Tech Report of Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines

Yuhao Zhang
University of California, San Diego
yuz870@eng.ucsd.edu

Arun Kumar
University of California, San Diego
arunkk@eng.ucsd.edu

ABSTRACT

Recent advances in Graph Neural Networks (GNNs) have changed the landscape of modern graph analytics. The complexity of GNN training and challenges of GNN scalability has also sparked interest from the systems community, with efforts to build systems that provide higher efficiency and schemes to reduce costs. However, we observe that many such systems basically "reinvent the wheel" of much work done in the database world on scalable graph analytics engines. Further, they often tightly couple the scalability treatments of graph data processing with that of GNN training, resulting in entangled complex problems and systems that often do not scale well on one of those axes.

In this paper, we ask a question: How far can we push existing systems for scalable graph analytics and deep learning (DL) instead of building custom GNN systems? Are compromises inevitable on scalability and/or runtimes? We propose Lotan, the first scalable and optimized data system for full-batch GNN training with *decoupled scaling* that bridges the hitherto siloed worlds of graph analytics systems and DL systems. Lotan offers a series of technical innovations, including re-imagining GNN training as query plan-like dataflows, execution plan rewriting, optimized data movement between systems, a GNN-centric graph partitioning scheme, and the first known GNN model batching scheme. We prototyped Lotan on top of GraphX and PyTorch. An empirical evaluation using several real-world benchmark GNN workloads reveals a promising nuanced picture: Lotan significantly surpasses the scalability of state-of-the-art custom GNN systems, while often matching or being only slightly behind on time-to-accuracy metrics in some cases. We also show the impact of our system optimizations. Overall, our work shows that the GNN world can indeed benefit from building on top of scalable graph analytics engines. Lotan's new level of scalability can also empower new ML-oriented research on ever larger graphs and GNNs.

1 INTRODUCTION

Graph Neural Networks (GNNs) have drastically shifted the landscape of advanced graph analytics. GNNs can provide powerful learned representations of the semantics of nodes and graphs. In about a decade, they have dominated many graph analytics leaderboards [17] for tasks ranging from node-edge level tasks, such as node classification and edge prediction, to graph-level tasks like graph classification or even graph generation. Applications span from video analytics [19], recommender systems [49], to drug discovery [24] and pandemic data analysis [46]. Interests across different domains are rising rapidly because many data are naturally represented as graphs, such as social networks and molecular structures.

However, GNN models are tricky to scale [15, 44, 45], because of the sheer amount of computation and the immense memory pressure they put on GPUs. A plethora of GNN systems was proposed to tackle these challenges [9, 20, 25, 27, 34, 41, 43, 51]. They express GNN workloads primarily as advanced matrix multiplications and rely on GPUs to execute them. When GPU memory is insufficient to host the entire matrices and the intermediate results, one either resorts to distributed processing [20, 51] and/or spilling techniques [20, 41] that load/offload data from GPU accordingly.

What makes GNN training so hard to scale, and why do we need these dedicated systems for GNNs? 1. Graph data are irregularly shaped and non-IID, differentiating them from regular IID data modalities such as text and images, for which that state-of-art deep learning frameworks were designed. To tackle the data scalability issues, most DL frameworks employ distributed data-parallel schemes [1, 38]. However, data parallelism no longer applies to graph data: graph partitions are not independent, and the training process requires cross-partition communications, depending on the input graph structure. 2. On the other hand, neural network back-propagation requires caching intermediate data during forward propagation. Depending on the graph data, these caches could be huge; Unlike models for IID data such as CNN or RNN, where the shape of input data is normalized and uniform, GNNs are highly input-dependent. GNNs are extremely hard to accommodate, as workloads are highly versatile and vary significantly in scale. 3. the "neighborhood explosion" problem [5] is also tricky to bypass; the data dependency grows exponentially as the number of GNN layers grows, posing challenges on both efficiency and scalability.

In this paper, we make a critical observation that **many of the GNN's challenges are, in fact, challenges of managing, moving, and handling the underlying graph data.** Nonetheless, existing custom GNN systems try to mix and solve the graph data and deep learning challenges holistically and couple them as one. We observe several shortcomings of that worldview: 1. many of these systems "reinvent the wheel" of much work done in the database world on scalable graph analytics engines. 2. they often tightly couple the scalability treatments of graph data processing with that of GNN training, resulting in entangled complex problems and systems that often do not scale well on one of those axes. GNN workloads, though drastically different from regular DNN workloads in data access patterns, are not too far away from non-NN graph analytics such as PageRank. As pointed out by prior work [27], most of the popular GNNs can be expressed under extended versions of graph programming models such as Gather-Apply-Scatter (GAS). Scaling "shallow" graph analytics is not a new topic: many graph

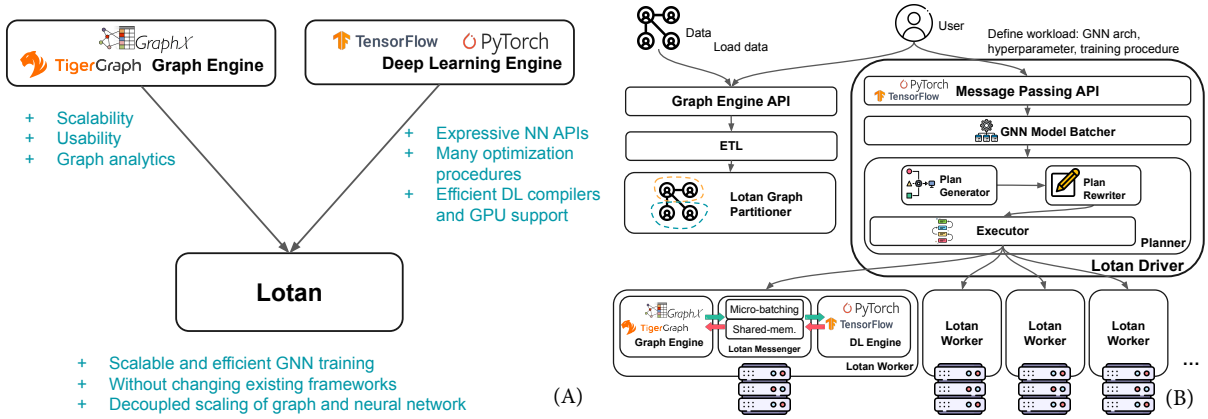

Figure 1: (A) Lotan bridges the gap between graph systems and DL systems. (B) Architecture of Lotan.

data systems were designed for that purpose [12–14, 28]. However, to the best of our knowledge, none of these graph analytics systems provide general GNN support, nor do they handle DL operations, which, nowadays, are better reserved for frameworks such as TensorFlow and PyTorch. Both stacks of software are needed: graph systems for graph challenges and DL systems for DL challenges. Our work trys to bridge the gap between them: Figure 1 (a).

System Desiderata. We envision a scalable GNN system to have the following desiderata: 1). Decoupled Scaling: scale graph and neural network parts by reusing existing industrial-strength tools for each. 2). Usability: Retain the ease of specification APIs of both graph and DL tools. 3). Non-disruptive Integration: No changes to internal code of those tools. 4). Speed and Accuracy: Fast runtimes without sacrificing DL accuracy.

In this paper, we seek to answer a fundamental systems question: *How far can we go by pushing existing systems' limits without modifications?* We propose a novel information system architecture for scalable GNN training with the **decoupling of graph and neural network**. Much like the famous **decoupling of compute and storage** in cloud computing, this decoupling enables us to tackle each side individually and enables them to scale independently. We carefully pick apart the graph and neural network dataflows in GNN training and re-imagine them as a "query plan" represented in our new intermediate-level *global operator graph*. We dispatch the execution plan to existing graph analytics systems and DL systems without needing to modify their internal code. We built a prototype distributed system called Lotan.

Overview of Lotan. Figure 1 (B) illustrates our system architecture. The user interacts with Lotan through the APIs to write their GNN workload. Our Planner then compiles it into graph and neural network operations and dispatches them to their separate execution Engines, which are existing graph and deep learning systems. Our Messenger handles the coordinates and communications between the Engines. This way, Lotan can preserve all functionalities of the execution Engines, especially the graph data management functionalities such as graph manipulation, partitioning, transaction, and other non-NN graph analysis methods.

Additionally, Lotan provides a series of system optimizations to increase the runtime performance. The most important two are GNN-centric graph partitioning and GNN model batching.

GNN-centric graph partitioning. Distributed graph processing comes with the problem of graph partitioning. To this end, we propose a GNN-centric graph partitioning scheme and the corresponding Reverse Graph Propagation scheme for GNN training. Our method works by taking account of the asymmetry in data size between forward- and backward propagation. We will describe it in detail in Section 5.1.

GNN model batching. GNN models, like other DL methods, require extensive hyperparameter tuning, which involves training multiple models on the same dataset. These models overlap extensively in their data access patterns, and opportunities exist because data access is quite costly for GNNs. We propose the first GNN Model Batching technique to improve GPU utilization and reduce runtime for GNN model selection workloads. As far as we know, Lotan is the first system to optimize for the GNN model selection/hyperparameter tuning workloads and the first to explore model batching for GNNs. We will discuss this technique in detail in Section 5.2

Our contributions can be summarized as follows:

- To the best of our knowledge, this work is the first attempt to bridge the gap between existing graph data systems and deep learning systems and the first to formally decouple the scaling of graph and neural networks in GNN training. It is also the first to utilize existing Graph data systems to tackle the scalability challenge that many GNN systems face while offering independent scaling of graph and NN that could open new GNN design freedom for the researchers.
- We re-imagine large-sale GNN training from a data management standpoint and unpack the dataflows into a "query plan" representation. We then devise novel query rewriting and optimization techniques techniques to improve scalability and efficiency.
- We also propose one of the first GNN-centric graph partitioning schemes to reduce graph node replication and communications during GNN training.

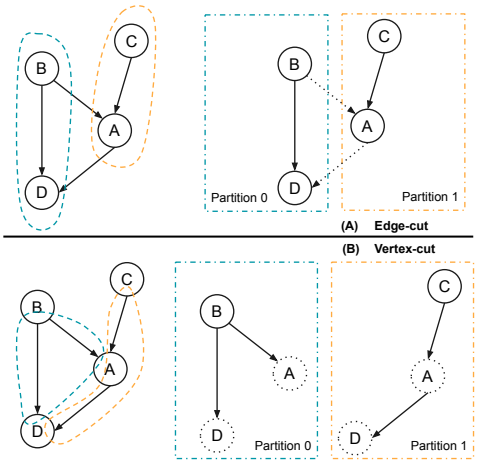

Figure 2: Two graph partitioning schemes.

- Furthermore, Lotan is the first GNN training system to treat model selection workloads holistically and explore model batching techniques to improve GNN training throughput.
- Last but not least, we did extensive empirical evaluations to compare Lotan to prior industrial-strength systems and studied the impact of our optimizations, showing our system's higher scalability and better runtime figures on multiple workloads.

2 BACKGROUND

2.1 Graph Neural Networks

Graph Neural Networks (GNNs) are neural networks on graph data. In a nutshell, a GNN always tries to summarize the graph structure and/or the graph properties into a compact numerical representation called the embeddings. GNNs can be categorized into spectral-based and spatial-based methods [45]. Spectral-based methods have roots in graph signal processing and rely on the graph Laplacian and Fourier transform for generating embeddings. Spatial-based methods are typically the applications of neural networks such as RNN, CNN, and GAN on graph data, with modifications to account for graph structure. The spatial-based methods are the more popular of the two categories [45] and will be the main focus of our system.

It is important to note that a GNN model can be ultimately expressed as a combination of graph processing (in the form of a modified Gather-Apply-Scatter programming model [27]). This is the basis of how our system attacks the problem; we compile a GNN training task into a global computational graph composed of graph operators and neural network operators and use existing systems for execution. More details on these concepts are in Section 4 and Section 3.2.

2.2 Distributed Graph Partitioning and Processing

GNN workloads still fall in the graph processing/analytics regime, as they resemble many classical graph processing problems and share very similar data access patterns. To tackle the many similar challenges in non-GNN graph data analytics, non-GNN graph data systems [6, 12, 14, 26, 40] rely on distributed processing, and a critical question is graph data partitioning.

Generally speaking, graph partitioning schemes can be classified into edge-cut and vertex-cut types. It is beyond the scope of this paper to fully cover the entire landscape of graph partitioning, so we only introduce the bare minimum background before we propose our own GNN-centric graph partitioning scheme in Section 5.1. Interested readers are directed to other literature on graph partitioning [3].

Edge-cut . Edge-cut partitioning affixes locations of the vertices, and the edges at partitioning boundaries are replicated (or need to be remotely fetched when needed). Figure 2 (A) illustrates this scheme. In Gather-Apply-Scatter workloads, the messages generated at vertices are sent across the partitioning boundary, resulting in cross-partition communication.

Vertex-cut. Vertex-cut partitioning is the alternative to edge-cut; it focuses on the edges and fixes their locations. As a trade-off, the vertices at the boundaries need to be replicated or at least remotely fetched when needed.

Note that a certain degree of cross-partition communication is inevitable, depending on the quality of the graph partitioning algorithm and the characteristics of the underlying graph. Graph data typically contain much more edges than vertices in the real world. Therefore vertex-cut partitioning, which avoids edge replications, sometimes seems more favorable in practice [14, 26, 37], and substantial performance gains were shown. A plethora of graph partitioning algorithms exist [3]. However, they are seldom designed for GNN workloads and, as a result, leave space for improvements. We will deep dive into the characteristics of GNN workloads and propose our graph partitioning scheme on top of the vertex-cut partitioning scheme in Section 5.1.

2.3 GNN Training Systems

Plenty of systems have been proposed to tackle the efficiency and scalability challenges brought by GNNs. Generally speaking, there are two main camps of GNN system research: First is the scalability camp, which aims to tackle the scalability issues of full-batch GNNs [20, 39, 52]; they are usually distributed systems and focus on providing the capability to run GNN workloads that fail on other systems. Second is the efficiency camp, which mainly focuses on runtime speed and does not often talk about scalability issues; they usually assume that the entire workload can comfortably fit in GPU memory/main memory [25, 27, 29, 34].

We summarize the comparisons between these systems in Table 1, and we will discuss them in more detail in Section 8. We evaluate these systems by several axes: 1. License, whether the system is open-sourced and usable for tests. 2. GPU, whether the system has GPU support. 3. Distributed, whether the system support distributed processing. 4. Sampling, whether the system targets full-batch or mini-batch GNN training. 5. Memory Hierarchy, whether the system is secondary storage-aware. We leave the performance tests and numbers to Section 7. In this paper, we argue that many of these systems are reinventing the wheel with custom-built graph data systems and are still facing scalability issues with larger

Table 1: Comparison of Lotan with prior art on key capabilities.

	License	GPU	Distributed	Sampling	Memory Hierarchy
Lotan	Open	✓	✓	Full	Disk-aware
DGL/DistDGL [43]	Open	✓	✓	Both	GPU-only
AliGraph/graph-learn [52]	Open	✓	✓	Mini-batch	GPU-only
GraphScope [47]	Open	✓	✓	Mini-batch	GPU-only
Sancus [34]	Open	✓	✓	Full	GPU-only
PipeGCN [42]	Open	✓	✓	Full	GPU-only
Dorylus [39]	Open	✗	✓(Serverless)	Full	N/A
ROC [20]	Open	✓	✓	Full	DRAM-aware
P^3 [10]	N/A	✓	✓	Mini-batch	GPU-only
DeepGalois [16]	N/A	✗	✓	Full	DRAM-only
Pytorch Geometric [9]	Open	✓	✗	Both	GPU-only
NeuGraph [27]	N/A	✓	✗	Full	DRAM-aware
PaGraph [2, 25]	Open	✓	✗	Mini-batch	DRAM-aware
MariusGNN [41]	Open	✓	✗	Mini-batch	Disk-aware

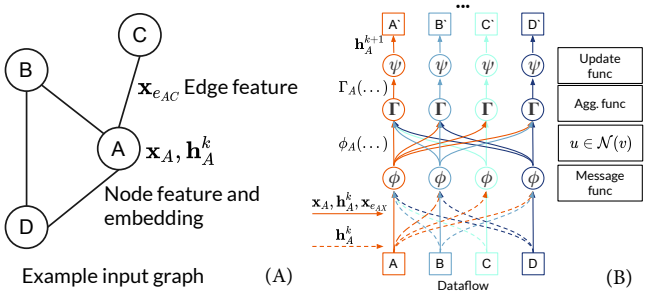

Figure 3: (A) An example input graph to a spatial-based GNN. (B) Dataflow diagram of a message passing GNN.

datasets or models. Lotan is more closely related to the scalability camp, but it differs from the prior art in technical contributions and our architecture design, which can utilize existing, established systems.

3 GNN APIS AND PROGRAMMING MODEL

3.1 GNN Interface

The first system issue we need to address is *how do we express a GNN model in a standardized way?* One of the most common and general abstractions is known as the Message Passing interface [11]. It is also widely adopted in GNN system literature and the de-facto standard. The Message Passing interface defines a GNN using *update rule*, an equation that tells us how to update a graph node's embedding:

$$\mathbf{h}_v^k = \psi(\mathbf{x}_v^k, \prod_{u \in \mathcal{N}(v)} \phi(\mathbf{h}_v^{k-1}, \mathbf{h}_u^{k-1}, \mathbf{x}_{e_{vu}})), \tag{1}$$

where ψ, ϕ, Γ are potentially learnable and differentiable functions, Γ is further required to be commutative and associative. ψ is called the update function, ϕ the message function and Γ the aggregate function. Note Equation 1 covers primarily spatial convolutional GNNs, on which we focus. Some other forms of GNNs, such as spectral-based methods [45], cannot be easily and efficiently expressed in the same framework. We leave the question of how to support those GNNs in future work.

Equation 1 is the interface we expose to the user. They will need to define the three functions ψ, ϕ, Γ. Depending on the nature of these functions, our system can do plan rewriting and optimizations to boost performance. More details are in Section 4.2. Figure 3 (A) shows an example input graph and Figure 3 (B) shows the conceptual dataflow of a GNN being learned on it.

Batched Message Passing. In practice, we find it is much more beneficial to rewrite Equation 1 in a batched, vectorized format, especially for better utilization of GPU:

$$\mathbf{H}_v^k = \Psi(\mathbf{X}_v^k, \overset{*}{\prod} \Phi(\mathbf{H}_v^{k-1}, \mathbf{H}_u^{k-1}, \mathbf{X}_{e_{vu}})), \tag{2}$$

where $\mathbf{H}_v^k, \mathbf{X}_v^k, \mathbf{H}_v^{k-1}, \mathbf{H}_u^{k-1}$, and $\mathbf{X}_{e_{vu}}$ are all matrices which are batched forms of their corresponding vectors, they have shape $B \times \_$, where B is the batch size and $\_$ is the dimension of the respective vectors. Ψ, Φ, Γ^* are the batched (vectorized) form of functions ψ, ϕ, Γ.

3.2 Lotan's Internal Programming Model.

It is not obvious how to parallelize Equation 1, mainly due to the neighborhood aggregation steps that are only native to graph processing systems. Some prior work re-cast the equation into bulk linear algebra [9, 43, 51, 52]. However, they often encounter huge scalability issues due to the potentially colossal graph size and the resulting sizes of matrix multiplications. On the other hand, it is not unfamiliar to see existing GNN systems using Gather-Apply-Scatter (GAS) abstraction or its extension [27] to translate a GNN training workload expressed as Equation 1 into an executable plan. However, none of these abstractions capture the potentially costly data transfer operations between the graph and DL execution backend Engines. We need to account for these operations correctly so that we can better understand the problem. Toward this goal, we propose a new programming model that roots in the decoupling of compute and storage and the separation of graph and NN. Our abstraction involves three main stages: 1. Graph propagation with Scatter-Gather-Collect, 2. DL propagation with ApplyEdge-Aggregation-ApplyVertex, and 3. Pipe and Join. To implement these abstractive operators, we build them upon existing operators in the graph processing and deep learning Engines.

Scatter-Gather-Collect. During this stage, the Graph Engine does a regular Scatter and Gather as in GAS for the graph propagation portion of a GNN. Instead of Apply in GAS, here it is followed with a Collect operation, where the graph engine, depending on the nature of the GNN, collects and packs relevant data to hand over to the DL engine. Conceptually, this stage is primarily for implementing the neighborhood scope $u \in \mathcal{N}(v)$.

ApplyEdge-Aggregation-ApplyVertex. In this stage, the DL Engine receives data from the graph engine and applies the user-defined GNN functions to the data. ApplyEdge is the per-edge function ϕ; similarly, ApplyVertex is the per-vertex function ψ . Aggregation is the neighborhood aggregation function Γ.

Pipe and Join. We need operations for data transfer at the Graph and DL Engine's boundary. From the Graph Engine to DL Engine,

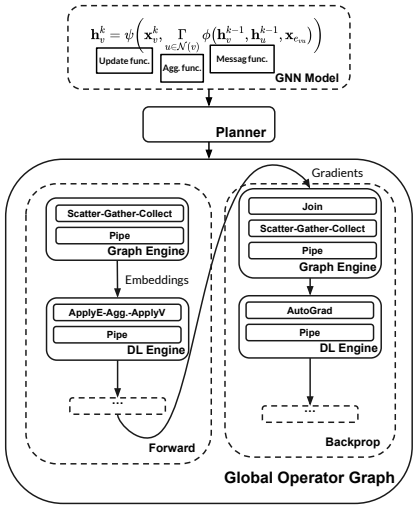

Figure 4: Global operator graph of end-to-end GNN training.

we need a Pipe operation that, as the name suggests, pipes data into the DL engine and pipes the results back to the graph engine. Then within the graph engine, a Join operation is needed to incorporate data piped back from the DL engine, as the order of data may not be preserved. We will cover these in detail in Section 4.3.

One important note is that this separation of stages is not fixed; some operations can be eliminated, some can be re-ordered, and some can be pushed down. We will explore all these opportunities for optimizations in Section 4.2.

3.3 Global Operator Graph and Execution

With all the introduced abstractions, we can now compile an entire GNN training workload into a computational graph consisting of the operators mentioned above. Such a graph would contain both graph-related and NN-related operators. A Planner, to be discussed in Section 4.2, will generate this graph from the user input expressed in the GNN message-passing interface.

Figure 4 shows the full operator graph for end-to-end GNN training, using the operators defined in Section 3.2. Data (embeddings during the forward-propagation, and gradients during the back-propagation) is sent back and forth between the Graph Engine and Deep Learning Engine. The Graph Engine is in charge of the graph aggregation by running Gather-Scatter-Apply under the hood for both forward- and back-propagation and collects all the necessary data for the DL Engine to consume, represented by the Collect operator. During the forward-propagation, the DL Engine handles the ApplyEdge, Aggregation, and ApplyVertex functions and subsequently does back-propogation with their AutoGrad capabilities. Both Engines run independently and are unaware of each other. They can run on the same set of machines, and the operators are parallelized. To coordinate the Engines and to provide a bridge for data communication, we build a Messenger component for our system, to be introduced in Section 4.3.

4 SYSTEM ARCHITECTURE

Lotan has 3 main components: 1. External Engines, where we adopt existing graph processing systems and deep learning frameworks **without modifying them**. 2. Planner, where Lotan creates and optimizes the execution plan of a GNN training workload. 3. Messenger, where Lotan reconciles the Graph Engine and the Deep Learning Engine and facilitates efficient data transmission between them.

4.1 External Engines

These engines are what Lotan relies on and improves on for tackling many scalability challenges of GNN training. We only use these engines' public interfaces and treat them as black boxes. This way, we will not need to modify these engines and drastically increase the portability and generality of Lotan while preserving all the features provided by both Engines. Furthermore, by building Lotan, we seamlessly bring GNN training capabilities to existing graph data systems without modifying them.

Graph Engine. The Graph Engine is an external graph data system that Lotan relies on for graph-related operations and scalability challenges. It can be a graph processing system or a graph DBMS, as long as it provides public interfaces for 1. Gather-Apply-Scatter (GAS) operators. 2. Operations that export data to external systems. Furthermore, it should provide scalable solutions for large-scale graph analytics. It can gracefully provide various data system features such as data partitioning and distribution, distributed processing, fault tolerance, memory management, and disk-spilling. Most of today's graph analytics systems/graph DBMS satisfy these criteria. Examples include Spark's GraphX [14], Giraph [12], TigerGraph [6], and Neo4j [33]. We choose GraphX for our prototype due to: 1. It is open-source software and has an active user community 2. It is easy to use and piggybacks on the popular and familiar Spark ecosystem. Our approach is general easily applicable to other graph analytics engines.

Deep Learning Engine. To handle the challenge from the neural network part of a GNN, we adopt an external deep learning system/framework. We use this system for forward propagation activation computing and back-propagation gradient computing with their autograd capabilities. By using an existing deep learning framework, we automatically make available the rich deep learning libraries and GPU support such a framework comes with. Furthermore, such a system can offer an out-of-box solution for distributed model training via their data-parallel capabilities [23, 38]. TensorFlow and PyTorch are both prominent examples of such a system and both are applicable. We choose PyTorch for our prototype due to its dominant popularity in the GNN community.

4.2 Planner

At the heart of Lotan is the Planner, inspired by query planners ubiquitous in data systems. Close to the concept of a DBMS query optimizer/planner, we need to weigh the potential query plans and choose the optimal one. The general idea is to assign relative costs for each stage of the execution and then determine the final cost estimate of a plan by summarizing the stage costs. However, in this case, the plan search space is much more limited, and it is favorable

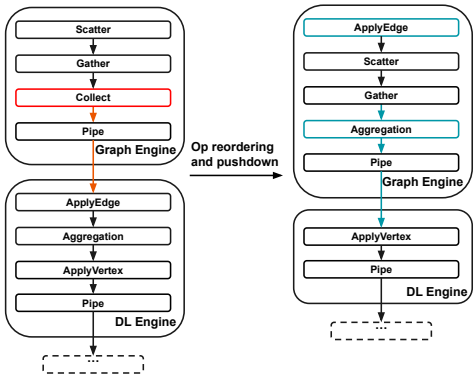

Figure 5: An example of plan rewrites.

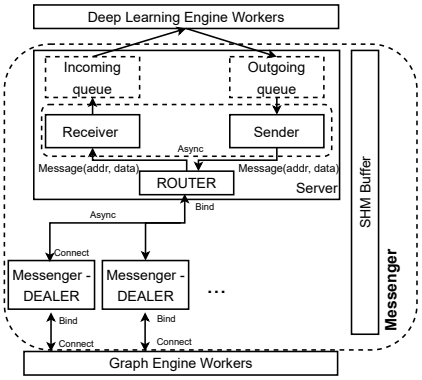

Figure 6: Messenger architecture.

to do operator pushdowns whenever possible. Therefore, we find simple heuristics sufficient and no sophisticated cost estimation is needed. To complete the study, we still try to model the costs, but primarily for curiosity and a deeper understanding of the problem. We will also verify some of the observations from our cost models in experiments.

Plan Generation and Rewrites. Plan generation is usually trivial, as GNN training comprises largely sequential stages, as Figure 4 shows. However, depending on the nature of the GNN model, the execution plan can be rewritten. We only consider the two most obvious cases of plan rewriting: operator reordering and pushdown.

Equation 1 gives the most general definition of a GNN, and Figure 3 (B) is the most stringent ordering of the three functions from the Message function to the Aggregation function to the Update function. However, because all these functions can be neural networks, they can only be handled by the DL Engine. Both the Message and Aggregation functions require neighbor information; we will also have to collect all the edges, features, and embeddings in the Graph Engine and ship them to the DL Engine. For this general case, our operator graph writes as Figure 4. This is an expensive plan due to the Collect operator and the size of data movement between the two engines. However, if the Message and Aggregation functions are both unparameterized and therefore do not require training, we can push down these functions to the Graph Engine and drastically save time. Figure 5 illustrates this scheme. We will evaluate the contribution of the plan rewrites with experiments in Section 7.2.1 and see that it contributes substantial performance gains.

Cost Estimation. To calculate the costs of a plan, we first evaluate the costs of individual stages respectively, then aggregate all the costs together. A stage is defined as the sub-operator graph between two boundaries of data movement. The costs are estimated using: 1. the data graph's information, such as the number of nodes and vertices and the average degree. 2. information of the GNN, such as the number of layers and the amount of parameters involved in the neural network. 3. The main and GPU RAM, network/disk bandwidth, and the number of concurrent CPU threads available (degree of parallelism). Due to space constraints, we leave the detailed breakdown of the costs to Section 6 of Appendix. Within the cost models, a few factors are associated with the specific workloads and, therefore, cannot be very well estimated. For more accurate costs, one can resort to logs of past runs of the same model and graph.

4.3 Micro-batch Processing and Messenger

One critical question of utilizing existing systems is how to reconcile them; each comes with its input/output interfaces, data formats, memory layouts, and other specifications. Further, the DL Engine heavily favors batched data input for higher utilization and throughput, while the data comes off the Graph Engine as streams to reduce memory footprint. This means we must convert the data stream to and from data batches. We also need to keep the order of data consistent during both the forward pass and backward pass stages.

To our best knowledge, this is the first time the data movement issues between graph data systems and DL systems being systematically studied. We adopt and synthesize existing techniques and optimizations to solve the novel problems mentioned above. We build a component we call Messenger, shown in Figure 6. We create one Dealer per Graph Engine worker to handle the data casting and batching dubbed micro-batch processing. Each data batch is hash indexed to verify the data orders. The Dealers then connect to a Router, which forwards data to and from the message queues, on which the DL Engine workers consume. We apply a series of system optimizations to the Messenger. It uses async sockets and shared memory to communicate with the DL Engine with minimal throttling.

5 SYSTEM OPTIMIZATIONS

5.1 Reverse Graph Backpropgation and GNN-centric Graph Partitioning

Graph partitioning is a vital part of distributed graph processing, as it dramatically impacts the amount of data replication and communications. Existing graph partitioning schemes are mostly not designed with GNNs in mind, resulting in suboptimal performance. We propose a novel graph partitioning and training execution scheme for GNNs, named Reverse Graph Backpropagation (RGB). This technique applies to vertex-cut-based Graph Engines [13, 14] such as GraphX, which our prototype of Lotan is based upon. Our method is based on two key observations: 1. Neural network training consists of a forward- and a back-propagation phase; the two

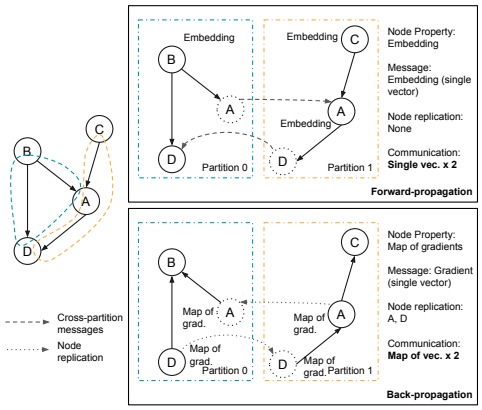

Figure 7: Regular 1D source hash partitioning and dataflow.

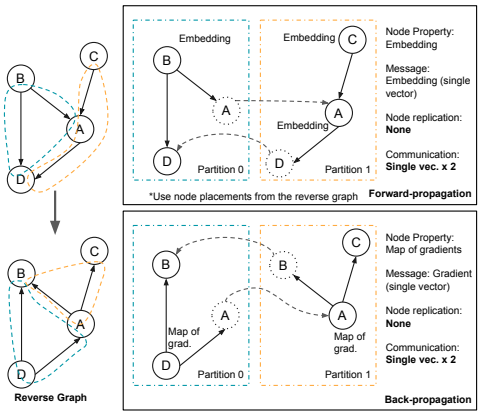

Figure 8: GNN-centric Graph Partitioning and dataflow.

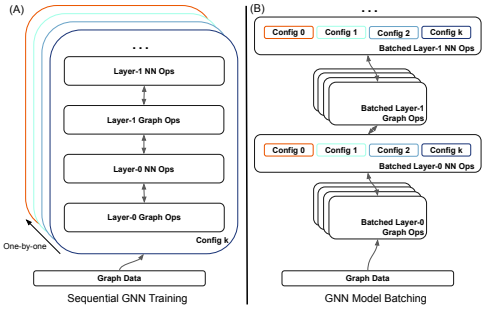

Figure 9: (A) Sequential training (B) Model Batching.

phases have inverted dataflow. 2. During GNN training, graph node data are updated, and the data size changes between phases, which leads to an asymmetry in replication costs.

We start from the well-accepted hash-based 1D edge partitioning [37], where we hash partition all the nodes and then colocate all edges based on their sources. Figure 7 illustrates the strategy. During forward propagation, each node's property and messages they send are its node embeddings, which are 1D vectors. No node replication happens, but two cross-partition messages take place. During the back-propagation, the data flow is inverted. However, each node updates its properties to gradients returned from the DL Engine (such updates happen in partition and do not incur cross-partition communications). These gradients are hash maps of vectors and, compared to the embeddings, are d (node degree) times larger. For a realistic graph, the average degree can easily be around 100. Because dataflow is inverted and the partitioning is not, heavy node replications and cross-partition communications happen.

To address this asymmetry issue, we propose our novel GNN-centric Graph Partitioning and the way to backpropagate through it, described as follows:

(1) Create a reverse graph (each edge reversed) from the original graph.
(2) Do a regular hash partitioning on the reverse graph: first, hash partition all the nodes and place them; second, partition the edges based on their sources so that all edges originating from the same source colocate in the same partition.
(3) Finally, we partition the original graph in the same manner.
(4) We run the forward propagation as usual on the original graph. However, we run the back-propagation on the reverse graph.

This way, there is no node replication during back-propagation, where the most significant bottleneck could arise. Depending on the circumstances, communication costs might increase for forward propagation but are offset by back-propagation savings. We keep the node placements consistent between phases, or extra cross-partition communication will incur. Figure 8 illustrates our approach. Regarding cross-partition communications, we only have single vectors instead of collections of vectors. We used a directed graph as an example, but the same logic still applies to undirected graphs.

5.2 GNN Model Batching

GNNs, like any other neural network, rely on careful and extensive hyperparameter tuning for the best accuracy performance. Consequently, the workloads are often multiple-model explorations. Each model has its set of neural network hyperparameters. When running hyperparameter tuning workloads, existing systems take a sequential approach: training them individually. Figure 9 (A) shows such an approach. There is wasted potential for improvements: 1. Models in a hyperparameter search workload share identical data access patterns, and re-using these routines can amortize the overheads. 2. Many GNN workloads have relatively low neural network components, often leaving the GPU underutilized. For DL methods on IID and Euclidean data, many systems [30, 50] have been developed to optimize for model selection workloads. However, these techniques do not apply as they assume IID data.

To address these issues, we propose GNN Model Batching. Model batching [32] is a technique to increase GPU utilization for IID models. To our best knowledge, we are the first to explore the same possibility for GNNs. We devise a model batching scheme to combine the models within a hyperparameter search workload. Figure 9 (B) depicts it. We run multiple models simultaneously on the model-batched version of the regular graph and NN operators.

All data transmitted between the Graph Engine and Deep Learning Engine are also batched together. The models can then share all the data access operations to amortize costs.

6 ANALYSIS OF COST MODELS

To better understand the problem, we model the various costs of GNN training: replication, computational, memory, and overheads. As mentioned earlier in Section 4.2, these models are not used in the Planner and only for deeper understanding and further experiment evaluation. Due to space constraints, we leave the tedious details and equations to Appendix. We now present a summary of two key observations about our cost model that we will later validate empirically.

6.1 Testable Observations

Effect of Number of Partitions. The number of partitions interplays with system performance in two ways: 1. for more data, more partitions are required to reduce memory pressure. 2. increasing the number of partitions will also increase the degree of parallelism and utilization because our Graph Engine uses one thread per partition.

To put it into an equation. We have the total computational cost for an execution plan with partitions:

$$W_P = \frac{W}{P} \max(\frac{P}{ML}, 1) + f_{overhead}(\frac{P}{ML}), \tag{3}$$

where W is the total amount of work (unit: time), $\frac{W}{P}$ becomes each partition's amount of work, $\frac{P}{M}$ is the total amount of tasks each machine gets, $\max(\frac{P}{ML}, 1)$ is the total amount of rounds each machine executes, depending on the degree of parallelism. Without losing generality, assume $f_{overhead}$ follows a monotonic increase along with P. We can then reason that as the number of partitions P increases; the overall runtime would first decrease and then increase.

We see precisely this behavior in Section 7.2.3. Due to the intertwined effects of this one parameter, the runtime behavior becomes non-linear and difficult to capture with simple cost models. Instead, we use rule-based heuristics to tune the number of partitions: we set it to be the same as the total number of CPU cores of the entire cluster unless more partitions are required to alleviate the memory pressure. Fortunately, GNN workloads are highly predictable in runtime and resource consumption. If necessary, our system can always do test runs (for 1 or 2 epochs of training is more than sufficient) to figure out the optimal config setting.

Effect of Model Batching. The intermediate embedding/gradient size of a GNN model greatly impacts runtime performance. Because of GNN Model Batching, Lotan can have inflated intermediates sizes. To be more precise, the intermediate sizes will be multiplied by the model batch size. Consequently, for model batching, we expect to see a scaling up when increasing model batching size due to higher utilization until the returns diminish due to overheads. We run experiments and show the results in Section 7.2.4 and will see the expected behavior.

7 EXPERIMENTS AND EVALUATION

Prior Art. Out of the distributed GNN training systems discussed in Section 2.3, we show comparisons to the SOTA: DistDGL [51], AliGraph [52], and Sancus [34]. We excluded all systems that do not support distributed training and those without public release. Despite the best effort, we could not set up and use ROC [20], with a similar situation reported in [39]. Sancus [34] and PipeGCN [42] should be comparable systems, both with approximated processing, while others (including ours) are exact, and we pick the former for benchmarking. Note that both DistDGL and AliGraph are primarily mini-batch GNN systems. Although DistDGL can run full-batch training, it fails almost all our workloads. Therefore, we use it with the mini-batch setting. Mini-batch training is mathematically different from full-batch training. But we put in our best effort for a fair comparison by tuning their mini-batch for their advantage and using standard benchmark metrics no matter the training algorithm.

Datasets. We use three of the standard benchmarking datasets from OGB [17], which has become the go-to place for graph datasets for benchmarking. We use ogbn-products, ogbn-arxiv, and ogbn-papers100M. The prior art also commonly use these datasets in their published papers. Table 2 first column shows brief statistics about the datasets.

Workloads. We define a GNN training workload with hyperparameter tuning factored in, which is an inevitable part of the end-to-end development of a GNN model. We primarily focus on two model architectures: GCN [21] and GIN [48] with various hyperparameter configurations. DistDGL and AliGraph additionally have the batch size to tune. In the corresponding literature, we found a batch from 128 to 8192 is common. We tried as much as possible to make the comparison apples-to-apples and tune the batch size beforehand for them. To not understate their performance, we set the batch size to be as large as they could handle before failing to enable the maximum possible throughput. This means mini-batch size 128 for DistDGL on ogbn-products+GCN and 8192 on ogbn-arxiv+GIN. And mini-batch size 128 for AliGraph on ogbn-arxiv+GCN. For Sancus, we can only test it on the GCN workloads as it does not have an existing implementation for GIN.

Experiment Setup. We use one cluster on CloudLab [36] with 8 worker nodes. Each node has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, and 10 Gbps network. Each worker node also has an Nvidia P100 GPU, which has 12 GB memory. We tried to get GPUs with larger memory but such resources are difficult and costly to obtain in academic clusters. Nevertheless, it is not a showstopper as Lotan's scalability is also agnostic to the underlying hardware. Furthermore, even with larger GPUs, workloads can still scale beyond GPU memory capacity and would not change our observations. All nodes run Ubuntu 20.04. We use Spark 3.2.0, Pytorch 1.10, and CUDA 11.0.

7.1 End-to-end Performance Study

We use a 3-layer GCN with a fixed hidden layer size of 256, as described in [17], dubbed GCN. We also include a variant of it with hidden size 512, which we call GCN-Large, to further distinguish between Lotan and Sancus. We skipped DistDGL and AliGraph with GCN-Large due to their crashes or much longer runtimes, and

Table 2: End-to-end test results. TLE: time limit exceeded. *For these tests, all models within the workload learned too slowly and are early terminated too soon. To not understate the system's capability, we make an exception and include a separate run that we run for a fixed 500 epochs and then pick the best valid accuracy checkpoint. †These tests would take unreasonable amount of time to finish, therefore we did not train it to converge and only report the throughput numbers.

Dataset Summary	Dataset	Model	System	Test Acc. (%)	Total Runtime (hr)	Throughput (epoch/hr)	CPU Util. (%)	GPU Util. (%)	Disk R/W (GB/hr)	Network (GB/hr)
#Nodes: 2,449,029 #Edges: 61,859,140 Avg. Degree: 50.5	ogbn-products	GCN	Lotan	**75.59**	63.82	16.22	49.58	1.75	6779.44	4989.52
			DistDGL	75.32	365.53	0.34	8.01	27.94	3.6	4548.72
			AliGraph	TLE	-	-	-	-	-	-
			Sancus	54.76	1.84	350.83	6.33	89.50	3.31	23086.58
		GCN-Large	Lotan	**75.89**	178.11	6.41	48.77	1.54	5917.79	3876.79
			Sancus	Fail	-	-	-	-	-	-
		GIN	Lotan	**75.75**	104.68	9.43	47.30	1.89	6735.52	4832.47
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
#Nodes: 169,343 #Edges: 1,166,243 Avg. Degree: 13.7	ogbn-arxiv	GCN	Lotan	**69.28**	1.90	924.85	18.47	5.79	4294.66	4203.44
			DistDGL	68.49	0.33	1912.50	21.39	15.80	3.45	5959.28
			AliGraph	68.60	171.44	1.59	5.35	6.73	3.48	44.46
			*Sancus	55.23	0.79	1855.67	5.98	89.97	3.00	19706.78
		GIN	Lotan	**71.22**	3.71	557.38	18.11	5.93	3123.74	4959.36
			DistDGL	43.64	0.15	1035.97	20.13	16.65	4.13	6109.54
			*DistDGL	69.26	2.76	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
#Nodes: 111,059,956 #Edges: 1,615,685,872 Avg. Degree: 29.1	ogbn-papers100M	GCN	†Lotan	-	-	0.08	25.03	0.22	2499.20	801.93
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
			Sancus	Fail	-	-	-	-	-	-
		GIN	†Lotan	-	-	0.04	24.16	0.11	2530.61	811.36
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-

since these tests would not provide extra insights. For GIN, we use one from [48] that is 4-layer. For the MLPs in GIN, we use a 2-layer MLP with dimensions $\{128, 256\}$ for the end-to-end study.

Following the standard practices in [17, 21, 48] for GNN training, we use an early termination of 10 epochs based on the validation set; that is to terminate if the validation accuracy does not increase for 10 consecutive epochs (with a tolerance of 0.01%). Further, we put a hard time limit of 48 hrs for each run. We also combine the hyper-parameters used in the papers above to form a grid search: learning rate $\in \{0.05, 0.01\}$, optimizer $\in \{Adam, Adagrad\}$, and dropout $\in \{0, 0.5\}$.

Table 2 summarizes the results of our end-to-end tests. On the ogbn-products + GCN workload, Lotan achieves 47x higher throughput than DistDGL while providing the same level of accuracy. There is no consensus from the GNN model research community on whether full-batch or mini-batch training is preferable. Further, note some of the models and systems adopted mini-batch training partly due to the scalability issues of full-batch training [4, 15]. Lotan is designed to mitigate the said issues. Our finding of full-batch training achieving the same or slightly higher accuracy than mini-batch training is in line with prior work [20, 42]. Sancus, though it runs fast, has severe issues in accuracy during our test, likely due to its approximate nature. Furthermore, it starts to fail on the GCN-Large workload, but Lotan can still scale. Lotan is also the only system to be able to handle GIN training, and all other systems fail due to GPU memory issues. Increasing GPU memory might fix their problems on these workloads. Still, it would not resolve the fundamental issues these systems have and the argument that Lotan has better scalability to handle large workloads.

On the ogbn-arxiv dataset, which is tiny, while Lotan can still provide the highest accuracy on both GCN and GIN, it no longer offers higher throughput than DistDGL. Lotan is not designed for small datasets. On the ogbn-papers100M dataset, which is one of the largest benchmark datasets available, Lotan is the only system to be able to run the workload. As far as we know, this is the first time for a GNN system to demonstrate full-graph GCN with a hidden size as large as 256 on this dataset. However, the execution is heavily bottlenecked, and a huge amount of disk spills happen. Consequently, we could not run the workload to converge in any reasonable amount of time. We only report the throughput numbers.

Figure 10 shows the learning curves for the best model out of each hyperparameter tuning workloads. On all workloads, Lotan converges fast and reaches the same or even higher accuracy as the prior art. When it comes to resource utilization. Lotan has high CPU utilization but generally less GPU utilization across the workloads. This is because Lotan only puts neural network operations on GPU and puts graph operations on CPU, while other systems put both on GPU. Except for Lotan, all other systems showed little to no disk R/W because they are not secondary-storage aware, where Lotan can utilize the disk for spilling. Sancus and DistDGL further utilizes GPU for communications, therefore resulting in even higher utilization.

7.2 Drill-down Experiments

To dig into the runtime figures, we also break down Lotan's runtime and investigate each portion's time costs in Figure 11 (A). The Graph Engine costs dominate, especially on the larger dataset. DL Engine and Pipe-Join costs are not as significant. This composition will change when we try scaling the model in Section 7.2.2.

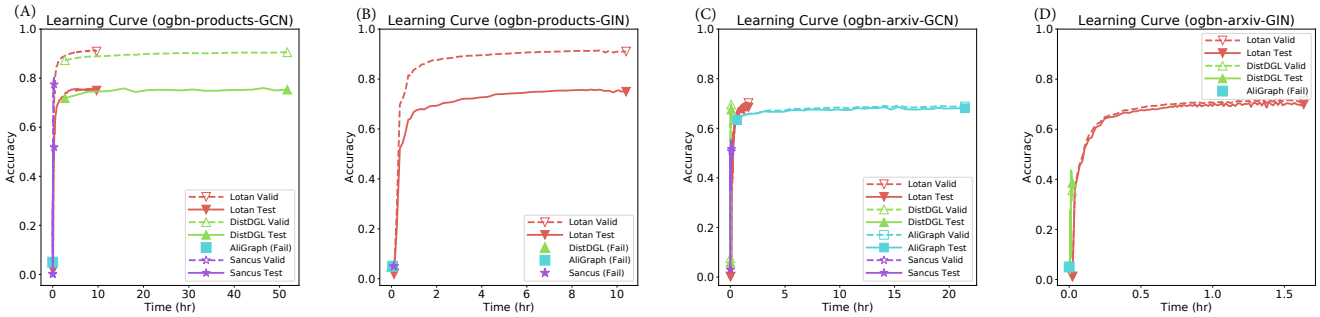

Figure 10: Learning Curves. (A) ogbn-products-GCN. (B) ogbn-products-GIN. (C) ogbn-arxiv-GCN. (D) ogbn-arxiv-GIN.

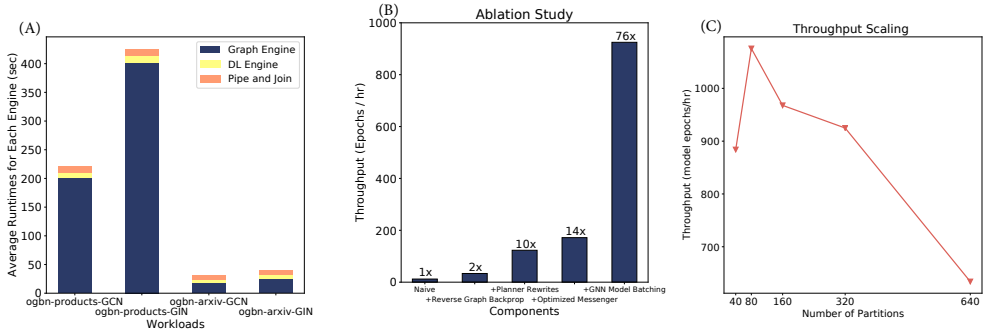

Figure 11: (A) Runtime component breakdowns. (B) Ablation study. (C) Effect of number of partitions.

7.2.1 *Ablation Study.*

To inspect each component's performance. We conduct an ablation study where we add our innovations to a naively implemented version of Lotan. We chose the ogbn-arxiv+GCN workload we used in our end-to-end tests for this test. Figure 11 (B) shows the results. We separate our technical innovations into four modules: 1. Reverse Graph Backprop (RGB) and the coupled GNN-centric partitioning scheme, as described in Section 5.1. 2. The execution plan rewrites by our Planner outlined in Section 4.2. 3. The various efforts we put into optimizing our Messenger architecture as described in Section 4.3. 4. Finally, our GNN Model Batching scheme was proposed in Section 5.2.

All of the components have substantial contributions to performance gains; Reverse Graph Backprop can boost the performance by 2x without any plan rewrites or other optimizations. With Planner rewrites introduced, we get another 5x speed-up due to the sheer amount of communication and computation saved. Furthermore, our Messenger optimizations boost the performance by another 40% by reducing overheads in I/O, IPC, and synchronization. Last but not least, GNN Model Batching contributes more than 5x speed-up due to amortized graph data access overheads. Overall, our technical innovations can boost the throughput of GNN model training by 76x, compared to a naively implemented system.

7.2.2 *Model Scalability.*

We now test Lotan's capability of scaling to larger neural network models. In practice, there are two primary ways to scale up a model: make it deeper by adding more (GNN) layers or increase the number of neurons in each layer. We call the first type depth scaling and

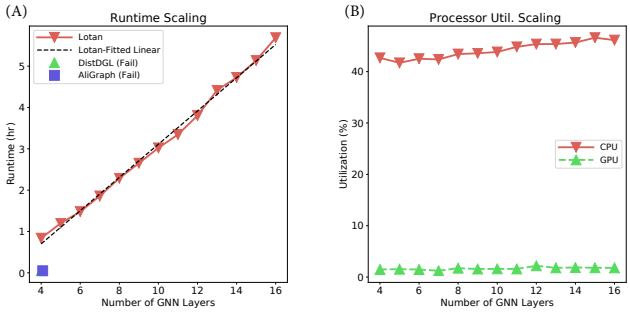

Figure 12: Depth Scaling. (A) Runtime. (B) Utilization.

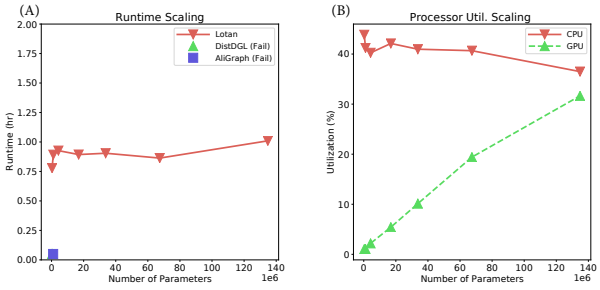

Figure 13: Width Scaling. (A) Runtime. (B) Utilization.

the latter type width scaling. Since Lotan disaggregates the graph operations from neural network operations, it has very different behavior for the two types of scaling. To thoroughly test it, we use two different workloads based on the GIN model we used earlier and trained on the ogbn-products dataset. For the depth scaling test, we test with different numbers of GNN layers ranging from 4 to 16. For the width scaling test, we fixed the model to be a 4-layer, and we varied the size of the MLP in GIN from 128 to 2^{17} (131072); we kept the embedding size also fixed as 256. This results in various models with hugely different sizes.

Depth Scaling. Figure 12 shows the results: Lotan can easily achieve almost linear scaling to 16 layers and even beyond, and there is minimal fluctuation in the processor utilizations. Note that the scaling is linear but not proportional; when the number of layers doubles, the runtime does not; this is because the scaling follows $y = kx + b$ with a non-zero intercept. It is to be expected as the amount of work grows linearly in this case, and Lotan shows resilience at scale. To the best of our knowledge, Lotan is the first system to demonstrate scale to 10+ layer GNNs with full batch training. It is important to note that the systems we compared all failed at 4 or more layers, as already discussed in Section 7.1.

Width Scaling. We show the width scaling results in Figure 13. Increasing the MLP size will not increase the amount of work on the Graph Engine, and since the GPU was under-utilized when the neural network is small, we see an almost constant scaling of Lotan. In this case, we see a dramatic increase in GPU utilization and almost constant CPU utilization. Thanks to the disaggregation of graph and neural networks, enlarging one size does not necessarily affect the other. Therefore Lotan can provide independent scaling and frees the user from scalability issues. It enables the user to design the GNN components separately and more freely. Furthermore, Lotan can gracefully handle a GNN model with 140M+ parameters with full batch training. To put it into perspective, this is the number of parameters of some early Transformer DL models have: BERT (110M) [7], and GPT-1 (117M) [35]. To our knowledge, Lotan is the first system to be able to handle this scale among the GNN systems. As shown in Section 7.1, other systems all failed at the very beginning.

7.2.3 *Effect of number of partitions.*

In our system, the number of partitions is a critical config parameter to tune. This parameter interplays with various components and subtly impacts the overall performance. We take the same ogbn-arxiv+GCN workload used in our end-to-end experiments and run it with various partitions. Figure 11 (C) shows the experiment results. As predicted in Section 6, the throughput first increases and then decreases. It increases likely due to increased parallelism at the beginning but then drops because of the overheads caused by the higher number of partitions. The network usage follows a similar trend, but the disk usage remains more stable. It is important to note that there exists a sweet spot of the parameter setting for maximum throughput. However, as discussed earlier, it is tough to model such non-linear behavior. So instead, we adopt the heuristics described in Section 6.

7.2.4 *Model Batching.*

We now inspect the effect of model batching on the workloads. For this test, we take the same ogbn-arxiv+GCN models used in Section 7.1 and create workloads with various degrees of model batching. Figure 14 shows the results. We first notice that the time costs scaling is all linear with constant overheads (manifested as the intercept), per our cost model described in Section 6. There is also a substantial gain in throughput, especially at the low degree of the model batching regime. The SGC and AAA costs scale far less steeply than the SGC costs; therefore, as the model batching size increases, the SGC costs become more and more dominant. This indicates that the biggest challenge is on graph data processing.

At a low degree of model batching (< 10), the time costs are dominated by their respective constant parts and not scaling as much with the model batch size. Therefore, the time costs only increase around 3x while the model batching size rises from 1 to 10, resulting in throughput gains. However, as the degree of model batching increases, the scaling parts of time costs dominate, and we see 2x increase in time costs when batch size increases from 10 to 20 (2x increase). Consequently, the throughput scaling plateaus out as a 2x model batched would mean 2x more runtime in this region.

8 RELATED WORK

GNN Systems. Many systems have been proposed to tackle the efficiency and scalability challenges of GNN training. Our work differs from them in our fundamental architecture design of separation of graph and neural network and our technical innovations. We have also conceptually compared them in Section 2.3. We have also already tested against some of the most related and state-of-art systems in Section 7. Most of their techniques are complementary to our work. DGL [43, 51], and PyG [9] are prominent examples of all-purpose GNN frameworks designed for generality and usability. AliGraph [52], and GraphScope [47] are GNN systems designed for industry-scale usage with an emphasis on sampling-based GNN training, which differs from Lotan's focus on full-batch training. NeuGraph [27] is one of the first systems to incorporate GNNs into an extended Gather-Apply-Scatter framework. It provides a scheduling scheme for shipping models/data in and out of multiple GPUs; its techniques are largely complementary to our work.

Other GNN systems proposed techniques ranging from memory management, communication reduction, approximated processing, and disk spilling. PaGraph [25] utilizes spare GPU memory for data caching to boost speed when the workload is relatively small. P3 [10] separates the graph metadata and graph properties and places them in a way to reduce communications. Sancus [34] proposes a communication reduction scheme via historical gradient caching and update skipping. Similarly, PipeGCN [42] uses pipeline parallelism with stale updates to speed up GNN training. They largely focus on the efficiency of GNN training via approximated processing and assumes the model and data can comfortably fit in GPU memory. PaGraph, P3, and Sancus are largely orthogonal to our work as Lotan is designed for large workloads, we do not assume an abundance of GPU memory. Dorylus [39] employs serverless functions to explore monetary cost-efficiency; our system is still for provisioned clusters, and we rely on existing data systems instead of custom-built ones used in Dorylus. Roc [20] uses main memory as swapping space to offload over-the-size data from GPU. MariusGNN [41] further proposes disk-spilling to increase

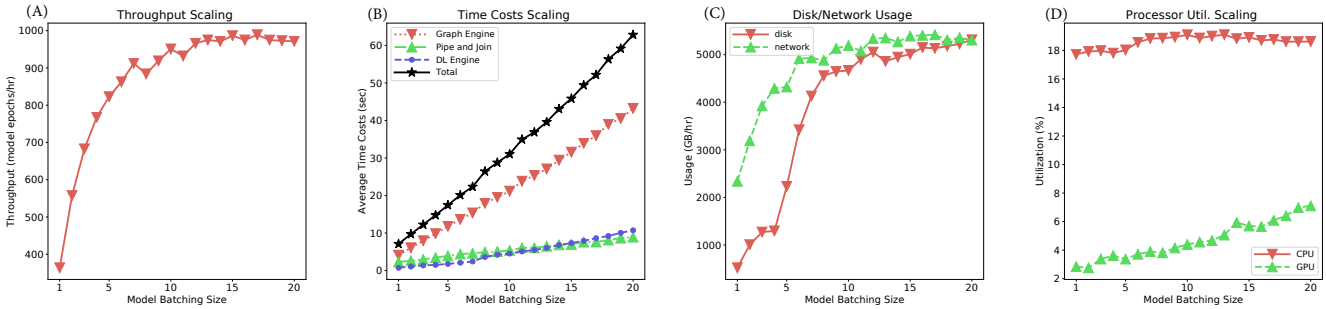

Figure 14: Scaling with GNN Model Batching. (A) Throughput. (B) Time Costs. (C) Disk and Network Usage. (D) Utilization.

the effective memory size. These techniques complement Lotan, and by employing a secondary-storage-aware graph data system, Lotan can naturally piggyback on its disk spilling capability.

Graph Analytics Systems. Prior to the GNN era, large-scale systems were built for non-GNN graph analytical workloads and data management. Ranging from graph DBMS [6, 8, 33], to classical graph analytics systems [12–14, 28], and Graph Embedding learning (not to be confused with GNN) systems [22, 29]. They are generally orthogonal to our work, as they target very different sets of workloads, and they seldom work with GNNs. Most techniques are workload-specific and not directly applicable to GNN training, but some may be complementary.

Faster and More Scalable GNNs. Ever since the first wave of GNN models arrived, algorithmic research has been active in tackling some of the scalability issues of GNNs by approximated processing and simplified architectures. This line of research is orthogonal to our work, as our goal is not to propose any new GNN model architecture but instead focus on the fundamental system challenges that will not be fixed by GNN model research alone. GraphSage [15] proposes mini-batch training and neighborhood sampling to reduce the data dependencies. FastGCN [4] runs even more aggressively IID sampling on the graph by directly controlling the number of nodes involved. SGC [44] challenges GNNs by proposing trivial two-layer architectures that reportedly could offer a similar accuracy performance. Graph coarsening techniques [18] has also been explored to preprocess and down-sample the input data.

9 LIMITATIONS, DISCUSSION, AND FUTURE WORK

Conclusions and limitations. Through careful abstracting, optimizing, and testing, in this paper, we have demonstrated that it is possible to bridge the gap between graph analytics systems and DL systems with high scalability and without modifying existing infrastructures. However, there are still two major limitations to our system Lotan: 1. Full-batch training only. Lotan currently is only optimized for full-batch training. Mini-batch training would require efficient graph sampling and filtering, posing another scalability challenge and potential query optimization questions. 2. Lotan is only designed for large workloads with large graphs and/or models. There is still some desire when the workloads are small and do fit in memory, where more leeway for caching and batching techniques exists. We leave the answers to the above limitations to future work.

Discussion. Our results showed that the graph data system bottlenecked many of our tests (see Figure 11). There are many sync barriers, costly data replications, and garbage collections. Graph data systems need to evolve to better support GNN workloads and property-rich graphs with high dimensional dense vectors. Furthermore, many recent advances from the ML Sys community, such as model hopper parallelism [30, 50] and pipeline parallelism [31], still need to be explored for GNN systems. It is unclear how these techniques designed for IID data could play with GNNs and graph data. However, with relaxed constraints and approximated processing, they might drastically amortize the runtime costs and play a vital role in the next generation of GNN systems.

Future work. Most of the GNN models and training schemes do not consider the partitioning scheme and locality of graph data. It can result in massive network communications and duplications of data. The main approach from the ML systems community has been GNN-centric graph partitioning schemes and cost reduction through caching or approximated processing. There is also a viable option of system-model co-design, where the GNN model knows about the partitioning scheme and respects the data locality. The system can then yield a more efficient execution plan by skipping or materializing specific data-accessing routines. This could be incredibly viable under transfer learning or neural architecture search contexts where GNN layers are shared across models.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283. USENIX Association, 2016.

[2] Y. Bai, C. Li, Z. Lin, Y. Wu, Y. Miao, Y. Liu, and Y. Xu. Efficient data loader for fast sampling-based gnn training on large graphs. *IEEE Transactions on Parallel & Distributed Systems*, (01):1–1, 2021.

[3] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. 2016.

[4] J. Chen, T. Ma, and C. Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *ICLR (Poster)*. OpenReview.net, 2018.

[5] J. Chen, J. Zhu, and L. Song. Stochastic training of graph convolutional networks with variance reduction. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 941–949. PMLR, 2018.

[6] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[8] X. Feng, G. Jin, Z. Chen, C. Liu, and S. Salihoğlu. Kùzu graph database management system. In *CIDR*, 2023.

[9] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[10] S. Gandhi and A. P. Iyer. P3: distributed deep graph learning at scale. In *OSDI*, pages 551–568. USENIX Association, 2021.

[11] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 2017.

[12] Giraph. Apache Giraph, Accessed Nov 2, 2022. https://giraph.apache.org/.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613. USENIX Association, 2014.

[15] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[16] L. Hoang, X. Chen, H. Lee, R. Dathathri, G. Gill, and K. Pingali. Efficient distribution for deep learning on large graphs. In *MLSys GNNSys Workshop*. mlsys.org, 2021.

[17] 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. In *NeurIPS*, 2020.

[18] Z. Huang, S. Zhang, C. Xi, T. Liu, and M. Zhou. Scaling up graph neural networks via graph coarsening. In *KDD*, pages 675–684. ACM, 2021.

[19] A. Jain, A. R. Zamir, S. Savarese, and A. Saxena. Structural-rnn: Deep learning on spatio-temporal graphs. In *CVPR*, pages 5308–5317. IEEE Computer Society, 2016.

[20] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[21] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (Poster)*. OpenReview.net, 2017.

[22] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large scale graph embedding system. In *MLSys*. mlsys.org, 2019.

[23] 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 *OSDI*, 2014.

[24] X. Li, X. Liu, L. Lu, X. Hua, Y. Chi, and K. Xia. Multiphysical graph neural network (MP-GNN) for COVID-19 drug design. *Briefings Bioinform.*, 23(4), 2022.

[25] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu. Pagraph: Scaling GNN training on large graphs via computation-aware caching. In R. Fonseca, C. Delimitrou, and B. C. Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 401–415. ACM, 2020.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349. AUAI Press, 2010.

[27] 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 *USENIX Annual Technical Conference*, pages 443–458. USENIX Association, 2019.

[28] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[29] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *OSDI*, pages 533–549. USENIX Association, 2021.

[30] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.*, 13(11):2159–2173, 2020.

[31] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *SOSP*, pages 1–15. ACM, 2019.

[32] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*, December 2018.

[33] Neo4j. Neo4j, Accessed October 12, 2021. https://neo4j.com/.

[34] J. Peng, Z. Chen, Y. Shao, Y. Shen, L. Chen, and J. Cao. SANCUS: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.*, 15(9):1937–1950, 2022.

[35] A. Radford and K. Narasimhan. Improving language understanding by generative pre-training. 2018.

[36] R. Ricci, E. Eide, and CloudLabTeam. Introducing Cloudlab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[37] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.

[38] A. Sergeev and M. D. Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *arXiv preprint arXiv:1802.05799*, 2018.

[39] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *OSDI*, pages 495–514. USENIX Association, 2021.

[40] Y. Tian. The world of graph databases from an industry perspective. *CoRR*, abs/2211.13170, 2022.

[41] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks, 2022.

[42] C. Wan, Y. Li, C. R. Wolfe, A. Kyrillidis, N. S. Kim, and Y. Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *ICLR*. OpenReview.net, 2022.

[43] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[44] F. Wu, A. H. S. Jr., T. Zhang, C. Fifty, T. Yu, and K. Q. Weinberger. Simplifying graph convolutional networks. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 2019.

[45] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 32(1):4–24, 2021.

[46] H. Xie, D. Li, Y. Wang, and Y. Kawai. Visualization method for the spreading curve of COVID-19 in universities using GNN. In *BigComp*, pages 121–128. IEEE, 2022.

[47] J. Xu, Z. Bai, W. Fan, L. Lai, X. Li, Z. Li, Z. Qian, L. Wang, Y. Wang, W. Yu, and J. Zhou. Graphscope: A one-stop large graph processing system. *Proc. VLDB Endow.*, 14(12):2703–2706, 2021.

[48] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*. OpenReview.net, 2019.

[49] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, pages 974–983. ACM, 2018.

[50] Y. Zhang, F. Mcquillan, N. Jayaram, N. Kak, E. Khanna, O. Kislal, D. Valdano, and A. Kumar. Distributed deep learning on data systems: A comparative analysis of approaches. *Proc. VLDB Endow.*, 14(10):1769–1782, 2021.

[51] 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 *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*, pages 36–44. IEEE, 2020.

[52] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow.*, 12(12):2094–2105, 2019.

A APPENDIX

A.1 Cost Models

A.1.1 Replication Factor. The vertex replication factor is a common measure of the quality of graph partitioning algorithms. It is defined as the average amount of logical presence each vertex has across partitions. However, replication factor defined this way does not consider the asymmetry during GNN forward- and backward-propagation highlighted in Section 5.1.

During forward-prop, the vertices merely have their embeddings, and replicating these vertices is relatively less expensive. However, during back-prop, the vertices are associated with maps of gradients that could be orders of magnitude larger than the embeddings. Replicating them would induce high costs. Hence vertex replication has different importance during forward-prop and back-prop. To account for this asymmetry, we define, as follows, a new metric composed of a weighted sum of the forward and backward replication costs.

Define the set of vertex partitions $\mathbb{V}_p = \{(v_i, p)\}$, each vertex v_i is accompanied by the partition number p. If a vertex is replicated, multiple tuples will be in the set with the same vertex but different partitions. Similarly we define the set of edge partitions as $\mathbb{E}_p = \{(v_i, v_j, p)\}$, where v_i is the source and v_j the destination.

Forward replication factor. During forward-prop, data flows from source vertices to destination vertices. Replication, if needed, happens during the scatter phase when the source and destination are not colocated. The same source vertex needs to be shipped over the network to each physical location where it is needed. Hence higher replication factor directly contributes to more networking needed. Define the forward replication cost R_f to be:

$$R_f := \frac{1}{n} \sum_i |\mathbb{A}_f(v_i)|, \tag{4}$$

where $\mathbb{A}_f(v_i) \subseteq \{p\}$ is the subset of partitions that v_i is mirrored to, and v_i has at least one outgoing edge that is co-located in that partition.

Backward replication cost. Similarly, the backward replication cost R_b can be defined as:

$$R_b := \frac{1}{n} \sum_i |\mathbb{A}_b(v_i)|, \tag{5}$$

where $\mathbb{A}_b$ is defined the same way as $\mathbb{A}_f$, except that instead of summing all v_i that have an out-going edge, we now sum those that have an in-coming one.

Total replication cost. Together, we take a weighted sum of R_f and R_b to obtain the total replication cost R:

$$R := \frac{1}{1+d} R_f + \frac{d}{1+d} R_b, \tag{6}$$

where d is the average degree of the graph. R_f and R_b now acknowledge the asymmetry between forward- and back-propagation. In practice, R_f and R_b can be measured using their definitions rather easily.

A.1.2 Memory Consumption. The most intensive memory consumption of the Graph Engine comes from the gather-scatter operations. We can model the relationship between memory consumption and the number of partitions.

$$M = \frac{f_{rep} P + N}{\max(\frac{P}{ML}, 1)}, \tag{7}$$

where M is the number of machines, L is the number of processing units per machine (degree of parallelism), P is the number of partitions for the data, and N is the total amount of data (in terms of the number of vertices). For simplicity, assume f_{rep} follows a linear relationship with P. Observations:

(1) At the very low amount of partitions ($P \leq ML$), increase P would increase memory footprint.
(2) When $P > ML$, the memory consumption would eventually begin to drop.

This means the memory consumption would rise and then fall as P increases.

A.1.3 Overheads. It is a non-trivial task to manage and operate on billions of objects in a distributed environment. Overheads such as object headers and one extra ephemeral copy of data are negligible in many systems. However, we cannot safely ignore them due to the "amplifying" effect of large graphs – any inefficiency would get repeated millions, if not billions of times, due to the number of vertices and edges in such a graph. Consequently, we realized our cost model has to have a better understanding and estimation of the overheads for a more accurate total cost estimation. We separate the overheads into two categories: constant and scaling. As the name suggests, the constant overheads are fixed costs associated with each specific type of operation; these could include process setup/destruction time. They usually do not scale with the amount of data and, therefore of little importance to our estimation.

On the other hand, the scaling overheads rise when the number of data increases. Note that the trend may not always be linear, as when the data scale approaches certain thresholds (disk/network throughput, RAM constrain), new overheads are induced due to network throttling, disk spilling, and garbage collections. All in all, these overheads further complicate the picture and are even harder to estimate. Our system relies on logs of past runs and specific heuristics to determine their costs.

Induced Overheads. Given the total memory consumption I, network and disk bandwidth usage J and K. And their respective resource limit $I_{max}, J_{max}, K_{max}$. The total induced overhead $O_{induced}$ can be summarized as:

$$O_{induced} := O_{memory} + O_{network} + O_{disk}, \tag{8}$$

and

$$O_{memory} = \mathbf{1}_{I>I_{max}} \cdot o_{memory}(I), \tag{9}$$
$$O_{network} = \mathbf{1}_{J>J_{max}} \cdot o_{network}(J), \tag{10}$$
$$O_{disk} = \mathbf{1}_{K>K_{max}} \cdot o_{disk}(K), \tag{11}$$

where $o_{memory}, o_{network}, o_{disk}$ are the respective functions for the overheads, and $\mathbf{1}_A$ is the indicator function defined as:

$$\mathbf{1}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases} \tag{12}$$

This means the induced overheads only exist when the required resource is above the resource limit and also scales along with the amount of resource requirement. There is no good way to estimate these functions and the resource limits beforehand, and we usually rely on runtime statistics. Even so, the estimation may still be tricky due to its non-linear nature. Hence a quick workaround is to give preference to execution plans with resource requirements below thresholds, which is implemented in Lotan.

A.1.4 Computational Cost Models. **Scatter-Gather-Collect Cost.**

$$W_{SGC}(k, forward) = l(k)N \cdot (c_0 d_{out} + c_1 R_f + c_2 d_{in}) \tag{13}$$

The first term is the scatter computation time, the second is the scatter data movement time (involving network and disk I/O), and the third is the gather and collect computation time. c_0, c_1, c_2 represent

the throughput of scatter, data movement and gather, respectively. Similarly, for back-propagation:

$$W_{SGC}(k, backward) = l(k)N \cdot (c_0 d_{in} + c_1 R_b f_e + c_2 d_{out}). \quad (14)$$

Note the asymmetry between forward and backward propagation, as highlighted previously in Section 6. Furthermore, we define an explosion factor $f_e \in \{d_{in}, d_{out}, 1\}$ to represent the potential explosion of data for specific plans. The actual value of f_e depends on the specification of the GNN, the execution plan, and the current prorogation direction.

Pipe-and-Join Cost.

$$W_{PJ}(k) = c_3 l(k) N (f_e^{out} + f_e^{in}) + c_4 N + c5 l(k) N, \quad (15)$$

The first and second terms (collected together) represent the pipe-to and pipe-from cost between the Graph Engine and Deep Learning Engine. Hence, to indicate the potential asymmetry, we have two explosion factors f_e^{out} and f_e^{in}. The third term represents the join cost of adding data back to the Graph Engine, and we always use a hash-join. The last term is the serialization/deserialization costs between different runtimes. c_3, c_4, c_5 are the pipe throughput, join operator, and serialization coefficients, respectively.

ApplyEdge-Aggregation-ApplyVertex Cost.

$$W_{AAA} = l(k) N f_e (w_0(k) + w_1(k)) + w_2(k) l(k) N, \quad (16)$$

where w_0, w_1, w_2 are the speed of the ApplyEdge, Aggregation, and ApplyVertex functions; they all depend on the GNN model specification.

Total Cost. To put everything together, we have the total cost of an execution plan written as:

$$W = \sum_k \sum_{direction} (W_{SGC} + W_{PJ} + W_{AAA} + O(N, d, l)), \quad (17)$$

where $O(N, d, l)$ is the non-negligible overheads associated with each stage. To compute the total cost, we need to gather statistics or estimate all the coefficients, compute the costs for each stage, and then sum them together.