of key graph algorithms such as PageRank and Breadth-First Search. This phase provided a solid foundation in graph data modeling, containerization, and algorithmic analysis.

In Phase 2, the system was expanded to support real-time data ingestion through a Kafka-based pipeline, seamlessly streaming trip data into a Neo4j instance deployed within Kubernetes. This phase introduced concepts of service orchestration, dynamic data processing, and distributed system management, culminating in a live graph environment capable of continuous analytics.

By successfully integrating diverse technologies such as Docker, Neo4j, Kafka, Kubernetes, and Kafka Connect, the project achieved its goals of scalability, availability, and real-time analytical capabilities. It reflects an end-to-end pipeline capable of handling both batch and streaming graph data, highlighting the practical application of modern data processing techniques in solving real-world challenges.

References

[1] Neo4j, *The Neo4j Graph Data Platform Documentation*, 2024. Available at: https://neo4j.com/docs/

[2] Confluent, *Apache Kafka and Kafka Connect Documentation*, 2024. Available at: https://docs.confluent.io/

[3] Kubernetes, *Kubernetes Documentation*, 2024. Available at: https://kubernetes.io/docs/

[4] Apache Software Foundation, *Apache Arrow - PyArrow Documentation*, 2024. Available at: https://arrow.apache.org/docs/python/

[5] New York City Taxi and Limousine Commission, *TLC Trip Record Data*, 2024. Available at: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[6] Docker Inc., *Docker Documentation*, 2024. Available at: https://docs.docker.com/