IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received December 27, 2021, accepted January 17, 2022, date of publication January 31, 2022, date of current version March 2, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3148126

GDLL: A Scalable and Share Nothing Architecture Based Distributed Graph Neural Networks Framework

DUONG THI THU VAN[1], MUHAMMAD NUMAN KHAN[1], TARIQ HABIB AFRIDI[1], IRFAN ULLAH[1], AFTAB ALAM[1,2], AND YOUNG-KOO LEE[1], (Member, IEEE)
[1]Department of Computer Science and Engineering, Kyung Hee University, Global Campus, Yongin 17104, South Korea
[2]Division of Information and Computing Technology, College of Science and Engineering, Hamad Bin Khalifa University, Qatar Foundation, Doha, Qatar

Corresponding author: Young-Koo Lee (yklee@khu.ac.kr)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government [Ministry of Science and ICT (MSIT)] (Development of a distributed graph Database Management System (DBMS) for intelligent processing of big graphs) under Grant IITP-2022-2021-0-00859.

ABSTRACT Deep learning has recently been shown to be effective in uncovering hidden patterns in non-Euclidean space, where data is represented as graphs with complex object relationships and interdependencies. Because of the implicit data dependence in the big graphs with millions of nodes and billions of edges, it is hard for industrial communities to exploit these methods to address real-world challenges at scale. The skewness property of big graphs, distributed file system performance penalty on small k-hop neighborhood subgraphs, and varying size of subgraph makes Graph Neural Networks (GNNs) training further challenging in a distributed environment using parameter servers. To address such issues, we propose a scalable, layered, fault-tolerance, and in-memory distributed computing-based graph neural network framework called Graph Distributed Learning Library (GDLL). The base layer utilizes an optimized distributed file system and a scalable graph data store to reduce the performance penalty. The second layer provides distributed graph processing using in-memory graph programming models while optimizing and hiding the underlying complexity of information complete subgraph computation. In the third layer, GNN modules are deployed on top of the first two layers for efficient distributed training using parameter servers. Finally, we evaluate and compare GDLL with the state-of-the-art solutions and outperform it significantly in terms of efficiency while maintaining similar GNN convergence.

INDEX TERMS Graph neural networks, GNN, big graph learning, distributed GNN, large scale graph, graph learning.

I. INTRODUCTION

Graph topologies may naturally represent real-world data in a variety of applications. In Euclidean data, deep learning is efficient at finding hidden patterns. But what about applications that use data from non-Euclidean domains and represent it as graphs having high interdependencies and compound object relationships? This is where GNNs, one of the most prominent deep learning frameworks for graph data modeling and analysis, comes in as a substantial solution [1]. This promising technique's core idea is to get the target node's embedded representation by preparing, aggregating, and merging information from its local neighborhoods, which is akin to graph embedding [2]. GNN has been widely utilized in many industrial problems such as recommendation systems [3], social network analysis, and anomaly detection [4] because of its great expressiveness and impressive performance.

The social network graph, for example, contains up to two billion nodes and a trillion edges [5]. By estimating associated features with those nodes and edges, the graph data at such a scale may result in 100 TeraBytes of data. While in GNN, the data are infeasible to store and process on a single computer. To use GNN methods to solve real-world problems by processing industrial-scale graphs, we require to build a graph learning system with scalability, fault tolerance, and integrity

The associate editor coordinating the review of this manuscript and approving it for publication was Senthil Kumar.

of fully-functional GNNs training computations. Nevertheless, the computation graph of GNNs are fundamentally different from conventional learning tasks because of data dependency within a graph. In existing traditional Parameter Server (PS) frameworks [6], [7] assuming data-parallel, each sample's computation is independent of other samples. But for convergence, the GNNs are highly dependent on the information complete independent subgraphs. Furthermore, because real-word graphs follow the Power-Law, the independent subgraph-based solution results in load-balancing issues in a distributed environment. Similarly, for quick data access, we must keep the graph data in memory. Thus we can no longer keep training or inference samples on disk and retrieve them via pipelines [5]. As a result, we are unable to create a learning and inference system for graph learning tasks based on existing PS frameworks that merely maintain model integrity in PS and perform workloads in data-parallel in each worker.

In the recent past, several attempts have been made to develop scalable systems, both in the academia and in industry. The present scalable learning solutions may be divided into two categories: shared memory-based distributed GNN frameworks (DGL [8], PyG, AliGraph [9], FeatGraph [10], and [11]) and shared-nothing architecture-based distributed GNN frameworks (NeuGraph [12] and AGL [5]). The former frameworks have inherited issues (such as limited scalability) because of the design architecture and lacking suitability for some real-world scenarios. For example, PBG fails when the graphs have rich attributes over nodes and edges. PyG and DGL are designed as a standalone powerful-machine systems to deal with industrial-scale graphs. AliGraph implements distributed in-memory graph storage engine, which requires standalone deployment before training a GNN model. In the latter class, NeuGraph [12] is based on the Scatter Apply Gather [13] graph processing model. This approach is suitable for information complete subgraph PS distributed training. Likewise, AGL is a general-purpose learning framework and is based on MapReduce distributed computing architecture. AGL also tries to inference in the distributed environment but it utilizes a disk-based storage approach and does not efficiently use the distributing file system.

Overall, existing solutions require in-memory storage of graph data either in a standalone powerful monster machine that could not manage real-world big graphs or in a customized graph store that could lead to a massive amount of communications between graph stores and workers. This makes them not scale to bigger graph data. Furthermore, state-of-the-art solutions do not well exploit existing shared-nothing distributed architecture, such as MapReduce framework aiming scalability, deployment of GNNs against big graphs, and fault tolerance. Finally, existing frameworks focus more on the training of graph learning models but overlook system integrity and generalizability.

To address such issues and to fill the research gap, in this paper, we present GDLL, a scalable, layered, fault-tolerance, in-memory, and shared-nothing architecture-based framework for distributed GNNs training.

The proposed framework is composed of three layers, i.e., Graph Data Layer (GDL), Graph Optimization Layer (GOL), and Graph Learning Layer (GLL). The GDL is the base layer (section IV-A) and has been designed on top of an optimized distributed file system (for k-hop based subgraphs) called Flock-HDFS for Subgraphs (F-HDFS) and a scalable graph data store to manage large scale raw data, big graphs, and subgraphs. F-HDFS is novel file aggregation approach designed on top of Hadoop Distributed File System (HDFS) [14]. Flock is designed as the I/O unit merging a group of related k-hop based subgraphs aiming to effectively access subgraphs for efficient GNNs training in a distributed environment. The second layer (section IV-B) is called GOL, which provides distributed graph processing on top of an in-memory MapReduce framework (Apache Spark [15]) while optimizing and hiding the underlying complexity of advanced message passing, k-hop-based subgraph computation, and graph sampling techniques. The third layer (section IV-C) is called GLL which deploy GNN modules on top of the first two layers for fast and efficient training while considering PS.

The main contributions of this paper are many fold and can be summarized as:

- We propose a scale-out, share-nothing architecture-based distributed GNN framework.
- We propose the notion of F-HDFS to improve the read-write performance of HDFS in the context of k-hop-based large-scale GNNs training.
- We address issues of distributed k-hop based GNNs training, i.e., varying size subgraph while assuming parameter server.
- In this work, we also develop an open-source echo system and API abstractions for scalable GNNs.
- We implement the proposed GDLL framework and conduct extensive experiments to validate our claims.

II. RELATED WORK

The past few years have also seen notable advancement in GNNs, in solving problems from diverse domains e.g., social network analysis [16], computer vision [17], chemistry [18], medicine [19], health [20], etc. However, deploying GNNs against big graphs is challenging [21]. Efforts have been made to propose a scalable system for GNNs. Existing methods adopted two types of approaches for scalable GNN training, i.e., Shared memory-based distributed GNN (systems that allow multiple processors to share the same memory location), and shared-nothing architecture based distributed GNN frameworks (where each server operates independently and controls its memory and disk resources against a common goal).

In the former case, attempts have been made on a single monster machine by fully utilizing the computing resources i.e., GPU, memory, and CPU to quickly train the GNNs models. This includes PyTorch Geometric (PyG) [22] and

Deep Graph Library (DGL) [8] where both of them are based on the message-passing paradigm. They have been optimized aiming to maximize the utilization of the CPU and GPUs. However, these systems cannot scale and are not applicable on big graphs consisting of considerable features. Similarly, PinSage [23], inspired from GraphSAGE, samples the neighborhood against a node exploiting localized convolutions and deploys MapReduce pipeline for fast inference but it is also based on a standalone machine.

In multiple machine setups, recently, researchers have tends to opt for a distributed design. In this context, Facebook offered PyTorch-BigGraph (PBG) [11], a large-scale graph embedding framework. The PBG can generate node embedding in an unsupervised manner from multi-relation data. Yet, it lacks in dealing with rich attributes over nodes and edges which is largely the case in real-world graph applications. DistDGL [24] is based on DGL, proposed a mini-batch-based scalable GNNs training on a cluster of machines. It adopts METIS [25] to partition and distributes graph data across all machines. Furthermore, to achieve better GNN convergence they followed the synchronous training approach on indigenously designed key-value store servers. However, their graph partition method results in severe imbalance partition and thus needs extreme optimization for load balancing. DistGNN [26] also based on the DGL, proposed a full-batch training strategy on an optimized shared memory implementation. They tend to bypass the communication by employing a family of delay update algorithms and minimum vertex-cut for further communication reduction. AliGraph [9] currently deployed at Alibaba for products recommendation, presented three layers system to support GNN algorithms consisting of the storage layer, sampling layer, and an operator layer. The storage layer provides a caching mechanism for neighbors of important nodes. The sampling layer provides Neighborhood, traverse, and negative sampling methods in a distributed setting. The last operator layer provides optimized aggregate and combines operators. Still, AliGraph cannot scale out, and the same has not been reported in the paper.

Likewise, scale-out systems have been proposed based on shared-nothing architecture, i.e., NeuGraph [12] and AGL [5]. The Neugraph is based on a GPU-based distributed memory system and uses a parallel, multi-GPU standalone system for GNNs training on large graphs. It proposes SAGA-NN (Scatter-ApplyEdge-Gather-ApplyVertex with Neural Networks), a programming model for GNN, and is inspired by vertex-centric parallel graph abstraction GAS [27]. It uses min-cut Metis partitioning for the input graph. Similarly, AGL [5] is a framework designed to process industrial-scale GNNs while utilizing the Hadoop map-reduce framework [28]. Along with this, AGL exploits the default structure of distributed file systems which can be a significant bottleneck. AGL and AliGraph tried to solve the scalability issue, however, they are still far from near-real-time inferences for the large industrial graph.

To summarize, by default, big data challenges are inherited by big graphs. Recent solutions demand in-memory graph data storage, either in a standalone powerful computer that can't manage real-world enormous graph data or in a customized graph store that can lead to huge communication overhead among graph stores and workers. As a result, they are unable to scale to massive graph data. Furthermore, MapReduce-based GNNs solutions do not fully utilize existing shared-nothing distributed architecture. Finally, existing frameworks focus more on graph learning model training while neglecting system integrity and generalizability. Also, they do not address the issues of k-hop based GNN training, i.e., varying size subgraph distributed GNN training. In this research work, we propose GDLL framework aiming to fill this research gap.

III. PRELIMINARIES

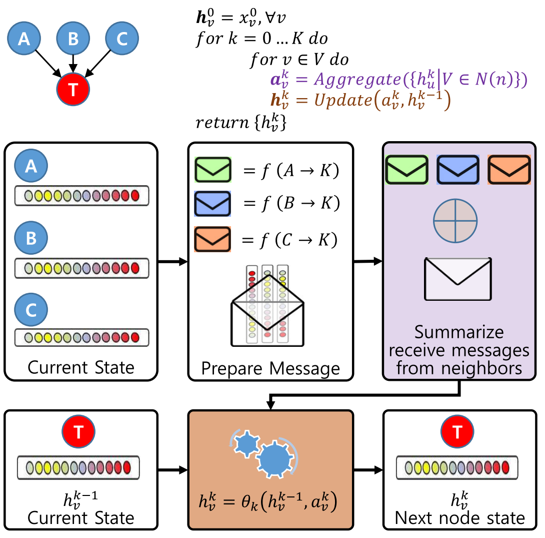
To better understand this research work, in this section, we introduce some notations and the background of GNNs. Then we articulate the foundation of HDFS [14], and Apache Spark [15]. Finally, we introduce the concept of K-hop neighborhood to help realize the data independence and the formation of information complete subgraph required in graph learning tasks.

A. NOTATIONS

We start from a directed, attributed and weighted graph, defined as $G = \{\mathcal{V}, \mathcal{E}, A, X, E\}$. The vertex set and edge set can be defined as $\mathcal{V}$ and $\mathcal{E} \in \mathcal{V} \times \mathcal{V}$, separately. A represents a weighted and sparse adjacency matrix and is defined as a $A \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, where $A_{v,u} > 0$, represents the weighted directed edge from vertex u to vertex v (i.e., $(v, u) \in \mathcal{E}$). Similarly, $A_{v,u} = 0$ represents there is no edge ($i.e.$, $(v, u) \notin \mathcal{E}$). X and E are vertex and edge feature matrices and are defined as $X \in \mathbb{R}^{|\mathcal{V}|\times n}$ and $E \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|\times e}$, where n and e are their corresponding feature sizes. We denote vertex v feature vector as x_v and edge feature vector as $e_{v,u}$ from edge (v, u) if $(v, u) \in \mathcal{E}$. To represent undirected edge, we decomposed a directed edge (v, u) into two directed edges as (v, u), and (u, v) with the same edge feature as $e_{v,u}$. Furthermore, we represent the Diagonal matrix as D, graph embeddings as h, identity matrix as I, Attention weights as a, aggregation function as $AGGR$, and Activation method as σ, correspondingly. Finally, we use $\mathcal{N}_v^+$ to define the set of vertexes directly pointing at v, i.e., $\mathcal{N}_v^+ = \{u : A_{v,u} > 0\}$, $\mathcal{N}_v^-$ to defines the set of vertexes directly pointed by v, i.e., $\mathcal{N}_v^- = \{u : A_{v,u} > 0\}$, and $\mathcal{N}_v = \mathcal{N}_v^+ \cup \mathcal{N}_v^-$. In simple words, $\mathcal{N}_v^+$ defines the set of in-edge neighbors of v, while $\mathcal{N}_v^-$ defines the set of out-edge neighbors of v. We will use the term *in-edges* for the edges pointing at a target vertex v and *out-edges* for the edges pointed by the same vertex v.

B. GNNS AS MESSAGE PASSING

GNNs are the type of deep learning methods that can directly be applied on graphs. Over the years several GNNs have been proposed which we divide into two broader categories i.e. message-passing-based GNNs and GNNs based on Weisfeiler-Lehman (WLGNN) test. In this paper, our focus is



A
B
C
T
$h_v^0 = x_v^0, \forall v$
for $k = 0 \ldots K$ do
for $v \in V$ do
$a_v^k = Aggregate(\{h_u^k | V \in N(n)\})$
$h_v^k = Update(a_v^k, h_v^{k-1})$
return $\{h_v^k\}$
Current State
$= f(A \to K)$
$= f(B \to K)$
$= f(C \to K)$
Prepare Message
Summarize receive messages from neighbors
h_v^{k-1} Current State
$h_v^k = \theta_k(h_v^{k-1}, a_v^k)$
h_v^k Next node state

FIGURE 1. Message passing based GNNs working with pseudocode.

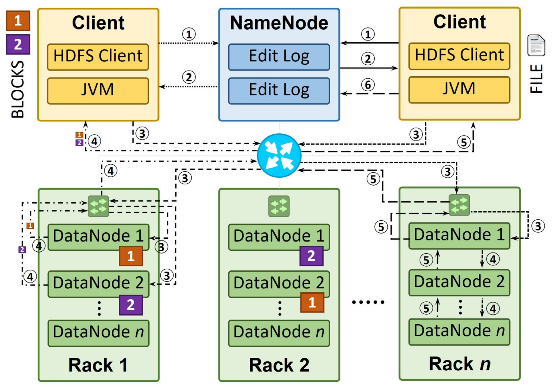


FIGURE 2. HDFS architecture.

on message-passing-based GNNs as WLGNN based GNNs are not scalable and its complexity increases polynomially with respect to graph size. Message-passing-based-GNNs tends to generalize the convolution operation to irregular domains in graph expressed as neighborhood aggregation or also called as message passing scheme as can be seen in Figure 1. The message passing scheme consists of two major steps i.e. aggregate, and update whereas it can be formulated as:

$$z_v^k = AGGR(h_v^{k-1}, v \in \mathcal{N}_v \cup \{v\}) \tag{1}$$

where *AGGR* must be differentiable and permutation invariant function, e.g., mean, sum or max. h_v^{k-1} denotes node v embeddings at $k-1$ layer or feature vector incase of k=1, z_v^k presents neighborhood aggregation at k^{th} layer and $\mathcal{N}_v$ presents the neighbors of node v.

The next step is to update the aggregated neighborhood and propagate such as:

$$h_v^k = \sigma(W^k z_v^k) \tag{2}$$

where W^k is the learnable weight matrix and σ is an activation function e.g ReLU.

In order to execute GNNs on a large graph, GraphSage proposed the notion of sampling before aggregation and update. In this paper, we implemented the Graph Convolution Network (GCN), GraphSage and Graph Attention Networks (GAT) in a similar message passing scheme.

C. HDFS AND APACHE SPARK

HDFS, inspired from Google File System [29], is an open-source distributed file system designed to store large-scale data reliably and to stream those data to the user application at a high speed. HDFS is high fault-tolerant and can scale out up to thousands of machines even on commodity hardware. The main components of HDFS are NameNode, DataNode, and Client, as shown in Figure 2. The NameNode is in charge of managing the file system namespace and the metadata, responding to various requests from clients. DataNodes are used for storage and provide a data replication mechanism to guarantee the availability of a system. When a client needs to write a file to HDFS, an RPC request is made to the NameNode to allow the request, if it does not exist. The files are divided into blocks by the client. For each block, the NameNode allocates several DataNodes according to its replication policy and creates the metadata including the location information in its memory. The client writes a packet to the DataNode. Similarly, for data reading, the client sends a read request to the NameNode and returns the metadata (contains the location information of file blocks) followed by the actual data. Finally, all file blocks are assembled into a complete file in the client.

Likewise, Apache Spark is an in-memory cluster computing platform that generalizes MapReduce model to provide efficient computation. Apache spark immensely outperforms traditional map-reduce procedures on platforms like Hadoop due to its heavy usage of in-memory computing. Resilient Distributed Dataset (RDD) that are stored in-memory and is mainly responsible for efficient memory abstraction mechanism. An RDD in Spark is an unchangeable distributed collection of elements. To perform any operation on Spark, all work is expressed by defining new RDD, transforming existing RDD, or calling methods on RDD. Every RDD are distributed into multiple partitions, which are computed on numerous nodes of the cluster. Furthermore, Spark processes are fault-tolerant, due to this feature large amounts of worker nodes can be killed without affecting the overall status of the job.

D. K-HOP NEIGHBORHOOD

Definition 1 (K-Hop Neighborhood): The K-hop Neighborhood is a subgraph G_v^k computed against a target vertex v, and is represented as the information complete subgraph of original graph G such that their node-set is $V_v^k = \{v \bigcup \{u : d(v,u) \leq k\}\}$, where $d(v,u)$ represents the length of the shortest path from u to v and its length is less than or equal to k. Its edge set from graph G consists of the edges that have both endpoints in its node set i.e., $\mathcal{E}_v^k = \{(u, u') : (u, u') \in \mathcal{E} \wedge u \in \mathcal{V}_v^k \wedge u' \in \mathcal{V}_v^k\}$. Moreover, it contains the feature vectors of the nodes and edges in the computed k-hop neighborhood, such as X_v^k and E_v^k. The 0-hop neighborhood for a target node v is the node v (its feature vector as well) itself.

The generated k-hop neighborhood provides sufficient and necessary information without compromising on data interdependency among nodes in the subgraph (see theorem 1 in [5]). It can similarly be denoted like an original graph such as $G_v^k = \{\mathcal{V}_v^k, \mathcal{E}_v^k, A_v^k, X_v^k, E_v^k\}$ where A_v^k is the k-hop neighborhood's adjacency matrix. Thus, this provides the ability to train the GNN model in workers independently in parallel using parameter server settings without compromising on the effectiveness of the GNNs models.

IV. PROPOSED GDLL FRAMEWORK

To tackle the complex problems in highly interdependent graph structure for scalable and efficient graph learning framework, we proposed the GDLL framework as can be seen in Figure 3. In this section, we present an abstract overview of the proposed framework whereas the technical details are elaborated in the following subsections.

The effectiveness of our proposed framework lies in the layer design as we proposed a three-layer framework, i.e., GDL, GOL, and GLL. It encourages progress in each layer separately to achieve scalable graph learning when integrated. To better comprehend the proposed GDLL framework, we have elaborated simple steps to perform scalable GNNs training, as can be seen in Figure 3. For handling large-scale graphs effectively during the GNNs training process, we first develop GDL layer, which provides methods to load and store the large raw-graph on top of HDFS, big-graphs into an indexed efficient Graph DS. Further, this layer is equipped with F-HDFS, which keeps the related subgraphs into the same blocks for fast loading during GNNs training.

The GOL layer is designed on top of an in-memory distributed computing engine, i.e., Apache Spark. This layer provides the functionality of the k-hop neighborhood, an information complete subgraph along with graph re-indexing and sampling strategies while hiding the complexity of distributed computing engines.

Finally, the GLL layer is responsible for PS-based GNNs training in distributed environment. The beauty of our proposed approach is that all variants of message passing based GNNs can be easily integrated (on top of GDL, and GOL) with our proposed shared-nothing framework.

A. GRAPH DATA LAYER

Graph Data Layer is the base layer of the framework and deals with big graphs' efficient persistence and retrieval. It consists of two main modules, i.e., Graph Access Controller, and Graph Data Persistence. The Graph Access Controller allows the upper layers to establish a connection with the Graph Data Persistence and store and persist the data in the Graph Data Persistence Store. This component is responsible for all the communication between the Graph Data Persistence and the subsequent upper layers. The Graph Data Persistence provides three kinds of data stores, i.e., Unstructured Data Store, Graph Data Store, and F-HDFS. Unstructured Data Store manages the storage and retrieval of raw graph data and GNN models on top of HDFS. Graph Data Store facilitates live on-demand graph data storage and retrieval. F-HDFS is an optimized HDFS for subgraph based GNN.

Real-world graph data is in a raw form stored in the RAW DS. In order to process this data, it needs to be mapped to the graph representation. This task is carried out by the Raw Data to Graph Mapper (R2G Mapper). The R2G Mapper transforms the raw graph data into graph format for distributed processing as shown in Algorithm 1. Algorithm 1 reads raw graph data from the RAW DS. The nodes data consists of node ids and its attributes, and edge data consists of source and destination node id and edge attributes. It then constructs a node object from each of the nodes' records in nodes data. The nodes' objects are put in the nodes list in the graph. Then the source and destination node IDs and edge attributes are obtained from the edge data, an edge object is created, and put in the edge list in the graph object. After loading, the graph is represented as an object in memory consisting of nodes with properties and edges with properties. This representation is handed over to the GOL to compute the information k-hop subgroups.

Algorithm 1: Graph Mapper

```
Input  : Node Data, Edge Data
Output: Graph Object
1  nodesData ← read(NodeData)
2  edgeData ← read(EdgeData)
3  graph ← empty
4  foreach Node n in nodesData do
5      id ← getNodeID(n)
6      node ← newNode(id)
7      attributes ← getNodeAttributes(n)
8      foreach Attribute a in attributes do
9          node.setAttribute(a)
10     end
11     graph.nodes.add(node)
12 end
13 foreach Node e in nodesData do
14     src ← getSource(e)
15     dst ← getDestination(e)
16     edge ← newEdge(src, dst)
17     attributes ← getEdgeAttributes(e)
18     foreach Attribute a in attributes do
19         edge.setAttribute(a)
20     end
21     graph.edges.add(edge)
22 end
```

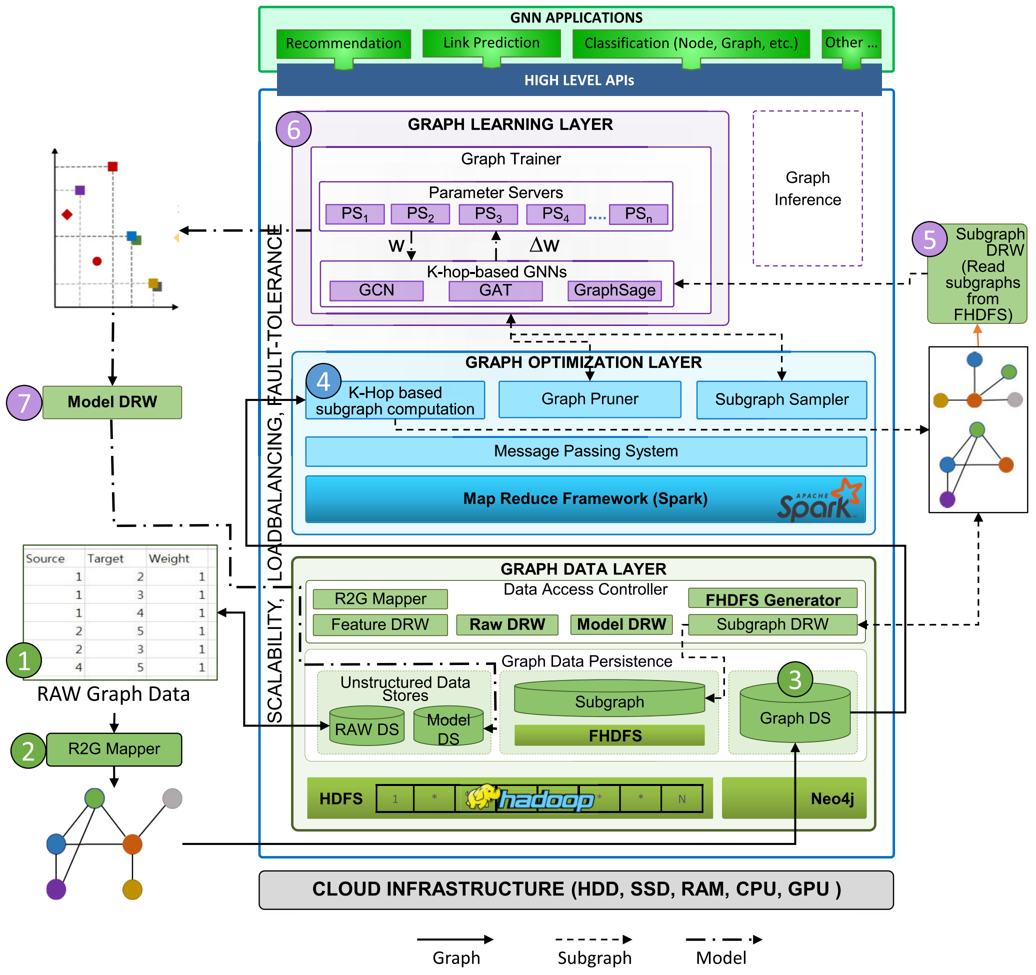


FIGURE 3. **The proposed GDLL framework has three layers. The first layer, GDL, is responsible for graph data management throughout the training lifecycle. In step-1, the raw data are persisted into the RAW DS, then the R2G mapper (Step-2) to map it into graph format for storing in Graph DS (Step-3). The GOL then load the Graph in distributed in-memory (Spark RDD) aiming to compute the information complete subgraph (Step-4). These subgraphs are then indexed in the F-HDFS (Step-5). These subgraphs are then retrieved by the GLL from F-HDFS for distributed training while utilizing parameter servers (Step-6). Finally, the trained model is persisted in the Model DS for inferences (Step-7).**

1) F-HDFS

The produced k-hop subgraph are large in number and need to be indexed so that to speed up the training process. The design principle behind HDFS is to work with large-size files which makes it suitable for Big-Data platforms mostly on commodity hardware. However, the k-hop subgraphs are mainly composed of small files in huge numbers. A massive number of small files causes two problems. First, for a large number of small files (as is the case with subgraphs), the NameNode gets overburdened and the DataNodes fail. Second, subgraph-based GNNs training demands rapid read access to the subgraph stored in files which consequently results in an exceeding amount of reading requests and hopping between DataNodes. This severely damages the I/O performance as shown in Figure 4. The throughput is very low for small files. Therefore, for scalable GNN distributed computing platforms, HDFS becomes a bottleneck. To solve this issue, we optimize the HDFS for distributed GNNs.

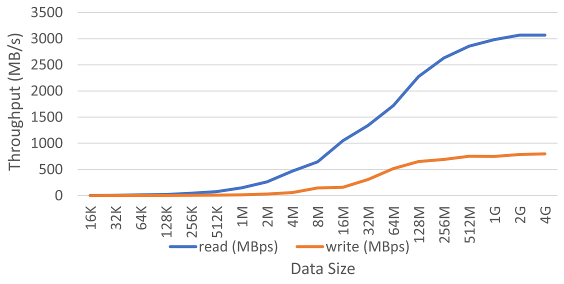


FIGURE 4. **HDFS throughput.**

Motivated by this issue, We designed F-HDFS as a scheme for optimized subgraph loading operations for the GNN training. The main notion is to systematically merge the generated subgraphs into data units called Flocks, which are

Algorithm 2: Flock Storage

```
Input  : Flock, Path
Output: Flock Persistance
1  $irecord.put("id", flock.id)$
2  $irecord.put("subgraphs", flock.subgraphs)$
3  $irecord.put("partial", flock.partial)$
4  $irecord.put("path", Path)$
5  $vertices \leftarrow \emptyset$
6  foreach $record \in flock.Index$ do
7      $vertices.put(record.id)$
8  end
9  $irecord.put("vertices", vertices)$
10 CreateIndexEntry($irecord$)
11 StoreInHDFS($flock, path$)
```

Algorithm 3: Flock Retrieval

```
Input  : VertexID
Output: Flock
1  $flock \rightarrow \emptyset$
2  $irecord \leftarrow Lookup(vID)$
3  if $irecord \neq \emptyset$ then
4      $path \leftarrow irecord.path$
5      $flock \leftarrow ReadFromHDFS(path)$
6      if $flock.partial \neq \emptyset$ then
7          while $flock.partial = \emptyset$ do
8              $path \leftarrow Find(flock.partial)$
9              $chunk \leftarrow ReadFromHDFS(path)$
10             Merge($flock, chunk$)
11         end
12     end
13 end
14 Return $flock$
```

then stored on the HDFS. We designed Flock format keeping the requirements of k-hop subgraphs in such a way that the required subgraphs packed in a Flock are loaded in bulk for the training operation which not only reduces the I/O cost but also considerably reduces the network overhead and as a result, speeds up the distributed GNNs training. The size of Flock is fixed to HDFS Block Size which we determined from the HDFS throughput shown in Figure 4. There is a significant improvement beyond 128MB, which is the HDFS Block Size in our case, and that is why we have fixed the Flock Size to 128MB.

The storage operation of Flocks is outlined in Algorithm 2. An index record is generated for each Flock consisting of Flock id, the number of subgraphs, the HDFS path where the Flock is to be stored, and the subgraph vertices that the Flock contains. This index record is indexed and is used for the retrieval operation. Algorithm 3 shows the Flock retrieval operation. First, the index record of the Flock containing the required vertex is retrieved. Then the corresponding HDFS location of the Flock is obtained from the index record, and the actual Flock is read. Finally, the Flock is unpacked, and the subgraphs are retrieved.

B. GRAPH OPTIMIZATION LAYER

Once large graph datasets are acquired and persisted to the GDL. Then to train GNNs in a distributed environment using PS, we need independent subgraphs having complete information. To accomplish this, we need to compute k-hop based independent subgraphs in the MapReduce environment while utilizing Apache Spark. For managing the k-hop based subgraph, we need a subgraph indexer to index the subgraph and to hold the information regarding subgraphs, i.e., access id, subgraph size, number of nodes, and edges. Further, as the real-world graphs are skewed thus the k-hop subgraph size might be large and becomes challenging to be processed on a single machine in a MapReduce cluster. Therefore, sampling strategies are required. Likewise, generated k-hops against targeted nodes contain overlapping nodes. This puts extra computation pressure during GNN training. To handle this problem, we develop a graph pruner to discard nodes from similar overlapping k-hop subgraphs for faster training. To address these issues, we design GOL on top of MapReduce Framework [28]. The GOL is composed of four components, i.e., k-hop subgraph generator, subgraph indexer, subgraph sampler, and subgraph pruner.

Another aim of this layer is to provide Application Programming Interfaces (APIs) abstraction on top of the Spark MapReduce framework for graph operations while optimizing and hiding the underlying complexity. The overall performance of the GNN training and inferences is proportional to the design of the GOL. The GOL main components are further elaborated in the following subsections.

1) K-HOP-BASED SUBGRAPH COMPUTATION

We observed that the $(k-1)$-hop neighborhood of node u is k neighborhood of v if existing an edge from u to v. Therefore, to provide a k-hop neighborhood for one node v we use $(k-1)$-hop neighborhood of the outgoing neighborhood nodes of v integrate with 1-hop of v.

We build two main functions that are in-edge merging and out-edge propagation. Where in-edge merging function is simply to merge in-edges into one set of in-edges. Out-edge propagation is to provide in-edge at $k+1$ for neighbors along out-edges [5].

To implement the k-hop generator using Spark MapReduce. First, we define (*key*, *value*) pairs, where node identity is key and node information is value. The node information consists of its k-hop, in-edge list, out-edge list. The reducer will provide k-hop by executing k-times. At k time, the reducer merges (k-1)-hop to in-edge at that time to provide k-hop. The k-hop generation process is described as in Figure 5. Given a graph dataset, and k. The map and reduce will be repeated k times. For the first round, we use MapToPair to provide (key, value) pair, then call ReduceByKey to merge in-edges of each node. To provide the input for the next round, we use the out-edge propagation function to

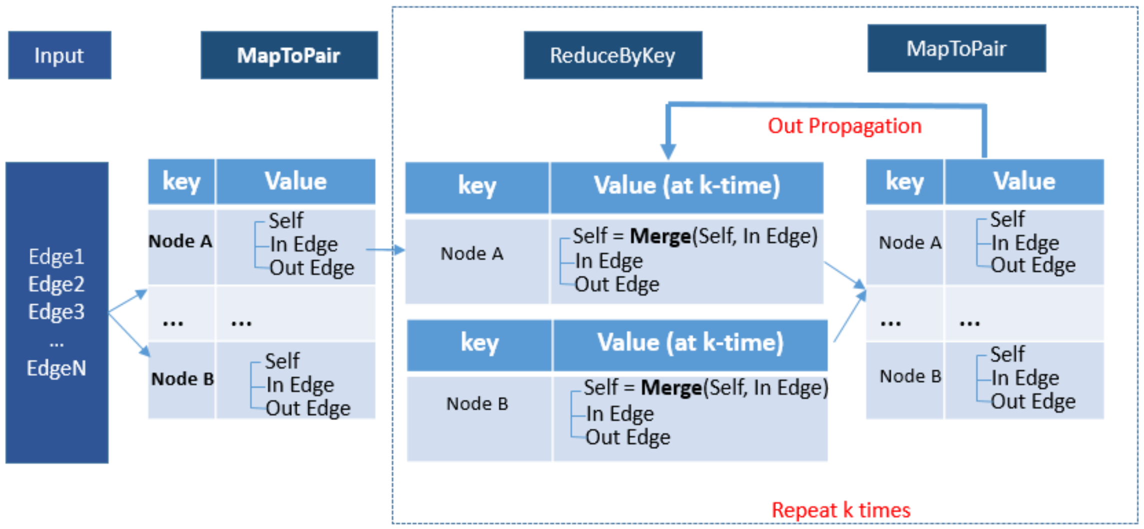


FIGURE 5. Generating k-hop neighborhood in distributed pipeline using spark.

provide in-edge at (k+1) for neighbors belongs to out-edge. The details of k-hop based subgraph generator are shown in Algorithm 4.

```
Algorithm 4: K-Hop Generator
   Input : input, k, output
 1 $lines \leftarrow read(input)$
 2 $node\_infos \leftarrow lines.map(l->get())$
 3 $node\_infor\_pairRDD \leftarrow node\_infos.mapTopair(n->$
     $create(nodeId, node\_info))$
 4 $results \leftarrow$
 5 $i \leftarrow 0$
 6 while $i \leq k$ do
 7     $i \leftarrow i+1$
 8     $results \leftarrow node\_infor\_pairRDD$
 9     $.reduceByKey((i,j)->inEdge_Merging(i,j))$
10     $node\_infor\_pairRDD \leftarrow result.FlatMap(s>$
         $outPropagation(s))$
11 end
12 $writeResult(output)$
```

2) SUBGRAPH SAMPLER AND PRUNER

Once the k-hop subgraph is too large and difficult to be executed on a single machine of the cluster then we use the subgraph sampler. Instead of analyzing the whole network, we can sample a small subgraph similar to the original graph. We use subgraph sampling to reduce the scale of the K-hop neighborhoods, especially for those "hub" nodes. In algorithm 4, subgraph sampling can be called inside inEdge-Merging() function if the number of "hub" of the target node is big. Several sampling techniques can be used such as random walk (select neighbors uniformly and randomly), bias random walk (select neighbors by graph properties like degree). Figure 6 shows the flowchart for sampling to return neighborhoods for a hub node *i*.

Graph pruning is to reduce the unnecessary computations in aggregating steps of the k-th layer while training. The graph pruning strategy is to reduce the non-zero values in the adjacency matrix of each layer. We include graph pruning as an optional function for the training phase.

C. GRAPH LEARNING LAYER

This layer aims to provide popular GNNs models implementation, facilitating distributed GNN training on a graph, and integration to the previous two-layer APIs for efficient training of GNNs. The GLL consist of two main components, i.e., K-hop-based GNNs Trainer, and PS.

1) K-HOP-BASED GNNS TRAINER

In this section, our focus is on expediting the large-scale efficient GNNs training. Large-scale GNNs training is a complicated problem due to the high inter-dependency of graph data that makes traditional distributed training challenging contrary to independent batch training in images and text. There are specifically two strategies to train a GNN model. One is whole-graph training [30], [31] and the other is sample-based training [9], [32]. The latter is generally adopted for large-scale graph learning and is further divided into a full batch training [26], and mini-batch training [24]. Yet, they have one disadvantage i.e., they fall short on inter-dependency within the nodes in samples thus compromising on the effectiveness of GNNs models. To compensate for this, researchers proposed the prior computation of k-hop neighborhoods based GNN training to overcome the issue of inter-dependency among nodes in the graph [5]. The K-hop neighborhood provides the complete information of the graph and also facilitates the sample-based traditional distributed training of GNNs using PS. Whereas, it induced new

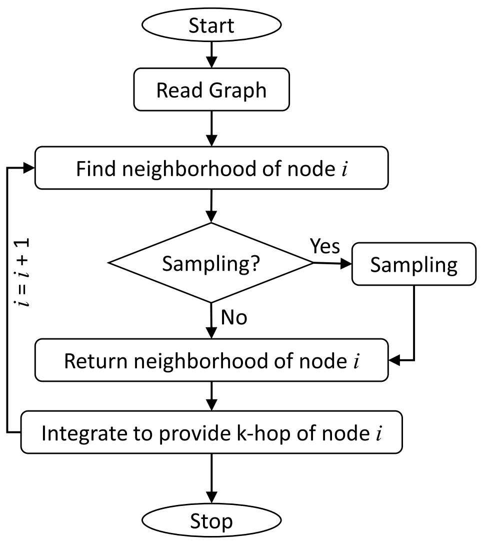

Start
Read Graph
Find neighborhood of node i
Sampling?
Yes
Sampling
No
Return neighborhood of node i
Integrate to provide k-hop of node i
$i = i + 1$
Stop

FIGURE 6. Flow chart of k-hop generation using subgraph sampling.

challenges such as varying size k-hops for different nodes, the issue of "hub" node (i.e., one node may get a large chunk of k-hop neighborhood relative to the original complete graph), and overlapping subgraphs among the computed k-hop neighborhoods.

The varying size of k-hop neighborhoods induces inconsistency in the PS-based distributed training, e.g., workers with fewer nodes in k-hop will have to wait for other workers to complete the computation. To overcome this issue, we ordered the k-hops on the number of nodes present in those k-hops so that all workers at a certain epoch get the identical length of k-hops. The 'hub' node issue is resolved through sampling from the GOL where larger k-hops are sampled, based on a certain threshold (i.e., the maximum number of nodes allowed in k-hop neighborhood subgraph), set according to the main memory of the systems. Finally, on overlapping subgraphs among the k-hops, we devise a setting in the training flow where we pick similar adjacent k-hops in the same batch for training with the help of our proposed F-HDFS.

The training of message-passing based GNNs on k-hop neighborhoods in the distributed environment has its challenges. To make it fully parallel and distributed, we prior computed the aggregations on node's direct neighbors and used them during model computation in GNNs training avoiding matrix multiplication method of message aggregation. However, it overloads the memory with slight improvements in training efficiency. Thus, we moved on from this way of implementation to the standard way of GNNs model computation. Yet, this way of implementation brings a new challenge in the k-hop subgraph vectorization because they are loaded extensively during the training phase. To overcome this, we made the two processes parallel i.e., the model computation and subgraph vectorization during the training phase. The batch size is now being kept small to accommodate both of these processes in memory to avoid the out-of-memory. Also, the F-HDFS storing and indexing of identical subgraphs makes the subgraph loading and vectorization more efficient. Additional improvements come from Apache Spark-based implementation which is several times faster than Hadoop MapReduce.

Our implementation is based on the message-passing paradigm where we divided the GNNs into three parts; 1-Sampling, which in our case is a k-hop neighborhood, and neighbor sampling incase of hub node, 2-Aggregation which aggregates the features of all the incoming neighbor nodes from the k-hop neighborhood, and 3-Combine- e.g., a fully connected neural network operation that obtains the updated feature of node v at k^{th} layer of GNNs, The GCN inspired from eq 1, where neighborhood aggregation is performed here by the below eq 3

$$h_v^k = \sigma\left(\hat{A}XW^k\right) \tag{3}$$

where $\hat{A}$ is obtained using eq 4 and self-loop is added using eq 5. X is the vertex feature matrix, and W is the learnable parameters. σ is any activation function such as Relu, Elu etc.

$$\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2} \tag{4}$$

here $\tilde{A}$ is an Adjacency matrix obtained by adding vertex self loop to it using eq 5 and $\tilde{D}^{-1/2}$ is a inverse diagonal matrix of $\tilde{A}$.

$$\tilde{A} = A + I_N \tag{5}$$

GNNs are a special type of deep learning methods designed especially for graphs. The most prevalent methods are GCN [30], GraphSage [32], and GAT [33]. The work on GNNs dates back to the first decade of the 2000s, but the breakthrough came after the proposal of GCN in 2017 [30]. We will discuss these methods and their integration with the information complete k-hop subgraphs in the coming section.

a: GRAPH CONVOLUTION NETWORK

The GCN introduced a semi-supervised deep learning approach that can be applied directly to the graph. The general idea of GCN is to aggregate information of a processing node and its direct neighbors and then combine them to generate new embeddings. The first step is to add the self-information by adding an identity matrix I as can be seen in eq 5. Next, for direct neighborhood aggregation, it multiplies the inverse diagonal matrix with obtained self-looped identity matrix as can be seen in eq 4. Finally, it injects the relevant direct neighborhood aggregated feature information into an MLP layer as can be seen in eq 3. The major limitation of GCN is its transductive property such as it was training on the complete graph which is in practical incapable to process large graphs at once in memory. Thus we trained it on k-hop subgraphs iteratively.

b: GRAPHSAGE

The GraphSage [32] proposed a neighbor sampling approach that enables large-scale graph training. For inductive learning, it provides three different aggregation functions as a convolution operation. The three aggregation functions are average aggregation, max-pooling, and LSTM where the average aggregation is a non-parametric aggregator, and the others two are learnable and parametric aggregators. The GraphSage due to its aggregation choices makes it adaptable for a graph that grows over time, unlike GCN. However, both of these methods are isotropic that is they have an equal contribution from their direct neighbors. But in reality, it is the opposite and neighbors may have an unequal impact on a target node, thus GAT [33], introduces an anisotropic method(unequal weightage) for neighbor's impact.

c: GRAPH ATTENTION NETWORKS

GAT introduces an attention strategy similar to the multi-head self-attention from transformer [34] where it gives more weight to the important nodes, unlike GCN and GraphSage treats all the neighboring nodes equally important. Thus, they induced a new learning parameter for learning neighbor impact in terms of how much weightage needs to be given to a certain neighbor. Though it gives better performance in terms of accuracy but computationally it is more expensive than GCN and GraphSage.

To formulate these methods, we start from GCN, where a graph convolution operation constructs the normalized sum of the node features of neighbors as can be seen in eq 6:

$$h^{k+1} = \sigma\left(\sum_{u\in N(v)} \frac{1}{c_{vu}} W^{(k)} h_u^{(k)}\right) \tag{6}$$

Here $N(v)$ is the set of its direct neighbors of node $\mathcal{V}_v$ along with a self-loop to add information from node itself, $c_{vu} = \sqrt{|N(v)|}\sqrt{|N(u)|}$ is a normalization constant based on graph structure, σ is an activation function (e,g, ReLU), and $W^{(k)}$ is a shared weight matrix for node-wise feature transformation. The GraphSage model applies identical update rule except that it set $c_{uv} = |N(u)|$.

As, GAT replaces normalized convolution operation with attention mechanism and computes the node embedding h_v^{k+1} of layer $l+1$ from layer k embeddings such as:

$$z_v^k = W^{(k)} h_v^k \tag{7}$$

where, z_v^k is a linear transformation of the lower layer embedding h_v^k and $W^{(k)}$ is its learnable weight matrix

$$e_{vu}^k = LeakyReLU(\vec{\alpha}^{(k)^T}\left(z_v^{(k)} \parallel z_u^{(k)}\right)), \tag{8}$$

Here e_{vu}^k is an additive attention score between two neighbors. where $\parallel$ denotes the concatenation of two nodes i.e. z embeddings. it then take a dot product concatenated z embeddings with $\vec{\alpha}^{(k)^T}$, where $\vec{\alpha}^{(k)^T}$ is a learnable weight vector. Finally, it applies LeakyReLU.

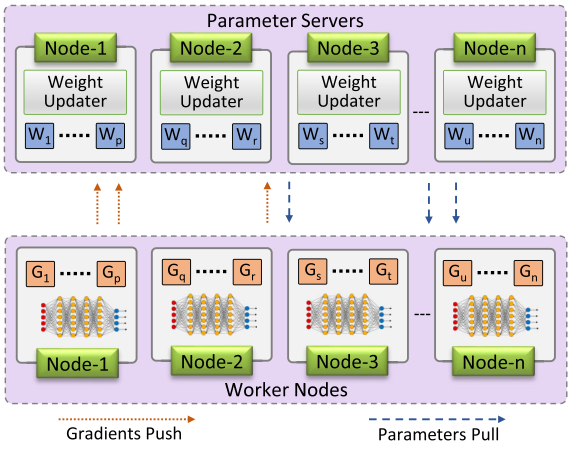


FIGURE 7. Parameter servers and data workers.

$$\alpha_{vu}^k = \frac{exp\left(e_{vu}^{(k)}\right)}{\sum_{m\in N(v)} exp\left(e_{vm}^{(k)}\right)} \tag{9}$$

Equation 9 employs a softmax to normalize the attention scores on each node's incoming edges.

$$h_v^{(k+1)} = \sigma\left(\sum_{u\in N(v)} \alpha_{vu}^{(k)} z_u^{(k)}\right), \tag{10}$$

Finally, equation 10 is similar to the GCN model, where embeddings from neighbors are aggregated collectively and scaled by the attention scores.

2) PARAMETER SERVER

The PS is the principal component of distributed machine learning applications because a single machine alone cannot cope with large-scale data. It is inevitable and essential when the data is independently distributed across workers as it maintains the globally shared parameters of the GNN model during distributed GNNs training. However, the PS assumes the data is independent and distributed among workers. But graph data due to its interdependency cannot be directly partitioned among the workers for which information sufficient subgraphs (i.e., k-hop neighborhood) must be computed earlier and then distributed among the workers [5].

In GLL, we employ multiple PS machines [35], because GNNs parameters are too numerous for a single parameter. Hence, we equally partition the parameters among multiple PS as shown in Figure 7, where each PS is only liable for holding the parameters in its partition. Accordingly, the gradient vector will be partitioned and sent as chunks to its affiliated parameter server by workers before sending a gradient. Likewise, workers will receive the related chunks from the updated model from PS. This setting is suitable for GNNs training on a large graph because the single PS tends to get overloaded.

TABLE 1. Graph data layer APIs.

API	Description
gdl.Graph.read(path)	Reads a Graph raw data from the hdfs
gdl.Graph.save(path)	Saves the graph object to the local or distributed storage
gdl.Graph.readFromDS(name, ds)	Reads a Graph with the given name in graph data store
gdl.Graph.saveToDS(name, ds)	Saves the given Graph in graph in the graph data store with the provided name
gdl.data.Mapper.convert(NodesData, EdgeList)	Convert Graph Data to Graph Object
gdl.data.Hdfs.read(HDFS_Path)	Reads Raw data or models on the hdfs
gdl.data.Hdfs.write(HDFS_Path)	Writes Raw data or models on the hdfs
gdl.Fhdfs.generateFlock(subgraph)	Optimize HDFS block according to k-hop based subgraphs
gdl.Fhdfs.Index.index(Flock)	Create index record for the Flock
gdl.Fhdfs.save(Flock,HDFSPath)	Persist the k-hop subgraphs Flock to the HDFS
gdl.Fhdfs.Index.lookup(vertexid)	Get The path of the F-HDFS from the index.
gdl.Fhdfs.getSubgraphs()	Get all the subgraphs from the Flock
gdl.Fhdfs.getSubgraph(Flock,vertexid)	Get specific vertex subgraph from the Flock

TABLE 2. Graph optimization layer APIs.

API	Description
gol.Khop.khopGenerator(inputPath,k outputPath)	Given an inputPath of a graph, a number k, and outputPath. This function generates k-hop for all nodes of the input graph, and saves the results at outputPath.
gol.EdgePartitioning.edgePartitioner(inputPath, k, outputPath)	Given an inputPath of a graph, a number k, and outputPath. This function provides k parts of edges of the input path such that edges that have the same destination nodes will be located in the same partition.
gol.GraphSampler.graphSampler(graph, strategy function object)	Given a graph, and strategy function name, tt returns smaller data from original data.

Under GDLL, the defined GNN model is distributed among workers and servers to be used in training. It involves stochastic gradient descent (SGD) computation for training distributed models. During each iteration, each worker performs model computation, and model parameters are sent to the PS server. These parameters are accumulated for the next iteration in the server and it maintains the current version of the GNN model parameters. As k-hop contains the necessary information to train the GNN model, the workers become independent of one another. This makes the GNN training on a graph similar to the conventional machine learning model training. Moreover, most of the k-hops are small subgraphs (the larger k-hop such as against a 'hub' node are also sampled) that enable us to deploy simple personal computers as workers in GNN training. We also provided helper functions for obtaining data, including getter/setter methods for gradients and weights.

D. GDLL LIBRARY AND SCENARIOS

In this section, we provide descriptions of the GDLL library. We show APIs for every layer in the GDLL (i.e., GDL, GOL, and GLL) as can be seen in Table 1, 2, 3. The flow and interaction among GDLL's layers are elaborated in the sequence diagram, shown in Figure 8, for better understanding.

Numerous GNNs-based domain-specific applications can be developed while utilizing the proposed GDLL framework. To understand the applications of the proposed system, we present some real-world domain-specific scenarios. Overall, all graph applications can mainly be divided into three tasks, i.e., node classification, link classification, and graph classification. Each of these tasks has countless applications. e.g., node classification in retail applications where products and customers can be viewed as nodes. Therefore recommending products to a customer can be done using node

TABLE 3. Graph learning layer APIs.

API	Description
gll.GML.gcnConv(in_feats, out_feats, bias=true, activation)	It will apply graph convolution over an input signal. Graph convolution is introduced in GCN.
gll.GML.gatConv(in_feats, out_feats, num_heads, bias=true, activation=None)	It will apply Graph Attention Network over an input signal.
gll.GML.sageConv(in_feats, out_feats, aggregator_method, bias=True, activation=None)	function for GraphSAGE layer from paper "Inductive Representation Learning on Large Graphs"
gll.GML.sumPooling(graph (features)	It applies sum pooling over the features that is summing the neighborhood features and reducing the size
gll.GML.meanPooling(features)	It applies mean pooling over the features that is averaging the neighborhood features and reducing the size
gll.GML.maxPooling(features)	It applies max pooling over the features that is choosing max values among the neighborhood features and reducing the size
gll.GML.Sequential(*args)	It will give support for sequential containers for stacking graph neural network modules.
gll.GML.neighborSampling(graph)	This generates neighbor sample
gll.GML.accuracy(pred_labels, orig_labels)	This method computes accuracy between two set of labels,
gll.GML.oneHotEncode(object)	This method computes one hot encoding to an array (both for labels and features)

TABLE 4. Parameter server APIs.

API	Description
gll.PS.constructor(lr,dropout)	It declares a model with a dropout rate of 'dropout'. It also declares an optimizer with a learning rate 'lr'.
gll.PS.applyGradients(gradients)	It, first, accumulates the gradients and then applies the accumulated gradients to compute new weight tensors for a model. These new weight tensors are sent to the data workers.
gll.PS.getWeights()	It gives the latest weight tensors of the current updated model. The data workers used them to set weights of their models.
gll.PS.constructor(lr,dropout)	It declares a model with a dropout rate of 'dropout'. It also declares an optimizer with a learning rate 'lr'.
gll.PS.computeGradients(weights)	First it sets the model with the given weights then It calculates the gradients of the model. These are the gradients that are sent to the parameter server.
gll.PS.getGradients()	It gives the gradients of the current model.
gll.PS.setWeights(weights)	It sets the current model with the given weights.
gll.PS.setGradients(gradients)	It sets the model with the given gradients.
gll.PS.getWeights()	It gives the weights of the current model.

classification. However, GNNs has its challenges especially deploying on a large-scale graph. Thus, the proposed system could be utilized for GNNs-based applications by industries ranging from small to mid and even large having resource constraints.

V. GDLL RESULTS AND EVALUATION

In this section, first, we present the experimental environment, datasets being used, the performance of the proposed GDLL. Finally, the scalability and the performance of the proposed system in terms of accuracy and speed have been

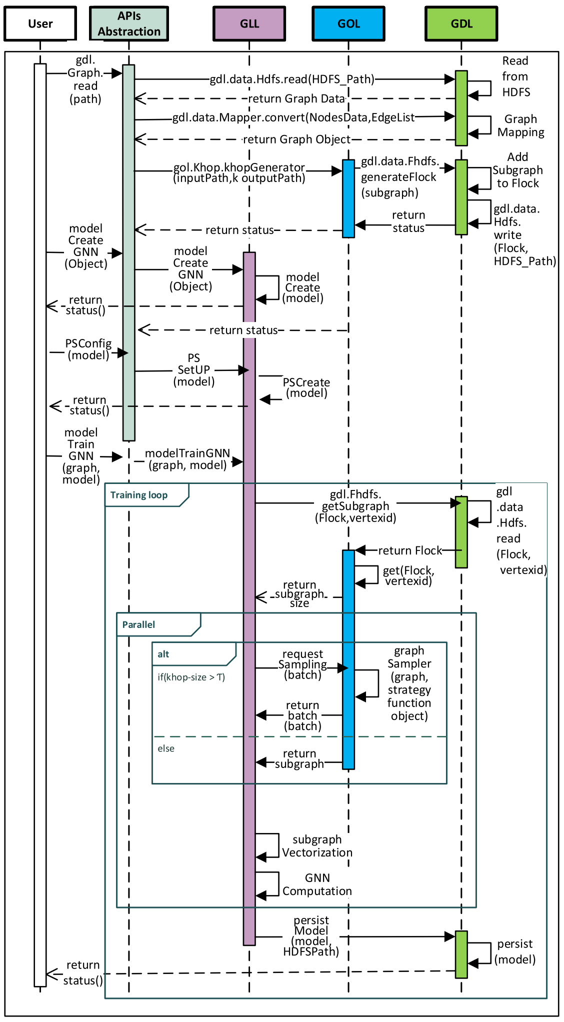


FIGURE 8. GDLL sequence diagram.

evaluated. Further, the time cost per iteration is improved with comparable accuracy with the state-of-the-art.

A. EXPERIMENTAL SETUP

We set up an indoor cluster of 11 machines running Hortonworks Data Platform (HDP) [36] as shown in Figure 9. Each node is running Ubuntu 18 with a Core i5 processor with 4 cores and 32 GB Ram and 1TB disk. The nodes are connected with high-speed switch. The cluster consists of five types of nodes, i.e., one server node, one NameNode, one Graph DB Server, four worker nodes, and four Parameter Servers. The Server node runs Ambari Server, F-HDFS Index, and F-HDFS Generator instances. F-HDFS Manager and Spark Server are running on NameNode. The worker nodes run Spark client, and the Parameter Server instances run on the Parameter Servers. GDLL instances are deployed on the respective nodes. In Figure 9, Clients, and Data Node services are configured on Worker Nodes. Data Node is the HDFS node whereas clients are the instances of Spark, Yarn [37], and Zookeeper [38]. Further, we set the value of different parameters of the GDLL, as shown in Table 5.

TABLE 5. Variables.

	Variable	Value
HDFS	Block replication	3
	HDFS Block Size	128 MB
	Java heap size	1 GB
Spark	Spark Demon Memory	2 GB
	Spark IO Compression lz4 BlockSize	128 KB
	Spark Shuffle IO Back-Log	8192

TABLE 6. Datasets used for GDLL evaluation.

Indices	Cora	PPI	OGBN
Nodes	2708	56944	2,449,029
Edges	5429	818716	61,859,140
Features	1433	50	100
Classes	7	121	47
Train	140	44,906	196,615
Valid	500	6,514	39,323
Test	1,000	5,524	2,213,091
Layers	2	3	3
Embedding	16	64	256
Epochs	200	200	20

B. DATASET

We use three different size of datasets to evaluate our proposed framework, consisting of two small datasets in standalone settings (i.e., cora [39] and PPI [40]), and one big datasets in distributed settings (OGBN-product [41]). Details about those datasets are summarized in Table 6.

1) CORA

The Cora dataset consists of 2708 scientific publications which belong to seven classes and 5429 connections amongst them corresponding to 2708 nodes and 5429 edges.

2) PPI

The PPI dataset is protein-protein interaction (PPI) graphs. There are 24 independent graphs with 56944 nodes belonging to 121 classes and 818716 edges in total, where each graph expresses different human tissue.

3) OGBN

The dataset OGBN is created by collecting Amazon products. There are total of 2,449,029 nodes and 61,859,140 edges where nodes represent products, and edges connect two products if the products are purchased together. Each node belongs to one of 47 classes and is described by 100-dimensional features.

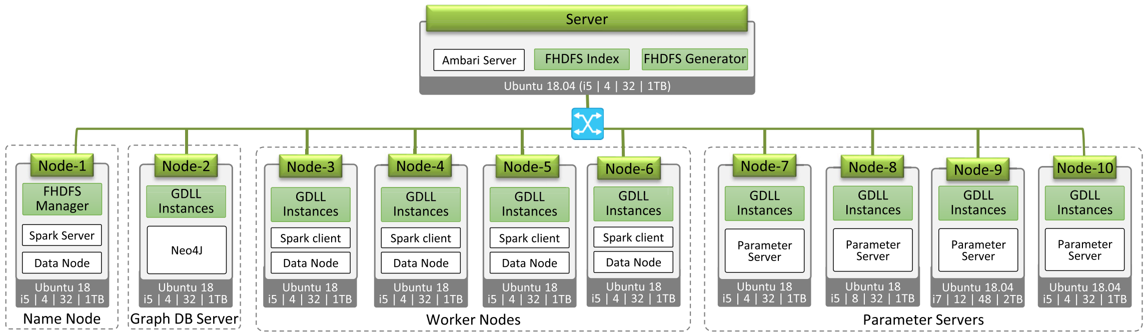


FIGURE 9. In-house cluster setup for GDLL evaluation.

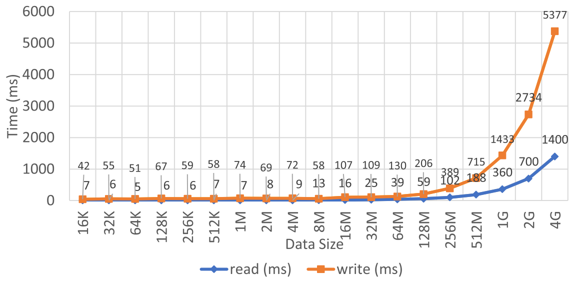


FIGURE 10. HDFS I/O performance.

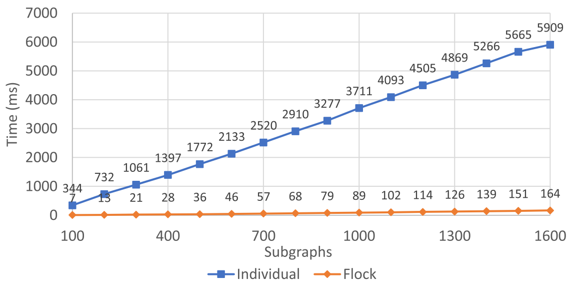


FIGURE 11. FHDFS read performance.

C. GDL EVALUATION

Figure 10 shows the HDFS performance in terms of read/write time with respect to different data sizes. For smaller data sizes, both the read and write operations take very little time, but the time increases as the data size increases beyond the HDFS block size. From the experimental result, we can see that the I/O time increases exponentially when the size of data exceeds the HDFS block size, which is 128MB in our case. It is worth mentioning that these experiments were conducted for a single read/write operation and not a bunch of read/write operations. One of the bottlenecks of the HDFS is the number of simultaneous I/O operations it can handle, for which we designed the F-HDFS.

Figure 11 shows the performance of F-HDFS in terms of read time concerning the number of subgraphs and Flocks. From the results, it can be seen that the read time increases linearly with the individual subgraphs whereas the read time remains significantly small in the case of Flocks.

D. GOL PERFORMANCE

The *k-hop* performance has been evaluated on a single system and in a distributed environment. We show *k-hop* performance, on a standalone machine, on Cora and PPI datasets, as shown in Figure 12. Here *k* is the value of *k* in *k-hop*. In this experiment, we evaluated the performance of the *k-hop* ranging from 1 to 5 *k-hops*. From the Figure, it is clear that Spark-based *k-hop* has significantly better performance than Hadoop. The AGL utilizes Hadoop MapReduce, whereas our proposed GDLL is based on Apache Spark. This experiment aims to show that performance improvement which will also affect the overall performance of the proposed GDLL in a distributed environment.

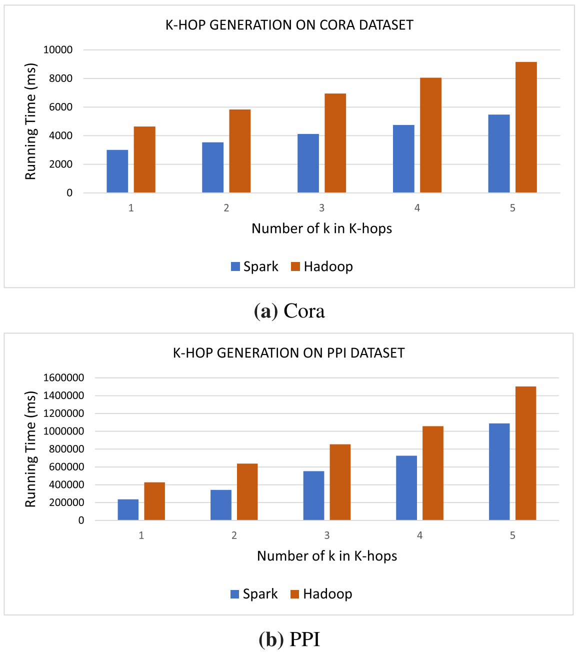


FIGURE 12. K-Hop generation evaluation using cora and PPI dataset (standalone).

TABLE 7. Popular GNNs training, effectiveness with different scalable systems.

Datasets	Methods	Base	PyG	DGL	AliGraph	AGL	GDLL (ours)
Cora (Accuracy)	GCN	0.813	0.818	0.811	0.802	0.811	0.807
	GAT	0.830	0.831	0.828	0.823	0.830	0.829
PPI (micro-F1)	GraphSAGE	0.598	0.632	0.636	-	0.635	0.633
	GAT	0.973	0.983	0.976	-	0.977	0.976
Ogbn (Accuracy)	GCN	0.757	-	-	0.723	0.744	0.738
	GraphSAGE	0.780	-	-	0.745	0.775	0.772

TABLE 8. Time per epoch (in second) on PPI dataset in standalone mode.

	GCN			GraphSage			GAT		
	1-layer	2-layer	3-layer	1-layer	2-layer	3-layer	1-layer	2-layer	3-layer
AGL	0.42	1.13	1.52	0.34	0.88	1.35	4.63	13.73	18.63
GDLL (ours)	0.33	0.89	1.21	0.27	0.70	1.07	3.75	11.27	15.23

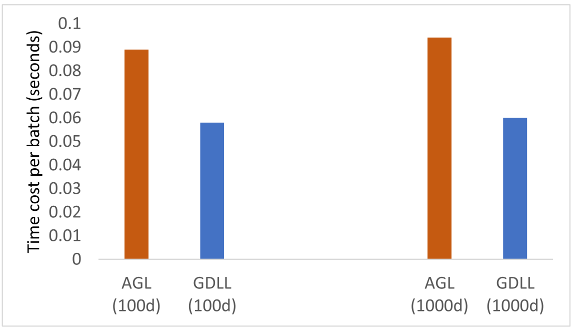


FIGURE 13. Time per batch comparison on OGBN-product dataset in distributed environment.

E. DISTRIBUTED GNN TRAINING

In this section, we show that our proposed GDLL is significantly efficient to the existing frameworks such as AGL while maintaining similar effectiveness in terms of GNNs convergence. Here, we demonstrate the effectiveness and efficiency by evaluating GNNs training in both standalone and distributed settings. In a standalone mode, we performed our experiments on two public datasets (i.e., Cora and PPI), as can be seen in Table 7. We achieved comparable accuracy on the Cora by training GCN and GAT in a standalone environment on a similar configuration used in AGL. Correspondingly, we achieved a similar micro-F1 on a multi-label PPI dataset by employing GraphSage and GAT. This shows our proposed GDLL framework is effective in terms of convergence to the existing graph learning frameworks such as PyG, DGL, AliGraph, and AGL. To demonstrate the scalability of GDLL, we also experimented on OGBN-product for distributed GNNs training and achieved similar performance in terms of effectiveness, as shown in Table 7.

To show the efficiency of our proposed GDLL framework in terms of speed, we illustrated per epoch time in the standalone environment on Cora, PPI, and in distributed mode on OGBN-product, respectively. For a fair comparison, we kept a similar configuration to AGL. Table 8 shows the efficiency in standalone mode and GDLL superior training speed can be seen. In the distributed mode, we employed in-house 4 workers and 4 servers in the parameter server for OGBN-product. We evaluated AGL and then our proposed GDLL on our in-house setup. Figure 13 shows the per batch (similar to AGL, i.e., 20 neighbors sample for a certain node) time in a distributed mode in comparison to AGL. For a fair comparison, we also expand the original 100-dimensional feature to 1000 dimensional by appending zeroes. Figure 13 shows the GDLL superior efficiency in comparison to the existing state-of-the-art framework i.e., AGL. The GDLL having similar effectiveness demonstrates superior efficiency in terms of speed, both in standalone and distributed mode. The optimization of the GDL, and GOL layers in the context of GNN training played a significant role in the overall efficiency of the proposed GDLL framework.

VI. CONCLUSION

In this study, we proposed a shared-nothing architecture-based scalable, layered, and distributed framework for GNNs called GDLL. The proposed framework is composed of three layers. In GDL, we optimized the HDFS for subgraph indexing and retrieval while proposing the notion of F-HDFS. In GOL, we utilized an in-memory distributed computing engine for k-hop neighborhood-based information complete subgraph. In addition, the k-hop neighborhood ensures independence among nodes in the graph, thus making us simply train the model with parameter servers in GLL. The results are encouraging in terms of performance, i.e., the accuracy has been maintained, while the efficiency, i.e., time per epoch, is significantly improved.

In the future, we will research to make the GNN inference efficient (e.g., by employing message passing to evade repeated computing) as there is also a need to make the inference more efficient for near real-time inferences for industrial-scale graphs. Likewise, we have specialized graph processing engines for scalable graph analytics. Also, we will investigate how to exploit such engines for scalable GNNs.

REFERENCES

[1] Z. Li, Y. Xing, J. Huang, H. Wang, J. Gao, and G. Yu, "Large-scale online multi-view graph neural network and applications," *Future Gener. Comput. Syst.*, vol. 116, pp. 145–155, Mar. 2021.

[2] S. Pan, R. Hu, S.-F. Fung, G. Long, J. Jiang, and C. Zhang, "Learning graph embedding with adversarial training methods," *IEEE Trans. Cybern.*, vol. 50, no. 6, pp. 2475–2487, Jun. 2020.

[3] A. Pucci, M. Gori, M. Hagenbuchner, F. Scarselli, and A. Tsoi, "Applications of graph neural networks to large-scale recommender systems some results," in *Proc. Int. Multiconference Comput. Sci. Inf. Technol.*, vol. 1, 2006, pp. 189–195.

[4] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song, "Heterogeneous graph neural networks for malicious account detection," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, 2018, p. 2077. [Online]. Available: https://www.overleaf.com/project/615fe81fae0c55ba6f37e557–2085

[5] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi, "AGL: A scalable system for industrial-purpose graph machine learning," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3125–3137, Aug. 2020.

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proc. 11th Symp. Oper. Syst. Design Implement.*, 2014, pp. 583–598.

[7] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big Learn. NIPS Workshop*, vol. 6, 2013, p. 2.

[8] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv:1909.01315*.

[9] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "AliGraph: A comprehensive graph neural network platform," 2019, *arXiv:1902.08730*.

[10] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, "FeatGraph: A flexible and efficient backend for graph neural network systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–13.

[11] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "PyTorch-BigGraph: A large-scale graph embedding system," 2019, *arXiv:1903.12287*.

[12] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "NeuGraph: Parallel deep neural network computation on large graphs," in *Proc. Annu. Tech. Conf.*, 2019, pp. 443–458.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," 2012, *arXiv:1204.6078*.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. 26th Symp. Mass Storage Syst. Technol. (MSST)*, 2010, pp. 1–10.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, nos. 10–10, p. 95, Jun. 2010.

[16] Y. Liu, X. Shi, L. Pierce, and X. Ren, "Characterizing and forecasting user engagement with in-app action graph: A case study of snapchat," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 2023–2031.

[17] X. Chen, L.-J. Li, L. Fei-Fei, and A. Gupta, "Iterative visual reasoning beyond convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 7239–7248.

[18] K. Do, T. Tran, and S. Venkatesh, "Graph transformation policy network for chemical reaction prediction," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 750–760.

[19] A. M. Fout, "Protein interface prediction using graph convolutional networks," Ph.D. dissertation, Dept. Comput. Sci., Colorado State Univ., Fort Collins, CO, USA, 2017.

[20] E. Choi, Z. Xu, Y. Li, M. Dusenberry, G. Flores, E. Xue, and A. Dai, "Learning the graphical structure of electronic health records with graph convolutional transformer," in *Proc. AAAI Conf. Artif. Intell.*, 2020, vol. 34, no. 1, pp. 606–613.

[21] H. Li, Y. Liu, Y. Li, B. Huang, P. Zhang, G. Zhang, X. Zeng, K. Deng, W. Chen, and C. He, "GraphTheta: A distributed graph neural network learning system with flexible training strategy," 2021, *arXiv:2104.10569*.

[22] M. Fey and J. Eric Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.

[23] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2018, pp. 974–983.

[24] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "DistDGL: Distributed graph neural network training for billion-scale graphs," in *Proc. IEEE/ACM 10th Workshop Irregular Appl., Archit. Algorithms (IA3)*, Nov. 2020, pp. 36–44.

[25] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[26] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "DistGNN: Scalable distributed training for large-scale graph neural networks," 2021, *arXiv:2104.06700*.

[27] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th Symp. Oper. Syst. Design Implement.*, 2012, pp. 17–30.

[28] *Apache Hadoop*. Accessed: Aug. 9, 2021. [Online]. Available: Available: https://hadoop.apache.org/

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.

[30] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[31] M. Serafini and H. Guan, "Scalable graph neural network training: The case for sampling," *ACM SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, pp. 68–76, Jun. 2021.

[32] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.

[33] P. V. kovi, G. Cucurull, A. Casanova, A. Romero, P. Liü, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.

[34] A. Vaswani, N. Shazeer, and N. Parmar, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[35] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *Proc. Symp. Opera. Syst. Design Implement.*, Carlsbad, CA, USA, Oct. 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz

[36] (Dec. 2021). *Hortonworks Data Platform*. [Online]. Available: https://docs.hortonworks.com/HDPDocuments/HDP3/HDP-3.1.0/index.html

[37] V. Vavilapalli, A. Murthy, and C. Douglas, "Apache Hadoop yarn: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.

[38] S. Haloi, *Apache Zookeeper Essentials*. London, U.K.: Packt, 2015.

[39] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Mag.*, vol. 29, no. 3, p. 93, 2008.

[40] M. Zitnik and J. Leskovec, "Predicting multicellular function through multi-layer tissue networks," *Bioinformatics*, vol. 33, no. 14, pp. 190–198, Jul. 2017.

[41] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," 2020, *arXiv:2005.00687*.

DUONG THI THU VAN received the B.S. degree in information technology from the HCMC University of Technology and Education, Vietnam. She is currently pursuing the combined master's and Ph.D. degree with the Data knowledge and Engineering (DKE) Laboratory, Department of Computer Science and Engineering, Kyung Hee University, Global Campus, South Korea. Her research interests include graph mining, big data analytic, and distributed computing.

MUHAMMAD NUMAN KHAN received the bachelor's degree in computer science with a specialization in data-science and web engineering from the Department of Computer Science, University of Peshawar, Peshawar, Pakistan. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Kyung Hee University, Global Campus, South Korea. His research interests include big-data analytics, distributed computing, computer vision, video retrieval, machine and deep learning, information service engineering, knowledge engineering, knowledge graph, and graph mining.

TARIQ HABIB AFRIDI received the B.S. degree in computer and information sciences from PIEAS, Islamabad, Pakistan, and the master's degree in computer science from the National University of Sciences and Technology (NUST), Islamabad. He is currently pursuing the Ph.D. degree with the Data knowledge and Engineering (DKE) Laboratory, Department of Computer Science and Engineering, Kyung Hee University, Global Campus, South Korea. His research interests include deep learning, computer vision, transformers, vision-language multimodal, graph neural networks, big data analytics, and distributed learning.

IRFAN ULLAH received the master's degree in computer science from the National University of Sciences and Technology (NUST), Islamabad, Pakistan. He is currently working as a Research Assistant at the Department of Computer Science and Engineering, Kyung Hee University, Global Campus, South Korea. His research interests include natural language processing, social computing, crisis informatics, machine/deep learning, OS design and optimization on memory systems, big data analytics, and distributed computing.

AFTAB ALAM received the B.S. degree in computer science from the University of Malakand, Khyber Pakhtunkhwa, Pakistan, the M.S. degree in computer science from the University of Peshawar, Khyber Pakhtunkhwa, and the Ph.D. degree in computer science from Kyung-Hee University, South Korea. He is currently working as a Researcher at the Department of Computer Science and Engineering, Kyung-Hee University, Global Campus, South Korea. His research interests include big data analytics in the cloud, distributed computing, information retrieval, social computing, machine/deep learning, information service engineering, knowledge engineering, complex event analysis in the multi-stream environment, and graph mining.

YOUNG-KOO LEE (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 1992, 1994, and 2002, respectively. Since 2004, he has been with the Department of Computer Science and Engineering, Kyung Hee University, Global Campus. His research interests include data mining, online analytical processing, and big data processing.