University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

KubeGraphBench: Benchmarking Graph Databases for Kubernetes Observability

1st Charles O'Brian
School of Computer Science
University College Cork
Cork, Ireland
121448002@umail.ucc.ie

2nd Tarek Zaarour
Dell Research
Dell Technologies
Cork, Ireland
Tarek.Zaarour@dell.com

3nd Ahmed Khalid
Dell Research
Dell Technologies
Cork, Ireland
Ahmed.Khalid@dell.com

4th Ahmed H. Zahran
School of Computer Science
University College Cork
Cork, Ireland
ahmed.zahran@ucc.ie

***Abstract*—Graph Databases (Graph DBs) are increasingly adopted due to their efficiency in handling complex, interconnected data. They offer a robust alternative to traditional relational Databases (DBs), which often struggle with evolving schemas and deep, multi-hop relationships. The objective of this work is to evaluate the performance and capabilities of two different Graph DB technologies, namely Neo4j (Neo4j) [1] and Memgraph (Memgraph) [2]. The performance evaluation presented builds upon existing infrastructure for constructing and maintaining a knowledge graph representation of Kubernetes clusters. We focus solely on benchmarking the read and write performance of the DBs under increasingly larger data workloads, capturing metrics including latency, memory and CPU usage, and throughput. The DBs are tested fairly while ensuring they are placed under the same set of conditions. The differences in technologies between the DBs are understood and have been taken into account before drawing conclusions. Only the freely available versions of the DB technologies are used. The drawbacks of these versions are taken into account compared to their paid counterparts. Through experimentation, it is concluded that Memgraph, the in-memory storage graph database, outperformed Neo4j in write-heavy benchmarks, but during read-heavy benchmarking, it only outperforms Neo4j in the case of small graphs or simple queries.**

***Index Terms*—Benchmarking, Graph Database, Cloud Computing, Kubernetes, Resource management.**

I. Introduction

The growing complexity and interconnectivity of data in modern systems have accelerated the adoption of Graph DBs [3]. Graph DBs inherently store relationships as first-class entities (edges) alongside data points (nodes). This design is ideal for relationship traversal queries leading to consistent high performance, even as data volume and relationship depth grows [4]. The flexible schema of Graph DBs allows for new node types, relationships, or properties to be added without disrupting existing data models or access patterns. This flexibility is crucial in rapidly evolving systems. However, Graph DBs may not be as efficient for simple CRUD (Create, Read, Update, Delete) operations on individual data points, or attribute-based queries that do not rely on traversing relationships, especially when dealing with numeric data [5]. Additionally, in use cases with high event rates and complex real-time analytics, such as streaming graph workloads, traditional Graph DB designs may exhibit low throughput and latency due to their focus on transactional guarantees (e.g., atomicity and durability) and rich data models [6], [7].

Distributed computing systems are undergoing a fundamental transformation from centralized cloud architectures to a Computing Continuum (CC) that spans Cloud, Fog, Edge, and IoT tiers [8]. In this continuum, computing resources are distributed across the infrastructure to reduce latency and optimize bandwidth consumption. Workloads have also evolved from monolithic applications to microservice-based architectures to facilitate scalability, agility, and resilience [9]. This transformation was enabled by the rise of lightweight containerization platforms, e.g., Docker, and orchestration platforms, e.g., Kubernetes, which enable efficient resource utilization, rapid deployment, and cross-environment portability. These transformations define complex, evolving systems that require flexible, extensible data models, such as knowledge graphs, to facilitate intelligent, adaptive resource management decisions.

CC systems exhibit high dynamics for application deployment and infrastructure scaling. The operation of CC systems spans two planes: the provisioning & capacity management plane and the orchestration & scheduling plane. The former involves decisions relevant to scaling infrastructure (adding/removing compute, storage, or network resources) to optimize for resource demands and operating costs. The latter involves decisions related to the allocation of active infrastructure to application deployment requests to balance Quality of Service (QoS) or energy efficiency. Zaarour et al. [10] proposed maintaining the state of the CC infrastructure using a Knowledge Graph (KG) stored in a Graph DB. This graph is continuously updated in response to provisioning/orchestration actions as well as the dynamics of hosted workloads that impact the resource state. Additionally, the graph is queried for the latest state to drive new deployment and adaptation decisions. As such, the performance of Graph DB technologies becomes an essential factor in enabling timely and reliable decision-making in dynamic cloud-edge environments.

In this paper, we present *KubeGraphBench* as an extensible and configurable framework for benchmarking Neo4j [1] and Memgraph [2] when used to store the state and topology of

This research has been funded in part by the project CLEVER (Project ID 101097560), which is supported by the Key Digital Technologies Joint Undertaking and Enterprise Ireland (EI).

Kubernetes clusters. KubeGraphBench is easily configurable to vary cluster scale (using KWOK [11]) and to benchmark arbitrary Cypher queries. It is also designed to log a wide variety of metrics, including latency, throughput, and CPU/memory utilization. We performed an exhaustive performance evaluation considering *write-* and *read-heavy* operations. This evaluation enables validating the use of Graph DBs to maintain the state of Kubernetes clusters when executing provisioning and orchestration actions. Our benchmark is open-sourced[1] and can be found on Github.

The rest of this paper is organized as follows. Section II presents background and related work. Section III describes the design and implementation of the proposed benchmarking framework, followed by the performance evaluation in Section IV. We finally conclude and present future work in Section V.

II. Background and Related Work

This section highlights background information on relevant technologies and related work.

A. Graph Databases

Graph DBs represent data as nodes and edges [12]. They emerged to address limitations in traditional database systems when handling multi-hop relationships and dynamic, schema-flexible domains. Relational databases such as MySQL or PostgreSQL, which store data in structured tables and rely heavily on joins to traverse relationships, have been highly optimized for transactional workloads and structured data. However, they are less suited for deep or irregularly connected data due to the performance cost of recursive joins and rigid schema requirements.

This paper explores the performance of two widely adopted Graph DB technologies, Neo4j [1] and Memgraph [2]. It is important to note that the evaluation is based on the freely available community editions of both systems. Neo4j, written in Java, is an on-disk graph database that supports the property graph model and uses the Cypher query language for expressive graph querying. It stores nodes, relationships, and properties as fixed-size records linked through pointers, using linked list structures to efficiently traverse relationships. Memgraph, a newer in-memory graph database implemented in C++, is designed for low-latency, high-throughput applications. The differences in storage architecture (on-disk vs. in-memory) and implementation language (Java vs. C++) are key factors influencing the expected performance variations. Our choice of these two DBs for benchmarking is also motivated by their popularity, active adoption, and the compatibility of Memgraph with the Neo4j Python driver integrated into our existing system. However, we must note that while Memgraph is largely compatible with Neo4j's Cypher syntax, some features are not fully supported. For instance, Figure 1 shows a Cypher query that identifies the set of Pods matched by a Service's selectors and stores them as `(:Service)-[:EXPOSES]->(:Pod)` relationships for faster and explicit traversal. As shown, `WHERE ALL` filtering with `count()`-based comparison is unsupported in Memgraph, which required us to rewrite parts of the query logic. Thus, Memgraph can serve as a near drop-in replacement, though minor adjustments may be needed depending on the Cypher features used.

[1]Available at https://github.com/tarzaa1/dkg

Listing 1: Neo4j Cypher Query

```
MATCH (s:Service {id: $service_id})-[:HAS_SELECTOR]->(l:
    Label)
WITH s, collect(l) AS selectorLst
MATCH (p:Pod)
WHERE ALL(label IN selectorLst WHERE (p)-[:HAS_LABEL]->(
    label))
MERGE (s)-[:EXPOSES]->(p)
```

Listing 2: Memgraph-Compatible Query

```
MATCH (s:Service {id: $service_id})-[:HAS_SELECTOR]->(l:
    Label)
WITH s, collect(l) AS selectorLst, count(l) AS
    selectorCount
MATCH (p:Pod)
OPTIONAL MATCH (p)-[:HAS_LABEL]->(lab:Label)
WITH s, p, selectorLst, selectorCount, collect(lab) AS
    podLabels
WITH s, p, selectorCount,
    [label IN selectorLst WHERE label IN podLabels] AS
        matchedLabels
WHERE size(matchedLabels) = selectorCount
MERGE (s)-[:EXPOSES]->(p)
```

Fig. 1: Cypher query logic adapted for Memgraph compatibility due to lack of support for `WHERE ALL` clauses.

B. Benchmarking Graph DBs

Several benchmarking frameworks have been developed to evaluate the performance of graph databases. These frameworks typically involve a mechanism for populating the DB with data, and issuing a suite of various queries to collect performance metrics. A critical factor of any graph database benchmark or study is the choice of dataset. In any given study there are three different types of datasets.

Real-world datasets. Most studies use some kind of real-world data set that has been sourced from a large public repository such as SNAP. These studies rely on datasets targeting different domains, such as road networks [13], social networks [14], [15], or commercial transactions [16].

Synthetic datasets. Fewer studies synthesize their datasets to ensure controlled benchmarking, e.g., ABCD benchmark [17]. This type of dataset is useful as it allows for a controlled environment and repeatable testing, however its drawback is that it can fail to mirror real-life scenarios that can occur.

Hybrid. Finally, there are benchmarking frameworks, such as the SocialSensor GraphDB-Benchmarks suite [18] that support both real-world and synthetic datasets by structuring their benchmarks to be dataset-agnostic.

The framework presented in this paper introduces a hybrid dataset design, based on Kubernetes resources. We use the KWOK framework [11] to synthesize Kubernetes resources and build a graph, then execute a suite of Kubernetes-oriented

queries. The synthesized dataset enables a controlled environment where we can run repeatable tests of different dataset sizes to explore how the Graph DBs compare. Importantly, the framework also supports evaluation on real Kubernetes resources: the same components and query suite can operate on live and operational clusters, allowing for real-time construction of property graphs, monitoring and performance analysis of GDBs as clusters are provisioned and workloads are orchestrated.

C. Kubernetes

Kubernetes (K8s) [19] is an an open-source platform that automates the deployment, scaling, and management of containerized applications. It has become the de-facto standard for container orchestration in cloud-native infrastructure. Kubernetes architecture includes control plane and worker nodes. Control plane nodes manage the cluster's state while worker nodes run workloads. Kubernetes defines various objects to simplify the deployment and management for microservice-based applications. A *Pod* represents the smallest deployable unit, hosting one or more containers; A *Deployment* is a declarative controller that manages *ReplicaSets* to maintain *Pod* replicas and handle updates; a *Service* provides a stable network endpoint for ephemeral *Pods*. These objects follow clear relationships, e.g., a *Node* belongs to a *Cluster*, a *Pod* is scheduled on a *Node*, and a Service exposes a Pod. The structure can be modeled as a graph whose nodes represent Kubernetes objects and edges represent their relationships. Additionally, nodes and edges may own attributes (properties) to capture their resources and state, such as CPU, memory and storage capacity.

Modeling Kubernetes environments as graphs offers a powerful abstraction for supporting tasks such as resource allocation, root cause analysis, security auditing, and infrastructure observability. Capturing the relationships between workloads, configurations, infrastructure components, and runtime metrics, in a graph enables administrators and developers to reason about their clusters in a more intuitive way. However, as clusters scale, their graph representations grow not only in volume but also in structural complexity. The highly dynamic nature of Kubernetes, including frequent changes in workloads and continuous cluster observability data such as performance metrics and resource utilization, results in continuous updates to the graph, making it heavily write-intensive. At the same time, users rely on the graph to make operational and development decisions, which often involve complex traversals and multi-hop queries over up-to-date system state. These demands place significant pressure on the underlying Graph DBs, both in terms of ingestion throughput and query performance.

III. KubeGraphBench Design and Implementation

We first present the objectives and guidelines for our benchmark design before detailing its implementation.

A. KubeGraphBench Design

Our benchmark is designed to offer a flexible and extensible benchmark for Neo4j and Memgraph when used to maintain the state of Kubernetes clusters. Flexibility is attained through exposing the ability to configure both the computing infrastructure setup and benchmark queries. To enable flexible infrastructure setup, we leverage KWOK (Kubernetes Without Kubelet) [11] to emulate any generic Kubernetes cluster setup. KWOK is a lightweight tool that emulates Kubernetes clusters with virtualized objects (e.g., nodes, pods, services, deployments, etc.) without running any actual workloads. Importantly, KWOK writes the full metadata of these virtualized objects to `etcd`, just like a real Kubernetes cluster, making it indistinguishable from a real environment from a data ingestion standpoint. Hence, KWOK enables us to emulate complex, multi-cluster environments without infrastructure costs using a single machine, making it ideal for our testing and benchmarking purposes. Additionally, it supports configuring a wide variety of static and dynamic parameters for each virtual object. Static parameters include node types, labels, taints, capacities, conditions, and resource requests for pods, among others, while dynamic parameters include aspects such as resource utilization, which can vary over time to simulate real-world behavior. Hence, it represents a generic, flexible, and lightweight approach to generating infrastructure in comparison to static alternatives, such as datasets.

Furthermore, a user can define new queries in a provided configuration file to enable benchmarking the Graph DB performance for user-specified Cypher queries. This features enables our framework user to investigate the performance of queries involved in typical system operations, such as resource scheduling and management decisions. A user can run their benchmark by defining key configuration parameters, such as the number of nodes and the number of pods per cluster. The user also includes a list of queries (all by default) along with a list of which database to benchmark. Once the configuration is defined, our implementation handles all the benchmarking steps automatically, from spinning up clusters to saving logs for various performance metrics. This design eliminates the need for user manual intervention during the benchmarking process.

B. KubeGraphBench Implementation

Our implementation extends an open-source implementation of a Graph-based Kubernetes state and topology management system (KubeGraph) [10], [20], illustrated in Fig. 2. This system enables the real-time modeling of Kubernetes clusters as a property graph representing various infrastructure and workload-related objects and their relationships using 5 containerized components deployed in each cluster:

- **KubeInsights** monitors and retrieves the state and metadata pertaining to various Kubernetes objects (e.g., nodes, pods, deployments, services, etc.) using the kube-api-server and submits updates to an event broker, i.e., **Kafka**.
- **KubeGrapher** consumes state updates from the event broker as JSON blobs; transforms and stores these updates as a property graph in **Neo4j/Memgraph** using the Cypher graph query language.

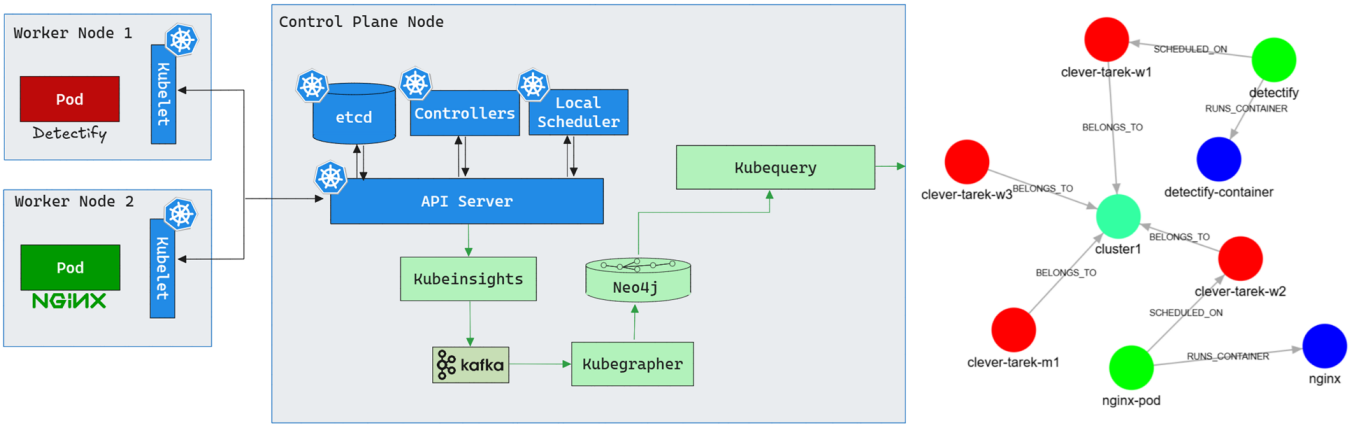


Fig. 2: System Architecture for Real-Time Kubernetes Knowledge Graph Construction and Querying

- **KubeQuery** implements a RESTful API which interfaces with the Graph DB to facilitate query operations via various endpoints that implement the appropriate Cypher query.

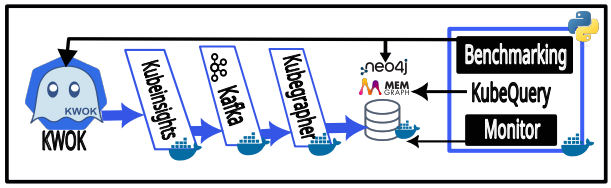


Fig. 3: Benchmark components and modifications.

To implement our benchmark, we replaced real infrastructure with KWOK emulated infrastructure (Fig. 3), developed monitoring components to facilitate logging performance metrics data, and extended **KubeQuery** to orchestrate the benchmarking process.

1) GDB performance metric collection: We collect a wide variety of metrics to capture Graph DB resource utilization (CPU and memory) and responsiveness (latency and throughput).

We implemented a generic monitoring script to capture CPU and memory utilization of tested Graph DB because many lightweight or open-source database editions lack built-in performance monitoring tools, our script collects raw CPU usage data from the */proc/stat* file, which provides interval-based measurements across CPU cores. This approach delivers finer granularity than other CLI tools, such as *ps*, which only report snapshot totals. For memory utilization, we extract information from the */proc/[pid]/statm* directory and combine it with values from */proc/meminfo* to compute memory consumption relative to the container's total memory. This method provides a more accurate and context-aware measurement of memory usage than relying solely on *top* [21] or *ps*. With tested databases containerized by design, the Docker Python library is used to execute system-level commands within the Graph DB container. Once started, the monitoring script begins capturing system statistics and writing them to a CSV file throughout each benchmark run. These statistics are sampled every 0.3 seconds. This script runs concurrently, in a separate thread alongside the main code, allowing continuous monitoring.

2) Query Design: To measure the query latency, we repeat each query 100 times with 0.1s inter-query idle period. For each query, relevant metadata, including parameters, duration, a start timestamp, an end timestamp, and response status, is saved to a CSV file for later analysis. Python's *time* module is used to record timestamps and calculate latency with millisecond precision. Note that a single "dummy" query is executed prior to benchmarking. This is intended to account for overhead incurred by the generation of a Cypher execution plan in Neo4j [22].

We also consider a query throughput metric to capture the DB's ability to process queries under high loads. To measure the throughput, we repeat sending the same query simultaneously and count the number of handled queries in 60 sec. This metric is expressed as queries per second.

The benchmark emphasizes **K-Hop Queries** [23] to evaluate neighborhood traversal performance, as they are most relevant for exploring structured resource relationships within Kubernetes-like environments.

Fig. 4 illustrates our data model comprising key graph nodes and relationships in a Kubernetes environment. The graph nodes represent physical and virtual computing entities, while graph edges represent the relationships between these entities. Both nodes and edges may maintain properties (key-value pairs) that represent their characteristics, state, and capabilities. Note that some graph nodes, such as compute nodes and pods, have an extensive property list capturing existing and available resources. Figure 5 highlights the K-hop considered in our evaluation.

3) Benchmarking Dataset: As mentioned previously, the dataset used in our benchmark was designed to emulate typical Kubernetes cluster configurations encountered in production environments. According to Kubernetes documentation, a large cluster is defined as comprising up to 5000 nodes, each hosting up to 110 pods [24]. In our benchmark, we test cluster sizes

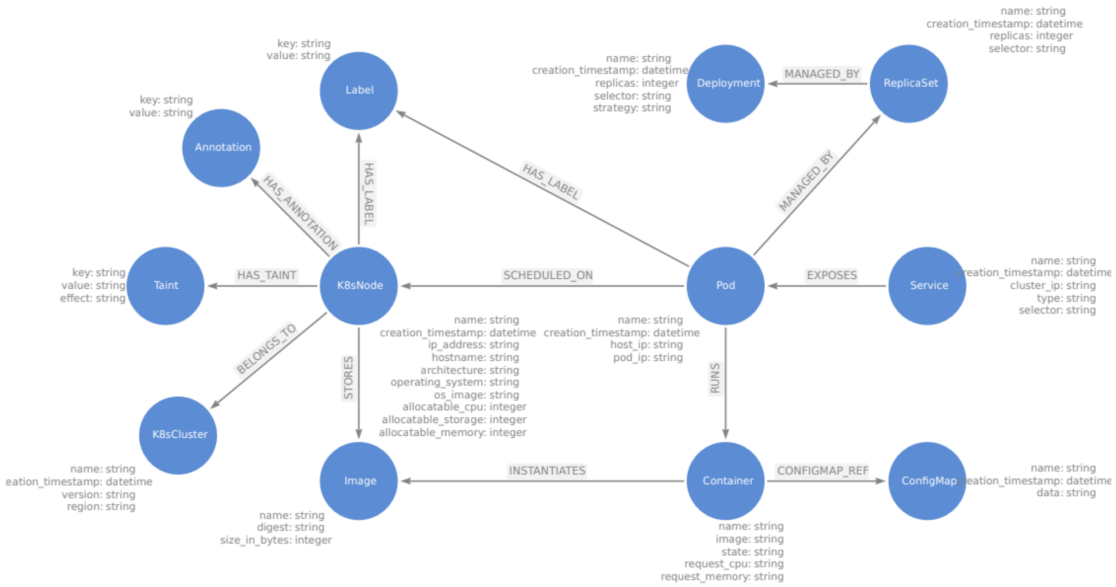


Fig. 4: Key graph nodes, edges, and properties representing a Kubernetes environment.

ranging from 10 to 1000 nodes, with 50 pods per node.

4) Benchmarking Orchestration: The benchmark is orchestrated using our modified **KubeQuery** implementation that operates as follows.

1) The user specifies a run configuration that includes the number of nodes and pods, the queries to run, and which database to benchmark.
2) Any existing KWOK clusters are deleted, and a new cluster is created with the specified resources.
3) The docker instances for Kafka and the graph database are restarted and any persisting data in the graph database is deleted.
4) A monitoring process is started, collecting meaningful CPU and memory utilization metrics from the graph database. The data is saved to a CSV file.
5) KubeInsights captures the state of the KWOK cluster and publishes it to Kafka.
6) KubeGrapher consumes the events from Kafka and inserts records into the graph database.
7) The benchmark queries specified by the user are executed against the graph database. Query latency and throughput are recorded in CSV files.
8) The process is repeated for the other graph database or with different configurations specified by the user.

IV. Performance Evaluation

We first present the evaluation setup before presenting the performance evaluation results.

A. Evaluation Setup

Experimentation setup. Our experiments are conducted using a MacBook M1 Pro (ARM64 architecture, 16GB RAM, macOS 15.3.1) machine. Docker Desktop (27.5.1) for Mac (Apple Silicon) is used to deploy containerized components that use Linux Debian/Ubuntu-based OS. We compared two Graph DBs, including Neo4j (5.26.0) and Memgraph (3.1.1) We consider one cluster with different numbers of nodes, including 10, 250, 500, 750, and 1000 nodes. Each node hosts a deployment involving 50 pods with each pod running a single container.

Let n denote the number of nodes and p, the number of pods per node in a benchmark run. Then the graph size is estimated as:

Vertices (graph nodes): $23 + (n * p) + (1 * p) + (8 * n)$
Edges: $15 + (10 * (n * p)) + 14 + (27 * (n - 1))$

TABLE I: Estimated vs Measured Graph Nodes and Edges. Note: Some data loss occurs when creating larger clusters and inserting the records into the graph, which may account for discrepancies between estimated and measured.

Config	Est. Nodes	Meas. Nodes	Est. Edges	Meas. Edges
1N 50P	81	81	529	529
250N 50P	14523	14010	131752	126640
1000N 50P	58023	57516	527002	521950

Our benchmark involves two stages: graph generation and query. In the generation phase, a cluster is generated based on the configuration file with CPU and memory utilization collected.

- **Latency** refers to the time it takes for the database to process a query and return its result. This metric reflects the responsiveness of the Graph DB and reflects its suitability for real-time, user-facing decisions.
- **Query throughput**, represented as queries per second, is the number of queries that can be executed within a 60 second time frame. This metric defines the Graph DB responsiveness to high query loads and large-scale systems.

Listing 3: Q1.1 – Return the cluster and all its nodes

```
MATCH (C:Cluster)<-[:BELONGS_TO]-(N:K8sNode)
WHERE C.id = "{cluster_id}"
RETURN C, N
```

Listing 4: Q2.2 – Return the cluster, all nodes, and their taints

```
MATCH (C:Cluster)<-[:BELONGS_TO]-(N:K8sNode)
WHERE C.id = "{cluster_id}"
MATCH (N)-[:HAS_TAINT]->(T:Taint)
RETURN C, N, T
```

Listing 5: Q3.3 – Return the cluster, nodes, pods on those nodes, and their ReplicaSets

```
MATCH (P:Pod)-[:SCHEDULED_ON]->(N:K8sNode)-[:BELONGS_TO]->(C
    :Cluster)
WHERE C.id = "{cluster_id}"
MATCH (P)-[:MANAGED_BY]->(R:ReplicaSet)
RETURN C, N, P, R
```

Listing 6: Q4.4 – Return the cluster, nodes, pods, ReplicaSets, and annotations on each ReplicaSet

```
MATCH (P:Pod)-[:SCHEDULED_ON]->(N:K8sNode)-[:BELONGS_TO]->(C
    :Cluster)
WHERE C.id = "{cluster_id}"
MATCH (P)-[:MANAGED_BY]->(R:ReplicaSet)-[:HAS_ANNOTATION]->(
    A:Annotation)
RETURN C, N, P, R, A
```

Listing 7: Q5.5 – Return the cluster, nodes, pods, their Deployments, and annotations on each Deployment

```
MATCH (P:Pod)-[:SCHEDULED_ON]->(N:K8sNode)-[:BELONGS_TO]->(C
    :Cluster)
WHERE C.id = "{cluster_id}"
MATCH (P)-[:MANAGED_BY]->(R:ReplicaSet)-[:MANAGED_BY]->(D:
    Deployment)-[:HAS_ANNOTATION]->(A:Annotation)
RETURN C, N, P, R, D, A
```

Fig. 5: Representative Cypher queries with increasing traversal depth (1–5 hops).

- **CPU Usage** represents the percentage of processing power consumed by the database when executing queries. Lower CPU usage under load generally indicates more efficient query processing.
- **Memory Usage** represents the percentage of RAM consumed during benchmarking. This includes memory used for storing graph structures (nodes, relationships, and properties), indexing, and query execution. Efficient memory usage is important for running large graphs or operating within resource-constrained environments.

B. Graph Generation

We consider various metrics, such as latency, CPU and memory usage, during graph generation as measures for resource utilization and intensive writing speed. We measure the graph generation latency starting after the building of the KWOK cluster, from when the first record is inserted until all corresponding records are inserted in the tested Graph DB. Fig. 6 shows that Memgraph features lower graph generation latency by 25%-37% when compared to Neo4j for different cluster sizes. This result indicates Memgraph's superiority with respect to intensive write operations, which is relevant to infrastructure scaling in federated computing systems.

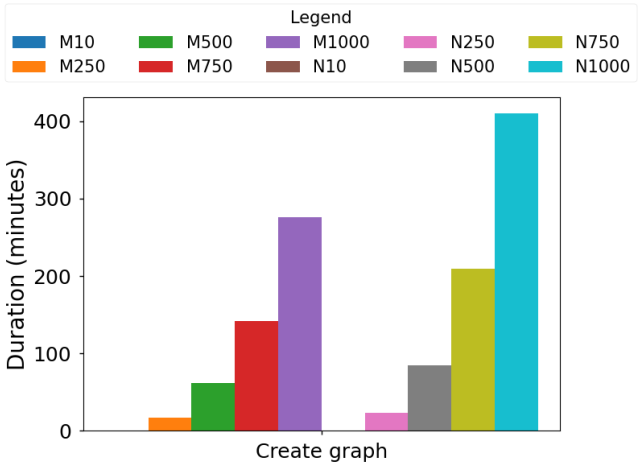


Fig. 6: Create Graph Durations across Tests

Fig.7 compares the CPU and memory utilization over time during graph creation for different clusters sizes. Across all dataset sizes, Memgraph exhibits a significantly lower memory footprint, never exceeding more than 2.5%. We see for Neo4j, that as the dataset size increases, the memory usage increases. Most notably in Fig.7c, for the 1000-node cluster, Memgraph used only 2% of memory compared to 15% for Neo4j.

Excluding 10-node cluster, both Graph DBs showed similar CPU utilization, with Neo4j exhibiting slightly higher usage when active. As shown in Fig. 7a, Memgraph uses a limited amount of CPU while Neo4j utilization reached 100% to process all insertion queries.

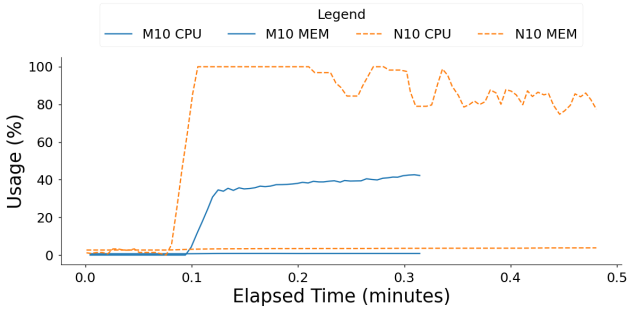
We believe Neo4j excess memory utilization is attributed to the use of multiple stores for properties, relationships, and nodes. Additionally, Neo4j runs on the Java Virtual Machine (JVM), which introduces overhead due to abstraction and garbage collection (GC). On the contrary, Memgraph is designed to use memory-efficient storage models and has no GC overhead as it is written in C++. These findings are consistent with Memgraph's recent BenchGraph benchmark report [25], where Memgraph demonstrated significantly lower memory utilisation than Neo4j.

C. K-hop Traversal Performance

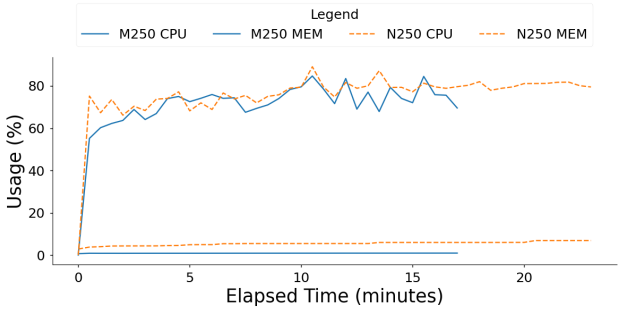
TABLE II: Number of tested K-hop queries.

K	1	2	3	4	5
Queries	10	7	8	3	2

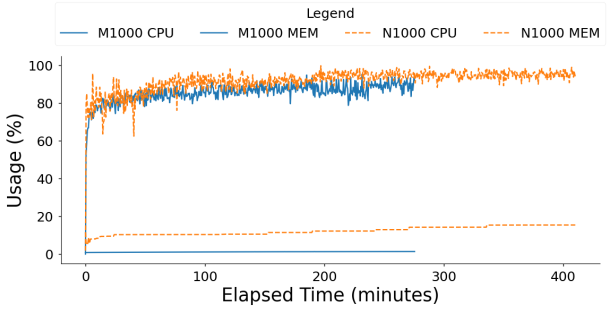
Our read performance is defined by the latency and throughput of various K-hop queries. We executed different number of queries for different K-hop scenarios, as illustrated in Table



(a) 10-Node Cluster.

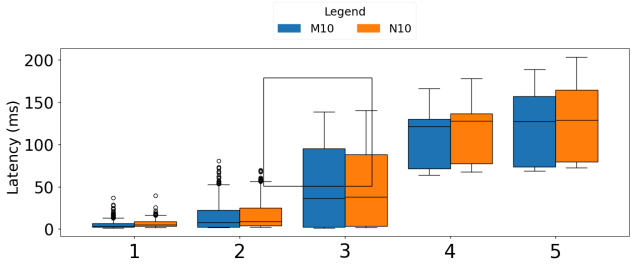


(b) 250-Node Cluster.

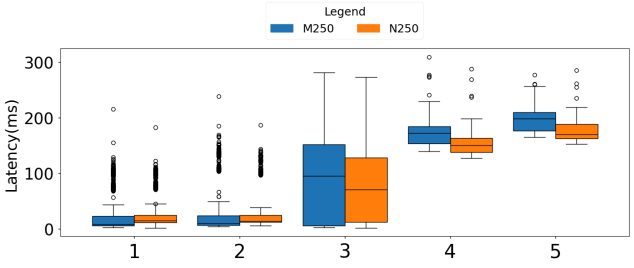


(c) 1000-Node Cluster.

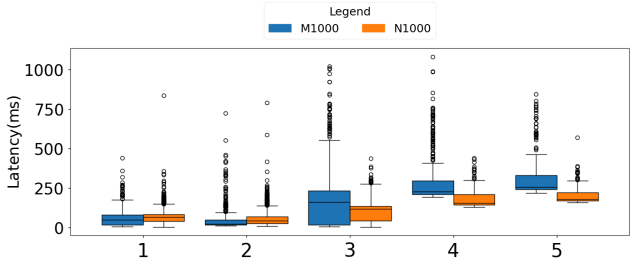
Fig. 7: CPU and Memory utilization for different cluster sizes.



(a) 10-node Cluster.



(b) 250-node Cluster.



(c) 1000-node Cluster.

Fig. 8: K-hop Query Latency Statistics

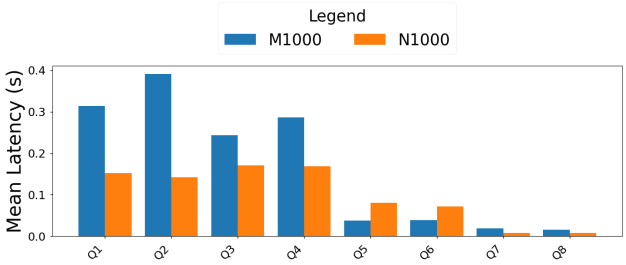


Fig. 9: 3-Hop Queries - Mean latency

II. Note that more possible combinations are possible around 3-Hop and 4-Hop queries. Fig. 8 shows the latency boxplot for different K-hop queries.

The figure indicates that for the smaller, 10-node Cluster seen in 8a both databases show similar latencies across all K-Hop queries.

However, when cluster size increases in 8b and 8c, Neo4j achieves lower latencies, particularly for deeper, multi-hop traversals (3-hop, 4-hop and 5-hop). Memgraph still occasionally achieves lower latencies for shallower traversals (1-hop and 2-hop).

Figure 8 also shows considerable variance in latency, particularly for Memgraph, with numerous outliers visible on the plot. This variance is attributed to the diversity in the latency of the individual queries in different K-hop groups. Figure 9 illustrates this diversity by plotting the mean latency across all 3-hop queries tested for a 1000-node cluster for both databases. We believe that this diversity is caused by the amount of data retrieved by different queries and data storage even though they target the same hop depth. We believe that Neo4j exhibits lower latency variance as its storage arrangement reduces retrieval latency when compared to Memgraph.

Fig. 10 shows the mean throughput of various K-hop queries across all of the tested cluster sizes. It is evident that the throughput drops as the we increase the cluster size. Memgraph maintains higher throughput at the 10-node scale, while Neo4j demonstrates stronger scalability, achieving

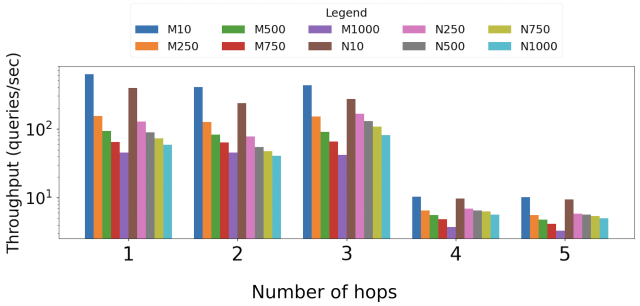


Fig. 10: Query throughput

significantly higher throughput at larger cluster sizes, especially for 4 and 5 hop queries. We believe Neo4j speedy performance results from relying on caching strategies (off-heap page caching (for disk data) and on-heap caches (query results, indexes)). On the contrary, Memgraph adopts a single-layer, pointer-based data structures in RAM that does not exploit any translation between storage and execution.

V. Conclusion and Future Work

We presented KubeGraphBench, an extensible framework for benchmarking Graph DBs as a backend data store for Kubernetes observability. We benchmarked two popular property graph databases, namely Neo4j and Memgraph. By emulating a cluster environment using KWOK, and constructing a knowledge graph from synthetic yet realistic Kubernetes resources, the benchmark provides a platform for stress-testing the graph databases in a way that closely mirrors actual observability and infrastructure monitoring workloads. As for our results, Memgraph consistently wrote data into the database faster than Neo4j across all dataset sizes. Memgraph demonstrated high throughput during the create graph phase. Neo4j consumed significantly more CPU and memory compared to Memgraph during the write phase. Memgraph maintained a low and stable memory footprint even at large scales. Read/query performance varied across workloads. At smaller scales and lower data volumes, Memgraph often achieved lower latency and higher throughput. However, at larger dataset sizes and deeper, multi-hop queries, Neo4j proved to be more reliable and scalable, consistently outperforming Memgraph in both latency and throughput.

Future work includes expanding the benchmarking to include additional Graph DB technologies. We also plan to evaluate the built-in graph algorithm libraries, such as the APOC in Neo4j and Mage in Memgraph. Another key direction is analyzing the impact of indexing on both read and write performance. Finally, we aim to explore query optimization techniques tailored to common graph patterns encountered in Kubernetes environments.

References

[1] Neo4j, "Neo4j - the world's leading graph database," https://neo4j.com/, accessed: 2025-04-15.

[2] Memgraph, "Memgraph - real-time graph streaming platform," https://memgraph.com/, accessed: 2025-04-15.

[3] G. Demarestn, "What is behind the surging interest in graph databases?" 2023, accessed: 2025-04-16. [Online]. Available: https://aerospike.com/blog/what-is-behind-the-surging-interest-in-graph-databases/

[4] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazan, and J. L. Larriba-Pey, "Survey of graph database performance on the hpc scalable graph analysis benchmark," in *International Conference on Web-Age Information Management*. Springer, 2010, pp. 37–48.

[5] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual ACM Southeast Conference*, 2010, pp. 1–6.

[6] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A performance evaluation of open source graph databases," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. PPAA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 11–18. [Online]. Available: https://doi.org/10.1145/2567634.2567638

[7] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1860–1876, 2021.

[8] S. Dustdar, V. C. Pujol, and P. K. Donta, "On distributed computing continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2022.

[9] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[10] T. Zaroour, A. Khalid, P. Preedep, and A. Zahran, "Using distributed ledgers to build knowledge graphs for decentralized computing ecosystems," 2024, accessed: 2025-04-16. [Online]. Available: https://dl.acm.org/doi/10.1145/3627673.3679644

[11] Kubernetes SIGs, "Kwok: Kubernetes without kubelet," https://github.com/kubernetes-sigs/kwok, 2024, accessed: 2025-04-15.

[12] M. Besta, R. Gerstenberger, E. Peter, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *ACM Comput. Surv.*, vol. 56, no. 2, Sep. 2023. [Online]. Available: https://doi.org/10.1145/3604932

[13] Network Repository, "road network dataset," 2024, accessed: 2025-04-16. [Online]. Available: https://networkrepository.com/road.php

[14] ——, "soc-twitter-2010 network dataset," 2024, accessed: 2025-04-16. [Online]. Available: https://networkrepository.com/soc_twitter_2010.php

[15] ——, "soc-livejournal1 network dataset," 2024, accessed: 2025-04-16. [Online]. Available: https://networkrepository.com/soc-livejournal.php

[16] K. Contributor, "Amazon product co-purchasing network graph," 2020, accessed: 2025-04-16. [Online]. Available: https://www.kaggle.com/datasets/wolfram77/graphs-snap-com-amazon

[17] B. Kamiński, P. Prałat, and F. Théberge, "Artificial benchmark for community detection (abcd): Fast random graph model with community structure," *arXiv preprint arXiv:2002.00843*, 2020. [Online]. Available: https://arxiv.org/abs/2002.00843

[18] S. Project, "Graphdb benchmarks," 2014, accessed: 2025-04-16. [Online]. Available: https://github.com/socialsensor/graphdb-benchmarks

[19] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," vol. 55, no. 7, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3539606

[20] T. Zaarour, "Kubegraph: Modeling kubernetes objects as a property graph," https://github.com/tarzaa1/dkg, 2025, accessed: 2025-07-04.

[21] Goran Jevtic, "How to check cpu utilization in linux with command line," https://phoenixnap.com/kb/check-cpu-usage-load-linux, 2024, accessed: 2025-04-15.

[22] Neo4j, "Understanding execution plans," 2025, accessed: 2025-04-16. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/planning-and-tuning/execution-plans/

[23] C. C. Warrior, "Graph databases: Are multi-hop queries really slower?" 2024, accessed: 2025-04-16. [Online]. Available: https://medium.com/@confusedcyberwarrior/graph-databases-are-multi-hop-queries-really-slower-90193acf5e49

[24] K. SIGs, "Considerations for large clusters - kubernetes documentation," 2024, accessed: 2024-04-26. [Online]. Available: http://spahttps://kubernetes.io/docs/setup/best-practices/cluster-large/

[25] Memgraph, "Benchgraph - memgraph's graph database performance benchmark," https://memgraph.com/benchgraph/base, accessed: 2025-04-15.