GraphSnapShot: Masked Graph Structure Pre-training for Scalable and Fast Graph Learning

1 Introduction

In our recent research, we have developed a framework called GraphSnapShot, which has been proven an useful tool for graph learning acceleration. The core idea of GraphSnapShot is to capture and update the state of local graph structures dynamically, just like taking snapshots of graphs.

GraphSnapShot is designed to efficiently capture, store and update the dynamic snapshots of graph data, enabling us to track patterns in the structure of graph networks. This technique is useful for most graph learning tasks that relies on topology analysis or networks are constantly evolving, such as social media analysis, biological networks, or any system where the relationships between entities change over time.

The key components of GraphSnapShot is the GraphSDSampler. GraphSDSampler can efficiently capture, update, retrieve and store graph snapshots of topology while doing computation at the same time, which makes graph learning computation significantly faster.

In experiments, GraphSnapShot shows efficiency. It can promote computation speed significantly compared to traditional K-hop Sampling Methods in local topology storage and retrival (in 2-hop, 3-hop and 4-hop domain), with little loss of accuracy. It shows that the GraphSnapShot has potential to be a powerful tool for large graph acceleration.

2 Background

Local structure analysis in graph learning applications are fundamental for understanding and analyzing complex networks in various domains. This concept pertains to the idea of capturing the localized patterns and relationships within a large graph structure. Efficient graph structure learning are crucial in various applications, from social network analysis to bioinformatics, where useful hidden pattern are discovered by graph structure learning.

The GraphSnapShot framework we proposed provide insight to study graph learning by using dynamically updated local graph snapshots. GraphSnapShot is particularly valuable in large-scale networks where global analysis can be computationally intensive and less informative for certain types of inquiries. By focusing on local structures, researchers can detect community patterns, identify influential nodes, and understand local clustering behaviors in a more efficient way.

Applications of local structure learning are diverse and impact several fields:

- **Social Network Analysis:** Understanding individual or entity interactions within a network to reveal social dynamics and community formation.
- **Bioinformatics:** In biological networks, local structures assist in identifying functional modules or predicting protein interactions.
- **Recommendation Systems:** Improving recommendation accuracy by focusing on immediate user-item interaction patterns.
- **Network Security:** Analyzing local patterns for detecting anomalies or potential security breaches.

3 Motivation

Inspired by masked language modeling [STQ+19] [WGZC23], and masked vision learning [LFH+23] [HCX+21], we develop GraphSnapShot, which is a masked graph structure learning for scalable graph pretraining solution.

GraphSnapShot is the first research work to introduce masked graph structure for graph machine learning acceleration. Graph masking process involves two phases: graph sampling and removal. The efficiency of the first phase—graph sampling—has a profound impact on the overall system performance.

Traditional sampling methods in graph analysis, particularly for multi-hop domains, struggle with efficiency. Node-wise sampling methods, including well-known ones like GraphSAGE [HYL18], are not optimized for multi-hop scenarios due to the computational burden that grows exponentially with each additional hop—a phenomenon known as the "neighbor explosion." This makes them unsuitable for large-scale graphs where the capture of extended neighborhoods is crucial. Layer-wise sampling, such as in FastGCN [CMX18], although designed to mitigate this explosion by sampling nodes per layer, often loses valuable connectivity information, leading to a sparse and inaccurate representation of the graph's multi-hop structure. Furthermore, subgraph sampling methods such as Cluster-GCN [CLS+19] focus on localized computations and overlook vital inter-subgraph connections, failing to capture the broader topology necessary for a comprehensive multi-hop domain analysis.

GraphSnapShot represents a significant advancement in the field by addressing these inefficiencies. It introduces a hybrid sampling method that combines an initial comprehensive preprocessing of the graph with an intelligent dynamic sampling of multi-hop neighborhoods. This strategic approach allows for the maintenance of structural detail and computational efficiency, even as it scales to capture wider local topologies. GraphSnapShot dynamically adjusts its sampling based on the graph's evolving structure, which not only enhances the accuracy of the multi-hop neighborhood representation but also significantly reduces the computational load. Demonstrated by its considerable reduction in computation time compared to traditional methods, GraphSnapShot provides a new strategy for efficient graph masking in complex multi-hop domains.

4 Model Construction

Algorithm 1 Static Sampling and Dynamic ReSampling

function PREPROCESS(G)
    $sG \leftarrow$ STATICSAMPLE(G)
    return sG
end function
function DYNAMICSAMPLE(G, α, N)
    $num \leftarrow \alpha \times N$
    $sNodes \leftarrow$ SAMPLEFROMDISK(G, num)
    return $sNodes$
end function
function RESAMPLE(sG, α, N)
    $num \leftarrow \alpha \times N$
    $sNodes \leftarrow$ SAMPLEFROMMEMORY(sG, num)
    return $sNodes$
end function

Algorithm 2 Local Snapshot Swap

function ADD(G, $Nodes$)
    $G \leftarrow G \cup Nodes$
    $Nodes \leftarrow$ MOVETOMEMORYFROMDISK($Nodes$)
    return G
end function
function REMOVE(G, $Nodes$)
    $G \leftarrow G - Nodes$
    $Nodes \leftarrow$ MOVETODISKFROMMEMORY($Nodes$)
    return G
end function

Algorithm 3 GraphSnapShot Process

function PROCESS(G, α, N)
  $sG \leftarrow$ PREPROCESS(G)
  while ($InComputation$) **do**
    $dNodes \leftarrow$ DYNAMICSAMPLE(G, α, N)
    $rNodes \leftarrow$ RESAMPLE(sG, $(1-\alpha)$, N)
    $uG \leftarrow$ COMBINE($dNodes$, $rNodes$)
    $sG \leftarrow$ REMOVE(sG, $rNodes$)
    $sG \leftarrow$ ADD(sG, $dNodes$)
    $result \leftarrow$ COMPUTE(uG)
  end while
  return $result$
end function

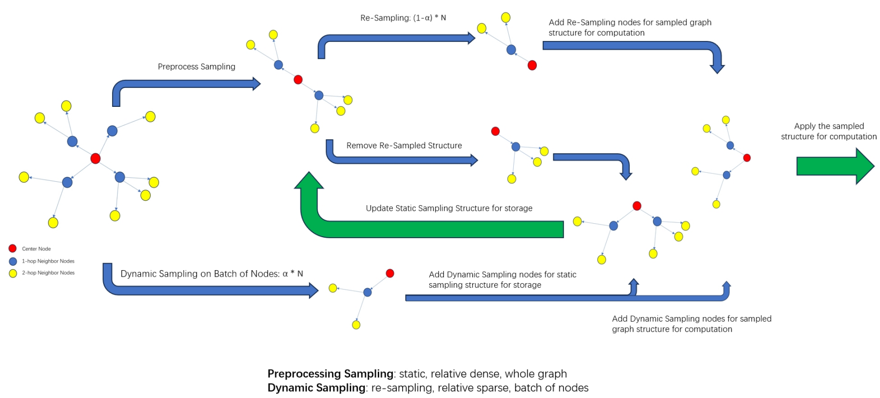

Figure 1: GraphSnapShot Model[1]

GraphSnapShot is designed to solve the inefficacy for current graph masking methods on large graphs. Typical distributed graph processing System such as Marius [MWX+21], will re-sample all the local structure each time, and load corresponding node embeddings from disk, which is time-consuming. GraphSnapShot proposing a new method for quick local structure retrieval by integrating the benefits of both static and dynamic processing. In the preprocessing phase, a static graph snapshot is sampled and stored in memory from the disk. Then, in the computation phase, the algorithm re-samples the graph snapshot, using a mix of data partially from memory and partially from disk. Concurrently, a smaller, dynamic snapshot is sampled from the disk. This dynamic snapshot is

[1] GraphSnapShot Code: `https://github.com/NoakLiu/GraphSnapShot`.

crucial as it is used to swap with the portion of the graph stored in memory. This process ensures the computation can always learn "fresh" snapshot without overwhelming the memory capacity. GraphSnapShot allows the algorithm to continuously update and adapt to changes in the graph while maintaining a manageable memory footprint, making it an effective tool for processing large, dynamic graph data.

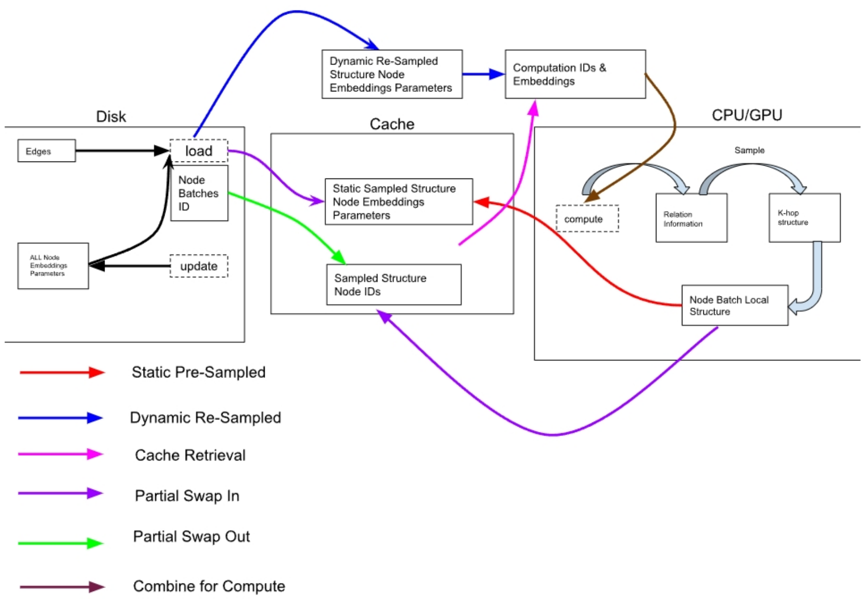

Figure 2: Disk-Cache-CPU/GPU Architecture for GraphSnapShot

In the GraphSnapShot system, disks are used to store edge and node embedding information, while computational resources (CPU/GPU) are utilized for neural network computations (such as backpropagation) and local structure masking. Additionally, the cache is employed to store the K-V pairs information of the sampled structure.

- Disk: Persistent storage of graph structure information, such as edges and node embedding parameters. Those embeddings are stored long-term and loaded into memory when needed for further processing and analysis.
- Cache: Store frequent K-V pairs of GraphSnapShot to accelerate graph learning. These K-V pairs include the identifiers of the sampled structure nodes and their embedding parameters, allowing for more efficient repeated access to the graph data.
- CPU/GPU: Computational Resources to responsible for performing local

structure computations on the graph, including neural network forward and backward propagation as well as the dynamic resampling and masking of the graph's local structure.

As for the cache retrieval and refreshing methods, we develop three methods to simulate different strategies.

The first method is FBL, which is short for full batch load. FBL is the traditional method where most graph processing systems current are deployed. Those system does not caching any information and resample graph structure each time from disk to memory for computation.

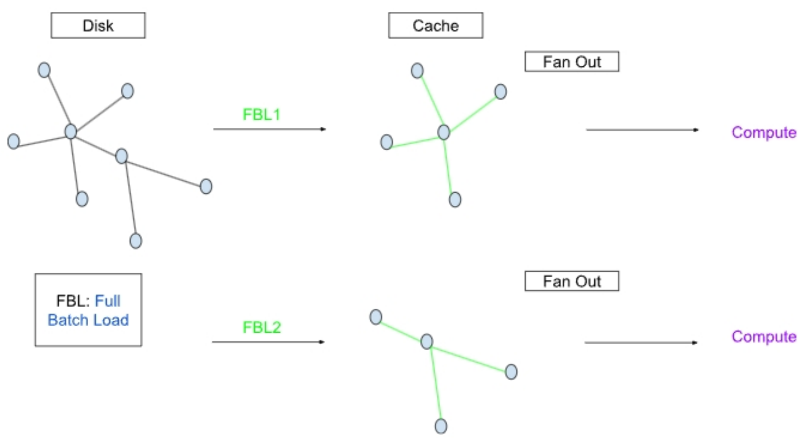

Figure 3: FBL Cache Snapshot Strategy

The second method, known as "On The Fly" (OTF), updates portions of the cache dynamically while leaving other segments of the cache information unchanged.

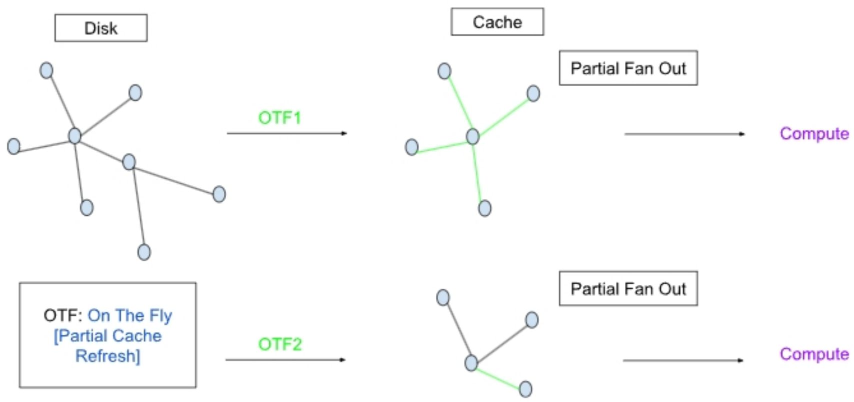

Figure 4: OTF Cache Snapshot Strategy

In detailed implementation of OTF, we design 4 different mode of it for real world application. They are (Full Fetch, Partial Fetch)x(Full Refresh, Partial Refresh).

The third method is FCR, which is short for "fully cache refreshing" in a batch. For FCR, in each batch, the the structure information are sampled from cache for computation, and between each batch, the cache is fully refreshed.

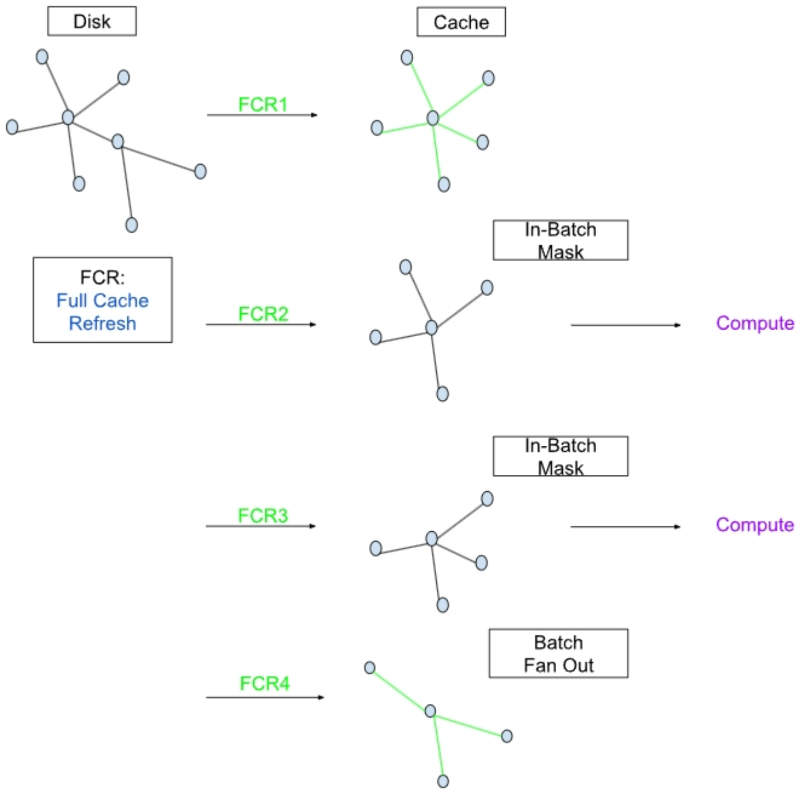

Figure 5: FCR Cache Snapshot Strategy

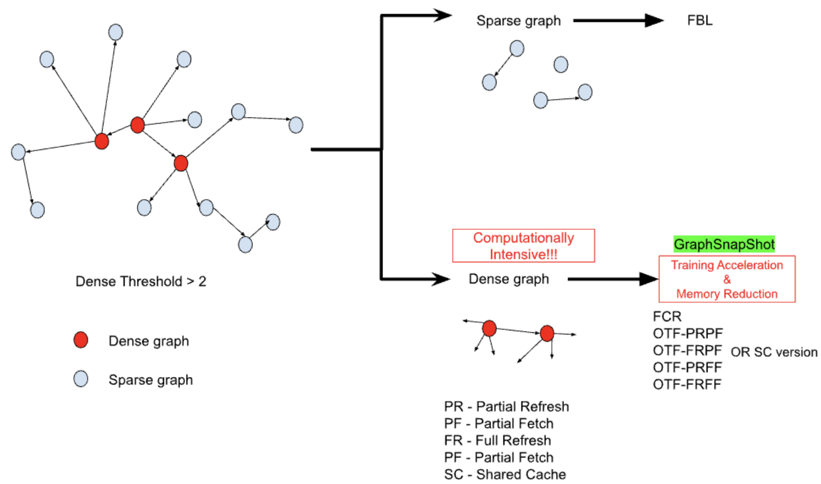

Figure 6: Graph Processing by Density

Our approach involves splitting a graph into two subgraphs based on the node degree threshold. Nodes with a degree higher than a specified threshold are categorized into a dense subgraph, while those with a degree below the threshold form a sparse subgraph. This division allows for the application of different sampling techniques to each subgraph, optimizing the computational resources and processing time.

For dense subgraphs, we utilize advanced sampling methods in GraphSnapShot such as FCR, OTF, FCR-SC, OTF-SC, etc, which are designed to efficiently manage the memory cache while maintaining the quality of the sampled neighborhood. These samplers dynamically adjust their sampling rates and cache refresh rates based on the learning phase and node centrality, ensuring efficient processing of dense regions.

For sparse subgraphs, we employ a Full Batch Load (FBL) approach, akin to DGL's NeighborSampler. This method is suitable for areas of the graph with lower degrees, where the computational overhead is inherently less, allowing for direct processing without the need for intricate sampling techniques.

We use the Deep Graph Library (DGL [WZY+20]) framework to implement new samplers in GraphSnapShot, leveraging its efficient handling of graph data structures and operations. Our implementation involves loading the graph, applying the graph splitting based on node degree, and then processing each subgraph with the respective sampling technique. We evaluated the performance of our new models on datasets in ogbn-benchmark [HFZ+21]. The results show that GraphSnapShot can significantly reduce training time and memory usage without compromising the quality of GNN training.

Bridging the Gap Between Pure Dynamic Algorithms and Static Memory Storage with GraphSnapShot: Dynamic graph algorithms, such as GraphSAGE, which require resampling the entire graph for each computation, incur significant overhead. This is primarily due to the need for constant, full-graph resampling to accommodate the ever-changing nature of dynamic graphs. GraphSnapShot addresses this issue by establishing a static snapshot at preprocessing phase and use dynamic processing to update partial snapshot at computation. This approach significantly reduces the computational burden by providing a stable, memory-stored snapshot (sG) as a baseline. GraphSnapShot ensures efficient computation and maintains the flexibility needed to handle dynamic changes in the graph.

Tradeoff Between Dynamic Sampling and Resampling: In GraphSnapShot, the balance between dynamic sampling ($DynamicSample$) and resampling ($ReSample$) plays a pivotal role in maintaining an up-to-date representation of the graph. Dynamic sampling involves integrating new nodes from the disk into memory, capturing the latest updates and changes. Resampling, on the other hand, focuses on sampling and processing existing nodes from memory, which are then rotated back to disk storage. However, if $DynamicSample$ is too large, it is close to traditional sampling, resulting in prolonged disk retrieval times. Conversely, while resampling processes memory-stored nodes, an excessive $ReSample$ may overly rely on memory data, potentially leading to a decrease in accuracy. Striking this balance is essential: moderating dynamic sampling to prevent inefficiencies and ensuring resampling maintains graph accuracy without an undue dependence on memory. In short, this is a tradeoff between optimizing memory usage (quick accessing) and model accuracy.

Local Snapshot Swap Strategy: The *Add* and *Remove* functions are designed to simulate the swap of nodes between memory and disk, reflecting the dynamic shifts between active and less active parts of the graph. This local snapshot swapping strategy is crucial for handling large-scale dynamic graphs, as it reduces the demand on memory and computational resources.

In conclusion, the design of our GraphSnapShot Process algorithm considers the complexity and variability of dynamic graph analysis. By combining static sampling with dynamic resampling, the algorithm effectively manages and analyzes graphs whose structures evolve over time, demonstrating robust performance and flexibility in handling large-scale network analyses.

Dense & Sparse Graphs Processing: Our approach shows significant improvements in computational efficiency compared to traditional methods. The dual sampling strategy for dense and sparse graphs allows for targeted resource allocation, effectively handling dense and sparse regions within the graph. By using FBL, sparse graph processing can avoid resampled on same node set; by using GraphSnapShot, dense graph processing can have smaller cache size and quicker processing time. This results in faster training times without compromising the accuracy of the GNN models. Belows are figures to visulize the sample efficiency for Dense & Sparse Processing.

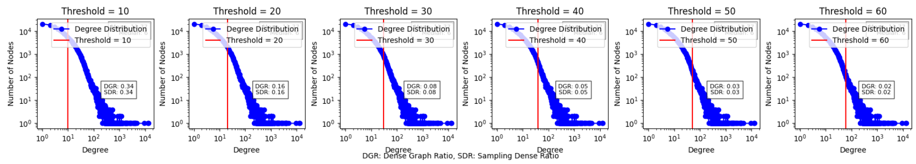

Figure 7: Dense and Sparse Processing for ogbn-arixv

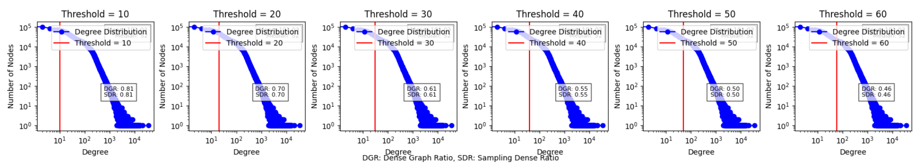

Figure 8: Dense and Sparse Processing for ogbn-products

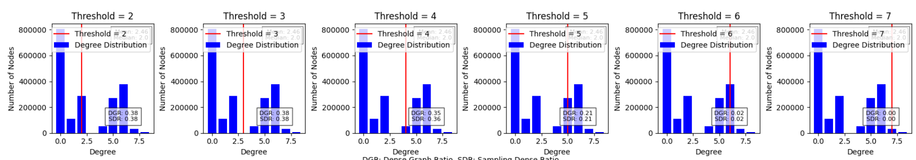

Figure 9: Dense and Sparse Processing for ogbn-mag

Training Time Reduction Analysis The provided table presents a detailed comparison of training time acceleration for various computational methods relative to the baseline method FBL. It quantitatively illustrates the percentage reduction in training times when employing FCR, FCR-shared cache, OTF, and OTF-shared cache methods across three distinct settings: `[20, 20, 20]`, `[10, 10, 10]`, and `[5, 5, 5]`. Each entry reflects the efficiency gains achieved by the respective method, highlighting how advancements in computational techniques can significantly decrease the time required for model training.

Table 1: Training Time Acceleration Percentage Relative to FBL

Method/Setting	**[20, 20, 20]**	**[10, 10, 10]**	**[5, 5, 5]**
FCR	7.05%	14.48%	13.76%
FCR-shared cache	7.69%	14.33%	14.76%
OTF	11.07%	23.96%	23.28%
OTF-shared cache	13.49%	25.23%	29.63%

The training time comparison table reveals that the **OTF-shared cache** method consistently provides the most substantial time accelerations, achieving up to **29.63%** faster training times than the FBL method, particularly in the smallest setting. This performance underscores its effectiveness in scenarios demanding quick model training. The **FCR** methods, both with and without shared caching, offer steady improvements, enhancing training speeds by approximately **7% to 14.76%**. Notably, the **OTF** method shines in smaller settings, suggesting increased efficiency in less complex computational tasks. The positive impact of shared caching across the FCR and OTF methods indicates that optimized resource management can lead to significant reductions in training times, making these methods highly beneficial in high-performance computing environments where reducing time is essential.

Runtime Memory reduction Analysis From the visualizations and data provided in the table of runtime memory reduction analysis, it is evident that the computational methods FCR, FCR-shared cache, OTF, and OTF-shared cache demonstrate significant improvements in reducing runtime memory usage compared to the baseline FBL method up to 90%.

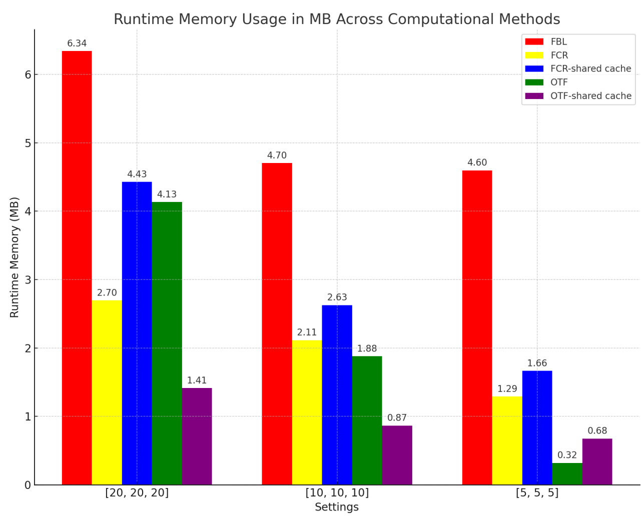

Figure 10: Runtime Memory Reduction

GPU Reduction Analysis This significant reduction in GPU memory usage demonstrates the effectiveness of the GraphSnapShot methodology. By intelligently balancing the trade-off between full storage of sparse graph regions and sampling of dense graph regions, the proposed approach greatly enhances memory efficiency, which is particularly beneficial for large-scale graph processing tasks.

Table 2: GPU Storage Optimization Comparison

Dataset	**Original**	**Optimized**	**Compression**
ogbn-arxiv	1,166,243	552,228	52.65%
ogbn-products	123,718,280	20,449,813	83.47%
ogbn-mag	5,416,271	556,904	89.72%

5 Experimental Result

5.1 Theoretial Analysis of Traditional Disk-Memory Strategy and Our Disk-Cache-Memory Strategy

In the traditional Graph System like Marius [MWX$^+$21], a disk-cache model is deployed, where the graph structure is resampled each time from the disk into the memory for computation. In our current GraphSnapShot model, we deploy a disk-cache-memory model where K-V pair for reused graph structure are stored in the cache to promote the system performance.

Theoretically, if we denote the batch processing size as $S(B)$, the cache size as $S(C)$ and the cache refreshing rate as α, and the processing speed of cache as v_c, and the processing speed of memory as v_m. Then we can get the batch average processing time for disk-memory model as $\frac{S(G)}{v_m}$ and the batch average processing time for disk-cache-memory model as $\frac{S(B)-S(C)}{v_m} + \frac{(1-\alpha)*S(C)}{v_c}$.

5.2 Experimental Analysis for Different Cache Refreshing Strategies in GraphSnapShot

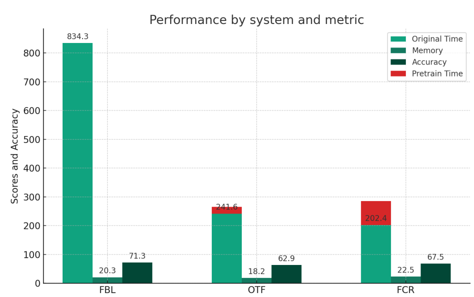

Figure 11: System Performance Evaluation

In our experiments, we assess the performance of different cache refreshing strategies, wherein the cache is either partially or fully updated, or not utilized at all. The experimental results show that the strategy of partially updating the cache yields the best performance in terms of speed and memory efficiency while achieving similar accuracy compared to the other two strategies.

5.3 Experimental Analysis of SSDReS Performance

In our experiments, we explored the efficacy of the GraphSnapShot algorithm by testing the tradeoff between static and dynamic resampling. Static fetch (proportional to fr) focuses on quick access to graph topology information stored in memory. Dynamic resampling (proportional to rr), on the other hand, is a measurement for the memory swap, it addresses the real-time changes in the graph, frequently requiring access to disk-stored data and swap in the memory to update the in-memory topology.

Analyzing the outcomes of these experiments provides insights into the algorithm's performance:

- **Accuracy and fetch rate**: Our results showed a general increase in accuracy as fetch rate increased, which shows dynamic resampling has critical roles in enhancing model precision. A higher rate of dynamic resampling implies a greater reliance on the stable structure of the graph, aiding in capturing patterns in graph topology. A relative high dynamic resampling rate is beneficial for the model's training stability and generalizability.

- **Loss and fetch rate**: Loss decreased with an increase in fetch rate, further validating the importance of dynamic resampling in improving model performance. Lower loss indicates reduced error in data fitting, suggesting that static resampling helps the algorithm to learn and predict the graph's structure more accurately. This could also imply that dynamic resampling offers a stable learning environment, potentially reducing the risk of overfitting.

- **Training Time and refresh rate**: Our experiments indicated the existence of an optimal refresh rate value between 0 and 1, which minimizes training time, and with a little loss in acuracy. We call this point as "trade-off" point between static and dynamic resampling. Over-dependence on dynamic resampling (low refresh rate) could increase the computational burden due to frequent updates required for rapidly changing graph structures, while excessive reliance on static resampling (high refresh rate) might cause the model to overly learn from limited portion of the graph structure in memory, deduce its ability to effectively learn from the entire graph structure.

5.4 GPU Reduction Visualization

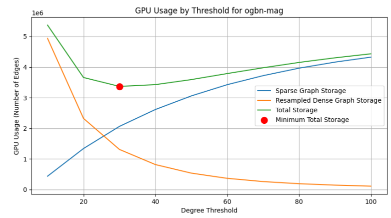

Figure 12: GPU Reduction Visualization for ogbn-mag

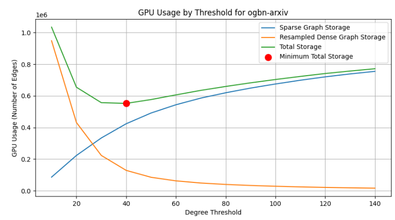

Figure 13: GPU Reduction Visualization for ogbn-arxiv

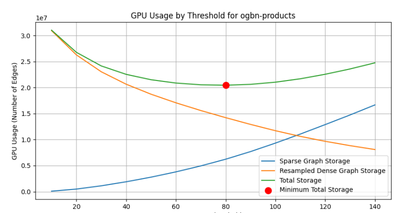

Figure 14: GPU Reduction Visualization for ogbn-products

The effectiveness of the GraphSnapShot algorithm lies in the way that it deploys disk-cache-memory architecture to store and update intermediate graph local structure (GraphSnapShot). By caching the K-V pairs of the reused graph structure, the graph machine learning is significantly accelerated. Besides, specifically, for the caching performance, there is a balance between static and dynamic resampling, which is controlled by α and $1 - \alpha$ respectively. This balance is key for handling the trade-off between using fast, in-memory data for stable and consistent performance (achieved through static resampling), and accessing data from disk to keep up with the latest updates (achieved through dynamic resampling). Fine-tuning the α parameter allows for calibration of the algorithm's focus between stability and adaptability. This adjustment is essential for achieving optimal performance across various graph analysis tasks. The

GraphSnapShot excels in processing dynamic graph data and tasks that require exploration of graph structures, particularly for applications with complex and large-scale network structures.

References

[CLS+19] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining*. ACM, jul 2019.

[CMX18] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling, 2018.

[HCX+21] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners, 2021.

[HFZ+21] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs, 2021.

[HYL18] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[LFH+23] Yanghao Li, Haoqi Fan, Ronghang Hu, Christoph Feichtenhofer, and Kaiming He. Scaling language-image pre-training via masking, 2023.

[MWX+21] Jason Mohoney, Roger Waleffe, Yiheng Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Learning massive graph embeddings on a single machine. *CoRR*, abs/2101.08358, 2021.

[STQ+19] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation, 2019.

[WGZC23] Alexander Wettig, Tianyu Gao, Zexuan Zhong, and Danqi Chen. Should you mask 15

[WZY+20] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

A Appendix

A.1 dgl Experiments

A.1.1 datasets

Table 3: Overview of OGBN Datasets

Feature	**OGBN-ARXIV**	**OGBN-PRODUCTS**	**OGBN-MAG**
Type	Citation Net.	Product Net.	Acad. Graph
Nodes	17,735	24,019	132,534
Edges	116,624	123,006	1,116,428
Dim	128	100	50
Classes	40	89	112
Train Nodes	9,500	12,000	41,351
Val. Nodes	3,500	2,000	10,000
Test Nodes	4,735	10,019	80,183
Task	Node Class.	Node Class.	Node Class.

A.1.2 Training Time Acceleration

This table provides a comparison of the training times ("training time") of different computational methods (FBL, FCR, FCR-shared cache, OTF, OTF-shared cache) across three settings (`[20, 20, 20]`, `[10, 10, 10]`, and `[5, 5, 5]`). It also calculates the time acceleration percentage relative to FBL for methods other than FBL, expressed as a percentage.

Table 4: Training Time Acceleration Through Each Method

Method	**Setting**	**Training Time (s)**	**Time Acceleration (%)**
FBL	[20, 20, 20]	0.2766	-
	[10, 10, 10]	0.0747	-
	[5, 5, 5]	0.0189	-
FCR	[20, 20, 20]	0.2571	7.05
	[10, 10, 10]	0.0639	14.48
	[5, 5, 5]	0.0163	13.76
FCR-shared cache	[20, 20, 20]	0.2554	7.69
	[10, 10, 10]	0.0640	14.33
	[5, 5, 5]	0.0161	14.76
OTF	[20, 20, 20]	0.2460	11.07
	[10, 10, 10]	0.0568	23.96
	[5, 5, 5]	0.0145	23.28
OTF-shared cache	[20, 20, 20]	0.2393	13.49
	[10, 10, 10]	0.0559	25.23
	[5, 5, 5]	0.0133	29.63

Table 5: Detailed Settings for Each Computational Method

Method	**Detailed Settings**
FCR	[20, 20, 20], alpha=2, T=50 [10, 10, 10], alpha=2, T=50 [5, 5, 5], alpha=2, T=50
FCR-shared cache	[20, 20, 20], alpha=2, T=50 [10, 10, 10], alpha=2, T=50 [5, 5, 5], alpha=2, T=50
OTF	[20, 20, 20], amp_rate=2, refresh_rate=0.15, T=50 [10, 10, 10], amp_rate=2, refresh_rate=0.15, T=50 [5, 5, 5], amp_rate=2, refresh_rate=0.15, T=50
OTF-shared cache	[20, 20, 20], amp_rate=2, refresh_rate=0.15, T=50 [10, 10, 10], amp_rate=2, refresh_rate=0.15, T=50 [5, 5, 5], amp_rate=2, refresh_rate=0.15, T=50

- **OTF-shared cache** shows the highest time acceleration at the smallest setting (`[5, 5, 5]`), with a **29.63% improvement**, highlighting its efficiency for smaller datasets.
- Both **FCR** and **FCR-shared cache** methods consistently perform better than FBL, especially in medium and smaller settings, indicating significant improvements in training time.
- The time acceleration percentages are notably higher in the smallest setting for all methods, suggesting that these optimized approaches are particularly effective for smaller datasets.

A.1.3 Runtime Memory Reduction

Table 6: Comparison of Runtime Memory Reduction Across Computational Methods. RM stands for Runtime Memory, ar for amp_rate/alpha, and rr for refresh_rate.

Method	Setting	RM (MB)	Reduction (%)
FBL	[20, 20, 20]	6.3387	0.00%
	[10, 10, 10]	4.7041	0.00%
	[5, 5, 5]	4.5963	0.00%
FCR	[20, 20, 20], ar=2, T=50	2.6962	57.46%
	[10, 10, 10], ar=2, T=50	2.1149	55.04%
	[5, 5, 5], ar=2, T=50	1.2921	71.89%
FCR-shared cache	[20, 20, 20], ar=2, T=50	4.4287	30.13%
	[10, 10, 10], ar=2, T=50	2.6272	44.15%
	[5, 5, 5], ar=2, T=50	1.6645	63.79%
OTF	[20, 20, 20], ar=2, rr=0.15, T=358	4.1327	34.80%
	[20, 20, 20], ar=2, rr=0.15, T=50	4.1924	33.86%
	[10, 10, 10], ar=2, rr=0.15, T=50	1.8782	60.07%
	[5, 5, 5], ar=2, rr=0.15, T=50	0.3207	93.02%
OTF-shared cache	[20, 20, 20], ar=2, rr=0.15, T=50	1.4149	77.68%
	[10, 10, 10], ar=2, rr=0.15, T=50	0.8663	81.58%
	[5, 5, 5], ar=2, rr=0.15, T=50	0.6762	85.29%

The reduction percentage is calculated by comparing the runtime memory usage of a method under a specific setting to the baseline method's usage at the same setting. The formula used to compute the reduction is given by:

$$Reduction(\%) = \left(\frac{BaseRM - CurrentRM}{BaseRM}\right) \times 100\%$$

where:

- **Base RM** represents the runtime memory (in MB) of the baseline method for a given setting, which is the FBL method in this case.

- **Current RM** represents the runtime memory of the current method under the same setting.
- **Baseline Setting (FBL)**: All calculations are compared against the FBL method as the baseline, where the reduction is always 0% because it is compared against itself.
- **FCR Method**:
  - For the setting [20, 20, 20], memory reduction is 57.46%, showing significant effectiveness of this method at larger settings.
  - For the settings [10, 10, 10] and [5, 5, 5], memory reductions are 55.04% and 71.89% respectively, indicating effective memory reduction across various settings, especially at smaller scales.
- **FCR-shared cache Method**:
  - This method typically uses more memory than FCR in a shared cache situation but still shows significant memory reductions, especially in the setting [5, 5, 5], with a reduction of 63.79%.
- **OTF Method**:
  - For the setting [20, 20, 20], whether at a longer cycle (T=358) or a shorter cycle (T=50), memory reductions are around 34%, indicating that the change in response time does not significantly affect memory reduction.
  - At smaller settings ([5, 5, 5]), memory is reduced by up to 93.02%, showing extremely high efficiency of the OTF method in handling small-scale tasks.
- **OTF-shared cache Method**:
  - This method shows very high memory reduction rates across all settings, particularly in [5, 5, 5] with a reduction of 85.29%, and 77.68% in [20, 20, 20], indicating that shared caching significantly enhances memory efficiency in the OTF method.

A.1.4 GPU Usage Reduction

Table 7: GPU Usage Simulation Results for `ogbn-arxiv`

Threshold	Sparse Graph Storage	Resampled Dense Graph Storage	Total Storage
10	86,053	948,495	1,034,548
20	223,031	431,520	654,551
30	333,599	223,305	556,904
40	423,513	128,715	552,228
50	491,462	85,125	576,587
60	543,682	62,415	606,097
70	585,752	48,810	634,562
80	619,468	39,855	659,323
90	649,094	33,255	682,349
100	674,965	28,290	703,255
110	698,646	24,225	722,871
120	720,183	20,940	741,123
130	738,536	18,420	756,956
140	755,044	16,335	771,379

Table 8: GPU Usage Simulation Results for `ogbn-products`

Threshold	Sparse Graph Storage	Resampled Dense Graph Storage	Total Storage
10	92,784	30,942,345	31,035,129
20	491,202	26,278,605	26,769,807
30	1,122,314	23,034,195	24,156,509
40	1,894,456	20,644,980	22,539,436
50	2,781,440	18,723,690	21,505,130
60	3,797,306	17,072,610	20,869,916
70	4,943,892	15,588,690	20,532,582
80	6,230,698	14,219,115	20,449,813
90	7,697,986	12,920,310	20,618,296
100	9,327,842	11,708,340	21,036,182
110	11,063,622	10,622,880	21,686,502
120	12,882,860	9,671,610	22,554,470
130	14,737,002	8,839,755	23,576,757
140	16,667,254	8,088,615	24,755,869

Table 9: GPU Usage Simulation Results for `ogbn-mag`

Threshold	Sparse Graph Storage	Resampled Dense Graph Storage	Total Storage
10	432,912	4,934,910	5,367,822
20	1,335,788	2,321,400	3,657,188
30	2,059,531	1,307,445	3,366,976
40	2,610,841	813,090	3,423,931
50	3,055,313	533,295	3,588,608
60	3,422,953	363,120	3,786,073
70	3,717,725	257,385	3,975,110
80	3,962,156	187,365	4,149,521
90	4,159,659	140,670	4,300,329
100	4,322,855	107,835	4,430,690

A.2 PyTorch Experiments

Dataset	Nodes	Edges	Features	Classes
PubMed	19,717	44,338	500	3
Cora	2,708	5,429	1,433	7
CiteSeer	3,312	4,732	3,703	6

Table 10: Comparison of PubMed, Cora, and CiteSeer in Terms of Nodes, Edges, Features, and Classes

Operation	Duration (seconds)	Simulation Frequency
Simulated Disk Read	5.0011	0.05
Simulated Disk Write	1.0045	0.05
Simulated Cache Access	0.0146	0.05
In-Memory Computation	Real Computation	Real Computation

Table 11: Simulation Durations and Frequencies

Operation	k_hop_sampling	k_hop_retrieval	k_hop_resampling
Simulated Disk Read	✓		✓
Simulated Disk Write	✓		✓
Simulated Memory Access		✓	

Table 12: Function Access Patterns for Different Operations

Table 13: Experimental Settings - Setting 1

Dataset	Alpha	Presampled Nodes	Resampled Nodes	Sampled Depth
CiteSeer	0.1, 0.2, ..., 0.9	100	40	1, 2, 3, 4
Cora	0.1, 0.2, ..., 0.9	100	40	1, 2, 3, 4
PubMed	0.1, 0.2, ..., 0.9	100	40	1, 2, 3, 4

Table 14: Experimental Settings - Setting 2

Dataset	Alpha	Presampled Nodes	Resampled Nodes	Sampled Depth
CiteSeer	0.1, 0.2, ..., 0.9	20	10	1, 2, 3, 4
Cora	0.1, 0.2, ..., 0.9	20	10	1, 2, 3, 4
PubMed	0.1, 0.2, ..., 0.9	20	10	1, 2, 3, 4

Table 15: IOCostOptimizer Functionality Overview

Functionality	Name	Description
`adjust_dynamic_cost`	Adjust Dynamic Cost	Adjusts the read and write costs based on the current system load.
`estimate_query_cost`	Estimate Query Cost	Estimates the cost of a query based on the number of read and write operations.
`optimize_query`	Optimize Query	Optimizes a given query based on the provided context ('high_load' or 'low_cost').
`modify_query_for_load`	Modify Query for High Load	Modifies the query to optimize it for high load situations.
`modify_query_for_cost`	Modify Query for Cost Efficiency	Modifies the query to optimize it for cost efficiency.
`log_io_operation`	Log I/O Operation	Logs an I/O operation for analysis.
`get_io_log`	Get I/O Log	Returns the log of I/O operations.

Table 16: BufferManager Class Methods

Method	Description
`__init__(self, capacity)`	Initialize the buffer manager with a specified capacity.
`load_data(self, key, data)`	Load data into the buffer.
`get_data(self, key)`	Retrieve data from the buffer.
`store_data(self, key, data)`	Store data in the buffer.