Karunarathna, Anuradha; Senarath, Dinika; Madhushanki, Shalika; Weerakkody, Chinthaka; Dayarathna, Miyuru; Jayasena, Sanath; Suzumura, Toyotaro;, "Scalable Graph Convolutional Network Based Link Prediction on a Distributed Graph Database Server", 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), pp. 107-115, 18-24 Oct. 2020 doi: 10.1109/CLOUD49709.2020.00028



# Scalable Graph Convolutional Network based Link Prediction on a Distributed Graph Database Server

Anuradha Karunarathna*, Dinika Senarath*, Shalika Madhushanki*,
Chinthaka Weerakkody*, Miyuru Dayarathna‡*, Sanath Jayasena*, and Toyotaro Suzumura†¶§
*Department of Computer Science & Engineering,
University of Moratuwa, Moratuwa, Sri Lanka
Email: {anuradha.15,dinika.15,shali.madhushanki.15,chinthaka.18,sanath}@cse.mrt.ac.lk
‡WSO2, Inc., USA
Email: miyurud@wso2.com
†IBM T.J. Watson Research Center, New York, USA / ¶MIT-IBM Watson AI Lab, Cambridge, MA, USA/
§Barcelona Supercomputing Center, Barcelona, Spain
Email: suzumura@acm.org

Abstract—Graph Convolutional Networks (GCN) have become a popular means of performing link prediction due to the high accuracy offered by them. However, scaling such link prediction into large graphs of billions of vertices and edges with rich types of attributes is a signif cant issue to be addressed due to the storage and computation limitations of the machines. In this paper we present a scalable link prediction approach which conducts GCN training and link prediction on top of a distributed graph database server called JasmineGraph. We partition graph data and persist them in multiple workers. We implement parallel graph node embedding generation using GraphSAGE algorithm in multiple workers. Our approach avoids facing performance bottlenecks in GCN training using an intelligent scheduling algorithm. We show our approach scales well with an increasing number of partitions (2,4,8, and 16) using four real world data sets called Twitter, Amazon, Reddit, and DBLP-V11. JasmineGraph was able to train a GCN from the largest dataset DBLP-V11 (> 9.3GB) in 11 hours and 40 minutes time using 16 workers on a single server while the original GraphSAGE implementation could not process it at all. The original GraphSAGE implementation processed the second largest dataset Reddit in 238 minutes while JasmineGraph took only 100 minutes on the same hardware with 16 workers leading to 2.4 times improved performance.

Keywords-Machine Learning; Graph databases; Database management; Distributed databases; Graph theory; Graph Convolutional Neural Networks; Deep learning; Link Prediction;

## I. Introduction

The ability to handle massive graphs has become a necessity for conducting analytics on sensor networks, social networks, web graphs, citation graphs, etc [4] [21]. Recently, graph databases have obtained the capabilities of conducting analytics which involve large scale graphs of millions or even billions of vertices [5].

On the other hand, Convolutional Neural Networks (CNN) has found applications especially in the areas of link prediction and node classif cation with the advent of Graph Convolutional Neural Networks (GCN) [11] (i.e., Graph Convolutional Networks). Graph Convolutional Network is an improvement made over Convolutional Neural Networks with the aim of encoding graphs.

Link prediction predicts the probability of having a link between two nodes in a graph [15]. State-of-the-art link prediction approaches based on GCN are focused on small graphs with a few thousands of vertices and edges. Furthermore, most of the existing researches focus on link prediction only using the graph structure without considering the node/edge attributes. Use of attributes enables for higher accuracy with link prediction. However, existing systems are unable to handle such link prediction on large scale graphs (millions and billions of edges, varieties of rich node/edge attributes, gigabytes or even terabytes in size) due to the need for excessive memory and CPU resources [6]. No study has been made on eff cient scheduling of such link prediction tasks on large attribute graphs.

As a solution, in this work we propose an approach of distributing graphs across multi-machine clusters and perform deep learning and link prediction on distributed graph partitions. We develop a scheduling algorithm which eff ciently conducts the GCN training process of the graph partitions in the worker nodes. We design and implement our approach on top of a distributed graph database server called JasmineGraph [1] which partitions and stores rich graph data in multiple nodes.

We use the popular GraphSAGE algorithm for node embedding construction [9]. GraphSAGE algorithm not only considers the neighbourhood structure of each node, but also considers node attributes. While the original GraphSAGE algorithm generates node embedding in non-distributed manner, in our system we do node embeddings generation separately for each graph partition. This allows us to scale

[1]https://github.com/annonrepo/jasminegraph

the embedding generation. Since the edges in the original graph are partitioned in a manner that minimum number of edges are cut, the subgraphs are dense and each node has a rich neighbourhood. Therefore, the embeddings generated will be equally rich compared to the embeddings generated when the whole graph is considered. Therefore, we show that partitioning does not reduce the accuracy significantly. When graph datasets are too large to be handled in a single machine, our approach provides a good solution. This reduces the computation power of training because each worker shares a portion of training. The existing distributed deep learning approaches share model parameters across the workers and require global synchronization at each training iteration. This communication of model parameter updates across workers is a major bottleneck of such systems. But in this work, we show that we can get equally rich node embeddings by training independent models for graph partitions without having to share model parameters. It is because with our partitioning approach, the edges between partitioned subgraphs are kept minimal. Hence, each partition corresponds to a dense component of the graph with similar feature distribution and minimal connections to other partitions.

We show the proposed approach executes efficiently and is able to work with significantly larger graphs compared to the state-of-the-art such as Euler [1]. JasmineGraph was able to train a GCN from the largest dataset DBLP-V11 (size $>$ 9.3GB) in 11 hours and 40 minutes time using 16 workers on a single server while the original GraphSAGE implementation running on the same server could not process it at all. The second largest dataset Reddit was processed by the original GraphSAGE implementation in 238 minutes while JasmineGraph took only 100 minutes on the same single server with 16 workers leading to 2.4 times improved performance. From the scaling experiments done on 3 smaller graphs we observed that the GCN training process is a memory intensive operation than being CPU intensive. Specifically, the contributions of this paper can be listed as follows,

- *Scalable GCN based link prediction* - We implement and evaluate an approach for link prediction based on GCNs. We also describe multiple different performance optimizations which we conduct to do this process in a scalable manner.
- *Scheduling Mechanism* - We implement a scheduling algorithm based on bin packing and we demonstrate how this algorithm allows us to run the GCN training process in an efficient manner.
- *Evaluation* - We provide performance evaluation results of JasmineGraph system with experiments conducted using real world data sets.

The rest of the paper has been organized as follows. In the next section we describe the related work. Section III provides an overview to JasmineGraph. Section IV provides the implementation details. Section V provides the evaluation of the proposed approach. We discuss the results in Section VI. Section VII concludes the paper.

## II. Related Work

Link prediction is a technique which predicts the future connections of a network using the existing connections. Mainly, three types of graph features are important for graph link predictions. They are graph structure features which show the node and edge connectivity of the graph, latent features which are the node representations, and explicit features such as node attributes [26].

Embedding generation produces node embeddings which can be used for a variety of down-stream application such as link prediction. Network embeddings can be mainly discussed under three categories as matrix factorization [23], random walk [19], and deep learning approaches. Matrix factorization based algorithms obtain the node embedding by factoring a matrix such as an adjacency matrix which represents the network connection. Random walk based approaches like DeepWalk [19] and Node2Vec [9] perform a random walk through the neighbourhood of a selected node to compute the embedding. However, the factorization and random walk embeddings are unable to include node feature information.

Gori *et al.* [8] presented Graph Neural Networks (GNN) for the first time, and it got further extended by Scarselli *et al.* (2009) [21]. In order to learn a target nodes' representation under this approach, the neighbour information is propagated through a recurrent architecture iteratively until reaching a stable position. Even though the GCN has marked a significant success in learning graph data rather than GNN, some challenges exist. These are scalability, graph complexity and computation efficiency. In this work we propose a data partitioning based solution to fix these issues.

Zeng *et al.* presented a graph embedding method based on graph sampling [25]. They construct a new GCN on a small subgraph in every training iteration. They do not partition graph data as we do. However, they depend on specific hardware instructions to achieve optimal performance. Furthermore, they do not handle large graphs like DBLP-V11. A similar technique for node embedding generation was proposed by Lin *et al.* [17]. They use graph partitioning based approach like we do. However, the experiments have been done with small scale graphs hence they do not face system performance issues like we do.

PyTorch-BigGraph (PBG) is a graph embedding system that includes several modifications to the traditional multi-relation embedding systems which allows for scaling to graphs of billions of nodes and trillions of edges [14]. However, PBG depends on a shared file system. In contrast, JasmineGraph keeps data partitioned across multiple disks,

and it does not depend on such shared file system. Furthermore, JasmineGraph uses graph structure (i.e., neighbourhood information) vertex attributes as well for embeddings construction while PBG supports only for edge lists and edge attributes for embedding construction. Euler is a distributed graph learning system [1]. It can work with Tensorflow and allows users to train models on very complex heterogeneous graphs. However, Euler has not been tested with large scale graphs such as DBLP-V11. Similar to PBG, Euler also depends on a shared file system whereas JasmineGraph does not have such requirement. PBG uses random partitioning scheme while Euler uses hash partitioning. Both these techniques do not consider graph structural features like JasmineGraph's partitioner does.

We have implemented the proposed link prediction approach on top of JasmineGraph server. Next section describes essentially brief summary of system design of JasmineGraph.

## III. An Overview to JasmineGraph

An overview of the design of JasmineGraph is shown in Figure 1. The architecture of JasmineGraph is very similar to Acacia distributed graph database server [2] [3]. However, JasmineGraph is developed completely in C/C++ whereas Acacia was developed using X10 programming language.

The system has been designed with two main component categories: Master and Worker (See Figure 1). Two types of communication protocols have been designed for communication between external clients and the JasmineGraph system, and another between the Master and the Worker as well as between Workers themselves.

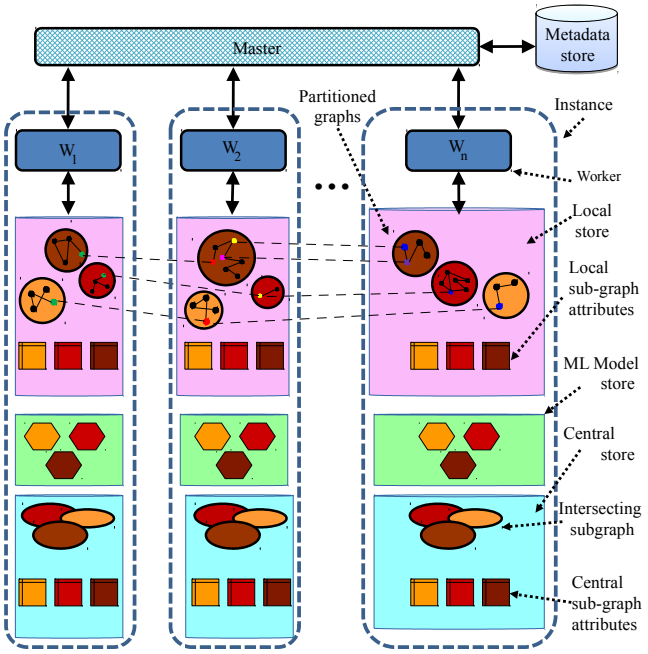


Figure 1: System Architecture of JasmineGraph.

The MetaDB of JasmineGraph is a standalone SQLite database which is used for storing metadata such as vertex count, edge count, graph path, partition id, vertex/edge counts of partitions, etc.

RDF partitioner and JSON attribute parser modules are used for processing the knowledge graphs (especially the attributes). Once the graph structure has been extracted by these two components, they get partitioned by using the Metis Partitioner. The plain edge lists are directly partitioned using Metis Partitioner [11]. Metis achieves high quality graph partitioning by using multilevel graph partitioning scheme. For more details, please refer [11].

Each worker consists of three data stores: Local, Central, and Model stores. The local store maintains the partitioned subgraphs. Central store maintains the edges where the starting and ending vertices stay in two different local stores. The attributes of each local/central partition also gets stored in the corresponding data store. During the machine learning (ML) model training process, the ML models are created for each partition. This means for a pair of local store and central store partitions, a corresponding ML model is trained. Such models get stored in the model store. These three categories of data stores are native stores that store data in a specialized efficient storage format. File Transfer Service module conducts data transfer between Master and the Workers. Embeddings generation is done by each worker. In the next section we provide the implementation details of our scalable link prediction system.

## IV. Implementation of High Performance Link Prediction

In this section first, we present the implementation details of storage format used for native store. Second, we present how the node embeddings are constructed. Third, we give specific details of the link prediction algorithm. Finally, we describe how the intelligent workload scheduling has been implemented for training.

### A. Graph Data Uploading

JasmineGraph has the capability to store the graph structure, the attributes of the graph nodes, as well as trained machine learning models in an effective manner. In order to achieve this, we do data partitioning. For the graph structure partitioning functionality, we use Metis graph partitioner [11]. Metis partitions an unstructured graph into a given number of partitions. We use Metis because it significantly reduces the communication overhead between compute nodes by reducing the amount of edge cut [3] [5]. If an edge has both of its nodes on the same partition, we store that edge in the local store. If not, it is stored in the central store. The trained models get stored in the model store.

### B. Implementation of Node Embedding Generation

The task of link prediction depends on the construction of embeddings for each and every vertex and then using those embedding to do the link prediction. We use the popular GraphSAGE algorithm to generate node embeddings [10]. GraphSAGE (Graph SAmple and aggreGatE) learns a function which creates embeddings by sampling and aggregation of features from a node's local neighborhood. It is an inductive node embedding algorithm which uses node features in order to learn an embedding function which generalizes to nodes which have not been seen before. After the train and validation phases are finished, the node embeddings are written to models store of JasmineGraph.

Since JasmineGraph stores an entire graph as partitions in several workers in a distributed manner, we use localized graph convolution modules to train each graph partition (Algorithm 1). The final outcome is node embedding for each node in the partition. We evaluate the accuracy of the prediction using Mean Reciprocal Rank (MRR) statistical measure. Moreover, the training happens in an unsupervised manner. We initiate the training process with the concatenation of local store graph partition and the corresponding central store partition. Then, node features of the concatenated graph and the graph structure is input to a neural network. The hidden layers of the neural network are structured in such a way to transform node features and aggregate the features over the graph to generate node embeddings.

The mechanism of storing edge cuts in central stores allows us to use all edges of the unpartitioned graph at the distributed training without any edge loss. Thus, for each partition's training, we use the aggregation of local store edges and corresponding central store edges. Moreover, the use of localized GCN algorithm for node embedding compensate for the loss of loosely connected nodes. Therefore, Metis partitioned high-quality sub graphs can be trained locally at a distributed setting to generate final node embeddings of the graph.

### C. Implementation of Link Prediction

The node embeddings generated using the Graph Convolutional Neural Network are used for the link prediction task. Since we generate the node embeddings by considering the graph structure and node identity features, the nodes which probably have a link will get closely similar node embeddings. Therefore, we perform nearest-neighbor search in the generated node embedding space to predict links. However, calculating the similarity indices for a given node with all other nodes in the graph individually, is not a scalable approach. Locality Sensitivity Hashing (LSH) is an efficient approximate nearest neighbor lookup algorithm for high dimensional space [20]. Here we use cosine similarity scoring to find the similarity of two node embeddings. Our link prediction algorithm is shown in Algorithm 2. The algorithm accepts a starting node denoted as query node ($q$) and it points out a list of predicted nodes as the output. In Algorithm 2, we use random projection method of LSH. First, an LSH object is created which can map high dimensional (*node_embed_dim*) node embeddings into lower dimensional (*hash_dim*) space and group the generated bit-wise hash values into a number of hash lookup tables. Then each node embedding is stored in the lookup table along with an index (*idx*).

**Algorithm 1: Embedding generation for a graph partition**

**Input** : localStoreGraphPortion $G_{local}(V_{local}, E_{local})$ , centralStoreGraphPortion $G_{central}(V_{central}, E_{central})$, epochs, minibatch-size, depth D, weight matrices $W^d, \forall d \in \{1,...,D\}$, non-linearity function $\sigma$, differentiable aggregator functions $AGGREGATE_d, \forall d \in \{1,...,D\}$, neighborhood sampler $N: v \rightarrow 2^V$

**Output** : Node embedding $z_v$ for all $v \in V$

**Description :**

```
G(V, E) ← G_local(V_local, E_local) + G_central(V_central, E_central)
features ← G.getFeatures()
validate_edges ← G.getValidationEdges()
test_edges ← G.getTestEdges()
train_edges ← E – validate_edges – test_edges
train_adjacency ← G.construct_adj(train_edges)
test_adjacency ← G.construct_adj(E)
for each epoch do:
    for each train_minibatch do:
        GraphSAGE_ForwardPropagation
            (train_adjacency,features,D,W^d,AGGREGATE_d,N)
        if validation_iteration:
            for each validate_minibatch do:
                GraphSAGE_ForwardPropagation
                (test_adjacency,features,D,W^d,AGGREGATE_d,N)
            end for
        end if
        optimizeParameters()
    end for
end for
return Node embedding z_v fo all v ∈ V
```

**Algorithm 2 : Link Prediction Algorithm**

**Input** : graphId, query node (q), no_of_results

**Output** : Predicted nodes

**Description :**

```
node_embeds ←collectNodeEmbeddings(graphId)
lsh ←LSHash(hash_dim, node_embed_dim, num_hash_tables)
for each node_emb do:
        lsh.index(node_emb, idx)
end for
predicted_nodes ← lsh.query(q, no_of_resuts)
return predicted_nodes
```

### D. Implementation of Intelligent Training Scheduler

Training multiple partitions in parallel may have performance issues when partitions do not fit into the memory. Therefore, the number of partitions that fit into the memory should be identified in order to support the desired level of parallelism. Our scheduling algorithm calculates the number of partitions that can be trained in parallel by estimating the memory requirements of each partition. Then each partition is assigned to an iteration in which its training will take place. When training is performed in a single node, this iteration schedule is generated for that single node. In the distributed setting, a separate schedule is generated for the partitions of each node.

The memory requirement for training a graph partition is estimated considering the metadata of the graph partition. These include vertex count, edge count, number of features, etc. and the memory usages attached with data structures and neural network architecture of the training algorithm. The available memory of the host node is obtained using the performance data collector of JasmineGraph. The scheduling of partitions is done in a manner that the available memory is fully utilized in each iteration, and all the partitions are trained in a minimum number of iterations. We developed a bin packing algorithm (See Algorithm 3) to assign partitions into the iterations (bins with the capacity of available memory) in such a way that the number of iterations is minimum. Our algorithm follows a 'best first' approach where each partition is assigned to the tightest spot (place where it requires minimal amount of resources) which can hold the partition. The output of the algorithm is a map of *(partitionID, iterationNo)* key-value pairs for each host machine. Next, we present the evaluation details of our link prediction system.

**Algorithm 3 : Scheduling graph partitions for training iterations**

**Input** : PartitionMemoryEstimationList L, availableMemory
**Output** : partitionTrainingIterationsMap (a map of (partitionID, iterationNo) key-value pairs)
**Description :**

```
partitionCount ← L.size()
binCount ← 0
binRemainingSpaces ← array[partitionCount]
for each partition in range(0, partitionCount) do
    minSpaceInBin ← capacity + 1
    bestBin ← 0
    for j in range(0, binCount) do
        if binSpaceLeft[j] >= L[i] and binSpaceLeft[j] - L[i] < min
                                                                then
            bestBin ← j
            minSpaceInBin ← binSpaceLeft[j] - L[i]
        end if
    end for
    if minSpaceInBin =  capacity + 1 then
        binSpaceLeft[binCount] ← capacity - L[i]
        partitionTrainingIterationsMap[partitionID] = binCount
        binCount++
    end if
    else
        binSpaceLeft[bestBin] ← binSpaceLeft[bestBin] - L[i]
        partitionTrainingIterationsMap[partitionID] = bestBin
    end else
end for
return PartitionTrainingIterationsMap
```

## V. Evaluation

We did two types of experiments to evaluate the scalability of the proposed approach, vertical scaling and horizontal scaling (i.e., scale out). The purpose of running both these types of experiments was to identify what hardware characteristics such as memory, CPU, network, disk, etc. mainly affects the system performance. Furthermore, we ran the original GraphSAGE code with the same data sets and compared the results with the results obtained from JasmineGraph. Moreover, we implemented hash partitioning as well on top of JasmineGraph and compared the effect of partitioning for the accuracy.

### A. Experiment Environment

The evaluations were conducted on a server computer as well as using a cluster of 5 computers.

The server had Intel®Xeon®CPU E7-4820 v3 @ 1.90GHz, 40 CPU cores (80 hardware threads via hyper threading), 64GB RAM, 32KB L1 (d/i) cache, 256K L2 cache, and 25600K L3 cache. It had 1.8TB Hard Disk Drive (HDD). It was running on Ubuntu Linux version 16.04 with Linux kernel 4.4.0-148-generic.

The compute cluster was running on 5 computers, four of which were running on GCloud compute engine's virtual machines (VM) with each VM having Intel®Xeon®CPU @ 2.00GHz, 30 GB RAM, 4 CPU cores (8 hardware threads), 10GB HDD. We name these computers as $W_1$, $W_2$, $W_3$, and $W_4$ since they run only JasmineGraph workers during the experiments. The fifth computer denoted as $M$ had Intel®Xeon®CPU E5-2695 v2 @ 2.40GHz, 16GB RAM, 2 CPU cores (4 hardware threads). All four computers denoted as $W_1$, $W_2$, $W_3$, and $W_4$ in this cluster had 32KB L1 (d/i) cache, 1024K L2 cache and 39424K L3 cache. $M$ had 32KB L1 (d/i) cache, 256K L2 cache and 30720K L3 cache. All the computers in the cluster were running with Ubuntu Linux version 16.04.

### B. Experiment Scenarios and Datasets

The datasets used in our experiments are shown in Table I.

Note that we do not evaluate the elapsed time for uploading the graph data sets to JasmineGraph in this paper. Rather we directly go ahead with evaluating the scalability of the

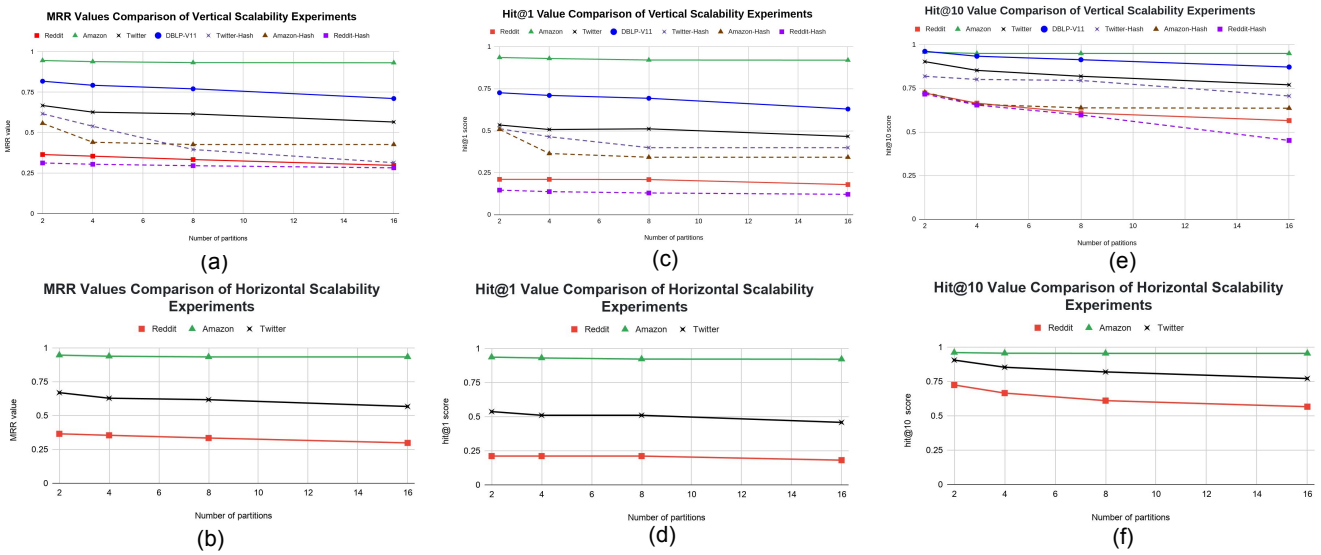


Figure 2: Overall Accuracy of GCN based Link Prediction

Table I: Data Sets.

| Dataset name | Vertices | Edges | No of features | Edgelist File Size (MB) | Feature Files Size (MB) |
|---|---|---|---|---|---|
| Twitter [18] | 81,306 | 1,768,149 | 1007 | 16 | 157 |
| Amazon-small [15] | 548,551 | 1,244,636 | 250 | 19.4 | 266 |
| Reddit [13] | 232,965 | 11,606,919 | 602 | 145 | 270 |
| DBLP-V11 [24] | 4,107,340 | 36,624,464 | 948 | 508 | 9523 |

training process because the aim is to evaluate the scalability aspects of the proposed scheduler. We conduct two types of experiments here. First, using only the server computer we do vertical scaling. We change the number of workers used in the values 2, 4, 8, and 16. We perform this experiment for all four datasets. Second, we use the cluster to run the experiments. We use Twitter, Amazon and Reddit datasets for this horizontal experiment and evaluate elapsed time and accuracy when trained in a different number of partitions. For GCN training we use 0.01 as the learning rate and set the number of epochs to 1. All the hyper-parameters were set according to the optimal default values in GraphSAGE.

### C. Experiment Results

We evaluate the link prediction accuracy to evaluate the generated node embedding quality. For that we use MRR and Hits@n metrics. In order to separate a set of test edges, we use the following approach. At each worker, for each graph partition we aggregate its corresponding central store portion to generate training subgraphs. Then 60% nodes of the subgraph are separated as train nodes, 20% as validation nodes and remaining as test nodes. Finally, the edges between test nodes are taken as test edges, edges between valid nodes are taken as validate edges and the rest of edges are used for the training process.

Table II: Experiments results on unpartitioned graphs using original GraphSAGE code on the server.

| Data set name | Time for training (s) | MRR | Hits@1 | Hits@10 |
|---|---|---|---|---|
| Reddit | 14260 | 0.686 | 0.541 | 0.951 |
| Amazon-small | 547 | 0.959 | 0.948 | 0.971 |
| Twitter | 2222 | 0.37 | 0.218 | 0.741 |
| DBLP-V11 | Cannot perform in the server | | | |

In Figure 2, (a), (c), and (e) charts show the MRR, Hits@1 and Hits@10 values obtained when the all four datasets are trained as several partitions in a single machine respectively. The accuracy results indicate that our system performs as expected in terms of the accuracy.

Charts (b), (d), and (f) in Figure 2, contain MRR, Hits@1 and Hits@10 values obtained for all datasets except for DBLP-V11 under the horizontal experiment respectively. For a particular number of partitions, we get similar accuracy values in both horizontal and vertical experiment scenarios. However, the accuracy values slightly decrease when the number of graph partitions increases. The reason for this accuracy reduction in all the cases is the neighbourhood information loss at edge cuts due to partitioning. This loss occurs because the sampling neighbourhood of boundary vertices (vertices at vertex cuts) is limited to the neighbourhood within the partition. Figure 3 shows the results from the scalability experiments. Increasing number of partitions (i.e., workers) reduces time taken for training process hence giving scalable result. However, this does not give linear scalability. Due to the large data size we were unable

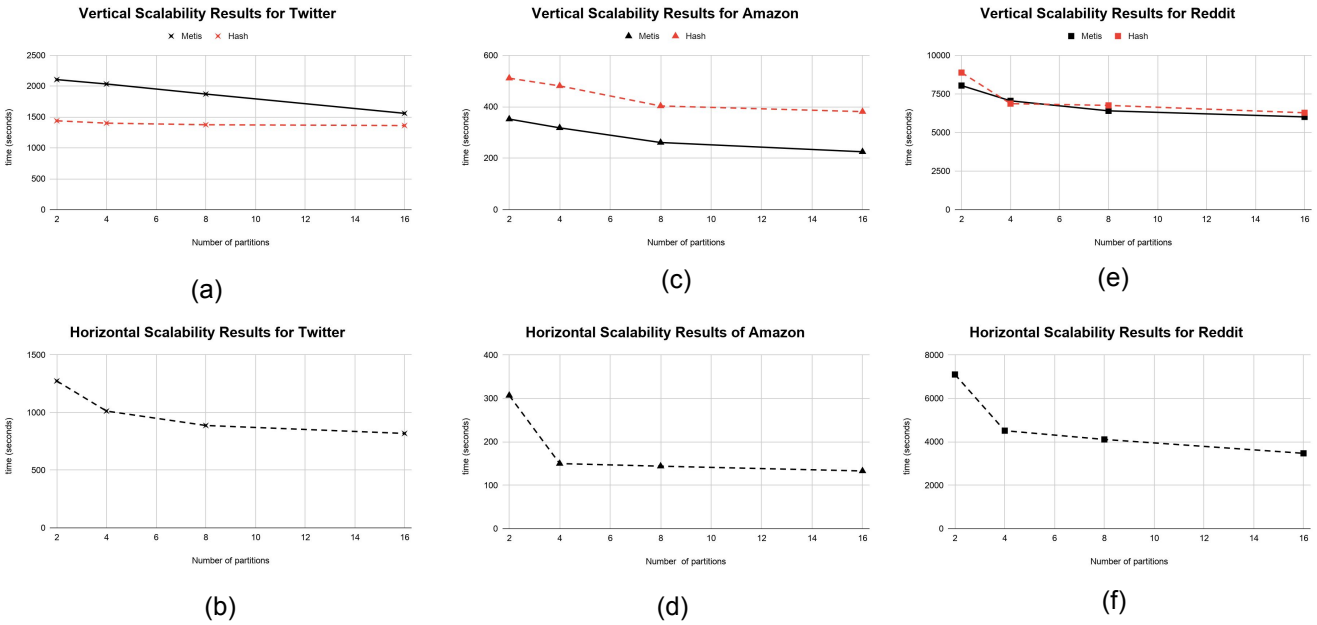


Figure 3: Vertical and Horizontal Scalability Results of GCN based Link Prediction for Twitter, Reddit, and Amazon graphs

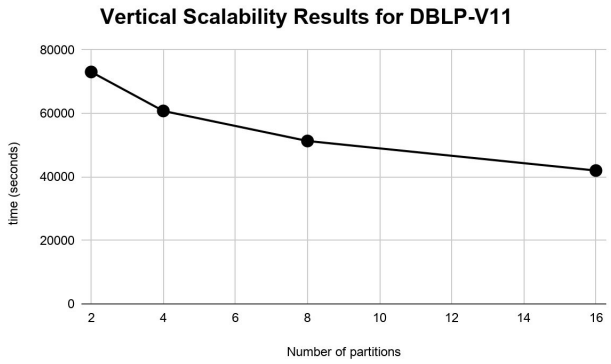


Figure 4: Vertical Scalability of GCN based Link Prediction for DBLP Dataset

to conduct horizontal scalability experiments with DBLP-V11. This is mainly due to each worker in the horizontal scalability cluster had only 10GB HDD space. However, the corresponding vertical scaling experiment ran successfully because the server had 1.8TB HDD space. Figure 4 shows the DBLP-V11 vertical scalability experiment results. We ran the original GraphSAGE code on the server. The results are shown in Table II. For all the data sets (except DBLP-V11) JasmineGraph provided significant improvements of training time with the highest being 243% improvement of elapsed time for Amazon-small dataset.

We conducted another round of experiments of vertical scalability using hash partitioning as the JasmineGraph's partitioning algorithm to evaluate the impact of graph partitioning algorithm for the training accuracy. The results are shown in dotted lines in Figure 3 (a), (c), and (e). All the hash partitioning experiments resulted in lesser accuracy values all the three metrics for the corresponding graph data sets. Elapsed times shown in Figure 3 (a), (c), and (e) indicate that it results in mixed outcome. Hence we plan to conduct further experiments of scalability in future to identify what characteristics of graph partitioning algorithm would result in lesser training times. However, both types of experiments indicated that graph partitioning allows for reducing the time taken for the training process significantly.

## VI. Discussion

In the current system we use two hop neighbourhood for node embedding construction. However, this is not possible for boundary vertices that are located at the vertex cuts. We can use only one hop neighbourhood using the information provided via central stores. Since central stores only record one hop neighbourhood, if we are to collect information beyond one hop neighborhood, we have to do a traversal which spans into the neighbouring partitions via the references provided via the central store. Although this could be implemented, in our system we keep it as a future work since there is no significant impact on the accuracy we observed due to this limitation in the current system.

## VII. Conclusion

In this paper we describe the scalable implementation of graph convolutional network (GCN) based link prediction on an open source distributed graph database server called

JasmineGraph [2]. Through this work we extended the well known GraphSAGE algorithm to run in a distributed parallel environment. We used graph partitioning to enable parallel execution of GCN training process. We found that densely connected components play critical role in determining the performance of the overall training process. With JasmineGraph's intelligent scheduler we were able to solve this issue by making sure each graph partition is scheduled with a group of partitions which fit into the memory of the worker, instead of training all partitions in parallel. We used four real world data sets Twitter, Amazon, Reddit and DBLP-V11 and showed performance improvements for our approach both in horizontal and vertical scaling experiments. The MRR and Hits metrics indicated that our system can perform with good accuracy and verified that the system is performing as expected. JasmineGraph was able to train a GCN from the largest dataset DBLP-V11 ($>$ 9.3GB) in 11 hours and 40 minutes time using 16 workers on a single server while the original GraphSAGE implementation could not process it at all. The second largest dataset Reddit was processed by the original GraphSAGE implementation in 238 minutes while JasmineGraph took only 100 minutes on the same hardware with 16 workers leading to 2.4 times improved performance. From the scaling experiments done with 3 smaller graphs we observed that our GCN training process is more memory intensive operation than being CPU intensive. In future we plan to extend our GCN based link prediction implementation to the domain of graph stream processing. We also hope to extend this work to the domain of privacy preserving machine learning.

## REFERENCES


[1] Alibaba. Euler. URL: https://github.com/alibaba/euler, 2019.

[2] M. Dayarathna, S. Bandara, N. Jayamaha, M. Herath, A. Madhushan, S. Jayasena, and T. Suzumura. An x10-based distributed streaming graph database engine. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 243–252, Dec 2017.

[3] M. Dayarathna, I. Herath, Y. Dewmini, G. Mettananda, S. Nandasiri, S. Jayasena, and T. Suzumura. Acacia-rdf: An x10-based scalable distributed rdf graph database engine. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 521–528, June 2016.

[4] M. Dayarathna, C. Houngkaew, H. Ogata, and T. Suzumura. Scalable performance of scalegraph for large scale graph analysis. In *2012 19th International Conference on High Performance Computing*, pages 1–9, Dec 2012.

[5] M. Dayarathna and T. Suzumura. Towards scalable distributed graph database engine for hybrid clouds. In *2014 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 1–8, Nov 2014.


[2]https://github.com/miyurud/jasminegraph


[6] M. Dayarathna and T. Suzumura. *High-Performance Graph Data Management and Mining in Cloud Environments with X10*, pages 173–210. Springer International Publishing, Cham, 2017.

[7] H. Gao, Z. Wang, and S. Ji. Large-scale learnable graph convolutional networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '18, page 1416–1424, 2018.

[8] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2, July 2005.

[9] A. Grover and J. Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864. ACM, 2016.

[10] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[12] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[13] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 933–943, 2018.

[14] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019.

[15] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Trans. Web*, 1(1), May 2007.

[16] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 556–559. ACM, 2003.

[17] W. Lin, F. He, F. Zhang, X. Cheng, and H. Cai. Initialization for network embedding: A graph partition approach. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, WSDM '20, page 367–374, 2020.

[18] J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 539–547, USA, 2012. Curran Associates Inc.

[19] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710. ACM, 2014.

[20] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[21] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, Jan 2009.

[22] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, Sept. 2017.

[23] X. Shen, S. Pan, W. Liu, Y.-S. Ong, and Q.-S. Sun. Discrete network embedding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 3549–3555, 7 2018.

[24] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: Extraction and mining of academic social networks. In *KDD'08*, pages 990–998, 2008.

[25] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Accurate, efficient and scalable graph embedding. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 462–471, May 2019.

[26] M. Zhang and Y. Chen. Link prediction based on graph neural networks. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 5171–5181, USA, 2018. Curran Associates Inc.