

---

# Graph Neural Networks on Graph Databases

---

**Dmytro Lopushanskyy**

University of Oxford  
dmytro.lopushanskyy@keble.ox.ac.uk

**Borun Shi**

Neo4j  
brian.shi@neo4j.com

## Abstract

Training graph neural networks on large datasets has long been a challenge. Traditional approaches include efficiently representing the whole graph in-memory, designing parameter efficient and sampling-based models, and graph partitioning in a distributed setup. Separately, graph databases with native graph storage and query engines have been developed, which enable time and resource efficient graph analytics workloads. We show how to directly train a GNN on a graph DB, by retrieving minimal data into memory and sampling using the query engine. Our experiments show resource advantages for single-machine and distributed training. Our approach opens up a new way of scaling GNNs as well as a new application area for graph DBs.

## 1 Introduction

Graph-structured data is ubiquitous, underlying many of the complex systems we interact with daily, from social and transportation networks to molecular structures and knowledge graphs. As these datasets continue to grow in volume and complexity, there is an increasing need for machine learning systems that can identify meaningful patterns within graphs. Graph neural networks (GNNs) have emerged as a state-of-the-art solution for many of these graph machine learning tasks, driving advances in fields as diverse as recommender systems, fraud detection, and drug discovery.

Despite their success, GNNs still face significant challenges when it comes to scalability. Large real-world graphs often contain billions of nodes and edges, bringing an additional memory bottleneck. There exist several prior approaches that improve scalability. Compressing a sparse graph’s in-memory representation, for example, in Compressed Sparse Row (CSR)[1], reduces memory needed to store edges. Parameter sharing techniques and sampling-based GNNs[2–4] which reduce neighbourhood explosion improve training speed and hence the applicability of models to large graphs. Distributed GNN training frameworks [5, 6] enable one to scale training speed with respect to hardware resources available.

However, the scalability issues of GNNs have not been fully resolved. In-memory representation of large, heterogeneous and feature-rich graphs remains memory-bound. In distributed setups, new trade-offs arise between memory efficiency and computational speed, which are caused by data locality issues and inter-node communication overhead brought by graph partitioning. There are also other practical issues facing GNNs. Challenges in handling real-world dynamic graphs include linear time graph update (in CSR), ensuring data consistency, and synchronisation for partitioned graphs. The Extract, Transform and Load (ETL) layer adopted by graph ML remains inefficient, as it frequently involves the update and reloading large CSVs.

Separately, graph databases, query languages and query engines have evolved. A graph DB treats nodes and edges as primary entities in its graph schema, storage format, query language, as well as in the operators implemented by the query engine. This allows time- and space-efficient execution of native graph queries. For example, finding all nodes reachable from a given source node can be expressed and executed conveniently in a graph DB. If the data is instead modelled as tables and queried through SQLs, complex many-to-many joins are involved. Typical database properties that exist in relational databases, such as ACID transactions, online transactional processing (OLTP)

and online analytical processing (OLAP) workloads, are supported by graph DBs. Naturally, it is increasingly common to store large graphs in various graph DBs within the industry.

The lack of synergy between graph ML and DB is largely due to the historical fact that they started as separate fields. However, it becomes increasingly obvious that a tighter coupling improves graph ML pipelines. To start with, data copy from the DB into GNN workload is wasteful and slow. On a deeper level, both subjects are often solving the same underlying problems on graphs, which include efficient representation and retrieval of graph information.

We leverage graph DBs as a core component of GNN. We show how to formulate graph ML tasks like neighbour sampling and feature retrieval as queries for graph DBs. As a result, we significantly improve the memory efficiency of training GNNs.

## 2 Background

### 2.1 Graph neural networks

One of the dominant models for learning on graphs is message passing neural networks (MPNNs), a class of GNNs. An MPNN operates on all the nodes in a graph, for each node recursively collects information from neighbouring nodes, and aggregates and updates the node's information until convergence. Given a graph  $G = (V, E)$  with node initial features  $x_v^{(0)} \in \mathbb{R}^{1 \times n}$  for  $u \in V$ , an MPNN updates a node embedding  $x_v^k$  at layer  $k$  by the equation:

$$\mathbf{x}_v^{(k)} = \phi^{(k)} \left( \mathbf{x}_v^{(k-1)}, \bigoplus_{u \in \mathcal{N}(v)} \psi^{(k)} \left( \mathbf{x}_v^{(k-1)}, \mathbf{x}_u^{(k-1)} \right) \right), \quad (1)$$

where  $\psi$  is a message function,  $\bigoplus$  is an injective multiset function, and  $\phi$  is a update function, and  $\mathcal{N}(v)$  is some neighbourhood of  $u$  such as  $\{u \in V : (v, u) \in E\}$ .

Reducing the number of parameters in  $\phi$ ,  $\bigoplus$ ,  $\psi$  improves model scalability. The sampling of neighbourhood  $\mathcal{N}(v)$  further reduces the worst-case parametric complexity from  $O(|V|)$  to some configurable constant  $O(S_k)$  for any layer  $k$ . These methods underpin the majority of modern GNNs, with a good example being (mean-aggregate) GraphSAGE[2]:

$$x_v^{(k)} = \sigma \left( W^{(k)} \cdot \text{CONCAT} \left( x_v^{(k-1)}, \text{MEAN} \left( \{x_u^{(k-1)} \mid u \in \mathcal{N}^k(v)\} \right) \right) \right) \quad (2)$$

Here  $\sigma$  is some non-linear activation function,  $W^{(k)}$  is the only layerwise-learnable weight, and  $\mathcal{N}^k(v)$  is some layerwise sampled neighbourhood of  $v$ .

### 2.2 Implementation of GNNs

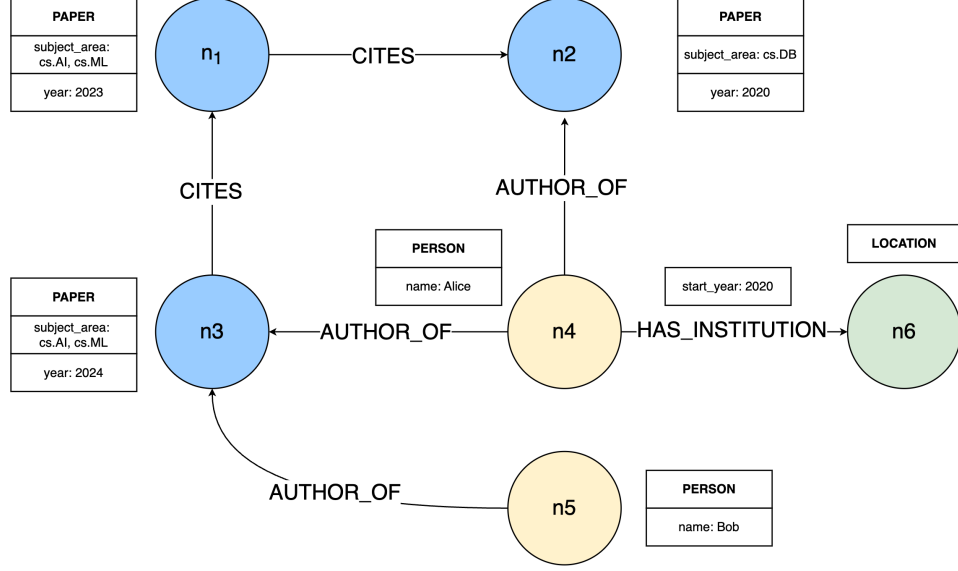
There exist several software libraries for GNNs, such as PyTorch Geometric (PyG) [7], the Deep Graph Library (DGL) [8] and TensorflowGNN [9]. A typical workflow is to load the entire graph from some files (CSVs) into memory for single-machine training.

Distributed training techniques have been developed to speed up the training process by scaling it with respect to the available hardware resources. The starting point is some graph partitioning algorithm, such as METIS [10, 11]. Such algorithms usually achieve a trade-off between good workload-balance to ensure all training processes are well saturated and min-cut across partitions to reduce inter-process communications [12]. Libraries such as DistDGL [5] and PyTorch Distributed Data Parallel (DDP) [6] support such setups. [13, 14] addresses the engineering bottlenecks in workload balancing, scheduling of pre-sampling steps and CPU utilisations.

### 2.3 Graph databases and graph query languages

There are multiple ways to model graphs within graph DBs. Two of the most common ones are Labelled Property Graph (LPG)[15] and Resource Description Framework (RDF)[16], each equipped with its own way of querying the graph.

An LPG is defined as a tuple  $(V, E, L, l_v, l_e, K, W, p_v, p_e)$ .  $V$  denotes the the set of vertices,  $E$  the set of edges,  $L$  the set of labels,  $l_v : V \rightarrow \mathcal{P}(V)$  the label assignment function on vertices,



**Figure 1:** An example labelled property graph. Simple citation graphs such as cora[17] or ogbn-papers100M[18] can be modelled by such schema.

```
MATCH (n:PAPER)-[:CITES]->(m:PAPER)
RETURN DISTINCT m
```

```
SELECT DISTINCT p.*
FROM PAPERS p
JOIN CITES c ON p.id = c.
cited_paper_id
```

**Figure 2:** Left: A Cypher query that returns all papers that are cited by at least one other paper. Right: An equivalent SQL query, assuming a reasonable table schema, such as one table PAPERS with properties columns and a second table CITES with columns citing\_paper\_id and cited\_paper\_id.

$l_e : E \rightarrow \mathcal{P}(E)$  the label assignment function on edges,  $K$  the set of keys (attribute or property names),  $W$  the set of values (attribute or property values),  $p_v : V \rightarrow K \times W$  the property key-value assignment functions on vertices,  $p_e : E \rightarrow K \times W$  the property key-value assignment functions on edges. Loosely speaking, LPG allows the most flexible modelling of any graph. For example, vertices and edges can share the same labels, property keys, and values. Figure 1 shows an example of an LPG graph.

There exist several query languages for LPG, such as Cypher[19, 20] and openCypher [21], which form the basis of the recent ISO GQL standard [22, 23]. The power of Cypher comes with graph pattern matching and the fact that nodes and edges are treated as first-class entities in LPGs. Figure 2 shows an example Cypher query.

An increasing number of industrial LPG graph databases support Cypher, such as Neo4j [24], ArangoDB [25], TigerGraph [26].

RDF, which originated from the semantic web, adopts a different graph data model. The central entities in RDF are triples, usually denoted as  $(s, p, o)$ , for subject-predicate-object. Therefore, RDF graph databases are also commonly referred to as triple stores. SPARQL[27] is a common RDF query language. Implementations of triple stores include RDFox [28]. While it is common to use Cypher over LPGs and SPARQL over RDFs, there exist implementations that allow one to execute one type of query language over another graph data model, such as AWS Neptune [29]. With minor modifications, Cypher queries can be translated into SPARQL.

```

MATCH (n)
WITH DISTINCT labels(n) AS NodeTypes, n
UNWIND NodeTypes AS NodeType
WITH NodeType, collect(distinct keys(n)) AS AllKeys
RETURN NodeType, reduce(s = [], k IN AllKeys | s + k) AS Attributes

```

**Figure 3:** Cypher query that returns all relevant metadata for all nodes.

### 3 Related Work

To the best of our knowledge, there is no previous work that trains a GNN directly on a graph database leveraging its query engine. There are several preliminary work that attempts to integrate graph DBs with ML workflows. Kùzu [30] extended PyG’s default `featureStore` and `graphStore`. Being an embedded graph DBMS, the entire graph is loaded into the memory of one machine and converted (in a zero-copy manner) to torch tensors, after which existing single-machine PyG training can be used.

LPG2vec [31] and RDF2vec [32] are shallow embedding methods that apply directly to the aforementioned graph representations, which can be seen as different heterogeneous extensions of node2vec [33]. [34] proposed extending (graph) query engines to handle algebraic operations on these embeddings. Relational deep learning[35] applies GNNs on tabular data by first forming a (hyper)graph.

As ML systems evolve and data grows in size, compound systems become increasingly common [36] as opposed to single monolithic models that are often small. For example, databases recently appeared to be an important retrieval component in the retrieval augmented generation (RAG)[37–39] framework for large language models. Our work could be viewed as introducing a more tightly coupled retrieval component within the graph ML pipeline.

While we focus on database support for GNNs, in-database learning [40–42] studies the use of relational databases to support the implementation of standard ML models, such as linear regression.

### 4 Training GNN on graph DB

We now present our approach, which offloads all retrieval steps of a typical GNN training pipeline to the graph DB. In particular, we only retrieve necessary metadata at the start and do not materialise any graph information. For each batch of nodes, we sample their neighbours and retrieve their features through executing queries. Our approach applies to a variety of GNNs, tasks, datasets and software libraries.

**Initial Data Retrieval.** The first step in training GNNs in existing workflows involves loading the graph, with initial features, into memory. Various representation formats, such as CSR, Coordinate Format (COO) and edge list, offer different memory and complexity tradeoffs. Two sets of configurations, one being user-defined hyperparameters, such as the number of layers, and the other being graph metadata, such as the number of nodes and initial feature dimensions, are used to initialise the dataloader and model.

Up until this point, the graph does not need to be fully materialised. Instead, we only retrieve the necessary graph metadata from the graph in the DB. Figure 3 shows the query that retrieves node metadata. Any edge metadata can be similarly obtained by another Cypher query which we include in Appendix A.

For example, when training ogbn-papers100M[18], our setup requires less than 1KB of RAM, instead of 48GB in the regular setup, during the initialisation stage.

**Neighbour Sampling.** Once the system is initialised with the necessary metadata, it proceeds with the mini-batch training steps. As captured by the update Equation 2, the key operations are sampling and feature extraction. Typically, the sampling operation has access to the full graph structure and is implemented in-memory. We offload the operation to the database. We construct a query template, as shown in Figure 4, which, when supplied with seed nodes and sampling parameters, gets executed on the database. As a result, only the absolute minimal information is materialised in the training process.

```

MATCH (node_0:$NODE_TYPE)
WHERE node_0.id IN $SEED_NODES
OPTIONAL MATCH (node_0)-[rel_1:$REL_TYPE]->(node_1:$NODE_TYPE)-[rel_2:$REL_TYPE]
->(node_2:$NODE_TYPE)
WITH node_0, node_1, node_2
ORDER BY rand()
LIMIT $MAX_NEIGHBOURS
RETURN
    node_0.id as src_id,
    node_1.id, node_1.features,
    node_2.id, node_2.features;

```

**Figure 4:** A Cypher query template that samples two-hop neighbourhoods of given seed nodes. Features of sampled nodes are returned in the same step.

Since there is no keyword in Cypher that is equivalent to `SAMPLE`, it is not immediately obvious how sampling can be done as part of the query. From a database point of view, sampling does not traditionally fit into either an OLTP or an OLAP workload. One might attempt to use a query similar to Figure 2, and then `LIMIT` the output to the number of samples. However, the semantics of `MATCH ... RETURN ...` do not have any ordering guarantee.

A single execution of the query with a specific set of seed nodes on a fresh database often returns results in nondeterministic ordering. On the other hand, repeated executions give the same ordering because of the caching mechanisms implemented by the majority of databases. Therefore, we utilise the combination of an in-line function `rand()`, `ORDER BY` and `LIMIT` to guarantee uniformly randomly sampled neighbourhoods.

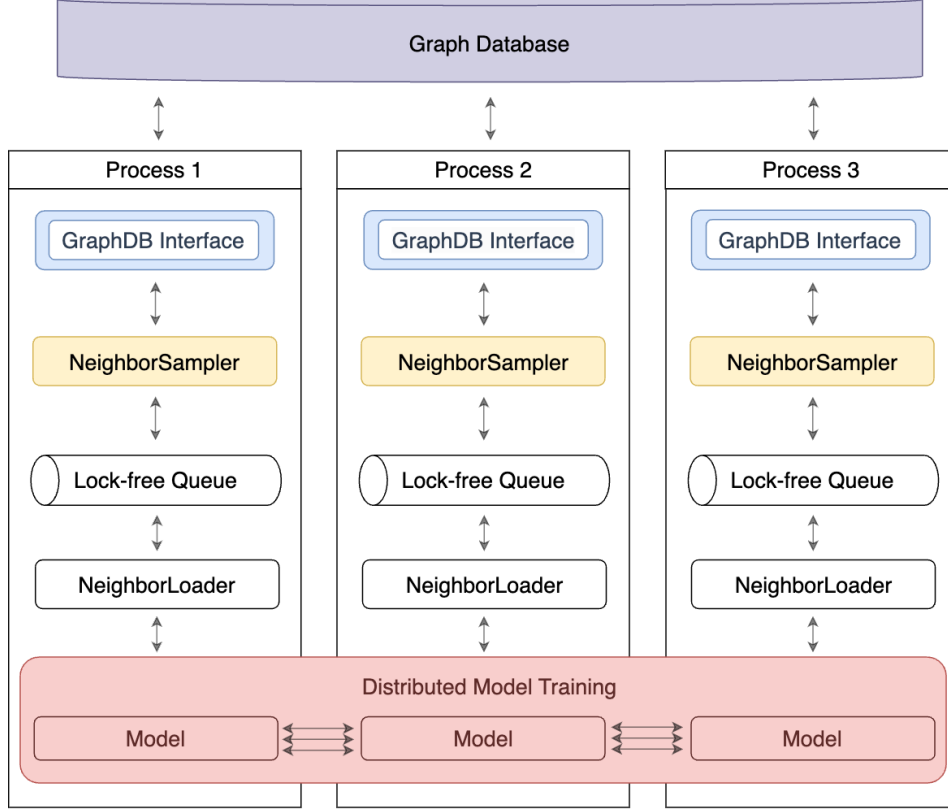
Multi-hop neighbourhoods are frequently used to allow nodes to aggregate information from farther away. One approach we could take is to separately query for each hop. Sampled neighbours returned are used as seed nodes for the next query (second hop), and so on. This query is shown in Appendix B. However, this increases latency and the resources required. More importantly, it eliminates the advantage of a query engine that is designed to execute multi-hop queries. Hence, our query performs multi-hop sampling by chaining together multi-hop pattern matching.

Node-wise uniform random sampling per layer, as implemented by the original GraphSAGE, requires a `LIMIT` clause at each hop. Consequently, the query engine has to materialise all intermediate neighbours. As an optimisation, we instead only limit the final neighbourhood size, which greatly improves the time and resource usage. Our query is shown in Figure 4.

**Feature retrieval.** Since we no longer store the entire graph in-memory, we also offload feature storage to the database. One option is to use separate storage for both the features (such as vector databases[43]) and the graph. However, the storage formats of most graph DBs and their query engines allow efficient retrieval of features when retrieving the node IDs since node attributes are co-located with node IDs in storage[24]. This allows us to use one graph DB as the storage, selecting relevant attributes as shown in Figure 4. The engineering advantage of not needing to set up a separate feature store and maintaining cross-DB consistency is non-negligible.

**Distributed training.** Distributed training is enabled by the graph partitioning methods introduced in Section 2.2. Existing graph partitioning methods are often slow and nondeterministic (which affects reproducibility). Additionally, they require re-partitioning for any modification of the graph. While inter-process communication overhead is minimised by workload-balancing min-cut partitioning, such communication still happens, which introduces unpredictable latency or could become a failure point in a system. Communication free partitioning[12] comes at the cost of additional halo nodes.

If we take a step back, this is one solution to the problem of *how to store data that allows efficient concurrent reads*. This is exactly what is solved by graph databases that support horizontal scaling for OLAP transactions[44]. The partitioning solution is commonly referred to as graph sharding[45, 46], while simply duplicating copies of the data is another viable option.



**Figure 5:** Our distributed training architecture. One graph database acts as a central graph and feature store. Multiple training processes can concurrently sample from the DB with no overhead. Our architecture closely mimics that of PyTorch Distributed Data Parallel[6] and applies to any other similar setup.

Since we have formulated sampling operations as a special type of read transaction, our method enables distributed training almost for free. One simply sets up the distributed training architecture as usual and configures each process to sample from the same database, as illustrated in Figure 5.

## 5 Experiments

We validate the applicability and scalability of our approach. We use ogbn-papers100M and ogbn-products [18], two real-world large graphs with different topology. We use Neo4j as our graph database. We refer to Appendix C for our database, hardware and other training configurations. A detailed ablation study of performance with respect to different software stacks and database vendors is out of the scope of this work. We use PyG and adapt its GraphStore and FeatureStore to be interfaces for Neo4j and implement a custom Neo4jGraphSampler. We benchmark our approach for both single-machine and distributed training setups. Our code is available on GitHub.<sup>1</sup>

### 5.1 Analysis of the query plan

Formulating sampling as a special type of retrieval query is a central component of our approach. We verify that our query is indeed correct. We utilise the profiling tool provided by the query engine (keyword PROFILE) to examine the execution details of all the operators involved.

As shown in Table 1, the correct number of initial and final rows are produced, and at each hop, we are able to verify the number of total neighbours (88, 18524). Two one-hop traversals (steps 8 and 6) are performed with filters applied optimally. An alternative execution plan could be first finding the

<sup>1</sup><https://github.com/DmytroLopushansky/ScalingGNNs>



**Table 1:** An example query execution plan for Figure 4 with a specific set of 10 \$SEED\_NODES and 100 \$MAX\_NEIGHBOURS. Operator are executed from the bottom up. Details represent the exact execution parameters. Estimated Rows represent expected rows produced. Rows represent actual rows produced. DB Hits measure the amount of work by the storage engine. Total database access is 38570, total allocated memory is 19120 bytes.

| Operator        | Id | Details   | Estimated Rows | Rows  | DB Hits |
|-----------------|----|---|----------------|-------|---------|
| +ProduceResults | 0  | node_0, node_1, node_2  | 100            | 100   | 1102    |
| +Top            | 1  | rand() ASC LIMIT 100  | 100            | 100   | 0       |
| +Projection     | 2  | rand() AS 'rand()'  | 8381           | 18524 | 0       |
| +Apply          | 3  |   | 8381           | 18524 | 0       |
| +Optional       | 4  | node_0  | 8381           | 18524 | 0       |
| +Filter         | 5  | NOT anon_1 = anon_0 AND node_2:PAPER                                      | 8381           | 18524 | 18524   |
| +Expand(All)    | 6  | (node_1)-[anon_1:CITES]-(node_2)  | 8466           | 18612 | 18738   |
| +Filter         | 7  | node_1:PAPER  | 291            | 88    | 88      |
| +Expand(All)    | 8  | (node_0)-[anon_0:CITES]-(node_1)  | 291            | 88    | 98      |
| +Argument       | 9  | node_0  | 10             | 10    | 0       |
| +NodeIndexSeek  | 10 | TEXT INDEX node_0:PAPER(id)<br>WHERE id IN \$autolist_0, cache[node_0.id] | 10             | 10    | 20      |

**Table 2:** Single-machine training. Batch Time measures the average time per batch. Sampled Nodes record average sampled nodes per batch and, similarly, Sampled Edges.

| Backend           | RAM           | Batch Time  | Sampled Nodes | Sampled Edges | Batch Size  |
|-------------------|---------------|-------------|---------------|---------------|-------------|
| No Backend        | 256 GB        | 7.6s        | 23 898        | 26 653        | 1024        |
| No Backend        | 128 GB        |             | Out-Of-Memory |               | 1024        |
| Embedded          | 128 GB        | 11s         | 31 768        | 32 670        | 1024        |
| Embedded          | 60 GB         |             | Out-Of-Memory |               | 1024        |
| Disk-based        | 256 GB        | 75s         | 24 138        | 48 907        | 1024        |
| <b>Disk-based</b> | <b>128 GB</b> | <b>76s</b>  | <b>24 081</b> | <b>48 899</b> | <b>1024</b> |
| Disk-based        | 12 GB         | 108s        | 23 979        | 48 905        | 1024        |
| <b>Disk-based</b> | <b>8 GB</b>   | <b>126s</b> | <b>23 678</b> | <b>48 909</b> | <b>512</b>  |

cross product of all possible node\_0, node\_1, node\_2 and then applying filtering by edges and labels, which is clearly suboptimal.

The full execution plan also includes additional measurements such Page Cache Hits/Misses, Time (ms) for each operator, which we leave out here. Examining the execution plan allows one to understand the optimality of the query, benchmark detailed runtime behaviour, and potentially find improvements in either the written query or the query engine implementation.

## 5.2 Single machine training

We now study the runtime characteristics of our setup for single-machine training. Our setup should support training extremely large graphs (ogbn-papers100M) on tiny machines with low RAM usage, since at any point in time, the total RAM required is no larger than representing the 2-hop sampled neighbourhood of batch-size number of nodes, plus some model weights. At the same time, as we equip the training machine with more hardware resources, we expect training speed to improve. Since the absolute wall-clock time is only meaningful to the specific hardware, we compare our setup using a Neo4j database with a common training setup using no database and with an embedded graph database [47]. In order to benchmark the runtime trade-off that results from using a DB, we use one CPU only. We train GraphSAGE represented by Equation 2 for node classification.

As shown in Table 2, our training works even when there is only 8GB of RAM available, using a batch size of 512. As we increase RAM usage, the average processing time decreases, or one could equivalently increase batch sizes. Therefore our setup enables a smooth trade-off between hardware

**Table 3:** Distributed training. Partitions represent the number of training processes.

| Backend    | Batch Size | Partitions | Epoch Time    | Sampled Nodes | Sampled Edges |
|------------|------------|------------|---------------|---------------|---------------|
| No Backend | 1024       | 2          | 1061s         | 406 372       | 713 840       |
|            |            | <b>8</b>   | <b>461s</b>   | 410 237       | 716 912       |
| Graph DB   | 512        | <b>2</b>   | <b>1 841s</b> | 9 024         | 46 073        |
|            |            | <b>8</b>   | <b>564s</b>   | 7 916         | 46 061        |
|            |            | 12         | 597s          | 7 403         | 46 065        |
|            |            | 16         | 629s          | 9 037         | 46 051        |
|            | 1024       | <b>2</b>   | <b>2 094s</b> | 7 027         | 92 107        |
|            |            | <b>8</b>   | <b>600s</b>   | 18 215        | 92 042        |
|            |            | 16         | 590s          | 15 563        | 92 108        |
|            | 8192       | 2          | 6 676s        | 104 425       | 737 118       |
|            |            | 8          | 1 702s        | 112 082       | 736 857       |

resources and training time for an arbitrary scale. For example, our approach should enable graph ML workloads for extremely large graphs, such as Graph500-scale34 [48] with more than 17 billion nodes and 274 billion edges, commonly used for HPC graph analytics but have not been achieved within the graph ML community.

We also examine what happens when there is an abundance of memory available that allows common single-machine training. As expected, the no-backend setup is on par with an embedded DB, both being significantly faster than using a separate graph DB. This suggests the IO processing and converting Cypher output into PyTorch tensor (an ETL step, which is a non-zero-copy operation) becomes the new bottleneck.

### 5.3 Distributed training

We implement Figure 5 and study the runtime characteristics of distributed training using a graph DB.

Since we already understand the trade-off between memory usage and processing power, we fix all experiments with 128GB RAM. A basic feature of any database is to support many concurrent reads. While each single query’s execution is single-threaded, we expect a large number of concurrent queries to utilise a large number of threads to saturate available CPUs.

To make sure we are benchmarking only the impact of introducing the database, but not the orthogonal concern of unpredictable model synchronisation and network latency, we use a single machine with 16 CPUs and multiple separate training processes to simulate the distributed setup.

As shown in Table 3, as we increase the number of parallel training processes from 2 to 8, training speed improves linearly by about 4 times. The optimal epoch time is achieved with 8 processes. Increasing the number of training processes no longer improves training speed due to the fact that all CPUs are now saturated.

We additionally check the correctness of our setup by examining the empirical distribution of sampled nodes in Appendix D and verify that it aligns with the degrees of sampled nodes. Our model gives an average of 58% accuracy after 10 epochs without hyperparameter and batch tuning for model weight synchronisation. Industry standard model architecture, hyperparameter and DDP configuration tuning should be used in real-world deployments to reach state-of-the-art scores.

## 6 Conclusion

In this paper, we introduce a new way of training GNNs by using graph DB as an integral component in the training pipeline. We formulate common data access operations in GNN training pipelines as



retrievals in graph DB. Concretely, initial full-graph loading is removed, and only metadata is loaded, in addition to multi-hop neighbourhood sampling with feature retrieval being executed as one query.

Our experiments validate that minimal memory is required to train arbitrarily large graphs. At the same time, there is a smooth scaling curve of training speed with respect to hardware resources. Our setup also seamlessly applies to distributed training, where similar memory advantage and scaling behaviour are observed. The separate pre-processing step of graph partitioning is not required since this is exactly what is offered by database horizontal scaling.

Most importantly, our approach leverages the techniques and tooling from graph DB to the benefit of graph ML and lays a foundation for future developments at the intersection of these two fields.

## 7 Future Work

There is a variety of future work that builds upon our approach, and we outline several promising follow-up directions.

**Engineering optimisations.** Since this is the first work that tightly couples a GNN training process with a graph DB, our setup is focused on verifying the method’s applicability and benchmarking the trade-offs. For real-world deployments, hardware accelerators such as GPUs/TPUs can be leveraged to further speed up the training process in a standard way. The common bottleneck of workload imbalanced between sampling and tensor operations could be addressed with scheduling methods similar to [13, 14]. In single-machine training with an embedded graph DB, memory I/O could be optimised by using shared memory layouts between GNN processing and DB query engine. Database indexes could be created in a way that matches the retrieval patterns of GNN workloads.

**Other sampling methods.** While our query in Figure 4 can be modified to support common node-wise sampling schemes, it is unclear whether more exotic sampling methods[49] can be formulated as queries. For example, there exist many heterogeneous GNNs [50–53], all of which are based on neighbourhood sampling with respect to some metapaths. A metapath is essentially a node-label and edge-type sequence, such as

$$\text{AUTHOR} \xrightarrow{\text{WRITES}} \text{PAPER} \xrightarrow{\text{CITES}} \text{PAPER} \xleftarrow{\text{WRITES}} \text{AUTHOR}$$

LPGs can naturally model any heterogeneous graphs. It is clear that such metapaths can be easily defined as a query, such as

$$\text{MATCH } (n1:\text{AUTHOR}) - [e1:\text{WRITES}] -> (n2:\text{PAPER}) - [e2:\text{CITES}] -> \\ (n3:\text{PAPER}) <- [e3:\text{WRITES}] - (n4:\text{AUTHOR})$$

Moreover, such heterogeneous long multi-hop queries are well supported by OLAP graph databases.

Both the formal study of which sampling methods can be modelled within query languages and the empirical experiments of which method sees the biggest gain from using graph DBs are interesting future directions.

**Further integration with graph DBs.** Our query in Figure 4 returns the multiset in Equation 2. The immediate next step in the training pipeline is to apply mean aggregation, which is also a basic aggregation operator in the query. Therefore, one could attempt to implement more parts of Equation 2, or perhaps all of it, within the database. The natural question is which particular instance of Equation 1 can not be implemented, and in which cases do we observe practical advantages.

**Inference.** Typical inference workflow mimics the forward pass of the training setup. As a result, classifying a single new node requires both recreating the graph and a full forward pass. It seems our setup could be extended to support extremely resource-efficient and fast inference of new data with minimal modification.

## Acknowledgements

The authors thank Michael Benedikt for the valuable comments and discussions on this work.

## References

- [1] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, page 233–244, 2009. doi: 10.1145/1583991.1584053. URL <https://doi.org/10.1145/1583991.1584053>. 1
- [2] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Neural Information Processing Systems*, 2017. URL <https://api.semanticscholar.org/CorpusID:4755450>. 1, 2
- [3] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.
- [4] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Ququan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems*, 32, 2019. 1
- [5] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020. 1, 2
- [6] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415530. URL <https://doi.org/10.14778/3415478.3415530>. 1, 2, 6
- [7] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019. 2
- [8] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. 2
- [9] Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Wai Lok Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, Jonathan Halcrow, Filipe Miguel Gonçalves de Almeida, Pedro Gonnet, Liangze Jiang, Parth Kothari, Silvio Lattanzi, André Linhares, Brandon Mayer, Vahab Mirrokni, John Palowitch, Mihir Paraskar, Jennifer She, Anton Tsitsulin, Kevin Vilella, Lisa Wang, David Wong, and Bryan Perozzi. TF-GNN: graph neural networks in tensorflow. *CoRR*, abs/2207.03522, 2023. URL <http://arxiv.org/abs/2207.03522>. 2
- [10] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998. URL <https://api.semanticscholar.org/CorpusID:3628209>. 2
- [11] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, September 1998. 2
- [12] Kaidi Cao, Rui Deng, Shirley Wu, Edward W Huang, Karthik Subbian, and Jure Leskovec. Communication-free distributed gnn training with vertex cut, 2023. URL <https://arxiv.org/abs/2308.03209>. 2, 5
- [13] Xinchun Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph gnn training for large graphs. *Proc. ACM Manag. Data*, June 2023. 2, 9
- [14] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytgnn: efficient graph neural network training at large scale. *Proc. VLDB Endow.*, page 1228–1242, February 2022. 2, 9
- [15] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases, 2017. URL <https://arxiv.org/abs/1610.06264>. 2

- [16] RDF. <https://www.w3.org/RDF/>, 2014. 2
- [17] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000. ISSN 1573-7659. doi: 10.1023/A:1009953814988. URL <https://doi.org/10.1023/A:1009953814988>. 3
- [18] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020. 3, 4, 6
- [19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018. doi: 10.1145/3183713.3190657. URL <https://doi.org/10.1145/3183713.3190657>. 3
- [20] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language cypher. *CoRR*, abs/1802.09984, 2018. URL <http://arxiv.org/abs/1802.09984>. 3
- [21] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. opencypher: New directions in property graph querying. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 520–523. OpenProceedings.org, 2018. doi: 10.5441/002/EDBT.2018.62. URL <https://doi.org/10.5441/002/edbt.2018.62>. 3
- [22] GQL. <https://www.iso.org/standard/76120.html>, 2024. 3
- [23] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ. *CoRR*, abs/2112.06217, 2021. URL <https://arxiv.org/abs/2112.06217>. 3
- [24] Neo4j. <https://neo4j.com/>. 3, 5
- [25] ArangoDB. <https://arangedb.com/>. 3
- [26] TigerGraph. <https://www.tigergraph.com/>. 3
- [27] SPARQL. <https://www.w3.org/TR/sparql11-query/>, 2013. 3
- [28] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfx: A highly-scalable rdf store. In *International Workshop on the Semantic Web*, 2015. URL <https://api.semanticscholar.org/CorpusID:7608029>. 3
- [29] AWS Neptune. <https://aws.amazon.com/neptune/>. 3
- [30] Ziyi Chen, Xiyang Feng, Guodong Jin, Chang Liu, and Semih Salihoglu. Kùzu: Graph learning applications need a modern graph DBMS. In *The Second Learning on Graphs Conference*, 2023. URL <https://openreview.net/forum?id=Eg3MthXzeT>. 4
- [31] Maciej Besta, Patrick Iff, Florian Scheidl, Kazuki Osawa, Nikoli Dryden, Michal Podstawski, Tiancheng Chen, and Torsten Hoefer. Neural graph databases. In *Learning on Graphs Conference*, pages 31–1. PMLR, 2022. 4
- [32] Petar Ristoski, Jessica Rosati, T. D. Noia, Renato De Leone, and Heiko Paulheim. Rdf2vec: Rdf graph embeddings and their applications. *Semantic Web*, 10:721–752, 2019. URL <https://api.semanticscholar.org/CorpusID:150293718>. 4
- [33] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016. 4
- [34] Hongyu Ren, Mikhail Galkin, Michael Cochez, Zhaocheng Zhu, and Jure Leskovec. Neural graph reasoning: Complex logical query answering meets graph databases. *arXiv preprint arXiv:2303.14617*, 2023. 4

- [35] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. Relational deep learning: Graph representation learning on relational databases. *CoRR*, abs/2312.04615, 2023. doi: 10.48550/ARXIV.2312.04615. URL <https://doi.org/10.48550/arXiv.2312.04615>. 4
- [36] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems. <https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/>, 2024. 4
- [37] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020. 4
- [38] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.
- [39] Costas Mavromatis and George Karypis. Gnn-rag: Graph neural retrieval for large language model reasoning. *arXiv preprint arXiv:2405.20139*, 2024. 4
- [40] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. Exploration of approaches for in-database ml. In *International Conference on Extending Database Technology*, 2023. URL <https://api.semanticscholar.org/CorpusID:253270276>. 4
- [41] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. Learning models over relational data using sparse tensors and functional dependencies. *arXiv preprint arXiv:1703.04780*, 2017.
- [42] Dan Olteanu. The relational data borg is learning. *arXiv preprint arXiv:2008.07864*, 2020. 4
- [43] Pinecone. <https://www.pinecone.io/>. 5
- [44] Maciej Besta, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Nils Blach, Berke Egeli, Georgy Mitenkov, Wojciech Chlapek, Marek Michalewicz, Hubert Niewiadomski, et al. The graph database interface: Scaling online transactional and analytical graph workloads to hundreds of thousands of cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18, 2023. 5
- [45] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.*, jan 2019. 5
- [46] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. G-tran: Making distributed graph transactions fast. *arXiv preprint arXiv:2105.04449*, 2021. 5
- [47] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. Kuzu graph database management system. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. URL <https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf>. 7
- [48] Graph500. <https://graph500.org/>. 8
- [49] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. Sampling methods for efficient training of graph convolutional networks: A survey. *IEEE/CAA Journal of Automatica Sinica*, 9(2):205–234, 2021. 9
- [50] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding. In *Proceedings of the web conference 2020*, pages 2331–2341, 2020. 9
- [51] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The world wide web conference*, pages 2022–2032, 2019.
- [52] Yuxiao Dong, Nitesh V. Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 135–144, 2017. doi: 10.1145/3097983.3098036. URL <https://doi.org/10.1145/3097983.3098036>.

- [53] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 793–803, 2019. doi: 10.1145/3292500.3330961. URL <https://doi.org/10.1145/3292500.3330961>. 9

## A Cypher query for retrieving edge metadata

Figure 6 shows an example Cypher query that returns all relevant edge metadata.

```
MATCH (a)-[r]->(b)
WITH DISTINCT type(r) AS EdgeType, labels(a)[0] AS SourceType, labels(b)[0] AS
TargetType, r
WITH EdgeType, SourceType, TargetType, collect(distinct keys(r)) AS AllKeys,
count(r) AS edge_count
RETURN EdgeType, SourceType, TargetType, reduce(s = [], k IN AllKeys | s + k) AS
UniqueKeys, edge_count
```

**Figure 6:** A Cypher query that returns all relevant metadata for all edges.

## B Single-hop sampling query

Figure 7 shows a Cypher query that performs one-hop neighbourhood sampling. The nodes returned can be used as seed nodes for subsequent sampling on the next hop.

```
MATCH (node_src:PAPER)-[rel:CITES]-> (node_dst:PAPER)
WHERE id(node_src) IN $SEED_NODES
RETURN id(node_dst), rand() as r
ORDER BY r
LIMIT $MAX_NEIGHBOURS
```

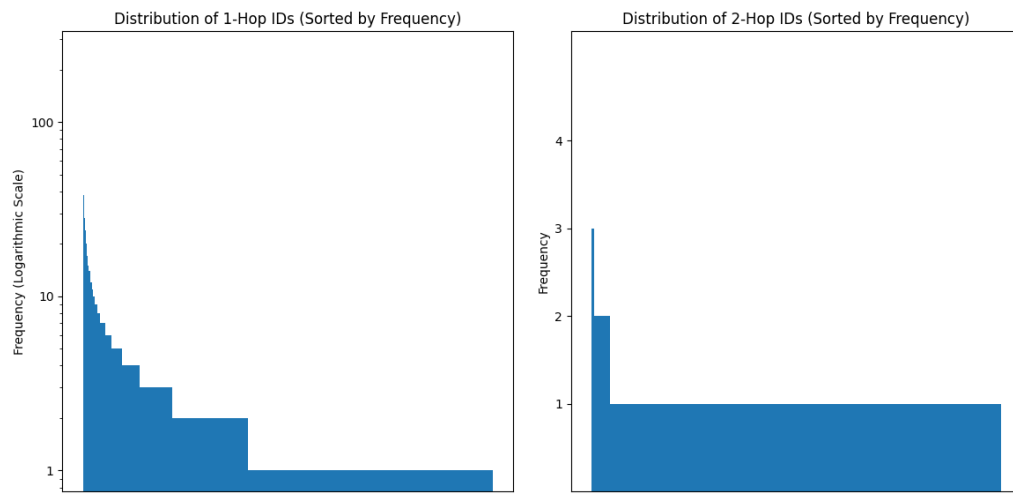
**Figure 7:** A Cypher query that samples one-hop neighbourhood for given seed nodes.

## C Database and training configuration

We use Neo4j Community Edition version 5.21. We configure `dbms.memory.heap.initial_size = 32GB`, `dbms.memory.heap.max_size = 32GB` and `dbms.memory.pagecache.size=32GB`. We use Kuzu version 0.5.2. All other Neo4j and KuzuDB configurations are kept as default. We use PyTorch 2.3.0 and PyG 2.5.0. When sampling two-hops we use  $12 \times 12 = 144$  number of neighbours.

## D Empirical distribution of sampled nodes

Figure 8 shows the plots of sampled nodes. We run the query in Figure 4 100 times on ogbn-products, with `$SEED_NODES` of 100 nodes from id 100000 to 100099. The empirical distribution of frequency of sampled nodes aligns with actual degree distribution, with a few high-degree nodes sampled frequently and the majority of nodes sampled once.



**Figure 8:** Empirical distribution of sampled nodes using query template Figure 4.