Scalable Architecture for Real-Time Graph-Based Analytics Using Neo4j, Kafka, docker, and Kubernetes

Simran Agrawal – sagar158@asu.edu
Arizona State University

Abstract

This project aimed to build a scalable, real-time graph analytics platform by integrating Neo4j, Apache Kafka, Docker, and Kubernetes. The work proceeded in two distinct phases. Phase 1 focused on constructing a batch-processing pipeline using a Dockerized Neo4j environment, where cleaned and filtered NYC Yellow Taxi trip data were modeled into a graph and analyzed using PageRank and Breadth-First Search (BFS) algorithms. In Phase 2, the system architecture transitioned into a dynamic real-time streaming pipeline by leveraging Kafka for continuous ingestion, Kafka Connect for automatic transformation, and Kubernetes for orchestration and scaling.

By methodically integrating these technologies, the resulting system achieved near real-time graph analytics with robust scalability and minimal latency. This project showcases how batch and streaming pipelines can be cohesively built for production-grade graph analytics, addressing the growing industry need for real-time decision-making systems.

1. Introduction

The rapid growth of interconnected data across industries demands systems capable of real-time analysis and actionable insight extraction. Graph databases like Neo4j have emerged as key technologies for managing complex, relational datasets, supporting use cases such as transportation networks, fraud detection, and recommendation engines.

This project set out to design a robust and scalable analytics pipeline capable of handling both static and dynamic data streams efficiently. Utilizing open-source technologies such as Neo4j, Docker, Kafka, and Kubernetes, the project aimed to simulate real-world conditions involving continuous data ingestion and live graph transformations.

The work unfolded in two distinct phases. Phase 1 established the foundational batch processing environment with Neo4j and Docker, enabling structured graph modeling and preliminary analysis. Phase 2 elevated the system into a real-time, streaming architecture orchestrated via Kubernetes, incorporating Kafka-based ingestion and dynamic analytics execution.

2. Methodology

2.1 Phase 1: Batch-Based Neo4j Graph Modeling

The project began by curating the March 2022 NYC Yellow Taxi dataset. Python's Pandas and PyArrow libraries were utilized to filter the dataset to retain only trips within the Bronx borough. Trips were further restricted to a minimum fare of $2.50 and a minimum distance of 0.1 miles to ensure data quality. Datetime fields were standardized to ISO 8601 format to maintain compatibility with Neo4j.

The cleaned dataset was then modeled into a graph structure where each unique pickup and drop-off location was represented as a :Location node, while each trip formed a :CONNECTS relationship enriched with attributes such as fare amount, trip distance, and timestamps.

To ensure environmental consistency, Neo4j was containerized using Docker. The custom Dockerfile included installations for Neo4j Community Edition and the Graph Data Science (GDS) plugin, along with necessary configurations for exposed ports and memory optimizations.

The **data_loader.py** script automated the ingestion process by mapping CSV entries into graph nodes and relationships. Analytical scripts in **interface.py** executed PageRank to identify central hubs and BFS to compute the shortest paths between any two locations, enabling effective transportation network analysis.

Validation through the **tester.py** script confirmed that the graph was correctly structured, containing precisely 42 nodes and 1,530 relationships. Analytical results aligned closely with expected real-world behavior, providing a solid baseline for advancing to real-time data handling.

2.2 Phase 2: Real-Time Streaming Architecture

Building upon the static pipeline, Phase 2 transitioned the system into a real-time, continuously updating environment. Minikube was used to establish a local Kubernetes cluster with enhanced resource allocations, ensuring stable performance of multiple services.

Zookeeper and Kafka were deployed using Kubernetes manifests, providing essential services such as distributed coordination and message brokering. Kafka was configured with internal and external listeners to enable seamless integration of producers and consumers across cluster boundaries.

Neo4j was deployed into the cluster using Helm charts, with a customized **neo4j-values.yaml** file enabling GDS plugin installation, persistent volume storage, and secure authentication. Kafka Connect, utilizing the custom Docker image **roy012299/kafka-neo4j-connect**, was deployed to act as a bridge between Kafka topics and Neo4j updates.

The **data_producer.py** script simulated real-world data ingestion by continuously streaming filtered taxi trip records into the **nyc_taxicab_data** Kafka topic at a rate of approximately one message every 0.25 seconds. Kafka Connect automatically transformed incoming JSON messages into Cypher queries, updating the Neo4j graph incrementally without manual intervention.

Interface analytics scripts were adapted to execute PageRank and BFS dynamically on the live, evolving graph, enabling near real-time analytical queries even as the underlying data structure continuously changed.

3.System Validation and Graph Verification

The graph structure was verified at multiple stages to ensure integrity and correctness. Visualization through the Neo4j Browser confirmed the presence of 42 :Location nodes interconnected by 1,530 :CONNECTS relationships. High-traffic pickup and drop-off locations were visually identifiable through thicker edge formations, reflecting expected urban mobility patterns.

Validation scripts reaffirmed the consistency of node and relationship counts, while PageRank and BFS outputs validated the logical connectivity and traversal capabilities of the graph. Node 159 emerged as the highest-ranked hub (~3.22825), highlighting its strategic importance within the transportation network, while node 59 consistently ranked lowest (~0.18247).

During streaming ingestion, continuous real-time validation was conducted to ensure that Kafka Connect reliably updated the graph structure without introducing inconsistencies, demonstrating the robustness of the end-to-end data pipeline.

4. Results and Achievements

The batch ingestion phase successfully produced a complete and validated graph database that accurately captured urban trip flows within the Bronx. Analytical results confirmed the identification of critical transportation hubs and commonly traversed routes, establishing a foundation for further real-time enhancements.

Transitioning to real-time streaming, the system achieved consistent ingestion of approximately 150 messages per second with an average end-to-end latency of around 200 milliseconds per update. Despite dynamic updates to the graph topology, analytical consistency was preserved across PageRank and BFS outputs, validating the architectural soundness of the pipeline.

These results demonstrate that combining batch-based modeling with real-time streaming is not only feasible but also scalable, offering a reliable solution for production-grade graph analytics systems.

5.Discussion

Several key lessons emerged throughout the project.

First, meticulous data preprocessing is essential to avoid schema mismatches, analytical inaccuracies, and ingestion bottlenecks later in the pipeline. Ensuring ISO 8601 time formats, deduplicated node mappings, and strict fare-distance filtering were critical success factors.

Second, containerization using Docker streamlined the development and deployment processes by eliminating environmental inconsistencies. Resource tuning, port forwarding, and authentication configurations proved essential for operational stability.

Third, orchestrating distributed services in Kubernetes introduced real-world complexities such as service discovery challenges, resource contention management, and external connectivity issues. Systematic debugging, YAML tuning, and persistent volume configurations were key to overcoming these obstacles.

Finally, the seamless integration of batch and real-time processing highlighted the importance of modular architecture designs that allow for gradual system upgrades without service disruption.

Future improvements could involve multi-node clustering of Neo4j and Kafka for increased fault tolerance, integration of real-time monitoring dashboards using Prometheus and Grafana, and enforcing schema evolution governance using Confluent Schema Registry.

6.Conclusion

This project successfully demonstrated the development of a scalable, production-ready graph analytics pipeline capable of both batch and real-time data processing. Beginning with a Dockerized Neo4j batch ingestion system and evolving into a Kubernetes-orchestrated, Kafka-driven streaming architecture, the project provided deep insights into distributed systems engineering, real-time analytics, and graph data science.

The hands-on experience gained in troubleshooting container orchestration, dynamic data ingestion, and scalable graph computation has laid a strong foundation for future work in cloud-native, high-performance data engineering domains.

7.References

1] Neo4j, The World's Leading Graph Database. Available at: **https://neo4j.com/**
[2] Apache Kafka Documentation. Available at: **https://kafka.apache.org/**
[3] Kubernetes Official Documentation. Available at: **https://kubernetes.io/docs/**
[4] NYC Taxi & Limousine Commission Trip Record Data. Available at: **https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page**
[5] Apache Parquet Documentation. Available at: **https://parquet.apache.org/**