**GCNSplit**: Bounding the State of Streaming Graph Partitioning

Michał Zwolak  
KTH Royal Institute of Technology  
Stockholm, Sweden  
michal.zwolak@hotmail.com

Zainab Abbas  
KTH Royal Institute of Technology  
Stockholm, Sweden  
zainabab@kth.se

Sonia Horchidan  
KTH Royal Institute of Technology  
Stockholm, Sweden  
sfhor@kth.se

Paris Carbone  
KTH Royal Institute of Technology  
Stockholm, Sweden  
parisc@kth.se

Vasiliki Kalavri  
Boston University  
Boston, Massachusetts, USA  
vkalavri@bu.edu

**ABSTRACT**

This paper introduces GCNSplit, a streaming graph partitioning framework capable of handling unbounded streams with bounded state requirements. We frame partitioning as a classification problem and we employ an unsupervised model whose loss function minimizes edge-cuts. GCNSplit leverages an inductive graph convolutional network (GCN) to embed graph characteristics into a low-dimensional space and assign edges to partitions in an online manner. We evaluate GCNSplit with real-world graph datasets of various sizes and domains. Our results demonstrate that GCNSplit provides high-throughput, top-quality partitioning, and successfully leverages data parallelism. It achieves a throughput of 430K edges/s on a real-world graph of 1.6B edges using a bounded 147KB-sized model, contrary to the state-of-the-art HDRF algorithm that requires > 116GB in-memory state. With a well-balanced normalized load of 1.01, GCNSplit achieves a replication factor on par with HDRF, showcasing high partitioning quality while storing three orders of magnitude smaller partitioning state. Owing to the power of GCNs, we show that GCNSplit can generalize to entirely unseen graphs while outperforming the state-of-the-art stream partitioners in some cases.

**CCS CONCEPTS**

- Computing methodologies → Unsupervised learning;  
- Information systems → Data management systems.

**KEYWORDS**

data streams, graph partitioning, graph neural networks

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).


---

**1 INTRODUCTION**

Graph streaming analytics is an emerging application area that aims to extract knowledge from evolving networks in a timely and efficient manner [7, 28]. Graph streams are (possibly unbounded) sequences of timestamped events that represent relationships between entities: user interactions in social networks, online financial transactions, driver and user locations in ride-sharing services. Graph streams are continuously ingested from external, often distributed, sources and are modeled either as streams of edges or as vertex streams with associated adjacency lists.

Graph partitioning has always been crucial for data management, especially since it enables parallel computation on high volumes of complex data. In the context of graph streaming, online graph partitioning methods [3, 24, 31, 34, 40] process graph streams and assign edges or vertices to partitions on-the-fly. To make high-quality partitioning decisions on streaming graphs, state-of-the-art algorithms either accumulate growing state or optimize for load balancing, sacrificing data locality. Existing solutions for online partitioning fall in one of two extremes, as depicted in Figure 1. On one end, stateless approaches such as the hash-based partitioners, achieve almost perfect load balance at the expense of degraded...
partitioning quality. At the other end, stateful methods, such as HDRF [24], yield low IO cost and good balance but have $O(|V|)$ state complexity or higher, where $|V|$ is the number of unique vertices.

While maintaining mutable partitioning state incurs a negligible performance overhead for static and slow graph streams, it poses a major bottleneck in modern applications dealing with high-throughput and possibly unbounded graph streams. In such a setting, the size of accumulated state can quickly exceed available memory causing lookups to hit secondary storage and slowing down updates. The reliance on mutable state further hinders the ability to parallelize the partitioning logic, as that would require expensive synchronization and locking.

This paper addresses the challenge of providing high-quality partitioning for high-rate, possibly unbounded graph streams, without the need to maintain growing mutable state. We propose GCNSplit, a streaming graph partitioning framework, that replaces the partitioning state with an immutable, fixed-size model, encoding the graph stream’s characteristics. GCNSplit targets attributed graphs encountered in several domains, including social networks, IoT applications, natural and medical sciences, citation networks, and online transaction systems [6, 12, 23, 25, 39]. Figure 2 summarizes its design and application setting. To achieve low cuts and load balance, GCNSplit leverages inductive Graph Convolutional Networks (GCNs) to learn vertex embeddings during an offline training phase. It then augments this embedding model with load constraints and assignment heuristics to support online partitioning decisions on continuous streams of edges. GCNSplit achieves high throughput by employing multiple parallel partitioners capable of independently assigning edges to partitions without synchronization. Further, by adopting inductive graph representation learning, GCNSplit’s models can be used to partition entirely unseen graph streams without retraining.

We summarize our contributions as follows:

- We provide a solution to online graph partitioning with bounded state that offers truly scalable execution and good quality, which is on par with state-of-the-art algorithms.
- We implement GCNSplit, an extensible framework that unifies the offline unsupervised training pipeline with the continuous partitioning serving one.
- We demonstrate that GCN-based partitioning can often generalize to unseen graphs. GCNSplit’s models can be used to partition not just unseen edges of the input graph but also entirely unseen graphs of similar structure and matching feature set dimensions.

We extensively evaluate the performance and partitioning quality of GCNSplit on various real-world and synthetic graphs, including the largest publicly available graph stream with features (cf. Section 6). Our results show that GCNSplit exhibits good partitioning quality on a par with state-of-the-art streaming partitioning algorithms, while maintaining well-balanced partitions. Using a real-world graph of 1.6 billion edges (Papers100M), GCNSplit achieves a throughput of 420K edges/s and scales linearly with the number of parallel processes. Compared with the best performing state-of-the-art algorithm which requires at least 116GB of state for the same task, GCNSplit has radically lower and constant state requirements of up to 385K B. Finally, GCNSplit generalizes well and can effectively partition unseen graphs.

We have released the code of GCNSplit as open-source and made models and experiments publicly available [2].

2 PRELIMINARIES

In this section, we revisit the problem of streaming graph partitioning and highlight recent advances in leveraging graph representation learning for partitioning static graphs. Table 1 summarizes the notation we use in the rest of the paper.
2.1 Streaming Graph Partitioning

Balanced graph partitioning is an instance of the graph partitioning problem that tries to optimize for both load balance and minimum cuts and it is a NP-hard problem [4]. Offline graph partitioning methods have access to the entire graph and iteratively refine partitions by re-assigning nodes and edges in each step. Techniques range from exact and slow to approximate and fast (heuristics) [5, 13, 36]. Streaming graph partitioning methods, on the other hand, ingest a graph as a stream of vertices or edges and partition it in an online fashion [3, 30, 31, 33, 34]. As edges and vertices arrive continuously, online partitioners cannot iterate over the entire graph and need to make assignment decisions on-the-fly. Thus, they rely on heuristics and state, so that the partitioning function can access the history of earlier assignment decisions.

Vertex partitioning algorithms ingest streams of vertices and assign them to partitions one after the other. As a result, edges might end up connecting vertices assigned to different partitions. The set of edges spanning multiple partitions is called the edge-cut. The goal of vertex-centric methods is to minimize the edge-cut. Non-trivial online vertex partitioning algorithms require a priori knowledge of the entire graph. This requirement makes such algorithms impractical for graph streams which are gradually revealed to the partitioner during ingestion [3].

Edge partitioning methods assign edges to partitions [9, 24, 40]. In that case, it may happen that edges having a common endpoint end up in separate partitions. As a result, vertices are copied and replicated across partitions. The number of replicated vertices is called the vertex-cut. The higher the vertex-cut the larger the communication overhead of computations performed on the partitioned graph. Thus, edge partitioning algorithms aim to minimize the number of replicated vertices.

The edge-centric stream representation of a massive graph is more convenient to process than its vertex-centric counterpart and does not require prior knowledge of graph properties. However, non-trivial online partitioning methods suffer from growing state size that needs to be kept in memory to make assignment decisions. Whenever the partitioning algorithm processes a new vertex it needs to update the current state. Such $O(|V|)$ memory complexity becomes a bottleneck for modern distributed stream processing systems as it incurs a large number of I/O operations. Thus, neither existing edge-centric partitioning algorithms nor vertex-centric ones can efficiently handle truly unbounded data.

2.2 Graph Representation Learning

Recent breakthroughs in graph representation learning, such as GraphSAGE [12], have enabled effective dimensionality reduction for large graphs and have shown promising predictive performance capabilities. The essence of inductive Graph Convolutional Networks (GCN) is to exploit features associated with vertices and edges as well as the graph structure to build convolutional neural networks that summarize the graph. Graph representation learning methods automatically learn to encode graph structure and properties into $d$-dimensional vectors. Such low-dimensional embeddings can be then processed by downstream machine learning tasks. Embeddings can encode nodes, edges, subgraphs or the entire graph. Node embedding techniques encode graph vertices so that certain node similarities are preserved in the embedding space. As such, their objective function aligns with that of partitioning algorithms, which aim to assign similar nodes to the same partition.

Due to the streaming nature of our problem, we apply inductive graph representation learning based on GCNs. This approach is capable of successful generalization to instances unseen during the training. Convolutional inductive methods represent nodes as functions of their neighborhood while utilizing node features or attributes. Given some neighborhood of the unseen node, such encoders can produce a meaningful embedding, making them scalable and amenable to parallelism.

GraphSAGE. We briefly describe GraphSAGE [12], the inductive GCN that lies at the core of GCNSplit. GraphSAGE has been successfully applied in various real-world scenarios [41] and relies on fixed-sized uniform neighborhood sampling. Restricting the neighborhood size makes it practical for large and skewed graphs.

GraphSAGE and similar GCN frameworks rely on the notion of neighborhood aggregation. Consider the generation of an embedding vector $z_v$ for an arbitrary node $v$. $z_v$ is initialized with the raw input features of $v$ and subsequently, its embedding is refined in an iterative manner as follows: at every iteration, $v$ gathers the embeddings of a subset of its neighbors and aggregates them into a single vector. The aggregated vector is then combined with the embedding vector $z_v$ which is updated by going through an arbitrary differentiable function, such as one defined by a fully connected neural network. As iterations proceed, the embeddings capture topological and feature information from distant neighbors [12].

3 DESIGN OF GCNSPLIT

We target the problem of balanced streaming graph partitioning. Formally, a graph stream is defined as a sequence of edges represented as $V \times V$ tuples $(v_i, v_j)$, where each vertex includes its id and feature vector $F$ of fixed dimensions, as such i.e., $v = (id, F)$. We further assume a fixed number of partitions $k \in \mathbb{N}$. The problem of balanced streaming graph partitioning is to define a single-pass mapping function $V \times V \rightarrow \{1, 2, \ldots, k\}$ over the input edge stream that minimizes 1) the normalized load and 2) the replication factor (Sec. 5.4), while keeping 3) space complexity independent of the size of the input stream.
To solve this problem, we present GCNSplit, a GCN-based partitioning framework for unbounded graph streams with bounded state. In the rest of this section, we explain how GCNSplit’s design makes GCNs applicable to the streaming setting. We provide an overview of GCNSplit’s functionality and architecture in Section 3.1. Section 3.2 describes how the offline training phase generates fixed-size models that can be used for partitioning unbounded edge streams. Finally, we discuss GCNSplit’s online operation and heuristics in Section 3.3.

### 3.1 Framework Overview

To accommodate the edge streaming setting, GCNSplit involves two core modules in taking online partitioning decisions: (1) graph embedding module and (2) graph partitioning module, as shown in Figure 3. The Graph Embedding Module is responsible for the continuous encoding of the non-euclidean input graph stream into vectors of defined size in latent space. Whereas, the Partitioning Module consists of an ML-based partitioning algorithm that assigns incoming edges to partitions. The partitioning module function aims to achieve good partitioning quality, while taking into account load balancing constraints via its assignment heuristics.

Both models used in the embedding and partitioning of the graph stream are trained jointly offline on a sample of the graph stream using the same objective function. The two models are trained on unlabeled data and have the goal of building embedding representations and an online partitioning model which minimizes cuts while not overloading the partitions. It is worth mentioning that in certain scenarios, the node embedding and partitioning modules could potentially be trained separately or replaced without the need to adjust other parts of the framework. At the final partition selection step, GCNSplit uses a set of heuristics to incorporate load metrics known at runtime and ensure load constraints are respected during unbounded executions.

#### 3.1.1 Graph Embedding Module

The graph embedding module uses GraphSAGE to encode input edges and their corresponding vertices into numerical, vector representations. It is therefore important that the input graph stream is annotated with features for our scheme to work efficiently. GCNSplit uses an element-wise max function to aggregate vertices’ vector representations and vector concatenation as the combine function. The trainable parameters of the node embedding generation module include the $W^l$ matrices of all $l$ layers of the GraphSAGE model. The dimension of the matrix from the first layer of the network ($W^1$) is dependent on the size of the node features. The sizes of the matrices from the following layers are equal to the embedding size, which is controlled by a hyperparameter. Thus, the number of the trainable parameters is solely tied to the dimensions of the feature set and the chosen embedding size, while being independent of the size and scale of the data served. The last layer of the embedding model yields a vector representing a particular vertex in a latent space that is passed as an input to the partitioning module. The list of parameters, including the number of network layers and the embedding size can be found in the project repository [2].

#### 3.1.2 Partitioning Module

The partitioning module converts graph input data from its $d$-dimensional vector representation in the embedding latent space to vectors $Y \in \mathbb{R}^{n \times k}$, where $Y_{ij}$ corresponds to the probability that a corresponding vertex $v_i$ belongs to the partition $j \in \{1, 2, \cdots, k\}$; $n$ is the number of items and $k$ is the number of all partitions. This module is implemented as a fully connected neural network with a softmax function at its end that turns the $d$-dimensional vector into a probability distribution over the $k$ partitions.

The trainable parameters of the partitioning module are the weights of the connections between neurons of consecutive layers in the neural network. The number of parameters depends on the number of layers, the vector size, and the number of partitions. The first layer of the partitioning network consumes the output of the embedding module, hence, the number of neurons in that layer is equal to the vector length. The number of neurons in the last layer of the network is equal to the number of partitions $k$.
The normalized cut cost for $k$ is $Y$ of nodes that belong to $S$ edge connections to all nodes, called GAP’s loss function and further discuss the proposed additions.

Unbounded streams. In this section we revisit the formulation of is capable of performing vertex-cut partitioning over possibly GCN-based loss function introduced in GAP [17, 18]. While GAP’s loss unlabeled graph data. The training function itself is responsible for indication of what a good partition assignment can be in any given origin, as long as they have matching feature sets.

Algorithm 1: Unsupervised Model Training

Data: dataset $X$ $\in \mathbb{R}^m$, number of partitions $k$, embedding size $d$, batch size $b$

1. while Loss not converged (eq 7) do
2. $b = \text{sample}(X)$, $D \leftarrow$ degree matrix, $A \leftarrow$ adjacency matrix
3. $z_1, ..., z_b; z_i \in \mathbb{R}^d \xrightarrow{} \text{GraphSAGE}(v_1, ..., v_b; v_i \in \mathbb{R}^m)$
4. $Y \leftarrow \text{PNN}(z_1, ..., z_b)$, $Y \in \mathbb{R}^k$
5. $\Gamma \leftarrow Y^T D$, $L \leftarrow \text{update}(Y, \Gamma, A)$

end

3.2 Model Training

GCNSplit allows training a partitioning model offline on unlabeled data. A loss function drives the model training towards edge-cut minimization. GCNSplit generates both embedding and partitioning networks in the training process (loss function re-use), while not restricting the training and application graphs to be of the same origin, as long as they have matching feature sets.

The main challenge for unsupervised learning is that there is no indication of what a good partition assignment can be in any given unlabeled graph data. The training function itself is responsible for automatically discovering the partition assignments that provide good cuts and balance. Our proposed model extends the edge-cut based loss function introduced in GAP [17, 18]. While GAP’s loss function is only applicable to static edge-cut partitioning, GCNSplit is capable of performing vertex-cut partitioning over possibly unbounded streams. In this section we revisit the formulation of GAP’s loss function and further discuss the proposed additions.

A naive formulation of a cut-based loss function can easily lead to unbalanced partitions as it favors disconnecting small sets of isolated nodes. To avoid this situation, GAP introduces a normalizing factor to reduce bias. The cut cost is defined as a fraction of the total edge connections to all nodes, called association [29] or volume [43]. The normalized cut cost for $k$ partitions looks as follows:

$$Ncut(S_1, S_2, ..., S_k) = \sum_{p=1}^{k} \frac{\text{cut}(S_p, \bar{S}_p)}{\text{vol}(S_p, V)}$$

The $\text{vol}(S_p, V)$ is defined as total edge connections between nodes in $S_p$ and $V$, which can be represented as e.g., total degree of nodes that belong to $S_p$ in graph $G$ ($d_p$ represents the degree of node $v$):

$$\text{vol}(S_p, V) = \sum_{v \in S_p} d_v$$

To utilize the normalized cut to train the model, several transformations have to be applied to Equation 1. The output of the model is $Y \in \mathbb{R}^{n \times k}$. First, we would like to represent the $Ncut$ in terms of $Y$. In the output of the partitioning network $Y_{ip}$ represents the probability that a node $v_i$ belongs to a partition $S_p$. The probability that the node $v_i$ does not belong to the partition $S_p$ is equal to $1 - Y_{ip}$. Therefore, the expected value of a cut looks as follows:

$$E[\text{cut}(S_p, \bar{S}_p)] = \sum_{v_i \in S_p} \sum_{v_j \in N(v_i)} Y_{iq}(1 - Y_{qj})$$

with $N(v_i)$ being the sampled neighborhood of $v_i$. Using an adjacency matrix notation (A), Equation 3 can be rewritten as follows:

$$E[\text{cut}(S_p, \bar{S}_p)] = \sum_{v \in S_p} (Y_{ip}(1 - Y_{ip}))^T \odot A$$

Equations 3 and 4 are equivalent, as element-wise multiplication with adjacency matrix ($\odot$A) ensures, that only the nodes from the sampled neighborhood are considered. The result of this product is a square matrix with side length equal to the number of nodes in the graph. The final value is a sum over the elements of that matrix.

Even though we managed to incorporate $Y$ into the cut equation we still need the normalizing factor. From Equation 2 we know that we need to utilize nodes’ degrees in order to conduct the normalization. Thus, let us assume that $D$ is a column vector where each value $i$ corresponds to node $v_i$’s degree. Using the product of matrices $Y$ and $D$ we can compute the expected value of volume for each partition as $E[\text{vol}(S_p, V)] = \Gamma_p$, where $\Gamma = Y^TD$ and $\Gamma_p$ is the $p$th element in the vector $\Gamma$.

According to Equation 1 we have both parts of the normalized cut - the minimum cut and the volume. Combining both of them results in an equation as follows ($\odot$ is an element-wise division):

$$E[Ncut(S_1, S_2, ..., S_k)] = \sum_{i=1}^{n} (Y \odot \Gamma)(1 - Y)^T \odot A$$

Minimizing the loss function of Equation 5 could lead to unbalanced partitions and even assign all nodes to the same partition. To address this issue, GAP introduces a balancing term, which acts as regularization. Given $|V|$ nodes and $k$ partitions, perfectly-balanced partitions would contain exactly $|V|/k$ nodes. The sums of columns of $Y$ correspond to the expected number of nodes in each partition. The equation which considers the perfectly-balanced partition size looks as follows:

$$\sum_{p=1}^{k} \sum_{i=1}^{n} Y_{ip} - \frac{n}{k} = \sum_{i=1}^{n} (1 - Y_{ip})^2$$

We get the following loss function by combining the normalized load (Equation 5) and the equally loaded partition error (Equation 6). In order to avoid outweighing one part of the loss over the other, we extended GAP’s loss function by introducing a set of configurable coefficients ($\alpha$, $\beta$) that are used to regulate the importance of: 1. a normalized cut and 2. cross-partition balance, as it can be seen in Equation 7.

$$L = \alpha \sum_{p=1}^{k} \sum_{i=1}^{n} (Y_{ip} - \frac{n}{k})^2 + \beta \sum_{i=1}^{n} (1 - Y_{ip})^2$$
The model and unsupervised loss function we have described so far can be used to partition static graphs by assigning nodes to partitions. However, our use-case targets edge-partitioning. As explained in Section 3.3, we derive edge assignments by applying this model to the endpoint vertices of every edge in a stream, thus, making this model applicable to edge streams. In addition, we introduce load constraints to ensure good load balance across partitions.

In Algorithm 1, we summarize the unsupervised training execution logic which is based on mini-batch gradient descent for fast convergence. The resulting embedding module (GraphSAGE) generates a single node embedding while the partitioning network (PNN in Algorithm 1) is used to generate the assignment probabilities (Algorithm 1 Line 4).

Neighborhood Sampling: The sampling method used on each step of the mini-batch execution (Algorithm 1 Line 2) is instrumental to the model training. Due to the multiplication with the adjacency matrix, if nodes were sampled arbitrarily, the probability of choosing non-adjacent nodes would be high. Therefore, for each vertex within a mini-batch, we also ensure to include its direct neighborhood in the sample. This way, we update the model effectively without nullifying the normalized cut of the loss function.

Loss Function Re-use: The training module produces two models using the same loss function: an embedding and a partitioning network. These can potentially be trained separately depending on the encoder used. However, we chose to execute the training of the two components jointly for simplicity and performance, by using the same loss function. As a result, we could exploit GraphSAGE’s ability to optimize its parameters based on a differentiable loss function.

Training Graph: Training in GCNSplit is executed offline on a snapshot or sample of a streaming graph. The training data is extracted using a fixed-size window or snapshot of the graph stream but it can also be a random sample. Furthermore, the training data does not need to come from the same origin graph that is partitioned. An entirely different graph can also be used for training as long as it shares a similar feature set with the target graph. In Section 3.4 we provide further insights on the generalization power of GCNSplit to unseen graphs.

3.3 Model Serving

Enforcing load constraints is a critical extension we made to GAP’s model. As we show in Section 6.1, GAP’s loss function leads to 50-226% higher normalized load (load imbalance) compared to GCNSplit when partitioning the same graph. To ensure that all partitions stay within limits during continuous inference, we regulate their sizes using assignment heuristics.

3.3.1 Assignment heuristics. We devise two heuristics to partition the edge stream using the models, HighestOrLeastLoaded and HighestAvailable. Figure 3 provides an example that shows how the two methods are applied.

HighestOrLeastLoaded. The HighestOrLeastLoaded heuristic first tries to assign an edge to the partition with highest assignment probability of its endpoint vertices. For each endpoint vertex, we retrieve the partition index with the highest assignment probability and try to assign the edge to it. That is, given two vector probabilities $Y_1$ and $Y_2$ produced by the partitioning module, the heuristic retrieves the partition index with the highest assignment probability for each endpoint $p_1 = \arg\max(Y_1)$ and $p_2 = \arg\max(Y_2)$. It first tries to assign the edge to partition $p$, where $p = \max(p_1, p_2)$. If the assignment exceeds the maximum load, we try assigning the edge to the highest ranked partition of the other endpoint vertex. If this assignment exceeds the load constraint again, we assign the edge to the currently least loaded partition.

HighestAvailable. The second heuristic, HighestAvailable, handles the maximum load constraint differently. First, given the edge endpoints $v_1$ and $v_2$, the corresponding assignment probabilities $Y_1$ and $Y_2$ are merged into a list of partition indices $Y$, which is sorted in descending order. Then, we iterate through $Y$, until we find the first partition which can accept the new edge without exceeding the load constraint.

3.3.2 Inference. In online partitioning, edges are processed one-by-one, or window-by-window, where a window is a group of consecutive edges. Thus, every edge assignment is done independently of other assignments. In Algorithm 2, we show the steps of the model serving process for the unsupervised-trained model. The model application scheme of GCNSplit receives unbounded edge streams and partitions them by first converting their counterparts (vertices) into latent space vectors (Algorithm 2 Lines 2-3) and then feeding them into the partitioning network (Algorithm 2 Lines 4-5). Applying GraphSAGE to vertices corresponding to newly added edges requires retrieving the nodes’ neighbourhoods. However, GCNSplit operates on a possibly unbounded stream of edges, where the evolving graph may not fit in main memory. Thus, we rely on the training graph to retrieve information for the new vertices. Each new edge is appended to the in-memory training graph at inference time (Algorithm 2 Line 1). Then, GraphSAGE uses this graph to retrieve the neighbourhood information of the vertices and generate embeddings. After the partitioning of the edge is completed, the ingested edge is removed from the in-memory training graph to ensure the state size remains constant over time (Algorithm 2 Line 6). As a result, the embedding network is capable of generating more informative embeddings, since it has more information about the node’s neighborhood, rather than just the attributes of the edge endpoints. Then, the inference is performed on the corresponding

Algorithm 2: Model Serving with Heuristic

<table>
<thead>
<tr>
<th>Data:</th>
<th>training graph $G$, vertices $v_1, v_2$, number of elements in partitions $S_1, S_2, ..., S_k$, maximum load $M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>partition ID $i$</td>
</tr>
</tbody>
</table>
1. $appendEdge(G, v_1, v_2)$ /* getting embeddings */
2. $z_1 = GraphSAGE(v_1)$;
3. $z_2 = GraphSAGE(v_2)$; /* getting assignment probabilities */
4. $Y_1 = PNN(z_1)$;
5. $Y_2 = PNN(z_2)$;
6. $removeEdge(G, v_1, v_2)$
7. $i = applyHeuristic(Y_1, Y_2, S, M)$
8. return $i$
assignment probabilities of the endpoint vertices. These probabilities are generated from the partitioning network which takes two separate embeddings of the endpoint vertices as input and produces two corresponding assignment probability vectors. Lastly, the chosen assignment heuristic assigns the edge to a partition (Algorithm 2 Line 7). In Figure 3, we summarize the partitioning logic in an end-to-end example of model application from the ingestion of an edge to its final assignment.

3.4 Partitioning Quality and Applicability

Like any ML-driven application, GCNSplit’s effectiveness depends on the quality of the training data and the characteristics of the examples it will be applied on. The graph embedding process is tuned so that similar vertices are represented by vectors that are close to each other in the embedding space. Similarity refers to the structural position of nodes in the graph, as well as the statistical similarity of their associated features. As a result, we expect GCNSplit to be particularly effective on graph streams whose structural characteristics and feature distribution remain relatively stable over time. Nevertheless, if GCNSplit is applied on a graph stream with major concept drift, it will—in the worst case—behave like hash partitioning. The partitioning classifier will assign vertices to partitions at random, yet it will still be guaranteed to produce balanced partitions, thanks to the enforcement of the load balance constraint. We empirically verify this claim in Section 6.1. The underlying assumption of GCNSplit is that the distribution of the data does not shift drastically over time. Otherwise, GCNSplit could either (1) integrate online learning techniques that let the model adapt over time [22, 37] or (2) resort to full model re-training.

With regards to generalization, GCNSplit is applicable to any unseen graph with the same feature set as the graph used for model training. As in the case of partitioning unseen vertices of the same graph, high structural and feature similarity between the target and training graphs is instrumental to achieving high partitioning quality. Our evaluation results (cf. Sec. 6.3) indicate that a richer set of features leads to better generalization. However, we believe this issue requires a further investigation that is beyond the scope of this paper.

4 IMPLEMENTATION

We now briefly outline the implementation of the GCNSplit framework. Training is a batch process that takes place offline, before partitioning. The user needs to provide a training graph. Partitioning is an online process that ingests a graph stream either edge-by-edge or in a micro-batch fashion. The ingestion window size is configurable. In addition, the user needs to set the number of target partitions and select the heuristic to use.

Model implementation. We implement our GCN-based partitioning models using PyTorch. Our partitioning models consist of two components: 1) the GCN component, which generates node embeddings of the incoming graph stream using GraphSAGE and 2) the partitioning component, which generates a probability vector for assigning the incoming edges to given partitions based on the previously generated embeddings using a 3-layered neural network. We adapted the GraphSAGE implementation from an existing code-base [1]. Both the GraphSAGE network and the partitioning network are built using PyTorch building blocks that provide support for implementing neural networks. Furthermore, PyTorch’s strong support for GPU makes operations such as matrix and vector multiplication fast for our neural network-based models.

Parallel model serving. Since models are immutable, GCNSplit can leverage data parallelism to allow for scalability and sustain high-throughput streams. In contrast to stateful algorithms like HDRF, GCNSplit does not need to remember past partitioning decisions and partitioner processes can operate independently of each other. The serving pipeline of GCNSplit is implemented as a set of processes that communicate via queues. At the top of the pipeline, a stream producer process ingests edge streams from a streaming source and pushes them into an ingestion queue. Next, a set of parallel stream consumer processes pull edges from the queue and invoke the partitioning method on them, going through the steps shown in Figure 3. Partitioning decisions are then written into an output queue that can be consumed by a downstream graph streaming application. These partitioning decisions are based on load values that are kept local by each partitioning process to ensure that the load does not exceed the set load limit. To implement the multi-process system and inter-process communication we utilized the torch multiprocessing package, which is a wrapper around the Python’s multiprocessing package optimized towards working with torch.Tensor data structure. In order to ensure equal distribution of computing power between the processes running on the same machine, we limit each process to using a single core.

State size configuration. Stateful streaming graph partitioning algorithms like HDRF keep partial vertex degrees and the previous assignments of the processed nodes so far as in-memory state. This state grows as more distinctive vertices appear in the input stream. Instead, GCNSplit keeps a constant state in memory which depends only on the size of the machine learning model it produced during training. The model size depends on the number of training parameters (which depend on the number of layers in the model’s network), the embedding size, and most importantly on the dimensions of the feature set. A dataset with a rich feature set will have a model size larger than that of another dataset with a less rich feature set. In all of our experiments, we found that effective models are no larger than a couple of MBs, and in most cases, they took only a few KBs of space in memory.

5 EVALUATION METHODOLOGY

We evaluate GCNSplit’s efficiency, scalability, and partitioning quality in various scenarios. Before presenting the results, we first describe our experimental setup and evaluation methodology in this section. We present the datasets and baseline algorithms we use for our experiments and the partitioning quality evaluation metrics. The configuration parameters used during model training can be found in the Appendix [2].

5.1 Experimental Setup

We trained our models using an on-premises physical machine consisting of an Nvidia RTX 2070 Super GPU with 8GB of internal memory. We served the models using a machine comprising of an AMD Ryzen Threadripper 2920X 12-Core processor with 128GB
Table 2: Graph datasets used for evaluation

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>No. of features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitch DE</td>
<td>9.4K</td>
<td>153K</td>
<td>2.5K</td>
</tr>
<tr>
<td>Twitch PTBR</td>
<td>1.9K</td>
<td>31K</td>
<td>2.5K</td>
</tr>
<tr>
<td>Twitch ENGB</td>
<td>7.1K</td>
<td>35K</td>
<td>2.5K</td>
</tr>
<tr>
<td>Twitch RU</td>
<td>4.3K</td>
<td>37K</td>
<td>2.5K</td>
</tr>
<tr>
<td>Twitch ES</td>
<td>4.6K</td>
<td>59K</td>
<td>2.5K</td>
</tr>
<tr>
<td>Twitch FR</td>
<td>6.5K</td>
<td>112K</td>
<td>2.5K</td>
</tr>
<tr>
<td>Deezer RO</td>
<td>41K</td>
<td>125K</td>
<td>84</td>
</tr>
<tr>
<td>Deezer HU</td>
<td>47K</td>
<td>222K</td>
<td>84</td>
</tr>
<tr>
<td>Deezer HR</td>
<td>54K</td>
<td>498K</td>
<td>84</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>203K</td>
<td>234K</td>
<td>165</td>
</tr>
<tr>
<td>Reddit</td>
<td>230K</td>
<td>5.9M</td>
<td>602</td>
</tr>
<tr>
<td>Papers100M</td>
<td>110M</td>
<td>1.6B</td>
<td>128</td>
</tr>
<tr>
<td>Synthetic</td>
<td>930M</td>
<td>1.3B</td>
<td>64</td>
</tr>
</tbody>
</table>

RAM. All modules of the system are implemented using Python 3.8. We used Pandas 1.0.2 [20] to read edges from CSV files and NumPy 1.16.2 [19] to process arrays containing the node’s features. We used PyTorch 1.4 [21].

5.2 Datasets

Table 2 shows the characteristics of the datasets we use to evaluate GCNSplit, ordered by their number of edges. GCNSplit expects an input graph with associated features, as well as timestamps. Of those publicly available, we have selected graphs from different domains to evaluate partitioning quality and of various sizes to verify that the quality of partitioning does not degrade for large graphs. We have also chosen graphs with different feature sets to study how the state size changes.

Twitch: The Twitch dataset [26] represents a user-to-user network of the platform where streamers broadcast their activities live. The dataset consists of six different networks based on the user’s language. The node attributes consist of a user’s location, streaming habits, and activity information and all networks have the same feature set representation.

Deezer: The Deezer dataset [27] represents the user’s friendship network on a music streaming service. The dataset contains three different networks based on the user’s country and the features consist of users’ preferred music genres.

Bitcoin: The Bitcoin dataset [39] represents Bitcoin transactions mapped to real entities. A node in the graph represents a transaction and edges represent the flow of Bitcoins between these transactions. Node features consists of the timestamp associated with each transaction, the number of inputs/outputs, transaction fees, and other transaction information.

Reddit: The Reddit dataset [12] represents a post-to-post network of Reddit, which is a social media platform where users post and comment on topics of their interest. Edges represent comments between posts and node features consist of the post title and the number of comments.

Papers100M: The Papers100M dataset [38] represents the citation network between computer science arXiv papers. Each node has a feature vector of 128 dimensions created by embedding the paper’s title and abstract. We order the edges of the dataset using their publication year to have a time-based set of edges.

Synthetic: To evaluate the performance of GCNSplit in a more challenging scenario, we generate a large random graph with 1.3B edges and 930M nodes. The graph has 64 random synthetic features and we use a subset of 10K edges for model training. We use this graph to demonstrate that GCNSplit’s throughput and state size is independent of the graph size.

5.3 Baseline partitioning methods

We compare GCNSplit with two baseline methods that prioritize different quality metrics: a hash-based stateless method that favors load balancing and a state-of-the-art stateful method, namely HDRF [24], that optimizes cuts. Our goal with GCNSplit is to strike a balance between the two, providing both well-balanced and high-quality partitions.

5.4 Quality Metrics

We evaluate the quality of partitioning methods with the following metrics. We use the normalized load \( \rho \) on the highest loaded partition, \( \rho = \frac{\text{load on the highest loaded partition}}{\text{Total number of vertices} \cdot k} \), where \( n \) is the number of edges in the graph and \( k \) is the number of partitions. \( \rho = 1 \) indicates that the load is well-distributed across the partitions. To evaluate partitioning quality, we use the replication factor \( \sigma \), which indicates how many vertex copies, i.e., the vertex-cuts, are created by the partitioning algorithm. It is defined as \( \sigma = \frac{\text{Total number of vertex copies}}{\text{Total number of vertices}} \). The lower the \( \sigma \), the better the partitioning quality.

5.5 Model Parameters

In our experiments, we trained models using different configurations for each dataset. The Appendix [2] contains all training configurations. The training sets are created based on dataset’s granularity that includes years, epochs and timestamps etc. For example, the Reddit dataset models were trained on the first 10,000 edges based on timestamp order and Bitcoin models were trained on the first 9,164 edges based on the first dataset epoch.

All models used the Adam optimizer [14] with a learning rate equal to 0.0001. The embedding size is 64. The number of layers of the embedding network is 2. The number of layers in the partitioning network is 3 and the number of neurons in a hidden layer of the partitioning network is 64. The maximum load parameter is set to 1.01, which means that the ratio between the number of edges in the highest loaded partition and the ideal \((\frac{k}{n})\) partition was always below 1.01. Certainly, this setting is different for experiments regarding the trade-off between the maximum load value and the replication factor. Finally, we set the \( \lambda \) parameter of the HDRF algorithm to 1 and \( \epsilon = 10^{-3} \), based on its default configuration.

6 EVALUATION RESULTS

We organize the evaluation section in the following three parts. First, we evaluate partitioning quality and show that GCNSplit is on par with HDRF in terms of replication factor, while it maintains well-balanced partitions (Sec. 6.1.). Second, we present performance results that demonstrate how GCNSplit provides high-throughput
partitioning and scales with the number of parallel processes while having small and constant state requirements (Sec. 6.2.). Third, we evaluate the model generalization and demonstrate that GCNSplit not only produces high-quality partitions for unseen graphs but also outperforms HDRF in multiple instances (Sec. 6.3).

6.1 Partitioning Quality

We evaluate GCNSplit’s partitioning quality in terms of replication factor and load balance and compare it with HDRF, hash partitioning, and a baseline approach that uses GAP’s loss function. We also study how the replication factor is affected by the maximum load constraint, the number of edges in the graph stream, and the number of partitions. Finally, we evaluate how the heuristics we introduce in Section 3.3 affect the replication factor.

Replication factor. We use HDRF, Hash, and GCNSplit to partition the graphs into 6 partitions. We set the maximum load limit for GCNSplit to 1.01, effectively forcing the creation of well-balanced partitions. We measure the replication factor of the resulting partitions and plot the results in Figure 4. GCNSplit outperforms Hash on all real graphs and exhibits marginally higher \( \sigma \) than HDRF on the Reddit and Bitcoin graphs. The HDRF baseline ran out of memory before completing the partitioning of Papers100M and Synthetic graphs. The synthetic graph represents a worst-case scenario for GCNSplit, having random structure and features, yet, its worst-case performance falls back to that of hash partitioning (cf. Section 3.4).

Effect of the number of partitions. Next, we evaluate the sensitivity of \( \sigma \) to the number of partitions. We partition the Reddit graph into an increasing number of partitions, using GCNSplit, HDRF, and Hash, and we measure the replication factor of the resulting partitions. Figure 5 shows that \( \sigma \) increases with the number of partitions across approaches. Hash produces the highest \( \sigma \) value, while HDRF and GCNSplit perform similarly with the best cut ratio and lowest \( \sigma \).

Effect of the number of edges. We now study how the replication factor changes over time, as the streaming algorithms partition continuously arriving edges from the graph stream. We stream the Reddit graph in timestamp order and fix the number of partitions to \( k = 3 \). We measure the \( \sigma \) after certain intervals based on the x-axis ticks as shown in Figure 6. As expected, \( \sigma \) increases with the number of edges for all algorithms. However, once again, GCNSplit performs as well as HDRF, producing the lowest number of cuts.

Load balance. In our experiments so far, we set the maximum load limit constraint to 1.01. We now study how much further we can improve GCNSplit’s replication factor by increasing the maximum load constraint to control the normalized load, \( \rho \). Hash partitioning exhibits \( \rho \approx 1 \), while HDRF achieved \( 1 \leq \rho \leq 1.0314 \), in our experiments, depending on the dataset and number of partitions. For this experiment, we use the Reddit graph and we set the number of partitions to \( k = 3 \). We measure the effect of changing the maximum load limit constraint on the replication factor \( \sigma \) and plot the results in Figure 7. Overall, \( \sigma \) decreases with an increasing maximum load limit.

Effect of heuristics. Our experiments so far have used the HighestOrLeastLoaded heuristic. In this section, we evaluate how this choice compares with the HighestAvailable heuristic and how both methods behave on different graphs. Figure 7 plots the replication factor on the Reddit graph in a streaming scenario where edges arrive continuously and are assigned to 3 partitions. HighestOrLeastLoaded performs consistently better than HighestAvailable throughout the duration of the experiment.
Effect of window size. The throughput of streaming applications largely depends on the input batch size of the streams they process. We examined GCNSplit’s throughput in respect to window size, using the Reddit graph with \( k = 6 \) partitions and \( p = 16 \) processes. In Figure 9b it is observed that in this setting throughput caps at 1K edge windows, which we adopt as a constant in all experiments.

State size. We further compare GCNSplit and HDRF model state size for various graphs. Table 3 shows the results for 6 partitions. As expected, HDRF accumulates much larger state than GCNSplit, proportional to the size of the graphs. GCNSplit, on the other hand, has orders of magnitude smaller state requirements, requiring just 115KB for Synthetic (1.3B edges) and 147KB for Papers100M (1.6B edges) to store the models corresponding to the biggest graphs.

We also observed the state size of the algorithms while ingesting the edge stream. In the synthetic graph, HDRF’s state would grow continuously during ingestion and eventually exceed the available memory before completing the partitioning. Using 32 processes, GCNSplit could successfully partition the entire graph keeping its total memory requirements below 11GB, while sustaining 100K edges/s throughput. When it comes to the Papers100M graph, GCNSplit achieved an average partitioning throughput of 430K edges/s, using up to 14GB of memory.

It is noteworthy to discuss the effect of the graph structure on the partitioning throughput. As detailed in Table 2, the Papers100M dataset contains 9 orders of magnitude fewer nodes compared to the highly dense synthetic graph. Combined with the fact that the datasets are streamed in time windows, the probability of encountering duplicate graph nodes in the same window is much higher in the case of Papers100M than that of the synthetic graph, therefore, leading to less unique embedding lookups per batch and higher throughput. Thus, the synthetic graph prescribes a worst-case throughput performance scenario due to its sparsity and randomized edge ordering.

Training time. GCNSplit performs training offline on a snapshot of the streaming graph and re-training is not required when partitioning graphs with the same feature set (c.f. Section 3.2). Table 3 reports the training times for the models we use. We found that the training performance depends on the snapshot size rather than the resulting model size. Training the largest model (Reddit) takes 36 minutes on a snapshot of 16K nodes. Training the Papers100M model takes 6.5× longer for a 62.5× bigger snapshot with 1M nodes.

Generalization to Unseen Graphs

So far we have shown GCNSplit’s capability of producing high-quality partitions for unseen nodes of a streaming graph, using a fixed-size model. Here, we evaluate GCNSplit’s ability to effectively partition completely unseen graphs. Specifically, we train GCNSplit on a subset of a graph and then use its model to partition a different graph stream, albeit with a common feature set.

For this experiment, we use the Twitch and Deezer networks and set \( k = 6 \). We train the models on the complete Twitch-DE graph and 10K edges of the Deezer-RO graph and use the rest of the networks for inference. Figure 10 plots the replication factor of

### Table 3: Partitioning state size (k=6) and training times.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GCNSplit state</th>
<th>HDRF state</th>
<th>Training (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitch DE</td>
<td>1.6MB</td>
<td>4.3MB</td>
<td>22</td>
</tr>
<tr>
<td>Deezer RO</td>
<td>126KB</td>
<td>5.4MB</td>
<td>38</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>166KB</td>
<td>19MB</td>
<td>10</td>
</tr>
<tr>
<td>Reddit</td>
<td>385KB</td>
<td>47MB</td>
<td>36</td>
</tr>
<tr>
<td>Papers100M</td>
<td>147KB</td>
<td>&gt;116GB</td>
<td>233</td>
</tr>
<tr>
<td>Synthetic</td>
<td>115KB</td>
<td>&gt;116GB</td>
<td>13</td>
</tr>
</tbody>
</table>
GCNSplit, alongside that of HDRF and Hash. GCNSplit generalizes well to unseen graphs having matching feature sets and has a low replication factor across experiments. More importantly, GCNSplit outperforms HDRF for all but one instance of the Twitch network. A comparison across graphs also provides an insight into the effect of the feature set, suggesting that the richer the features the better the partitioning quality. Recall that Twitch contains 2.5K features, while Deezer exhibits a mere 83 features per node.

Our observations showcase an important finding. GCNSplit is capable of producing high-quality partitions for entirely unseen graphs whose feature set matches that of the training graph. Whereas, the state of the art algorithms accumulate state tailored to a single graph and cannot be re-used in other partitioning instances.

7 RELATED WORK
To the best of our knowledge, GCNSplit is the first attempt to leverage inductive graph representation learning for online partitioning. Other than the offline approaches outlined in the GAP papers [17, 18], we are not aware of closely related work. First, as GAP requires prior knowledge of the full graph during the training phase, it cannot be directly used on continuously ingested graph streams. Further, GAP is a vertex partitioning method and applying its loss function to edge partitioning leads to high load imbalance as proven previously in Figure 8. GCNSplit overcomes these limitations by providing (i) offline training on a small graph sample coupled with continuous and scalable online inference, (ii) two assignment heuristics that combine vertex embeddings to make edge assignment decisions and (iii) load constraints to ensure good load balance across partitions.

Within the space of ML-enabled data management research, our work resembles that of learned indexes and data structures [8, 15, 16]. GCNSplit relies on learned characteristics of the input data (the graph) to improve the performance of a data management task, which in our case is partitioning rather than search, as in existing work. We now summarize existing work in devising graph representation learning approaches that can serve as alternatives to existing work. We now summarize existing work in devising graph representation approaches that can serve as alternatives to existing work.

Transductive methods [6, 10, 11, 23, 32] are of little use in the streaming context, as they can only generate representations for nodes used during the training phase. Nevertheless, they could be leveraged in static graph scenarios and when node features are absent. On the other hand, inductive methods are more suitable for a streaming setting and numerous methods exist that could replace GraphSAGE in GCNSplit. The algorithms differ in the way they aggregate and sample neighbors, as well as in their approach to generating and combining representations. With regards to the problem of partitioning, we believe that graph attention networks [35] and position-aware graph neural networks (P-GNN) [42] are particularly interesting. The first approach utilizes the attention strategy to conduct neighborhood aggregation. Trainable parameters are applied to a node’s neighbors to weight contributions from neighbors differently. In P-GNNs, nodes are represented by their relative distances to a chosen anchor set, which consists of one or several randomly chosen nodes acting as reference points for other nodes.

8 CONCLUSION AND FUTURE WORK
We presented GCNSplit, an ML-driven streaming graph partitioning method that overcomes the problem of increasing state size for unbounded streams without sacrificing quality. GCNSplit leverages node features to make partition assignment decisions and benefits from recent advances in GCNs. GCNSplit can successfully generalize to unseen nodes and graphs, as long as they bear matching feature set dimensions.

Further Work. We believe that GCNSplit paves the way for exciting future work towards fully-dynamic graph streaming frameworks able to tune their configuration parameters. GCN-based partitioning can be exploited to facilitate the evolution of distributed graph query engines and databases with stream ingestion. Furthermore, the quality of our techniques can be improved via feature analysis or online learning methods to allow the periodic update of the model when significant changes are detected (new features and structural properties). Another interesting direction is exploring how to dynamically change the number of partitions over time. Elasticity could be achieved by training with a large number of virtual partitions. An additional layer could then be used to map physical partitions to virtual ones without updating the model.

ACKNOWLEDGMENTS
This work was partially supported by the RISE AI Research Center, the Wallenberg Foundation project “Data-Bound Computing”, the Swedish Foundation for Strategic Research under Grant No.: BD15-0006, a Red Hat Collaboratory Research Incubation Award (ID: 2022-01-RH08), a Samsung MSL UR collaboration grant, and a Google DAPA award.

REFERENCES