# A Learned Sketch for Subgraph Counting

Kangfei Zhao*, Jeffrey Xu Yu*, Hao Zhang*, Qiyan Li§, Yu Rong†
* The Chinese University of Hong Kong, {kfzhao, yu, hzhang}@se.cuhk.edu.hk
§ Wuhan University, qiyan.li@whu.edu.cn
† Tecent AI Lab, yu.rong@hotmail.com

## Abstract

Subgraph counting, as a fundamental problem in network analysis, is to count the number of subgraphs in a data graph that match a given query graph by either homomorphism or subgraph isomorphism. The importance of subgraph counting derives from the fact that it provides insights of a large graph, in particular a labeled graph, when a collection of query graphs with different sizes and labels are issued. The problem of counting is challenging. On one hand, exact counting by enumerating subgraphs is NP-hard. On the other hand, approximate counting by subgraph isomorphism can only support 3/5-node query graphs over unlabeled graphs. Another way for subgraph counting is to specify it as an *SQL* query and estimate the cardinality of the query in *RDBMS*. Existing approaches for cardinality estimation can only support subgraph counting by homomorphism up to some extent, as it is difficult to deal with sampling failure when a query graph becomes large. A question that arises is if subgraph counting can be supported by machine learning (ML) and deep learning (DL). The existing DL approach for subgraph isomorphism can only support small data graphs. The ML/DL approaches proposed in *RDBMS* context for approximate query processing and cardinality estimation cannot be used, as subgraph counting is to do complex self-joins over one relation, whereas existing approaches focus on multiple relations.

In this paper, we propose an Active Learned Sketch for Subgraph Counting (ALSS) with two main components: a sketch learned (LSS) and an active learner (AL). The sketch is learned by a neural network regression model, and the active learner is to perform model updates based on new arrival test query graphs. We conduct extensive experimental studies to confirm the effectiveness and efficiency of ALSS using large real labeled graphs. Moreover, we show that ALSS can assist query optimizer to find a better query plan for complex multi-way self-joins.

## CCS Concepts

• **Information systems → Information systems applications**.

## Keywords

Subgraph counting; Machine learning; Graph neural network

**ACM Reference Format:**
Kangfei Zhao*, Jeffrey Xu Yu*, Hao Zhang*, Qiyan Li§, Yu Rong†. 2021. A Learned Sketch for Subgraph Counting. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457289

## 1 Introduction

Graph has been widely used for modeling real applications in commercial, biological, lexical, and social networks [58, 75, 78, 86, 99]. As one of fundamental problems, subgraph counting is to count how many subgraphs in a labeled data graph that match a user-given query graph by either homomorphism or subgraph isomorphism. The subgraph counting provides insights to understand a large complex data graph, as users can issue a collection of query graphs with different sizes and labels. It has a wide range of applications in network analysis, to name a few, designing graph kernels for graph comparison and representation [72, 81], building probabilistic models for computer vision tasks (e.g., photo cropping and image segmentation) [31, 94, 95], and analyzing chemical and biological networks (e.g., molecular properties prediction and phylogeny construction) [18]. In addition to supporting applications, subgraph counting can also be used in a system for query optimization to estimate the cardinality for complex large joins with cycles, where a query is to enumerate subgraph matchings in a large data graph stored in a relational system [5, 57].

Exact subgraph counting by subgraph isomorphism is challenging. As surveyed in [69], there are enumeration methods [11, 14, 15, 17, 41, 57, 71] and analytical methods [8, 56, 62, 66]. Full subgraph enumeration is difficult as determining whether a given query graph exists in a data graph by subgraph isomorphism is NP-complete [16, 24]. The analytical approaches do counting by decomposing a query graph into smaller subgraphs, and counting the query graph based on the counts obtained for the smaller subgraphs. The analytical methods are designed for some query graphs up to a certain size (e.g., 5-node query graph), and is difficult to have a general form for any size of query graphs. Instead of exact subgraph counting, approximate subgraph counting has been studied in recent years [19, 20, 23, 40, 85]. These approaches are designed for query graphs with 3-5 nodes over simple unlabeled data graphs, and are non-trivial to be extended to support labeled graphs due to the intrinsic techniques they use for approximation (e.g., color coding) [19, 20].

The subgraph counting can be supported in *RDBMS* using *SQL* [60, 74], multi-way joins [21, 50, 84, 98], and processed as cardinality estimation. First, the approaches in [21, 60, 74, 84] decompose a query graph into a set of small subgraphs, count the subgraph matchings for the smaller subgraphs, and aggregate the counts to

**Table 1: ML Approaches for AQP and Cardinality Estimation**

| Approach | Supported Queries | | | | Learning Strategy | Model | Model Update |
|---|---|---|---|---|---|---|---|
| | Join | Selection | Aggregate | Group By | | | |
| *DBEst* [54] | precomp. join | num., cate. | cnt,sum,avg,etc. | ✓ | supervised & unsupervised | KDE & GBDT | ✗ |
| *DeepDB* [34] | precomp. join | num., cate. | cnt,sum,avg | ✓ | unsupervised | SPN | ✓ |
| Thirumuruganathan et. al. [79] | ✗ | num., cate. | cnt,sum,avg | ✓ | unsupervised | VAE | ✗ |
| Kiefer et. al. [43] | Equi-join | num., cate. | cnt | ✗ | unsupervised | KDE | ✓ |
| Kipf et. al. [44] | PK/FK join | num., cate. | cnt | ✗ | supervised | MSCN | ✗ |
| Dutt et. al. [25] | ✗ | num., cate. | cnt | ✗ | supervised | NN, GBDT | ✗ |
| Sun et. al. [76] | PK/FK join | num., cate., str. | cnt | ✗ | supervised | Tree LSTM | ✓ |
| Hasan et. al. [32] | ✗ | num., cate. | cnt | ✗ | unsupervised | MADE | ✗ |
| *Naru* [90] | precomp. join | num., cate. | cnt | ✗ | unsupervised | MADE, Transformer | ✓ |

approximate the final count. They rely on an assumption that the counts of the smaller subgraphs are independent, which is impractical for real large graphs so that it incurs inaccurate estimation. Second, the approaches in [50, 98] make use of join sampling that draws matchings from the data graph. These approaches face drastically increasing sampling failure in the intermediate join steps when the query graphs and the distribution of the data graph are complex (i.e., complex topological structure and label distribution), although the sampling is independent. The limitations of these approaches cause the deterioration of estimation accuracy as well as efficiency on counting for complex subgraphs. In [64], a benchmark *G-CARE* for cardinality estimation of subgraph matching (homomorphism) is presented, which investigates estimation approaches for graphs [23, 60, 74] as well as relational data [50, 84, 98]. With *G-CARE* open source, we verify that join sampling has its limitation for complex subgraph matching even by homomorphism when it is used to test for large query graphs. The difficulty of cardinality estimation comes from complex data distribution [48].

A question that arises is if Machine Learning (ML) and Deep Learning (DL) can be effectively used for subgraph counting. Liu et. al. in [52] study neural subgraph isomorphism counting using a dynamic memory network. Their models are trained on synthetic graphs which are up to 512 nodes (Table 4 in [52]). Their approach is computationally expensive to be used for large data graphs. Below, as subgraph counting (homomorphism or subgraph isomorphism) can be specified as an *SQL* query, we discuss if DL approaches studied in *RDBMS* context for approximate query processing (AQP) or cardinality estimation can be used to support subgraph counting.

Table 1 summarizes the ML/DL approaches studied for AQP (the top three) and cardinality estimation (the bottom six). The *SQL* queries they support include join, selection, and group-by and aggregation. Both supervised learning and unsupervised learning are studied. The supervised learning approaches are query-driven to learn a function mapping from query features to its cardinality [25, 44, 76], and the unsupervised learning approaches are data-driven to focus on learning a joint probability distribution of the underlying dataset [32, 34, 43, 79, 90]. By supervised learning [44, 76], *SQL* queries are encoded based on the fixed database schema where all the relations, attributes and join attributes need to be given beforehand. By unsupervised learning [34, 54, 90], the model learns the joint probability distribution of one relation. If queries contain joins across multiple relations, multiplying the probabilities relies on the independent assumption of relations [34]. Otherwise, models need building on precomputed join results. None of the approaches in Table 1 can be used to support subgraph counting directly for the following reasons. First, a query graph can be any with different patterns, sizes, and labels whereas the joins cannot be predetermined or precomputed. Second, the approaches listed in Table 1 learn joins across different relations in general, whereas the subgraph counting is by self-joins between an edge relation and itself. There can be many joins but all are based on one relation with the same distribution. The models used for joins across different relations are not effective to learn self-joins on one relation, and versa visa. It is difficult to support subgraph counting with homomorphism by existing ML/DL approaches, and it is even difficult to support subgraph counting with subgraph isomorphism by *SQL* due to the additional constraints.

In this paper, we propose an Active Learned Sketch for Subgraph counting (ALSS) with two main components, a sketch learned for subgraph counting (LSS) and an active learner (AL), to learn a sketch for subgraph counting over a labeled undirected data graph $G$, where subgraph counting can be by either homomorphism or subgraph isomorphism. Below, we focus on node-labeled data and query graphs. We will discuss the extension to both node/edge-label graph in this paper.

**Sketch Learning**: We learn a sketch by a neural network regression model. Inspired by the decomposed-based subgraph counting approaches [8, 9, 23, 40, 46, 66, 74], we decompose a user-given query graph, $q$, into smaller subgraphs. To avoid confusing, we call a smaller subgraph decomposed as a substructure below. The learned sketch firstly extracts features for each individual substructure as a fixed-length vectorized representation, then predicts the count of the query graph $q$ by aggregating the representations of all substructures. Specifically, we adopt a graph neural network (GNN) [13, 100] to represent labeled substructures, and learn the aggregate function by self-attention mechanism, followed by a multi-layer perceptron (MLP) that predicts the final count. Our learned sketch has the following advantages. First, due to the universal approximation capability of neural networks [36, 37], the error from a simple combination of the substructure counts under the independence assumption can be greatly reduced. Instead, the learned sketch conducts the aggregation by evaluating the query-specific importance of the substructures. Second, the sketch learns via error feedback from its estimations over a set of query graphs where each query is with the exact count, by a powerful optimizer (e.g., stochastic gradient descent). Third, the learned sketch captures the characteristics of learning task by exploiting the inductive bias of the subgraph counting tasks. Here, GNN models self-joins (or self-join graphs) on one relation via parameter sharing. The learned sketch is permutation-invariant regarding the set of the substructures (either by homomorphism or subgraph isomorphism). Fourth, our learned sketch will be adapted to a specific workload and its underlying distribution by learning, and does not serve the purpose of being a general estimation formula to perform well on any workload.

**Active Learning**: A main issue by supervised learning in LSS is that it requires a large training dataset of query graphs together with label and the exact count. In fact, the number of possible labeled query graphs grows exponentially w.r.t. the number of nodes and distinct labels, which implies it is almost impossible to have a rather complete training set to learn a model. To address this issue, we adopt active learning [7, 70]. The core of active learning is a strategy that selects a subset from a pool of unlabeled data, based on a metric of informativeness regarding the model, to enrich its training data for prediction improvement. But, active learning has its own issue. Recall that we learn a neural network regression model for LSS. Existing active learning approaches for regression models measure the informativeness based on expected variance reduction, expected model change, or model ensemble [22, 47, 53, 55]. Although these approaches are effective on classical ML models (i.e., linear regression, random forests, and single-layer artificial neural network), they suffer from high computation cost. To avoid such cost, we take a new approach in devising a specialized active learning strategy for LSS, which is not only effective but also efficient. In brief, we incorporate a classification task into LSS to predict the magnitude of subgraph counts. In other words, our LSS has two learning tasks. One is to predict the subgraph count for a given query graph. The other is to help AL to decide how to select test query graphs for further learning. In the framework of uncertainty sampling for active learning [49], our active learner AL leverages the uncertainty of the classification task and the cross-task consistency between the two interrelated tasks without performing model ensemble or computing gradients. Note that the target of AL in LSS is not to narrow down the data complexity as statistical learning does [30, 59]. Instead, AL of LSS aims to perform more efficient model updates compared with passive learning, under equivalent data volume.

**Contributions.** The main contributions of ALSS are summarized as follows. ① We propose a supervised learning model LSS, for estimating subgraph counts. To the best of our knowledge, LSS is the first ML/DL based approach for subgraph counting over a large labeled data graph, dealing with either homomorphism or subgraph isomorphism. ② We devise a simple yet effective active learning strategy, specialized for LSS, aiming to improve the model's generalization in contrast to passive learning. ③ We conduct extensive experimental studies to verify the effectiveness and efficiency of LSS and its active learning strategy, via comparison with relational-based and native graph algorithms. We verify that existing approaches fail to handle large and complex query graphs, due to the failure of their sampling strategies. On the contrary, our LSS makes accurate estimation stably from small to large queries, and the performance is robust under varying training workloads. ④ To further study the application of LSS, we equip LSS as a cost estimator of a multi-way join system. Testing on 3 real data graphs shows that LSS can improve the quality of query plans up to 3 orders of magnitude over 1,560 diversified query graphs.

**Roadmap.** §2 gives the problem statement. In §3, we present an overview of the learned sketch. Then, we elaborate on the learned sketch model and the corresponding active learning strategy in §4 and §5, respectively. §6 reports the experimental results. Finally, we review the related works in §7 and conclude the paper in §8.

## 2 Problem Statement

We model a graph as a labeled undirected graph $G = (V, E, L, \Sigma)$. Here, $V$ is a set of nodes, $E$ is a set of undirected edges, $\Sigma$ is a set of labels, and $L$ is the mapping function that maps a node $u \in V$ to a label denoted as $L(u)$. We denote neighbors of node $u$ in $G$ as $N(u) = \{v | (u, v) \in E\}$.

Given a data graph $G = (V, E, L, \Sigma)$ and a query graph $q = (V_q, E_q, L, \Sigma)$. A homomorphism of $q$ to $G$ is a function $f : V_q \mapsto V$ such that (1) for every $u \in V_q$, $L(u) = L(f(u))$, and (2) for every $(u, v) \in E_q$, $(f(u), f(v)) \in E$. A subgraph isomorphism of $q$ to $G$ is a homomorphism of $q$ to $G$ under the condition that $f$ is an injective function, where $f(u) \neq f(v)$ for any pair of $u$ and $v$ in $V_q$ if $u \neq v$. A homomorphism (or subgraph isomorphism) function $f$ of $q$ induces a subgraph $G_f = (V_f, E_f, L, \Sigma)$ in $G$, where $V_f$ is the set of nodes, $f(u)$, for every $u$ in $V_q$, and $E_f$ is the set of edges, $(f(u), f(v))$, for every edge $(u, v)$ in $E_q$. We say $G_f$ is a subgraph matching of $q$ to $G$ by homomorphism (or subgraph isomorphism) if $f$ is a homomorphism (or subgraph isomorphism) function.

**Subgraph Counting**: Given a data graph $G$ and a query graph $q$, subgraph counting is to find the total number of subgraph matchings of $q$ to $G$, denoted as $c(q)$. A node in a query graph is allowed to be unlabeled. When a node in $q$ is unlabeled, we assign a special label, which is interpreted as **any** label.

We build a machine learning model from a given data graph $G$ and a set of query graphs and the corresponding counts, $Q = \{(q_1, c(q_1)), (q_2, c(q_2)), \cdots\}$, where $q_i$ is a query graph and $c(q_i)$ is its exact count. The model will learn a sketch for subgraph counting by either homomorphism or subgraph isomorphism, but not both. In other words, for subgraph counting by homomorphism (subgraph isomorphism), every count $c(q_i)$ must be for homomorphism (subgraph isomorphism). A model will give an estimated count $\hat{c}(q)$ for an unseen query graph $q$. We use q-error to evaluate the accuracy of the estimated value.

$$\text{q-error}(q) = max\left\{\frac{c(q)}{\hat{c}(q)}, \frac{\hat{c}(q)}{c(q)}\right\} \tag{1}$$

Intuitively, q-error quantifies the factor by which the estimated count differs from the true count. It is symmetrical and relative so that it provides the statistical stability for true counts of various magnitudes. Here, we assume $c(q) \geq 1$ and $\hat{c}(q) \geq 1$.

## 3 An Overview

The overview of Active Learned Sketch for Subgraph counting (ALSS) is shown in Fig. 1, which has two main components, a learned sketch (LSS) and an active learner (AL). Here, LSS is a neural network regression model for predicting the counts of queries, and AL selects some informative query graphs $q_i$ based on a selection strategy from new arrival test queries. The selected queries $q_i$ and their exact count $c(q_i)$ are collected for possible further learning. As shown in Fig. 1, initially, a training query workload,

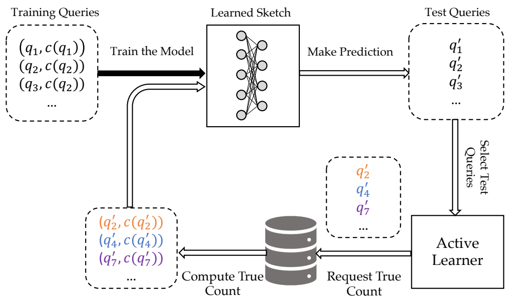


Figure 1: Overview of the Active Learned Sketch

$Q = \{(q_1, c(q_1)), (q_2, c(q_2)), \cdots\}$ is given over a data graph $G$, ALSS will learn a sketch LSS offline using a graph neural network (GNN) based model [13, 100]. The learned sketch LSS is then used to predict the count, $\hat{c}(q)$, for a test query graph $q$. Let $\hat{Q} = \{(q_1, \hat{c}(q_1)), (q_2, \hat{c}(q_2)), \cdots\}$ be the set of the given test queries and their corresponding predicted counts. The active learner AL will select a set of queries under a given budget, whose information is expected to help improve the model's performance, from $\hat{Q}$. Intuitively, a query $q_i$ is selected if the model's prediction has a larger q-error($q_i$). As the true count $c(q_i)$ is unknown yet, AL will evaluate the prediction uncertainty based on an uncertainty function and perform biased sampling on the uncertainty. Take $Q_\Delta$ as the set of additional selected queries with true counts computed. ALSS will update the sketch LSS incrementally by re-optimizing the loss function for the augmented training data, $Q \cup Q_\Delta$. This procedure repeats during online testing and can be fully automated without human-in-the-loop. The benefit of LSS actively learned is two folds. First, the increasing volume of training data improves the performance of LSS. Second, the added new training data motivates LSS to adapt to a varying workload.

## 4 The Learned Sketch

In this section, we discuss the Learned Sketch for Subgraph counting (LSS) on modeling, architecture, and node encoding.

### 4.1 LSS Modeling

The design of LSS is inspired by decomposed-based subgraph counting [8, 9, 23, 40, 46, 66, 74]. We decompose a query graph to smaller substructures, compute/estimate the counts for the substructures decomposed, and aggregate the individual counts for substructures decomposed as the final result. The key idea behind such a divide and conquer paradigm can be expressed by a formula in Eq. (2).

$$\hat{c}(q) = \phi\Big(\sum_{s_i \in S(q)} \sigma(s_i) \cdot w(s_i)\Big) \tag{2}$$

Here, a query graph, $q$, is decomposed into a set of substructures $S(q) = \{s_1, \cdots, s_n\}$. The function $\sigma(s_i)$ is an estimated count of the substructure $s_i$ and $w(s_i)$ is an optional weight. $\sum$ is an aggregation function that aggregates the weighted count followed by a possible post correction $\phi$. Based on the key idea, for exact counting algorithms [8, 9, 66], the formula of Eq. (2) is elaborated based on a specific decomposition strategy where the relationships between the substructures are determined to ensure the exact count. The existing approaches can support query graph up to 5 nodes, as the relationships between the substructures of a query graph can be complex when a query graph is large. For approximate counting algorithms [23, 40, 46, 74], they need to assume that the counts of the substructures are independent so that they can aggregate total count by summation or multiplication using Eq. (2). This independent assumption is the main cause of the estimation error and the accuracy degrades in particular when a query graph is large.

In this work, we model LSS as a regression model, and we learn an estimation formula by parameterizing the functions $\sigma(\cdot)$, $w(\cdot)$ and $\phi(\cdot)$ used in Eq. (2) using neural networks. As a regression model, LSS takes a query graph $q$ as input and outputs the estimation, denoted as $c_\Theta(q)$, in a similar fashion of Eq. (2). LSS is trained by minimizing the mean-squared-error (MSE) over a set of labeled training queries $Q$ as shown in Eq. (3).

$$\mathcal{L}_{reg}(Q;\Theta) = \frac{1}{|Q|}\sum_{q \in Q} |\log c(q) - \log c_\Theta(q)|^2 = \frac{1}{|Q|}\sum_{q\in Q} \log^2 \frac{c(q)}{c_\Theta(q)} \tag{3}$$

To achieve an average low relative error, the true count $c(q)$ and $c_\Theta(q)$ are transformed to logarithmic scale. It is worth noting that minimizing the average of $\log \frac{c(q)}{c_\Theta(q)}$ is equivalent to minimize the geometric mean of q-error, and minimizing the mean-squared-error (Eq. (3)) further imposes higher weights on larger q-error over the average due to the square [25].

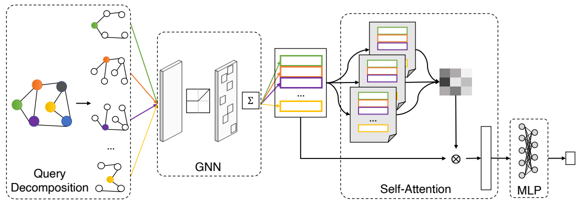


Figure 2: The LSS Architecture

### 4.2 LSS Architecture

The architecture of LSS is shown in Fig. 2. First, a query graph $q = (V_q, E_q, L, \Sigma)$ is decomposed into a set of substructures $S(q) = \{s_1, \cdots s_n\}$, where $s_i$ is a subgraph $s_i = (V_i, E_i, L, \Sigma)$. The decomposition should be complete in a sense that $V_q = \cup_{s_i \in S(q)} V_i$ and $E_q = \cup_{s_i \in S(q)} E_i$. In addition, in order to learn the interrelated substructures for counting, it is expected that $s_i \cap s_j$ is not disjoint for $i \neq j$. We take a simple but effective approach to determine a substructure $s_i$. In brief, for a $n$-node query graph $q$, we construct a $l$-hop *BFS* (Breadth-First Search) tree rooted at every node in $q$, and there are in total $n$ *BFS*-trees in $S(q)$. Second, a multi-layer graph neural network (GNN) is constructed to encode every $s_i \in S(q)$ into a fixed-length vector, which serves as the function $\sigma(\cdot)$ in Eq. (2). Third, a self-attention layer is used which is to learn a weighting function $w(\cdot)$ in Eq. (2), from the $n$ substructures of a $n$-node query graph $q$. Fourth, we sum up the encoding of substructures, $\sigma(s_i)$, weighted by their weights $w(s_i)$ to acquire query-level representation vector. Fifth, a multilayer perceptron (MLP) finally takes this query-level representation as input and outputs the estimated count corresponding to the function $\phi(\cdot)$ in Eq. (2).

The LSS forward propagation algorithm is listed in Algorithm 1, which takes a query graph $q$ as input and output the log-scale estimated count $\log c_\Theta(q)$. We elaborate on the details of each component of LSS step by step, where an input query graph $q$ is decomposed into a substructure set $S(q)$ as discussed in line 1.

**Algorithm 1:** LSS Forward Propagation

```
Input: a query graph $q$, aggregate functions $f_{\mathcal{A}}^{(1)}, \cdots, f_{\mathcal{A}}^{(K)}$,
       combine functions $f_C^{(1)}, \cdots, f_C^{(K)}$, weights of attention layer
       $W_1$, $W_2$
Output: $\log c_\Theta(q)$
1  $S(q) \leftarrow \mathsf{Decompose}(q)$
2  for $s_i = (V_i, E_i) \in S(q)$ do
3      for $k \leftarrow 1$ to $K$ do
4          for $v \in V_i$ do
5              $a_v^{(k)} \leftarrow f_{\mathcal{A}}^{(k)}(\{\mathrm{e}_u^{(k-1)} | u \in N(v)\})$
6              $\mathrm{e}_v^{(k)} \leftarrow f_C^{(k)}(\mathrm{e}_v^{(k-1)}, a_v^{(k)})$
7      $h_{s_i} \leftarrow \mathsf{Readout}(\{\mathrm{e}_v^{(K)} | v \in V_i\})$
8  $H_q \leftarrow \mathsf{Concat}(h_{s_1}, \cdots, h_{s_n})$
9  $A \leftarrow \mathsf{softmax}(W_2 \mathsf{tanh}(W_1 H_q^T))$
10 $\mathsf{E}_q \leftarrow A * H_q$
11 $\mathrm{e}_q \leftarrow \mathsf{Flatten}(\mathsf{E}_q)$
12 $\log c_\Theta(q) \leftarrow \mathsf{MLP}(\mathrm{e}_q)$
```

**GNN.** A $K$-layer GNN is constructed to process one substructure $s_i$ (line 3-7), and generate a per-substructure representation $h_{s_i}$ (line 7) at a time. The $K$-layer GNN [28, 83, 88] follows a neighborhood aggregation paradigm to update the representation of a node by aggregating the representations of its neighbors in $K$ iterations. Note that GNN uses the graph structure and node features to learn a representation vector for each node, or for the entire graph. Let $\mathrm{e}_v^{(k)}$ denote the representation of a node $v$ generated in the $k$-th iteration. In the GNN $k$-th iteration (layer), for a node $v$, an aggregate function $f_{\mathcal{A}}^{(k)}(\cdot)$ aggregates the representations of the neighbors of $v$ that are generated in the ($k$-1)-th iteration (line 5). Then, a combine function $f_C^{(k)}(\cdot)$ updates the representation of $v$ by the aggregated representation $a_v^{(k)}$ and the previous representation $\mathrm{e}_v^{(k-1)}$ itself (line 6). The functions $f_{\mathcal{A}}^{(k)}(\cdot)$ and $f_C^{(k)}(\cdot)$ are neural networks, e.g., linear transformation with non-linearity and optional Dropout [73] for preventing overfitting. To obtain a representation for the entire substructure $s_i$, GNN applies a function Readout to aggregate the node representations from the final iteration (line 7). The function is a simple permutation invariant function such as summation or a more sophisticated pooling function [91]. To summarize, GNN processes every substructure $s_i$ and outputs a representation $h_{s_i}$ for each $s_i$, respectively.

We investigated popular GNNs, including the vanilla Graph Convolutional Network (*GCN*) [45], Graph Attention Network (*GAT*) [83], *GraphSAGE* [28], and Graph Isomorphism Network (*GIN*) [88]. Different GNNs are different from their aggregate ($f_{\mathcal{A}}(\cdot)$), combine ($f_C(\cdot)$) and Readout functions. [88] proves that the distinguishing capability of GNN is as powerful as the graph isomorphism test, Weisfeiler-Lehman (WL) test [87], if the aggregate, combine and Readout functions are injective. Different from *GCN*, *GAT* and *GraphSAGE*, *GIN* uses MLP as the aggregate and combine functions and summation as Readout, which is proved to be injective. Therefore, *GIN* is as powerful as WL test. More specifically, it indicates that for two substructures $s_1$ and $s_2$, if $s_1$ and $s_2$ are isomorphic, their graph-level representation $h_{s_1}, h_{s_2}$ are equal under zero training error assumption. The property of *GIN* is consistent with the inductive bias of the subgraph counting task, i.e., if $s_1$ and $s_2$ are isomorphic, the true count of $s_1$ and $s_2$ are equal for both homomorphism and isomorphism. To this end, we use *GIN* as the GNN component of LSS.

We can extend LSS to support edge-labels by leveraging edge features in the aggregation function in line 5:

$$a_v^{(k)} \leftarrow f_{\mathcal{A}}^{(k)}(\{\mathsf{Concat}(\mathrm{e}_u^{(k-1)}, \mathrm{e}_{uv}^{(0)}) | u \in N(v)\}) \tag{4}$$

Here, $\mathrm{e}_{uv}^{(0)}$ is the initial edge features of edge $(u, v)$, which is concatenated with the node representation and transformed by the neural network defined by $f_{\mathcal{A}}^{(k)}$ of the $k$-th layer. Since vector concatenation is also injective, this extension does not violate the expressive power of *GIN*.

**Self-attention and MLP.** In the self-attention stage, the $n$ substructure representation vectors are concatenated to a matrix $H_q$ (line 8) and $H_q$ is processed by the self-attention mechanism [51, 82]. Intuitively, the attention mechanism is for multiple experts to rate the importance of the substructures in independent perspectives, with which a rating matrix $A$ is generated (line 9). In line 9, the self-attention mechanism is implemented by a two-layer MLP without bias, where the first layer has weight matrix $W_1$ and is activated by the nonlinear $\mathsf{tanh}$, and the second layer has weight $W_2$ and is activated by $\mathsf{softmax}$. By multiplying the rating matrix $A$ with the concatenated matrix $H_q$, we obtain a final query-level representation $\mathsf{E}_q$ in line 10. The size of matrix $\mathsf{E}_q$ is independent to the size of a query graph and is only controlled by the model's hyper-parameters. It is worth noting that the self-attention layer is invariant to permutations of the substructure set $S(q)$. In the end, $\mathsf{E}_q$ is flattened to a long vector $\mathrm{e}_q$ for the query graph $q$ (line 11) and pass through an MLP to generate the estimated count (line 12).

## 4.3 Feature Encoding

Encoding initial node/edge representation $\mathrm{e}_v^{(0)}/\mathrm{e}_{uv}^{(0)}$ for a substructure $s_i$ is important in learning. The one-hot encoding by GNN is to represent the attribute features [45, 88] for node classification and link prediction. Such sparse encoding is lack of insight for the analytical subgraph counting, which is related to both a larger data graph and a smaller query graph with labels. It is worth noting that the labels of a query node/edge serve as the predicates of the query, and are used to filter nodes/edges on the data graph. We propose frequency-based encoding and pre-trained embedding-based encoding to encode label information and topological structure.

**Frequency-based Encoding.** The frequency-based features encode the filter capability of a query node/edge regarding the data graph. We illustrate the encoding using node-labels, which can be adapted for edge-labels in a similar way. For a data graph $G$, we denote the number of occurrence of a node-label $l$ as $F(l) = |\{v \mid l \in L(v) \text{ for } v \in V\}|$. The node representation for $v$ in a query graph $q$, $\mathrm{e}_v^{(0)}$, is encoded as a $|\Sigma|$-dimensional vector, $\mathrm{e}_v^{(0)} \in \mathbb{R}^{|\Sigma|}$, where the $i$-th dimension corresponds to the $i$-th node-label $l_i \in \Sigma$. The value of $\mathrm{e}_v^{(0)}[i]$ is the fraction of the nodes in $G$ can be matched to $v$. In detail, if node $v$ is associated with a node-label $l_i$, $\mathrm{e}_v^{(0)}[i]$ will be set to $F(l_i)/|V|$, otherwise $\mathrm{e}_v^{(0)}[i]$ will be set to 1.0.

**Embedding-based Encoding.** The frequency-based encoding takes the attribute frequency of the data graph into account, but fails to

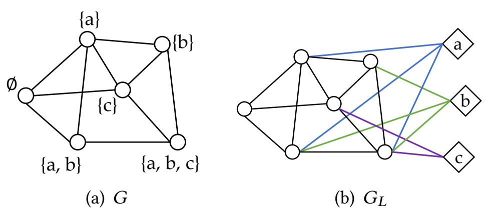


**Figure 3: A data graph $G$ & its label-augmented graph $G_L$**

leverage the topology of the data graph $G$. An encoding to encode the topological structure of the data graph together with its labels is needed. As feeding GNN a pre-trained and unsupervised embedding as node features can boost the performance, we pre-train a node-label embedding for the data graph $G$ to enhance the query graph encoding. To preserve the topological property of the data graph $G$ together with its labels ($\Sigma$), we construct a label augmented graph $G_L = (V \cup V_L, E \cup E_L)$ for $G = (V, E, L, \Sigma)$. Here $V_L$ is a set of nodes where a node represents a label in $\Sigma$, and there are $|\Sigma|$ nodes in $V_L$. $E_L$ is a set of edges where an edge is between a node, $v$, in $V$ with a node, $l$, in $V_L$, if $v$ has the node-label $l$ that the node $l$ corresponds to. Fig. 3(b) shows the label-augmented graph $G_L$ for a small data graph $G$ in Fig. 3(a). We use a scalable, task-independent graph embedding algorithm (e.g., *DeepWalk* [65], *node2vec* [27], *ProNE* [93]) to pre-train a node embedding for the label-augmented graph $G_L$. With the pre-trained label embedding, we encode every node in a query graph $q$. For a node $v$ in $q$, we set $\mathrm{e}_v^{(0)}$ to be $\sum_{l \in L(v)} \mathrm{e}'(l)$, where $\mathrm{e}'(l)$ is the pre-trained embedding of the label $l$ in $G_L$ if $v$ has the label $l$. A node $v$ will have an all-zero vector if it does not have any labels.

**Concatenated Encoding.** The concatenated encoding, $\mathrm{e}_v^{(0)}$ for a node $v$ in a query graph $q$ is to encode a node by concatenating the frequency-based encoding and embedding-based encoding.

## 5 Active Learning for LSS

We first introduce active learning (AL) in brief. A regular ML task is to train a model $\Theta$ by minimizing the empirical loss measured by a loss function $\mathcal{L}$ on the training data , and produce the predictive value for the test data $X_U$, where $X_L$ and $X_U$ are assumed to be randomly sampled from an underlying distribution. The idea of AL is to enhance the limited number of training data by selecting a collection of informative test data $X_A \subset X_U$ adaptively based on the model $\Theta$ to update the model $\Theta$. With the enhanced training data $X_A$, the model tends to learn effectively and produce better prediction results. This procedure repeats until a stop criterion is reached (e.g., exhausting of labeling budget, reaching a fixed number of iterations, or a small enough loss). The key issue in AL is its strategy of selecting the informative test data, which has different trade-offs. Choosing an effective and efficient strategy depends on the specific ML task, the optimization goal, as well as the data distribution.

The active learning algorithm we use for LSS is by uncertainty sampling. It attempts to sample from the region of the data which has the most uncertainty. The algorithm takes an LSS model $\Theta$ and a set of unlabeled test query graphs $\hat{Q}$ as input, and uses an uncertainty measurement $\varphi(q; \Theta)$ to evaluate the informativeness of a query graph $q$ under the LSS model $\Theta$. We will discuss the uncertainty function $\varphi$ later in this section. The AL algorithm works in 4 steps iteratively. ① For each query graph $q$ in the unlabeled test set $\hat{Q}$, we compute its uncertainty score, denoted as $u_q$, using the function $\varphi(q; \Theta)$. ② The active learner uses the normalized $u_q$ as sampling weights to sample a batch queries $Q_\Delta$ from $\hat{Q}$, and label $q \in Q_\Delta$ with its exact subgraph count. ③ The training set $Q$ and the unlabeled test set $\hat{Q}$ are updated, where $Q$ is enlarged as $Q \cup Q_\Delta$. ④ The LSS model is updated with the enlarged training data $Q$. In other words, the LSS model is retrained by optimizing the loss function $\mathcal{L}(Q; \Theta)$. Finally, the stop criterion can be the loss convergence or an iteration time specified for limited retraining.

The framework of uncertainty sampling is simple. However, the challenge is how to efficiently evaluate the uncertainty of a query graph $q$ regarding the model $\Theta$ for a regression task that predicts a real-value scalar. The existing approaches are based on model ensemble [22, 47]. By model ensemble, first, a committee of models is trained over the same training dataset; second, an AL strategy selects unlabeled data that are with higher disagreement over the models [47] or that maximize the expected model change (*MEMC*) [22], where all models are retrained in every iteration of AL. But, ensemble learning is cost-inefficient for LSS, especially for *MEMC* where gradients computation is required. For a regression model, an AL strategy is expected variance reduction [55], which attempts to directly optimize the variance of the model. Unfortunately, it has a high computation complexity $O(|\hat{Q}|K^3)$, where $K$ is the number of model parameters, since estimating output variance requires inverting and multiplying parameter matrices for each test query graph. Note that LSS itself is a neural network model with a large number of parameters, thereby expected variance reduction is also intractable.

To fast evaluate uncertainty, we propose a simple yet effective approach using multi-task learning. The approach we take to help AL is to incorporate a classification task into LSS model that predicts the magnitude of the subgraph count. Intuitively, test queries with ambiguous distribution of count magnitude have more uncertainty w.r.t. the LSS model. As the classification and regression tasks are highly correlated, i.e., have constrained output, we share all the feature representations between them in LSS except the output by the multi-task learning model [97]. In the output layer of the MLP in LSS, in addition to one neuron which predicts the estimated count $\log c_\Theta(q)$, there are $m$ extra neurons used to perform a multi-class classification. The output of the $m$ neurons is activated by a softmax function to achieve a probability distribution $p_\Theta(y|q)$ where $p_\Theta(y_i|q)$ indicates the probability that the magnitude of $c_\Theta(q)$ is $y_i \in \{0, \cdots m-1\}$. We use the cross-entropy loss in Eq. (5) for this multi-class classification. Here, $p(y_i|q)$ is the empirical distribution of the count magnitude, which is directly determined by the true count $c(q)$.

$$\mathcal{L}_{cla}(Q; \Theta) = \frac{1}{|Q|} \sum_{q \in Q} \sum_{i=1}^{m} [-p(y_i|q) \cdot \log p_\Theta(y_i|q)] \tag{5}$$

The overall training loss of LSS is given in Eq. (6), where $\lambda \in [0, 1]$ is a coefficient used to balance the classification and regression loss.

$$\mathcal{L}(Q; \Theta) = (1 - \lambda)\mathcal{L}_{reg}(Q; \Theta) + \lambda \mathcal{L}_{cla}(Q; \Theta) \tag{6}$$

With this auxiliary classification task, our active learner supports three uncertainty functions for the classification task [7, 70] and one uncertainty function for cross-task [96]: ① *Confidence.* $\varphi_{\text{CON}}(q;\Theta) = 1 - max_i p_\Theta(y_i|q)$, to indicate the gap between a perfect prediction, 1.0, and the highest posterior probability of the model. ② *Margin.* $\varphi_{\text{MAR}}(q;\Theta) = p_\Theta(\hat{y_1}|q) - p_\Theta(\hat{y_2}|q)$, where $\hat{y_1}$ and $\hat{y_2}$ are the first and second most likely predictions under the LSS model $\Theta$, respectively. A higher margin indicates the classifier is doubt in differentiating the top two classes. ③ *Entropy.* $\varphi_{\text{ENT}}(q;\Theta) = -\sum_y p_\Theta(y|q)\log p_\Theta(y|q)$ – the entropy of the posterior probability distribution. A higher entropy indicates a flat distribution. ④ *Cross-Task Consistency.* $\varphi_{\text{CTC}}(q;\Theta) = |\hat{y_1} - \log_{10} c_\Theta(q)|^2$, where $\hat{y_1}$ is the most likely prediction of the classification task, $\log_{10} c_\Theta(q)$ is the magnitude of the estimated count in a real-value form, generated by the regression task. $\varphi_{\text{CTC}}$ evaluates the data informativeness by its inconsistency among the classification and regression task, as their outputs are intertwined [96]. We use the squared error to measure the discrepancy between the magnitudes predicted by the classification and regression task.

The four uncertainty functions are used for evaluating query graph uncertainty. In practice, models can be updated periodically by collecting batches of test queries followed by computing true count offline. A preemptive AL strategy, that select queries and compute the true count in parallel [68], can further reduce the delay.

## 6 Experimental Studies

In this section, we give the test setting (§6.1), and report the extensive experiments: ① Compare the accuracy of LSS with state-of-the-art approximate approaches (§6.2) ② Compare the efficiency of LSS with the approximate and exact counting approaches (§6.3) ③ Validate the effectiveness of the active learner (§6.4) ④ Study the robustness of estimation to varying workloads (§6.5) ⑤ Investigate the effectiveness of LSS on query optimization (§6.6).

### 6.1 Experimental Setup

**Implementation and Setting.** We give the settings of LSS. The number of GNN layers is 3, where each hidden layer has 64 units and a Dropout probability 0.5. Such hyper-parameters are set referring to GNN [28, 83, 88]. We use a two-layer MLP to encode the query-level representation and generate the final output. The hidden layer is activated by ReLU and the output layer uses the softmax to predict the probabilistic distribution for the classification task. We set the cross task loss coefficient $\lambda$ to 1/3. The number of hidden units in the attention layer and MLP is tuned within $\{32, 64, 128\}$. For a query graph, we decompose it by computing the 3-hop *BFS*-tree rooted at each node. For the embedding-based encoding, we try 4 scalable task-independent node embedding approaches, i.e., *DeepWalk* [65], *node2vec* [27], *ProNE* [93] and *NRP* [89], and choose *ProNE* for LSS due to its efficiency on large data graphs and stable performance. Following the default setting in [93], the dimension of the embedding is 128.

The learning framework is built on PyTorch [3] with PyTorch Geometric [4]. We use Adam optimizer with a decaying learning rate to train our models. For different datasets, the main hyper-parameters for training are tuned in their empirical range: learning

**Table 2: Real Data Graphs**

| Dataset | $\lvert V\rvert$ | $\lvert E\rvert$ | $\lvert\Sigma\rvert$ | $\lvert\Sigma_E\rvert$ | Ent($\Sigma$) |
|---|---|---|---|---|---|
| aids | 253,598 | 273,955 | 51 | - | 0.93 |
| yeast | 3,112 | 12,519 | 71 | - | 2.92 |
| youtube | 1,134,890 | 2,987,624 | 20 | - | 3.21 |
| wordnet | 76,853 | 120,399 | 5 | - | 0.66 |
| eu2005 | 862,664 | 16,138,468 | 40 | - | 3.68 |
| yago | 12,811,197 | 15,768,516 | 188,883 | 91 | - |

**Table 3: Query Sets**

| Type | Dataset | # Queries | Query Sizes | Range of $c(q)$ | Cov($\Sigma$) |
|---|---|---|---|---|---|
| Homo. | aids | 780 | {3, 6, 9, 12} | $[10^0, 10^6]$ | 0.03 |
| Homo. | yeast | 1,205 | {4, 8, 16, 24, 32} | $[10^2, 10^9]$ | 1.0 |
| Homo. | wordnet | 645 | {4, 8, 12} | $[10^5, 10^{13}]$ | 1.0 |
| Homo. | eu2005 | 518 | {4, 8} | $[10^5, 10^{14}]$ | 1.0 |
| Homo. | yago | 1,366 | {3, 6, 9, 12} | $[10^0, 10^6]$ | 0.1 |
| Iso. | youtube | 910 | {4, 8, 16, 24, 32} | $[10^3, 10^{13}]$ | 1.0 |
| Iso. | eu2005 | 566 | {4, 8} | $[10^5, 10^{14}]$ | 1.0 |

rate $\in [10^{-3}, 10^{-4}]$, epochs $\in [50, 150]$, batch size $\in \{1, 2, 4, 8\}$, L2 penalty of Adam $\in [10^{-3}, 10^{-5}]$. We conduct grid search to choose configurations that have minimal range of q-error for 75%-25% quantiles under 5-fold cross validation. It is worth mentioning that the performance of LSS is not sensitive under these configurations. Both training and prediction are performed on a Linux server with 32 Intel(R) Xeon(R) Silver 4215 CPUs and 128G RAM.

**Datasets.** We test our LSS and ALSS on five real datasets (Table 2). The dataset aids and yago are obtained from [1] while others are obtained from [2], for both data and query graphs. These datasets cover different domains, including biological networks (aids, yeast), social network (youtube), web graph (eu2005), lexical network (wordnet) and Knowledge Graph (yago). aids, yeast, wordnet contain node-labels. For youtube and eu2005, node-labels are randomly drawn and assigned from a label set, following previous approaches for node-labeled subgraph matching [15, 29]. In Table 2, for the data graphs except yago, each node has exact one label. yago also contains edge-labels, as shown in the column $|\Sigma_E|$. We compute the label entropy (Ent($\Sigma$)) to evaluate the distribution of labels on data graph. The entropy is $\sum_{l\in\Sigma_V} p(l)\log p(l)$, where $p(l)$ is $F(l)/|V|$, the fraction that a node is attached to label $l$. The higher the entropy, the more skew the node label distribution. The query graphs are generated by randomly extracting connected subgraphs from the data graph, where the details can be found in [64, 77]. To obtain the exact count, we use state-of-the-art enumeration algorithms, *Graphflow* [57] and *GraphQL* [33] to compute homomorphism and subgraph isomorphism count, respectively. The original query set contains 1,800 queries except aids and yago. Here, we collect the queries whose true count can be computed in 2 hours. Table 3 presents the 7 query sets. The last three columns show the properties of these query sets, where the last column, Cov($\Sigma$), denotes the number of labels per query node on average.

**Baseline Approaches.** To comprehensively evaluate the effectiveness and efficiency of LSS, we compare LSS with a recent benchmark for subgraph matching cardinality estimation, *G-CARE* [1], which is originally built for subgraph homomorphism matching.

*Homomorphism.* The *G-CARE* benchmark contains 3 algorithms for graph data, CSET, SumRDF, IMPR and 4 relational-based approaches: CS, WJ, JSUB and BS. We briefly introduce these 7 algorithms. ① Characteristic Sets (CSET) [60] decomposes a query graph into a set of star-shaped substructures and estimates the count of individual substructure by an index, i.e., characteristic sets. Based on the independence assumption, the counts of the substructure are aggregated as the final count. ② SumRDF [74] builds a summary graph for the data graph, which groups the nodes with similar labels and proximity. The count is estimated as the expectation of matchings over all possible data graphs of the summary graph. ③ IMPR [23] samples visible subgraphs from the data graph by random walks, and counts the number of matchings in these subgraphs. The weighted sum of the counts is returned as an estimation. Note that IMPR only supports query graph with 3-5 nodes, and does not support node-labeled query and data graphs. *G-CARE* revises IMPR to perform sampling on labeled graphs. ④ Correlated Sampling (CS) [84] samples tuples for each edge of the query graph in a correlated fashion, and performs joins on the sampled tuples. ⑤ Wander Join (WJ) [50] performs random walks over the data graph to sample subgraph matchings. It estimates the counts of matchings by Horvitz-Thompson estimator [38]. ⑥ Join Sampling with Upper Bounds (JSUB) [98] extracts a maximal acyclic subgraph from the query graph and estimates the count of this subgraph as the upper bound of the original query. In *G-CARE*, its sampling approach is similar to WJ. ⑦ Bound Sketch (BS) [21] utilizes a set of bounding formulas [42] to estimate the upper bound of a query. In a net shell, CSET, SumRDF and BS are summary-based approaches while IMPR, CS, WJ and JSUB are sampling-based approaches.

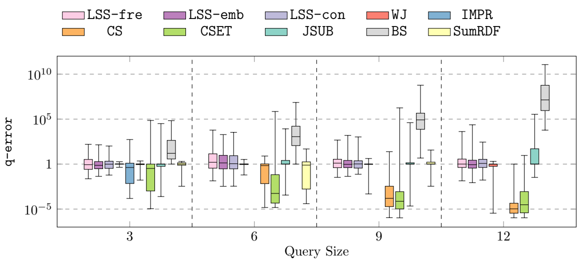
*Isomorphism*. We modify two baselines in *G-CARE*. One is the empirical best approach, WJ. The other is the native graph approach IMPR. We modify them to let them sample isomorphism matchings.

Following the setting of *G-CARE*, we set the sampling ratio to 3% for sampling-based approaches and the timeout limit to 5 minutes.

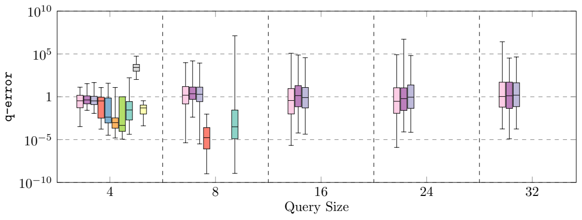
## 6.2 Accuracy on Real Graphs

We investigate the counting accuracy of LSS, including the frequency-based (LSS-fre), embedding-based (LSS-emb) and the concatenated (LSS-con) encoding variants. For LSS models, we split the queries into 80% for training and 20% for testing by stratified sampling on different query sizes, and the test results are collected for the whole query set by 5-fold cross validation. Models are trained without AL. For yago, edge-labels are incorporated into LSS-emb based on Eq. (4) with the frequency-based edge encoding, LSS-fre and LSS-con are not applicable due to the high dimension of a frequency-based node feature encoding.

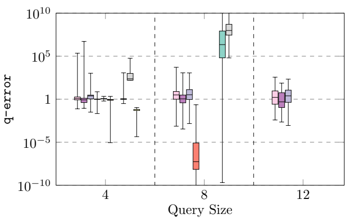
**Homomorphism**. Fig. 4 shows the statistical distribution of q-error of LSS compared with the 7 baseline approaches on the 5 query sets in Table 3. The state-of-the-art baseline WJ performs best on 3/6-node query graph of aids and yago, but easily fails on larger and complex queries, leading to a q-error as large as the true count. In contrast, LSS consistently predicts accurate counts, and the medians of q-error are smaller than 3 across various sizes of the 4 query sets. More specifically, in Fig. 4, a key observation made is that the sampling-based algorithms perform well on aids (Fig. 4(a)). However, for queries of yeast (Fig. 4(b)), wordnet (Fig. 4(c)) and eu2005 (Fig. 4(d)), their performance is degraded drastically, which is caused by severe *sampling failure*. yago (Fig. 4(e)) does not face severe sampling failure but still has an underestimation problem. Note that sampling failure indicates the estimator fails to sample



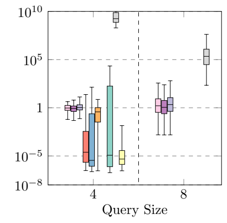
(a) aids



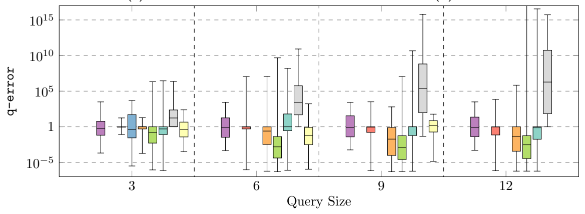
(b) yeast



(c) wordnet

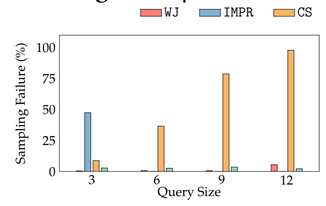


(d) eu2005

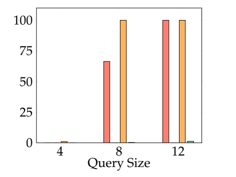


(e) yago

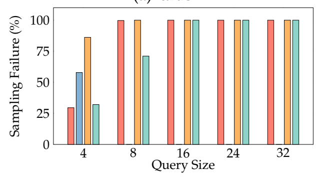
**Figure 4:** q-error **of Homo. Counting**



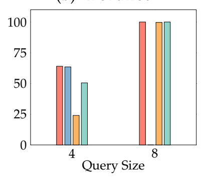
(a) aids



(b) wordnet



(c) yeast



(d) eu2005

**Figure 5: Percentage of Sampling Failure**

valid (partial) matchings for the query graph so that it returns 0 as the estimated count. We do not plot the estimation statistics of an approach for which all the queries suffer from sampling failure problem in Fig. 4. Sampling failure is highly related to data distribution, and the sampling-based approaches fail to explore the

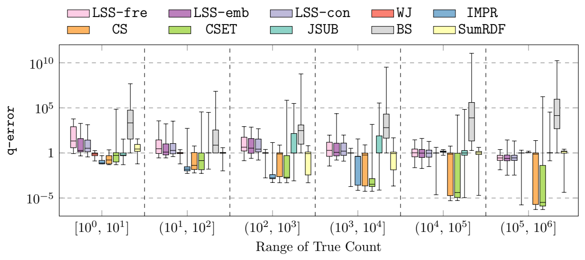


**Figure 6: Varying Range of True count on** aids

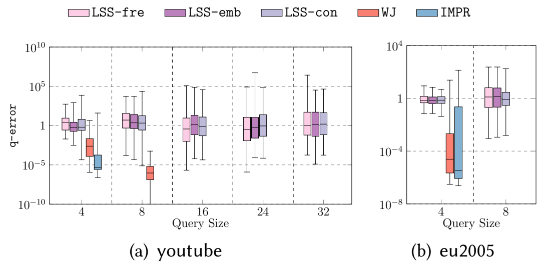


**Figure 7:** q-error **of Iso. Counting**

regions with complex data distribution, i.e., a large number of joins and predicates, by limited samples. On the contrary, modeling complex data distribution is the advantage of neural network models, which is the main reason LSS models can achieve stable accuracy across different datasets and query sizes. Compared with LSS-fre, LSS-emb and LSS-con can achieve better performance. This result indicates that the pre-trained *ProNE* label embedding is effective for the subgraph counting task, by leveraging the distributional similarity among the labels in the data graph.

A close observation in Fig. 5 further presents the percentage of queries that suffer from sampling failure for the 4 query sets. For aids (Fig. 5(a)), WJ and JSUB rarely fail. However, for yeast (Fig. 5(c)) and eu2005 (Fig. 5(d)), all queries with up to 8 nodes fail under WJ, CS and JSUB. For wordnet (Fig. 5(b)), the issue is not much severe; most 4-node queries are free of sampling failure and JSUB has less failure among the three subsets.

The sampling failure is jointly caused by complex label distribution on the data and query graphs, as well as their topology. Take WJ as an example, there is a random walker on the data graph that samples an edge at a time. When sampling one edge, the walker samples from the edges that are incident to the current node and the label constraint of the other node is satisfied. If no such edges, the sample is failed. The risk of the failure derives from one step random walk is unaware of whether or not future sampling can actually find a valid matching. For the query graph, the more attached labels (number of predicates) and the larger the query size (number of joins), the higher the failure risk. For the data graph, the more distinct labels and flatter label distribution, the higher the failure risk. aids has a skewed label distribution, referring to the low label entropy in Table 2. Note that the queries of aids have few labels, referring to the low label density in Table 3. Intuitively, on a skewed distribution, few predicates querying the top frequent labels makes the sampler more likely to sample a valid matching. Similar to aids, wordnet data graph has a skewed label distribution and almost all labels in the queries are the highest frequency labels in the data graph. Therefore, the sampling-based approaches perform well on its 4-node queries, as shown in Fig. 4(c). However, the accuracy drastically degenerates on larger queries. In our experiments, we also raise the sampling ratio from 3% to 10% for yeast, wordnet and eu2005, but the performance gain is slight. Fig. 6 shows the estimation statistics on various range of true count over aids queries. WJ has low q-error for queries with a small true count ($< 10^2$), where underestimation has little influence on the relative error.

Finally, we analyze the baseline approaches. For the summary-based approaches, CSET incurs large error of underestimation due to its independent assumption. SumRDF has the underestimation problem because it assumes the matchings are uniformly distributed on all possible data graphs. BS leads to an overestimation as it computes an upper bound for the join query. For the sampling-based approaches, apart from the best performed WJ, JSUB also performs well on small query graphs as it utilizes WJ as the sampler for acyclic subqueries. IMPR and CS both face underestimation problem because their sampling strategies fail to sample tuples/graphs actually contributing to the join results. Recall that IMPR cannot support query graphs with more than 5 nodes.

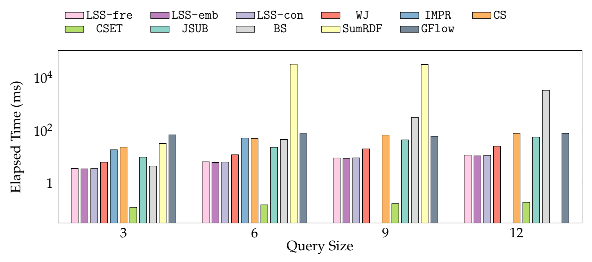
**Isomorphism**. We compare the accuracy of LSS models for isomorphism counting against the revised WJ and IMPR, as shown in Fig. 7. Similar to homomorphism, the severe underestimation of WJ and IMPR are caused by sampling failure, where all the youtube queries of up to 16 nodes are failed under WJ.

## 6.3 Efficiency

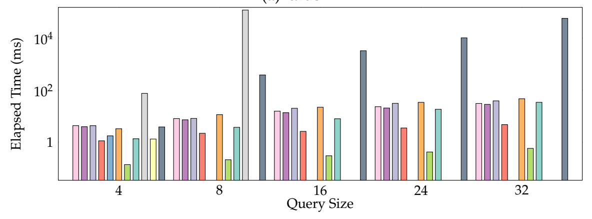
We compare the prediction time of LSS with the query execution time of the baseline algorithms. The setting of LSS models is the same as Section 6.2.

**Homomorphism**. The results of average elapsed time are shown in Fig. 8, where we omit those that timeout the whole query subset and IMPR's result for > 4-node queries in Fig. 8. We also compare with the exact matching algorithm *Graphflow* (GFlow). From the medium scale data graph aids (Fig. 8(a)) and wordnet (Fig. 8(c)) to large data graph eu2005 (Fig. 8(d)) and yago (Fig. 8(e)), our LSS models consistently outperform all the baselines except CSET. Compared with the state-of-the-art WJ, the prediction of LSS is 2-6 × faster on aids, wordnet and yago, and 2 orders faster on eu2005. CSET performs neither sampling nor matching for online queries. Instead, it builds indices for star-structure for the data graph offline, and query the indices for decomposed star substructures of the query graph. On small data graph yeast (Fig. 8(b)) with only thousands of nodes and edges, LSS models do not have many advantages, since sampling-based approaches are efficient. However, the larger the data graph, the more samples or data they need to process. In contrast, the prediction cost of a machine learning model is only determined by its architecture, e.g., number of layers, hidden units. Therefore, It is more suitable to deploy learned models for large data graphs. The exact algorithm, GFlow, still consumes long time on yeast since the true counts are large.

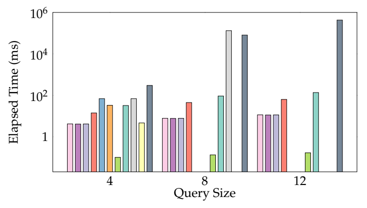
Considering the impact of query graph size. Both SumRDF and BS are timeout in processing large query graphs. SumRDF needs to search the matchings on the summarized data graph, which takes exponential time regarding the query size. BS has to evaluate multiple bounding formulas whose number is also exponentially increasing w.r.t. the query size. As query size increases, the elapsed time of both the sampling-based approaches and LSS grows slowly.



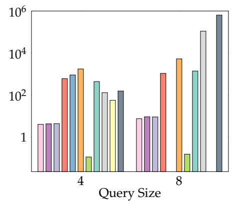
(a) aids



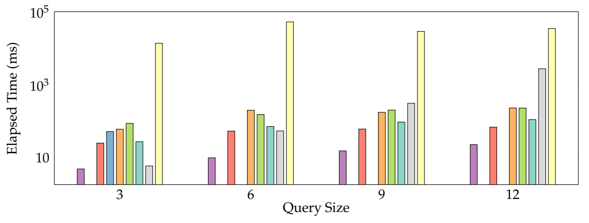
(b) yeast



(c) wordnet

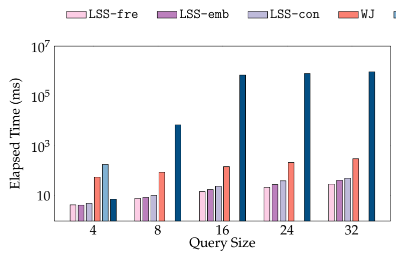


(d) eu2005

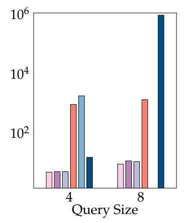


(e) yago

Figure 8: Elapsed Time (ms) of Homo. Counting



(a) youtube



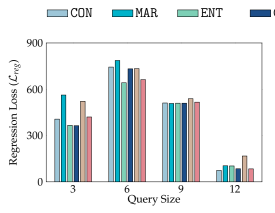
(b) eu2005

Figure 9: Elapsed Time (ms) of Iso. Counting

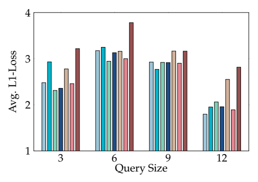
It is worth noting that, due to sampling failure, the sampling-based approaches would early terminate in intermediate join step, especially for large queries. Based on our query decomposition strategy, the number of substructures that needs processing grows linearly regarding the number of query node. For the 3 LSS variants, their prediction time is close. Theoretically, LSS-con could spend longer time as it takes longer vectors as input. In fact, we can only observe a slight difference on yeast. This also reflects the prediction time of neural networks is insensitive to its one layer configuration.

Table 4: Training Time (s) for Homo. Counting

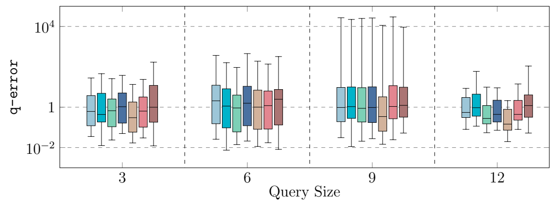
| Dataset | LSS Training | | | Embedding |
|---|---|---|---|---|
| | LSS-fre | LSS-emb | LSS-con | |
| aids | 352.10 | 359.66 | 375.40 | 54.84 |
| yeast | 1458.20 | 1262.93 | 1546.05 | 0.80 |
| wordnet | 390.30 | 380.52 | 386.04 | 20.06 |
| eu2005 | 250.75 | 266.45 | 268.92 | 719.13 |



(a) Total Regression Loss



(b) Average L1-Loss



(c) q-error

Figure 10: Different AL strategies on aids test set

**Isomorphism**. Fig. 9 presents the elapsed time of isomorphism counting on youtube (Fig. 9(a)) and eu2005 (Fig. 9(b)). Also, we report the query execution time of the exact algorithm, *GraphQL* (GQL). LSS outperforms WJ one order on youtube and two orders on eu2005. On large graphs, IMPR is even much slower than the exact algorithm and WJ is also slower than GQL for the smaller, 4-node query graph. The performance advantages of GQL benefit from its powerful filtering techniques.

**Training Time**. We report the training time of LSS on 32 CPUs. For the 4 homomorphism query sets, the training time averaged on cross validation for 50 epochs is shown in Table 4. The training time of LSS is mainly determined by the number of training queries and the epochs, and is independent to the size of the data graph. This is the advantage of supervised learning; as long as training queries can be collected, LSS models are scalable for large data graphs. For LSS-con, as the input of GNN is the longer concatenated vector, its training time is slightly longer than LSS-fre and LSS-emb. In addition, training on LSS-emb could be slightly faster than LSS-fre. We speculate the reason could be its 128 dimension input is well compatible with the parallel computing architecture. The last column 'Embedding' in Table 4 is the single-thread pre-training time of node embedding spend by *ProNE*. The time complexity of *ProNE* is linear regarding the number of nodes and edges in $G_L$, the label-augmented graph [93]. This time complexity makes it scalable for large data graphs, e.g., eu2005.

## 6.4 Active Learning

We compare the 4 strategies of ALSS, i.e., classification confidence (CON), margin (MAR), entropy (ENT) and cross-task consistency (CTC) with 2 strategies: random selection (RAN) and model ensemble (ENS). Specifically, RAN selects queries from the test query set randomly. This strategy is regarded as passive learning but is an effective heuristics in many cases [59]. For ENS, we train 5 LSS models at one

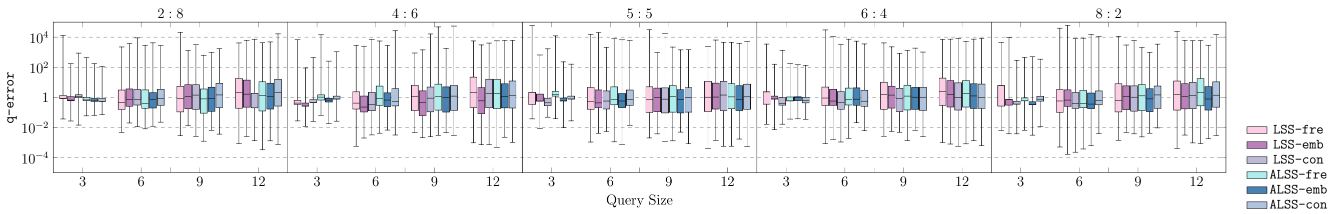


**Figure 11:** q-error **of** LSS **and** ALSS **models trained on varying workloads**

time and each model is fed to 80% of the training data split by 5-fold cross validation [47]. The final prediction is the geometric mean of that of the 5 models and the variance serves as the uncertainty. Once newly training data obtained, all the 5 models need retraining.

We test the aids query set as the queries are evenly distributed on various sizes and true counts. We split the whole query set into 60% for training a base LSS model, 20% as the selection pool $\hat{Q}$, and 20% as the test set $Q_T$. Given the base model (5 models for ENS) trained by 50 epochs, we apply 2 iterations uncertainty sampling since the retraining would not be frequent. In one iteration, 50 queries are selected from $\hat{Q}$ and added into the training set, followed by 50 epochs retraining for model update. Fig. 10(a) shows the final regression loss on the test query subsets, and Fig. 10(b) shows the average L1-loss compared with the original test result (ORI) on the base model. The average L1-loss, $\sum_{q \in Q_T} |\log c(q) - \log c_\Theta(q)| / |Q_T|$ reflects the average logarithmic q-error. Fig 10 demonstrates that adding new training data helps to improve the generalization capability of LSS, for both passive (RAN) and active learning (CON, MAR, ENT, CTC, and ENS). Compared with RAN, the active learning strategies are more effective in reducing the prediction errors. Among the 5 active learning strategies, ENT and CTC are most effective whereas MAR and CON have relatively worse performance. Recall that CON and MAR only consider the top-1 and top-2 likely magnitude of the count, respectively. As the estimated count always fall into two adjacent magnitudes, the uncertainty cannot be well reflected by only two probabilities. Note that the training cost of ENS is roughly 5 times of the strategies of ALSS. Fig. 10(c) further shows a detailed error distribution on different query sizes. Also, all the strategies attempt to shrink the min-max error range and CTC, ENT and ENS are more effective. We find that fine-tuning the model is not always beneficial to all queries, e.g., 9-node queries in Fig. 10(c).

Note that the target of AL in LSS is not to narrow down the data complexity. LSS is a complex neural network model, and we cannot make reasonable assumptions for the distribution of the query graphs or the true counts as statistical learning does [30, 59]. Instead, the active learning of LSS aims to perform more efficient model update compared with passive learning, under equivalent data volume.

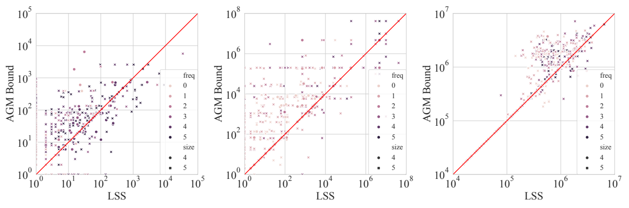
## 6.5 Robustness to Workload Shifts

To study the LSS robustness, we sample a large pool of queries by extracting 3/6/9/12-node subgraphs from the aids data graph. The number of testing queries is fixed 1,440, and is divided into 4 sets where one is 360. One set is used for testing 3/6/9/12-node queries respectively. The total number of training queries is 900. We trained the 3 LSS variants over 5 workloads independently, by varying the fraction of large/small queries within {2:8, 4:6, 5:5, 6:4, 8:2}, where 3/6-node queries are regarded as small and 9/12-node queries are large. For each LSS model, we further apply AL to update the model 2 times, one with 50 queries selected by cross-task consistency (CTC). Fig. 11 shows the testing q-error of the 30 models, i.e., 3 LSS and their ALSS variations trained over 5 workloads.

The key observation is that the testing performance is robust when workloads vary, particularly for LSS-emb. As query workload changes, the q-error varies over small queries mainly, but the fluctuation would not surpass one order for LSS-emb. We explain it. Due to query decomposition, large queries can benefit from small queries as the GNN is well trained. This also suggests that we can use more small queries or postpone training large queries in practice, because small queries are cheaper to process. As observed in testing, AL has the ability to balance the uneven query distribution by selecting more queries that appear less in the initial training set. Thus, ALSS always achieves better performance than LSS. And AL brings remarkable improvement on small queries since the possible world of small queries are rather smaller.

## 6.6 Query Optimization by LSS

We discuss how LSS can be integrated into query optimization in state-of-the-art systems [5, 6, 10, 92] to support complex self-joins over one relation. This is motivated by the fact that such join approaches [5, 6, 10, 92] are shown efficient to enumerate subgraph matchings by homomorphism. One representative system is *EmptyHeaded* [5, 6], which uses multi-way joins to speed up subgraph enumeration. In such systems, a key issue is how to support cyclic join queries, as self-joins are complex for subgraph enumeration. The technique used is to find a tree-structured join plan by tree-width decomposition (e.g., GHD for Generalized Hypertree Decomposition), where joins among the tree-nodes are acyclic, but joins in a tree-node may be cyclic. In brief, consider a cyclic join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$, where all $k$ relations are for one relation (e.g., edge table for the graph $G = (V, E, L, \Sigma)$) but renamed. GHD finds $l$ ($< k$) relations $R_{\tau_i}$, for $1 \leq i \leq l$, where $R_{\tau_i} = \bowtie \lambda(\tau_i)$ and $\lambda(\tau_i)$ is a subset of input relations of the cyclic join, such that $R_{\tau_1} \bowtie R_{\tau_2} \bowtie \cdots \bowtie R_{\tau_l}$ is an acyclic join whose result is the same as the original cyclic join. It is critical to select a good GHD, as it determines the query processing cost. In the GHD-based systems, the cost of each GHD-based plan is $\rho^* = \max_{i<l} \rho(\lambda(\tau_i))$, where $\rho(\lambda(\tau_i))$ is fractional edge covering number [26]. And, for any $R_{\tau_i}$, it satisfies $\left|R_{\tau_i}\right| \leq |E|^{\rho^*}$ based on AGM bound [12]. GHD finds a plan with smaller $\rho^*$ as it tends to contain smaller $R_{\tau_i}$, and smaller cost for generating $R_{\tau_i}$ and evaluation. However, such estimation can deviate far from real cost especially when selection predicates exist [64]. Instead of estimating the size $|R_\tau|$ based on AGM bound, in this work, we estimate $|R_\tau|$ using LSS. More specifically, we estimate $\left|R_{\tau_i}\right|$ by constructing a query graph $q = (V_q, E_q, L, \Sigma)$. The query graph $q$ will be fed into LSS to get its estimated count.



(a) yeast (b) wordnet (c) eu2005

**Figure 12: Cost of query plan:** AGM **vs.** LSS

We investigate how LSS reduces the cost of GHD. The testing is conducted on yeast, wordnet, and eu2005, ranging from small to large data graphs. For each data graph, we generate a training set including 202 3-node, and 608 4-node query graphs, whose labels are randomly assigned, and train a corresponding LSS model. For testing and comparison, 60 4-node and 210 5-node unlabeled patterns are generated. For each unlabeled pattern, to evaluate the influence of label distribution, we vary the number of nodes that the frequent labels are attached. Here, frequent labels are the labels whose frequency ranks the top 20% in Σ. We randomly attach frequent labels on zero to all the nodes, and the remaining nodes are randomly assigned to infrequent labels, resulting in 1,560 deduplicated queries in total.

Here, we use real cardinality as ground truth and set the true cost of each selected GHD to $\max_{i<l} |R_{\tau_i}|$. A close observation on the 1,560 test queries is shown in Fig. 12, where 'freq' denotes the number of nodes associated with frequent labels and 'size' denotes the query size. The queries above the line indicate LSS selects the better plan and the queries below the line indicate AGM selects the better plan. In general, LSS can recommend GHD with lower cost, even up to 3-4 orders of magnitude better than AGM over yeast (Fig. 12(a)) and wordnet (Fig. 12(b)). On the large graph eu2005 (Fig. 12(c)), LSS can effectively reduce the cost, especially for queries with higher cost ($> 10^6$). For wordnet (Fig. 12(b)) with a skewed label distribution over only 5 labels, AGM performs well on queries where most labels are frequent. These queries are close to unlabeled queries so that they are in accordance with the assumption of AGM. However, LSS still outperforms AGM on larger queries with infrequent or frequent-infrequent mixed labels. These results further confirm the superiority of LSS in modeling complex data distribution.

## 7 Related Work

**Subgraph Matching**. Subgraph matching has been extensively studied. A recent analysis for state-of-the-art in-memory algorithms [11, 14, 15, 17, 41, 71] can be found in [77]. Most of them follow Ullman's backtracing search [80], and are optimized in the aspects of filtering candidate nodes, matching ordering, enumerating partial results, etc. Another line of work transforms the subgraph matching query to a multi-way join query on a relational database. Database systems like *EmptyHeaded* [5] and *Graphflow* [57] use worst-case optimal join algorithm [61] to evaluate these subgraph matching queries in-memory. These algorithms can achieve the exact subgraph count by enumerating all the matchings.

**Exact and Approximate Subgraph Counting**. There are many exact and approximate subgraph counting approaches [69]. For exact counting, approaches in the literature are categorized into enumeration and analytic approaches. The enumeration approaches [35, 39, 63] obtain the count via enumerating all the subgraphs. In contrast, the analytical approaches solve the count in an analytical fashion by leveraging the counts of the same or smaller size query graphs, thereby avoid listing the matching. Two main analytical approaches are matrix-based [56, 62] and decomposition-based [8, 66]. For approximate subgraph counting, various estimation strategies have been explored such as path sampling [40, 85], color coding [19, 20], random walk [23], and graph summarization [60, 74].

Most of above approaches are designed to count graphlets [67], a.k.a. graph motifs, small, connected graph queries, over a simple undirected graph. The exact and approximate counting can support up to 5-node [66] and 10-node query graph [20], respectively. In addition, they cannot support labeled graphs and the extension is nontrivial, especially for the analytical-based and approximate approaches. Those impede their usage on larger general queries.

**ML/DL for Cardinality Estimation and AQP**. ML/DL models are exploited to perform cardinality estimation and AQP for *RDBMS*. We briefly review ML/DL approaches in Table 1. *DBEst* [54] builds kernel density estimation (KDE) models and tree-based regressors to conduct AQP. *DeepDB* [34] adopts Sum-Product Networks (SPNs) to learn the joint probability distribution of attributes. An SQL query is complied to a product of expectations or probability queries on the SPNs, where the product is based on independence of the attributes. [79] uses deep generative model, e.g., Variational Autoencoder (VAE) to model the joint probability distribution of attributes, which only support analytical aggregate query on a single table. [43] estimates multivariate probability distributions of a relation to perform cardinality estimation by KDE. Multiple joins can be estimated by building the KDE estimator on pre-computed join result or an estimation formula that leverages samples from the multiple relations as well as the models. Deep autoregressive model, e.g., Masked Autoencoder (MADE), is also adopted to learn the joint probability distribution [32, 90], which decomposes the joint distribution to a product of conditional distributions. Kipf et. al. propose a multi-set convolutional neural network (MSCN) to express query features using sets [44]. Anshuman et. al. [25] use MLP and tree-based regressor to express multiple attributes range queries. Sun et. al. [76] extract the features of physical query plan by Tree LSTM to estimate the query execution cost as well as the cardinality.

## 8 Conclusion

In this paper, we propose a neural network model as a learned sketch for subgraph counting over a large data graph, which can be used to either homomorphism or subgraph isomorphism counting. In addition, an active learning strategy is devised for the learned sketch, aiming to perform efficient model updates via selecting informative query graphs. Our extensive experiments demonstrate that the learned sketch provides fast and accurate estimation. The prediction is 2 orders faster than baseline approaches on large data graphs. And its medians of q-error are smaller than 3 across various query sets where state-of-the-art baselines easily fail on large and complex query graphs, resulting in a severe underestimation. Furthermore, our study on query optimization of a multi-way join system indicates that our learned sketch can help the system to recommend high quality query plans, whose cost can be up to 3 orders of magnitude cheaper than the classical approach.

## References

[1] https://github.com/yspark-dblab/gcare.

[2] https://github.com/RapidsAtHKUST/SubgraphMatching.

[3] Pytorch. https://github.com/pytorch/pytorch.

[4] Pytorch Geometric. https://github.com/rusty1s/pytorch_geometric.

[5] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[6] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Old techniques for new join algorithms: A case study in rdf processing. In *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, pages 97–102. IEEE, 2016.

[7] C. C. Aggarwal, X. Kong, Q. Gu, J. Han, and P. S. Yu. Active learning: A survey. In *Data Classification: Algorithms and Applications*, pages 571–606. 2014.

[8] N. K. Ahmed, J. Neville, R. A. Rossi, and N. G. Duffield. Efficient graphlet counting for large networks. In *Proc. ICDM'15*, pages 1–10, 2015.

[9] N. K. Ahmed, J. Neville, R. A. Rossi, N. G. Duffield, and T. L. Willke. Graphlet decomposition: framework, algorithms, and applications. *Knowl. Inf. Syst.*, 50(3):689–722, 2017.

[10] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB*, 11(6), 2018.

[11] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In *Proc. CPAIOR'19*, pages 20–38, 2019.

[12] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. In *Proc. of FOCS'08*, pages 739–748, 2008.

[13] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[14] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In *Proc. SIGMOD'19*, pages 1447–1462, 2019.

[15] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *Proc. SIGMOD'16*, pages 1199–1214, 2016.

[16] M. Bodirsky. Graph homomorphisms and universal algebra course notes. 2015.

[17] V. Bonnici, R. Giugno, A. Pulvirenti, D. E. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.*, 14(S-7):S13, 2013.

[18] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. Vishwanathan, A. J. Smola, and H.-P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1):i47–i56, 2005.

[19] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Counting graphlets: Space vs time. In *Proc. WSDM'17*, pages 557–566, 2017.

[20] M. Bressan, S. Leucci, and A. Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *Proc. VLDB*, 12(11):1651–1663, 2019.

[21] W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proc. SIGMOD'19*, pages 18–35, 2019.

[22] W. Cai, Y. Zhang, and J. Zhou. Maximizing expected model change for active learning in regression. In *Proc. ICDM'13*, pages 51–60, 2013.

[23] X. Chen and J. C. S. Lui. Mining graphlet counts in online social networks. In *Proc. ICDM'16*, pages 71–80, 2016.

[24] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[25] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB*, 12(9):1044–1057, 2019.

[26] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree Decompositions: Questions and Answers. In *Proc. of PODS'16*, pages 57–74, 2016.

[27] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proc. KDD'16*, pages 855–864, 2016.

[28] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proc. NeurIPS'17*, pages 1024–1034, 2017.

[29] M. Han, H. Kim, G. Gu, K. Park, and W. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proc. SIGMOD'19*, pages 1429–1446, 2019.

[30] S. Hanneke. A statistical theory of active learning. *Foundations and Trends in Machine Learning*, pages 1–212, 2013.

[31] Z. Harchaoui and F. R. Bach. Image classification with segmentation graph kernels. In *Proc. CVPR'07*, 2007.

[32] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Deep learning models for selectivity estimation of multi-attribute queries. In *Proc. SIGMOD'20*, pages 1035–1050, 2020.

[33] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. ACM SIGMOD'08*, 2008.

[34] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB*, 13(7):992–1005, 2020.

[35] T. Hocevar and J. Demsar. Combinatorial algorithm for counting small induced graphs and orbits. *CoRR*, abs/1601.06834, 2016.

[36] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[37] K. Hornik, M. B. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[38] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association*, 47(260):663–685, 1952.

[39] S. Jain et al. Impact of memory space optimization technique on fast network motif search algorithm. In *Advances in Computer and Computational Sciences*, pages 559–567. Springer, 2017.

[40] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proc. WWW'15*, pages 495–505, 2015.

[41] A. Jüttner and P. Madarasi. VF2++ - an improved subgraph isomorphism algorithm. *Discret. Appl. Math.*, 242:69–81, 2018.

[42] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. PODS'17*, pages 429–444, 2017.

[43] M. Kiefer, M. Heimel, S. Breß, and V. Markl. Estimating join selectivities using bandwidth-optimized kernel density models. *Proc. VLDB*, 10(13):2085–2096, 2017.

[44] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *Proc. CIDR'19*, 2019.

[45] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proc. ICLR'17*, 2017.

[46] T. G. Kolda, A. Pinar, and C. Seshadhri. Triadic measures on graphs: The power of wedge sampling. In *Proc. ICDM'13*, pages 10–18, 2013.

[47] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. In *Proc. NIPS'94*, pages 231–238, 1994.

[48] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB*, 9(3):204–215, 2015.

[49] D. D. Lewis and J. Catlett. Heterogeneous uncertainty sampling for supervised learning. In *Proc. ICML'94*, pages 148–156, 1994.

[50] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proc. SIGMOD'16*, pages 615–629, 2016.

[51] J. Li, Y. Rong, H. Cheng, H. Meng, W. Huang, and J. Huang. Semi-supervised graph classification: A hierarchical graph perspective. In *Proc. WWW'19*, pages 972–982, 2019.

[52] X. Liu, H. Pan, M. He, Y. Song, X. Jiang, and L. Shang. Neural subgraph isomorphism counting. In *Proc. KDD '20*, pages 1959–1969, 2020.

[53] L. Ma, B. Ding, S. Das, and A. Swaminathan. Active learning for ML enhanced database systems. In *Proc. SIGMOD'20*, pages 175–191, 2020.

[54] Q. Ma and P. Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proc. SIGMOD'19*, pages 1553–1570, 2019.

[55] D. J. C. MacKay. Information-based objective functions for active data selection. *Neural Comput.*, 4(4):590–604, 1992.

[56] I. Melckenbeeck, P. Audenaert, D. Colle, and M. Pickavet. Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations. *Bioinform.*, 34(8):1372–1380, 2018.

[57] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB*, 12(11):1692–1704, 2019.

[58] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[59] S. Mussmann and P. Liang. On the relationship between data efficiency and error for uncertainty sampling. In *Proc. ICML'18*, pages 3671–3679, 2018.

[60] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proc. ICDE'11*, pages 984–994, 2011.

[61] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. PODS'18*, pages 111–124, 2018.

[62] M. Ortmann and U. Brandes. Efficient orbit-aware triad and quad census in directed and undirected graphs. *Applied Network Science*, 2:13, 2017.

[63] P. Paredes and P. M. P. Ribeiro. Towards a faster network-centric subgraph census. In *Proc. ASONAM '13*, pages 264–271, 2013.

[64] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W. Han. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proc. SIGMOD'20*, pages 1099–1114, 2020.

[65] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *Proc. KDD'14*, pages 701–710, 2014.

[66] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: efficiently counting all 5-vertex subgraphs. In *Proc. WWW'17*, pages 1431–1440, 2017.

[67] N. Przulj. Biological network comparison using graphlet degree distribution. *Bioinform.*, 23(2):177–183, 2007.
[68] F. Regol, S. Pal, Y. Zhang, and M. Coates. Active learning on attributed graphs via graph cognizant logistic regression and preemptive query generation. In *Proc. ICML'20*.
[69] P. Ribeiro, P. Paredes, M. E. P. Silva, D. Aparício, and F. Silva. A survey on subgraph counting: Concepts, algorithms and applications to network motifs and graphlets. *CoRR*, abs/1910.13011, 2019.
[70] B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
[71] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB*, 1(1):364–375, 2008.
[72] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *Proc. AISTATS'09*, pages 488–495, 2009.
[73] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.
[74] G. Stefanoni, B. Motik, and E. V. Kostylev. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In *Proc. WWW'18*, pages 1043–1052, 2018.
[75] S. H. Strogatz. Exploring complex networks. *nature*, 410(6825):268–276, 2001.
[76] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB*, 13(3):307–319, 2019.
[77] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *Proc. SIGMOD'20*, pages 1083–1098, 2020.
[78] N. P. Tatonetti, P. Y. Patrick, R. Daneshjou, and R. B. Altman. Data-driven prediction of drug effects and interactions. *Science translational medicine*, 4(125):125ra31–125ra31, 2012.
[79] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das. Approximate query processing for data exploration using deep generative models. In *Proc. ICDE'20*, pages 1309–1320, 2020.
[80] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
[81] V. Vacic, L. M. Iakoucheva, S. Lonardi, and P. Radivojac. Graphlet kernels for prediction of functional residues in protein structures. *J. Comput. Biol.*, 17(1):55–72, 2010.
[82] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Proc. NeurIPS'17*, pages 5998–6008, 2017.
[83] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *Proc. ICLR'18*, 2018.
[84] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen. Join size estimation subject to filter conditions. *Proc. VLDB*, 8(12):1530–1541, 2015.
[85] P. Wang, J. Zhao, X. Zhang, Z. Li, J. Cheng, J. C. S. Lui, D. Towsley, J. Tao, and X. Guan. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Trans. Knowl. Data Eng.*, 30(1):73–86, 2018.
[86] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.
[87] B. Weisfeiler and A. A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.
[88] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *Proc. ICLR'19*, 2019.
[89] R. Yang, J. Shi, X. Xiao, Y. Yang, and S. S. Bhowmick. Homogeneous network embedding for massive graphs via reweighted personalized pagerank. *Proc. VLDB*, 13(5):670–683, 2020.
[90] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, P. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB*, 13(3):279–292, 2019.
[91] Z. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Proc. NeurIPS'18*, pages 4805–4815, 2018.
[92] H. Zhang, J. X. Yu, Y. Zhang, K. Zhao, and H. Cheng. Distributed subgraph counting: A general approach. *Proc. VLDB*, 13(11), 2020.
[93] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding. Prone: Fast and scalable network representation learning. In *Proc. IJCAI'19*, pages 4278–4284, 2019.
[94] L. Zhang, M. Song, Z. Liu, X. Liu, J. Bu, and C. Chen. Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation. In *Proc. CVPR'13*, pages 1908–1915, 2013.
[95] L. Zhang, M. Song, Q. Zhao, X. Liu, J. Bu, and C. Chen. Probabilistic graphlet transfer for photo cropping. *IEEE Trans. Image Process.*, 22(2):802–815, 2013.
[96] Y. Zhang. Multi-task active learning with output constraints. In *Proc. AAAI'10*, 2010.
[97] Y. Zhang and Q. Yang. A survey on multi-task learning. *CoRR*, abs/1707.08114, 2017.
[98] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *Proc. SIGMOD'18*, pages 1525–1539, 2018.
[99] G. X. Zheng, J. M. Terry, P. Belgrader, P. Ryvkin, Z. W. Bent, R. Wilson, S. B. Ziraldo, T. D. Wheeler, G. P. McDermott, J. Zhu, et al. Massively parallel digital transcriptional profiling of single cells. *Nature communications*, 8(1):1–12, 2017.
[100] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

## Acknowledgement

This work is supported by the Research Grants Council of Hong Kong, China under No. 14203618, No. 14202919 and No. 14205520.