# SPACE: Cardinality Estimation for Path Queries Using Cardinality-Aware Sequence-based Learning

MEHMET AYTIMUR, University of Konstanz, Germany
THEODOROS CHONDROGIANNIS, University of Konstanz, Germany and Norwegian University of Science and Technology, Norway
MICHAEL GROSSNIKLAUS, University of Konstanz, Germany and Thurgau Institute for Digitial Transformation, Switzerland

Cardinality estimation is a central task of cost-based database query optimization. Accurate estimates enable optimizers to identify and avoid expensive plans requiring large intermediate results. While cardinality estimation has been studied extensively in relational databases, research in the setting of graph databases has been more scarce. Furthermore, recent studies have shown that machine-learning-based methods can be utilized for cardinality estimation in both relational and graph databases. In this paper, we focus on the problem of estimating the cardinality of path patterns in graph databases, and we propose the *Sequence-based Path Pattern Cardinality Estimator* (SPACE). Our approach treats path patterns as sequences of node labels and edge types and assign similar cardinalities to path patterns with similar node and edge order. SPACE uses a dual approach: it encodes the sequence of nodes and edges to capture structural characteristics of the path pattern, while also incorporating a cardinality-based encoding to integrate cardinality information throughout learning. In a comprehensive experimental evaluation, we show that our method outperforms the state of the art in terms of both accuracy ($Q$-error) and training time.

CCS Concepts: • **Information systems** → **Query optimization**; • **Computing methodologies** → *Neural networks.*

Additional Key Words and Phrases: query optimization, cardinality estimation, machine learning

**ACM Reference Format:**
Mehmet Aytimur, Theodoros Chondrogiannis, and Michael Grossniklaus. 2025. SPACE: Cardinality Estimation for Path Queries Using Cardinality-Aware Sequence-based Learning. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 218 (June 2025), 26 pages. https://doi.org/10.1145/3725355

## 1 Introduction

Regardless of their data model, all database systems that support queries expressed in a declarative programming language rely on a query optimizer that translates such queries into efficient execution plans. To find an efficient execution plan, the optimizer enumerates equivalent plans and assigns an estimated cost to each alternative plan, ultimately selecting the plan with the lowest cost for execution. Leis et al. [25] have demonstrated that cardinality estimation, i.e., estimating the number of results returned by a query and all its sub-queries, is one of the most important sub-problems of query optimization. The precision of estimated cardinalities has a major impact on the runtime of the execution plan found by the optimizer.

Authors' Contact Information: Mehmet Aytimur, mehmet.aytimur@uni-kn, University of Konstanz, Germany; Theodoros Chondrogiannis, theodoros.chondrogiannis@uni.kn, University of Konstanz, Germany and Norwegian University of Science and Technology, Trondheim, Norway; Michael Grossniklaus, michael.grossniklaus@uni.kn, University of Konstanz, Germany and Thurgau Institute for Digitial Transformation, Kreuzlingen, Switzerland.


ACM 2836-6573/2025/6-ART218
https://doi.org/10.1145/3725355

Consequently, numerous cardinality estimation techniques in relational and non-relational database systems have been developed. Initial techniques that, in retrospect, are often referred to as *traditional approaches* were based on statistical summary data structures such as histograms or sampling. More recently, several so-called *learned approaches* have proposed different types of neural network models to estimate cardinalities. While many of these learned techniques address cardinality estimation in relational database systems, learned cardinality estimation in non-relational database systems, particularly graph database systems, has received less attention. One reason for this bias towards relational database systems might be that existing graph database systems are comparatively less mature and not as widely adopted. Another reason might be that graph database systems are more heterogeneous as RDF and Property Graph database systems co-exist with slightly different data models and query semantics.

We argue that the learned cardinality estimation techniques that do exist for path queries in graph database systems [9, 41] suffer from two major drawbacks. First, the embeddings used in these approaches map full path expressions to a single true cardinality. As a consequence, they are unable to capture the structural relationships between a path and its sub-paths, which can lead to very imprecise estimations. Second, existing techniques have been designed and developed with a specific graph data model and query semantics in mind and are, therefore, not general enough to be deployed in all existing graph database systems.

In this paper, we propose the *Sequence-based Path Pattern Cardinality Estimator* (SPACE), a novel approach to cardinality estimation for path queries in graph databases. According to an analysis of query logs conducted by Bonifati et al. [6], path queries are the most prevalent type of query in graph database systems. SPACE encodes a paths as an alternating sequence of node labels and edge types and learns a cardinality for each sequence prefix. Our experimental evaluation demonstrates that this encoding leads to much more precise cardinality estimates w.r.t. state-of-the-art learned techniques for graph database systems. Furthermore, the design of SPACE is general enough to accommodate different graph data models and query semantics. In particular, we will show that different query semantics can be supported purely by slightly changing the algorithm for training data generation.

More specifically, we claim the following technical contributions:

- We introduce *Sequence-based Path Pattern Cardinality Estimation* (SPACE) for estimating the cardinality of path pattern matching queries on labeled graphs (Section 4).
- We present a sampling method to extract path patterns from a given graph along with an algorithm to compute cardinality sequences that we use to train our model (Section 5).
- We conduct a thorough experimental evaluation where we demonstrate the improved accuracy achieved by SPACE compared to existing approaches and the positive effect of the improved estimates on query optimization (Section 6).

We begin in Section 2 with a discussion of the related work. We introduce preliminaries and formally define the problem addressed in this paper in Section 3, discuss the extensions and adaptations that have to be made to SPACE to deploy it in a production environment in Section 7, and give concluding remarks in Section 8.

## 2 Related Work

Cardinality estimation techniques can be classified along two axes. The first axis discerns whether a technique has been developed for relational or graph database systems, whereas the second axis distinguishes whether a traditional or a learned summary data structure is used. In this classification, the technique proposed in this paper falls into the class of learned summary data structure for graph database systems. However, since many techniques for graph databases draw inspiration from recent (learned) cardinality estimation techniques for relational databases, we also consider these approaches as related work.

## 2.1 Approaches for Relational Databases

Traditional methods for cardinality estimation on relational databases often rely on statistical summaries, such as histograms [2, 3, 30, 36, 47], sampling techniques [26, 49], and suffix tree-based methods [22, 24] to derive heuristics from the underlying data distribution. However, these approaches face a trade-off between the size of the summary data structure and the precision of the estimates, which is often difficult to balance in practice.

Deep-learning-based methods for cardinality estimation have gained popularity to overcome the limitations of traditional approaches. Early work by Lakshmi and Zhou [23] introduced neural networks for selectivity estimation, while Kraska et al. [21] used a mixture of neural networks to learn data distributions for fast indexing. More recently, Kipf et al. [20] proposed a set-based deep-learning model for estimating correlated join selectivities. Reiner and Grossniklaus [37] introduce a sample-efficient approach to estimate cardinalities in large data sets by leveraging geometric deep-learning techniques. Other recent works seek to tackle challenging types of queries, including `GROUP BY` [19] and range queries [10, 54]. Moreover, deep-learning-based approaches have gained popularity for `LIKE`-query cardinality estimation as well. Astrid [42] introduced two techniques: Astrid-EMBED, which creates cardinality-aware embeddings to estimate `LIKE`-query cardinality, and Astrid-NLM, which uses neural language models. LPLM [4] also addresses `LIKE` cardinality estimation using a novel neural language model. While estimating the cardinality of `LIKE`-queries is similar to estimating path cardinalities in graph databases, methods for `LIKE`-queries rely on fully materialized data (e.g., string columns). In contrast, enumerating all paths in a graph is impractical [5], making `LIKE`-query methods unsuitable for our problem.

## 2.2 Approaches for Graph Databases

While cardinality estimation has been widely studied in relational databases, research in graph databases is more scarce. Early works in graph cardinality estimation adapted traditional relational methods [34, 43, 45], which summarize dataset distributions and assume attribute independence. For example, Park et al. [34] adapted the Wander Join algorithm [26], a sampling-based method for online aggregation, to estimate cardinalities for subgraph matching queries by performing random walks to collect samples.

Several methods for cardinality estimation of subgraph matching queries have been proposed for graph databases. Characteristic Sets [32], used by the RDF-3X triplestore [33], estimates cardinality by decomposing queries into non-overlapping star sub-patterns, using precomputed counts for cardinality estimation and the independence assumption for remaining joins. SumRDF [44] builds a graph summary by merging similar nodes and recording edge multiplicities, with cardinality estimation performed on the summary graph. The accuracy of this method depends on the size of the summary. Regarding property graphs, Gubichev [15] introduced a technique adopted by Neo4j that requires minimal statistical information about node labels, relationships, and relationship types. Wörteler et al. [51] proposed a statistical model that, while requiring more information than Gubichev's method, considers how cardinality estimates propagate and transform along a subgraph pattern. Yakovets et al. [53] focused on path queries and explored histogram-based strategies to optimize bucket variance and improve estimation accuracy. Park et al. [34] evaluated multiple existing cardinality estimation methods for graph databases.

Chen et al. [7] proposed a traditional compression-based method that estimates cardinalities using a Markov table, which stores the cardinalities of all sub-paths up to a certain length. Mhedhbi and

Salihoglu [27] introduced a hybrid approach that combines efficient binary joins with worst-case optimal joins to estimate the cardinality of a query.

Recent advances have introduced learning-based techniques and probabilistic models to improve estimation accuracy for graph structures. LSS [56] uses ProNE [55] embeddings and a Graph Neural Network (GNN) [40] to estimate subgraph matching cardinalities, though it only supports undirected, unlabeled graphs. LMKG [9] applies deep learning to estimate cardinalities over knowledge graphs using both supervised and unsupervised learning methods. GNCE [41] utilizes RDF2Vec [38] embeddings and a GNN to predict cardinalities of conjunctive queries over knowledge graphs. LMKG and GNCE are the most closely related to our work.

## 3 Preliminaries and Problem Statement

This section introduces the concepts that the work presented in this paper builds upon and provides a formal statement of the problem addressed in this paper.

### 3.1 Path Pattern Matching

A *directed labeled graph* $G$ is defined as $G = (N, E, L, T, \rho, \lambda, \tau)$. $V$ is a finite set of nodes, $E$ is a finite sets of edges, $L$ is a finite set of node labels, $T$ is a finite set of edge types, $\rho : E \rightarrow N \times N$ is a function that assigns nodes to edges, $\lambda : N \rightarrow L$ is a function that assigns labels to nodes, and $\tau : E \rightarrow T$ is a function that assigns types to edges. A path $p$ is defined as an alternating sequence of nodes and edges $p = \langle n_1, e_1, n_2 \ldots, n_{k-1}, e_{k-1}, n_k \rangle$. Let the function $nodes(p)$ return the sequence $N_p = \langle n_1, \ldots, n_k \rangle$ of the nodes of $p$, the function $edges(p)$ return the sequence $E_p = \langle e_1, \ldots, e_{k-1} \rangle$ of the the edges of $p$, and the function $size(p)$ return the total number of elements (nodes and edges) in the path.

Similar to $G$, a graph pattern is a directed labeled graph $\Pi = (N_\Pi, E_\Pi, L_{,\Pi}\, T_\Pi, \rho_\Pi, \lambda_{,\Pi}\, \tau_\Pi)$ where $\rho_\Pi$, $\lambda_\Pi$, and $\tau_\Pi$ are defined analogously to $\rho$, $\lambda$, and $\tau$. A *subgraph matching query* $q(G, \Pi)$ on a directed labeled graph $G$ takes a graph pattern $\Pi$ as input and returns a set of matching subgraphs. A *path matching query* is a subgraph matching query, for which all possible matching subgraphs are paths. We refer to the nodes and edges of $\Pi$ as node and edge variables respectively. Similar to paths, $size(\Pi)$ returns the total number of node and edge variables in $\Pi$.

The exact results of a path pattern matching query depend on subgraph matching semantics used, i.e., *homomorphism*, *cyphermorphism*, or *isomorphism*. Under homomorphism semantics two or more node labels and two or more edge types in the pattern may be mapped to the same node and the same edge, respectively, in the retrieved path. Under cyphermorphism semantics two or more node labels in the pattern may be mapped to the same node in the retrieved path, but each edge type in the pattern must be mapped to a *single* edge in the retrieved path. Lastly, under isomorphism semantics both node labels and edge types in the pattern must be mapped to *unique* nodes and edges, respectively, in the retrieved path. Result paths under isomorphism semantics are *simple*, while result paths under homomorphism or cyphermorphism semantics may contain *cycles*.

In simpler terms, a path pattern $\Pi$ can be represented by a sequence $\Pi = \langle l_1, t_1, l_2, \ldots, l_{k-1}, t_{k-1}, l_k \rangle$ where $\{l_1, \ldots, l_k\} \in L_\Pi \cup \{\text{`\_'}\}$ are node labels, and $\{t_1, \ldots, t_{k-1}\} \in T_\Pi \cup \{\text{`\_'}\}$ are edge types. The symbol '_' is a wildcard that signifies that a node or an edge of the graph can be matched to a variable regardless of label or type, i.e., the node or edge variable in the pattern is non-labeled or non-typed, respectively. Given a directed labeled graph $G$ and a path pattern $\Pi$, the path matching query $q(G, \Pi)$ returns all paths $p = \langle n_1, e_1, n_2, \ldots, n_{k-1}, e_{k-1}, n_k \rangle$ in $G$ such that $\forall l_i \in \Pi : l_i = \lambda(n_i) \vee l_i = \text{`\_'}$ and $\forall t_i \in \Pi : t_i = \tau(e_i) \vee t_i = \text{`\_'}$. Throughout this paper, we use notation borrowed from the Cypher query language to represent path patterns, e.g.,

```
(n0:L1)-[r0:T1]->(n1:L2)-[r1:T2]->(n2:L1),
```

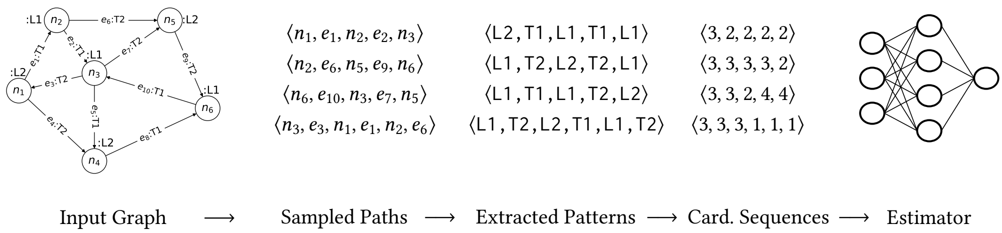


Fig. 1. Components of SPACE

where `n0`, `n1`, and `n2` are node variables, `L1` and `L2` are node labels, `r0` and `r1` are edge variables, and `T1` and `T2` are edge types.

### 3.2 Problem Definition

Given a *directed labeled graph* $G = (N, E, L, T, \rho, \lambda, \tau)$ and a path pattern $\Pi$, the total number of paths that a path pattern matching query $q(G, \Pi)$ returns is called the *result cardinality* or simply the *cardinality* $card(q)$ of $q$. The problem studied in this paper is to estimate the result cardinality of a path pattern matching query $q(G, \Pi)$, under the condition that $L_\Pi \subseteq L$, and $T_\Pi \subseteq T$.

### 3.3 Recurrent Neural Networks

In this paper, we propose an architecture based on a Recurrent Neural Network (RNN) [11, 39] to address the problem presented in Section 3.2. RNNs are designed to handle sequential data by maintaining a hidden state that evolves over time. At each time step $i$, the hidden state $h_i$ is updated based on the current input $x_i$ and the hidden state from the previous time step $h_{i-1}$. This process is described by the following equation:

$$h_t = \phi(W_h h_{i-1} + W_x x_i + b_h),$$

where $W_h$ and $W_x$ are weight matrices, $b_h$ is the bias, and $\phi$ is a non-linear activation function, such as tanh or ReLU. The hidden state $h_i$ captures information about the sequence up to time $i$. The output $y_i$ at time step $i$ is computed as:

$$y_i = \psi(W_y h_i + b_y),$$

where $W_y$ is the output weight matrix, $b_y$ is the bias, and $\psi$ is an activation function (e.g., sigmoid).

## 4 Sequence-based Path-Pattern Cardinality Estimation

We now present the *Sequence-based Path Pattern Cardinality Estimator* (SPACE), our approach for estimating cardinalities of path patterns using cardinality-aware learning. Figure 1 illustrates the overall architecture of our approach. The training process of SPACE is divided into two main steps.

The first step, *training data generation*, focuses on generating a summarized representation of a subset of all path patterns in a graph. SPACE first randomly samples path patterns up to a predefined size $k$. Subsequently, for each path pattern, SPACE computes a cardinality sequence that consists of the cardinality of the pattern and the cardinalities of all its prefix sub-path patterns.

In the second step, *cardinality-aware learning*, SPACE utilizes a recurrent neural network (RNN) architecture to learn from the path patterns and their associated cardinality sequences. The model jointly learns from both the syntactic structure of the paths and the cardinalities of all their prefix sub-path patterns.

### 4.1 Limitations of Existing Approaches

Existing learning-based models [9, 41] that support path pattern cardinality estimation on labeled graphs either generate embeddings that emphasize syntactic and structural relationships among path patterns or directly incorporate these similarities into the learning process without generating embeddings. GNCE [41] is an embedding-based method that generates fixed-length real-valued vectors for path patterns, capturing the structural relationships, including how nodes and edges connect within the graph. These embeddings provide a compressed representation of the graph's topology, allowing deep-learning models to train on the vectorized inputs for cardinality estimation. In contrast, LMKG [9] uses a direct-learning model to bypass the need for embeddings by encoding inputs directly from the graph structure itself. Instead of abstracting structural information into fixed-length embedded vectors, these methods concentrate on directly encoding graph entities along with their structural properties.

While the embedding effectively captures the syntactic structure and relationships among path patterns, GNCE overlooks cardinality information during the embedding phase. During training, cardinality information is only introduced in a post-hoc stage, where the embeddings are fed into the model. This methodology presents a significant limitation: it yields only one scalar cardinality, which fails to account for the dynamic nature of cardinalities that can vary along sub-paths of a path. Such an approach can lead to considerable inaccuracies during estimation, as different path patterns may share syntactic or structural similarities yet have markedly different cardinalities. For instance, consider the path patterns $\Pi_1 = \langle l_a, t_a, l_b, t_b, l_c \rangle$, $\Pi_2 = \langle l_a, t_a, l_b, t_c, l_d \rangle$, and $\Pi_3 = \langle l_a, t_a, l_b, t_e, l_f \rangle$ with cardinalities $card(\Pi_1) = 200$, $card(\Pi_2) = 190$, and $card(\Pi_3) = 10$, respectively. Although all patterns share the same prefix, i.e., $\langle l_a, t_a, l_b \rangle$, thus appearing structurally similar, only the cardinalities of $\Pi_1$ and $\Pi_2$ are similar; the cardinality of $\Pi_3$ is significantly different. Similarly, LMKG, which directly learns from syntactic and structural similarities, encounters the same problem. Specifically, LMKG maps every encoded path pattern to a single cardinality, which overlooks the inherent variability among the cardinalities of the sub-path patterns, thus leading to the same problem that GNCE also encounters. The results of our experiments presented in Section 6.2 (Table 9) further support this claim.

In summary, simply considering the syntactic or structural similarity of paths without considering their cardinality during the embedding or encoding phase fails to capture the variability in the cardinality of syntactically similar patterns. Such an omission can lead to inaccurate cardinality estimates.

### 4.2 Cardinality-aware Sequence-based Learning

To overcome these limitations of existing learning-based approaches, we propose the *Sequence-based Path Pattern Cardinality Estimator* (SPACE), a novel method for cardinality-aware learning that leverages a recurrent neural network architecture to capture the intricate relationships between path patterns and their corresponding cardinality. Our approach incorporates cardinality information directly into the learning process, allowing the model to learn from both the sequence of path pattern labels and types and the corresponding cardinalities of their prefix sub-path patterns. This dual-input representation ensures that cardinality variations along path patterns are captured throughout the learning process.

Our approach processes input sequences by dynamically linking each path pattern to its associated cardinality vector throughout the learning process. Recall that a pattern $\Pi$ is a sequence $\Pi = \langle l_1, t_1, l_2, \ldots, l_{n-1}, t_m, l_n \rangle$ where $\{l_1, \ldots, l_n\} \in L_\Pi \cup \{`\_'\}$ are node labels, and $\{t_1, \ldots, t_m\} \in T_\Pi \cup \{`\_'\}$ are edge types, while '_' acts as a wildcard. For each path pattern, we associate a vector of cardinalities, denoted as $C_\Pi = \langle c_1, c_2, \ldots, c_{m+n} \rangle$. Each cardinality $c_j$ corresponds to the number of results associated with the sub-path pattern that extends from the beginning of the path pattern up to the label at position $j$, if the $j^{\text{th}}$ element is a node label. If the element at position $j$ is an edge type, then the cardinality $c_j$ corresponds to the path pattern up to the label at position $(j+1)$ replacing the label with '_'.

To incorporate cardinality information into the learning process of SPACE, we utilize a recurrent neural network (RNN) to process path patterns alongside their cardinality sequences. More specifically, at each time step $i$, the RNN receives two components: the path pattern node label $l$ or edge type $t$ at position $j$ and its associated cardinality $c_j$. By maintaining a hidden state $h_{i-1}$, which retains information from all preceding time steps, our RNN can update hidden state $h_i$ to capture evolving patterns in the relationship between path patterns and their associated cardinalities. This evolution is formalized through the recurrence relation as:

$$h_i = \phi(W_h h_{i-1} + W_x x_i + b_h),$$

where

- $x_i = [p_j, c_j]$ is the input vector concatenating the element $p_j$, i.e., the label $l$ or type $t$ at position $j$, and its associated cardinality,
- $W_h$ and $W_x$ are weight matrices,
- $b_h$ is a bias vector, and
- $\phi$ is an activation function.

The output of the RNN at each time step is then computed as:

$$\hat{c}_{i+1} = W_y h_i + b_y,$$

where

- $\hat{c}_{i+1}$ is the predicted cardinality associated with the element at position $j+1$ of the path pattern,
- $W_y$ is the output weight matrix, and
- $b_y$ is the output bias.

To ensure that the model learns to align the predicted cardinalities with the actual cardinalities associated with each path pattern effectively, we consider the following cardinality vectors of a set of $k$ path patterns: $C = \{C_1, C_2, \ldots, C_k\}$ representing the true cardinality vectors and $\hat{C} = \{\hat{C}_1, \hat{C}_2, \ldots, \hat{C}_k\}$ representing the estimated cardinality vectors. The model's effectiveness is determined by minimizing the difference between the actual cardinality vectors and their predicted counterparts across all path patterns. This is achieved by reducing the cumulative error across all cardinalities in the sequence, measured by a function $\ell(C_i, \hat{C}_i)$ that quantifies the difference between each actual cardinality vector $C_i$ and its estimated vector $\hat{C}_i$ as:

$$Z(C, \hat{C}) = \min\left(\sum_{i=1}^{k} \ell(C_i, \hat{C}_i)\right),$$

where $Z(C, \hat{C})$ represents the overall error that quantifies the difference between the true cardinality vectors and the estimated cardinality vectors for the entire training set of path patterns.

### 4.3 Cardinality Estimation

The final component of our model is the cardinality estimator, which is trained on path patterns and their corresponding cardinality vectors. Specifically, given a set of path patterns $\langle \Pi_1, \Pi_2, \ldots, \Pi_k \rangle$ and their associated set of cardinality vectors $\langle C_1, C_2, \ldots, C_k \rangle$, our cardinality-aware sequence-based model learns from path patterns and their associated cardinality vectors. Figure 2 illustrates the architecture of our sequence-based model and presents the input format for the example path pattern ⟨L2,T1,L1⟩. Specifically, for a given input path pattern ⟨L2,T1,L1⟩ and its associated cardinality vector ⟨3, 2, 2⟩, 3 represents cardinality of sub-path pattern ⟨L2⟩, 2 represents cardinality of sub-path pattern ⟨L2,T1,'_'⟩, and 2 represents the cardinality of entire path pattern ⟨L2,T1,L1⟩.

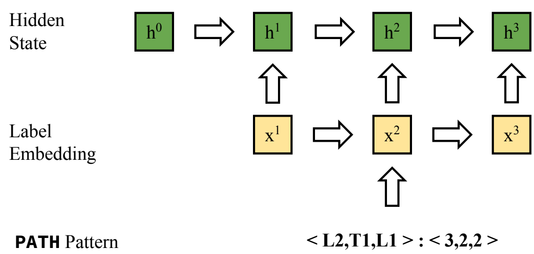


Fig. 2. SPACE model architecture during learning.

During cardinality estimation, our model takes a path pattern as input and generates a vector of estimated cardinalities. Each element in this vector corresponds to a label within the input path pattern. For example, given path pattern ⟨L2,T1,L1⟩, the output vector consists of three cardinalities: the estimated cardinality for ⟨L2⟩, the estimated cardinality for ⟨L2,T1,'_'⟩, and the estimated cardinality for ⟨L2,T1,L1⟩. Figure 3 illustrates estimated cardinalities for the input path pattern ⟨L2,T1,L1⟩. Specifically, for given input path pattern ⟨L2,T1,L1⟩, the output is a vector of estimated cardinalities, where *card*(⟨L2⟩) is estimated cardinality for ⟨L2⟩, *card*(⟨L2,T1,'_'⟩) is estimated cardinality for ⟨L2,T1,'_'⟩, and *card*(⟨L2,T1,L1⟩) is estimated cardinality for ⟨L2,T1,L1⟩. Our model assigns *card*(⟨L2,T1,L1⟩) as estimated cardinality to input path pattern ⟨L2,T1,L1⟩.

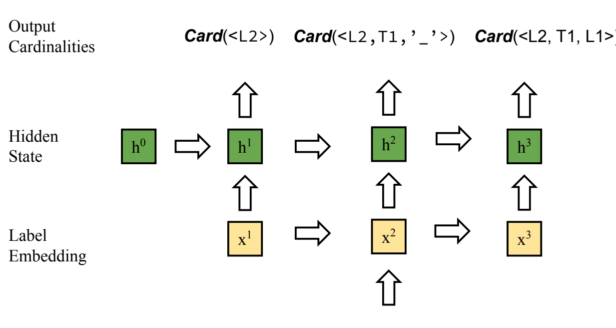


Fig. 3. SPACE model architecture during estimation.

***Encoding path patterns.*** In our model, each label in a path pattern is encoded using a one-hot encoding, which transforms categorical labels into binary vectors. Specifically, for a predefined set of unique labels $\mathcal{T}$, where $\mathcal{T} = L_\Pi \cup T_\Pi \cup \{\text{'\_'}\}$, the one-hot encoding for each label in $\mathcal{T}$ is defined as

$$[0, \ldots, 0, 1, 0, \ldots, 0] \in \mathbb{R}^{|\mathcal{T}|},$$

where $\mathbb{R}^{|\mathcal{T}|}$ represents the real coordinate space of dimension $|\mathcal{T}|$. In this representation, the position corresponding to the label $\mathcal{T}_i$ is set to 1, while all other positions are set to 0.

***Encoding Cardinalities.*** In our model, each cardinality $c_i$ in the vector $C = \langle c_1, c_2, \ldots, c_{m+n} \rangle$ is normalized to the range $[0, 1]$. Typically, the distribution of cardinalities is highly skewed, with certain path patterns being considerably more frequent in graphs than others. Training the model directly on this skewed data can lead to sub-optimal results. Therefore, we apply a log-transformation followed by min-max scaling to reduce the skew in the training data set. This approach is consistent with best practices found in several prior works [10, 20, 42]. Min-max scaling ensures that the log-transformed cardinalities are re-scaled to the range of $[0, 1]$ by assigning values of 0 and 1 to the smallest and largest cardinality, respectively.

***Deep Learning Architecture.*** A number of deep learning architectures, such as Transformers [48] and Recurrent Neural Networks (RNNs) [11, 39], have been proposed for sequence-based learning. We chose RNNs for our approach because they efficiently capture the sequential nature of the path patterns and model the evolving dependencies between node labels and edge types. While transformers handle long-range dependencies, they struggle with preserving the step-by-step progression of cardinalities. Moreover, probabilistic transformers, such as sequence-to-sequence models, predict the next token using prior context, capturing dependencies across the sequence. However, for path cardinality estimation, the context involves the entire set of paths, which must be enumerated for selectivity. Additionally, several RNN architectures, including Long Short-Term Memory (LSTM) [17] and Gated Recurrent Units (GRUs) [8] are available. In our approach, we use a GRU architecture. We conducted extensive experiments to evaluate different configurations, including varying the number of layers $(1, 2)$, and the number of hidden units $(128, 256, 512)$ and decided to use a GRU with one layer and 256 hidden states. Additionally, we evaluated our model with a range of batch sizes $(16, 32, 64, 128, 256, 512)$ and various learning rates $(0.1, 0.01, 0.001, 0.0001, 0.00001)$. The best performance was achieved with a batch size of 128 and a learning rate of 0.0001. The model was trained for 256 epochs.

***Loss Function.*** We use Binary Cross-Entropy (BCE) to measure the accuracy of our model. The binary cross-entropy is defined as

$$L_{BCE} = -\sum_{i=1}^{|p|} \left( y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \right), \tag{1}$$

where $y_i$ and $\hat{y}_i$ are the actual and the predicted normalized cardinalities, respectively, for the label in the path pattern $p$ occurring at position $i$. Our proposed method aims at minimizing the average binary cross-entropy over the entire set of path patterns, i.e.,

$$z = \frac{1}{N} \sum_{n=1}^{N} L_{BCE}, \tag{2}$$

where $N$ is the total number of path patterns in the training set.

## 5 Data Preperation

The goal of the data preparation phase is to collect a set of patterns and compute their representations in order to train SPACE. However, computing the cardinality sequences for all possible patterns up to a certain size $k$ would require enumerating all paths up to size $k$. This approach is clearly impractical. As such, we resort to computing the cardinality sequences for a subset of all patterns in a graph. The data preparation phase of SPACE consists of two steps: (a) the collection of path patterns from a given graph, and (b) the computation of the cardinality sequences.

### 5.1 Collection of Path Patterns

In order to collect patterns from a given graph $G$, we execute a number of random walks. We first define the maximum size $p_{max}$ of the patterns that we want to sample and the maximum number of random walks $r_{max}$ that will be executed. We then select a node of the graph at random and a desired pattern size $k$. We execute a random walk from the selected node by expanding outgoing edges until the required pattern size is met. If it is not possible to expand the current walk further, then the random walk fails, and no pattern is returned. If the random walk is successful, we generate a pattern from the labels and types of the nodes and edges of the path, which we add to our sample $P$. We repeat this process until $r_{max}$ walks have been executed. If a computed pattern has been generated by an earlier random walk, then it is not added to the sample, i.e., our sample contains every sampled pattern only once.

To generate a pattern with non-labeled nodes and/or non-typed edges, we first generate one that has all node labels and edge types. Then, we remove a random set of node labels and edge types in the pattern, replacing them with the '_' wildcard, which denotes that any node or edge can be matched regardless of its label or type, respectively. This idea bears similarity to query masking [31] that has been used to generate training data for learned cardinality estimation in relational databases. In contrast to query masking that involves the removal of query features to force the model to rely more on data features, we do not remove features to obtain queries with missing information, but we create more patterns.

### 5.2 Computation of the Cardinality Sequences

As soon as the sampling process finishes, we proceed to computing the cardinality sequences for all patterns in our sample. To do so, we have to enumerate all paths that match the given pattern. However, in contrast to the way that most graph database systems evaluate patterns, we have to process a given pattern in the exact sequence it is given, i.e., from the start node label to the end node label. The reason is that we do not only compute the overall cardinality of the entire pattern but also the cardinalities of the sub-patterns that are prefixes of the given pattern. The process to compute the cardinality sequence of a given pattern $\Pi$ is outlined in Algorithm 1.

The algorithm first initializes the cardinality sequence with as many zeros as the input pattern $\Pi$ in Lines 1–3. In Lines 4–16, the algorithm iterates over all the nodes of the graph. In Line 5, the label of the current node $n$ is checked and has to be equal to the first label in the pattern $\Pi$. If the node label does not match the first node label in the pattern, the algorithm proceeds to the next node of the graph. If the node is matched, then the algorithm executes a modified *Depth-first Search* from $n$ in Lines 7–16. Specifically, the algorithm maintains a stack $\mathcal{S}$ of pairs $\{n_c, i_c\}$ representing paths starting from $n$ with $n_c \in N$ being the endpoint of each path and $i_c$ is the index of the pattern $\Pi$ with which the last element of the current path has matched. The value of $i$ is always equal to $size(p_c) - 1$, where $p_c$ is the path represented by $\{n_c, i_c\}$. In Line 9, the top element of the stack is popped, and the outgoing edges of the current node $n_c$ are examined. If the type of an outgoing edge $e$ matches the associated element of the pattern $\Pi$ (Line 11), then the associated value in the cardinality sequence is incremented in Line 12. Similarly, if the label of the end node $n'$ of $e$ matches the associated element of the pattern $\Pi$ (Line 13), the subsequent element in the cardinality sequence is incremented (Line 14). Last, if the size of the path from $n_c$ to $n'$ is lower than the size of the pattern $\Pi$, the search continues, i.e., a new element is representing a path in added to the stack $\mathcal{S}$ in Line 16. Finally, the algorithm returns the sequence of cardinalities $C$ of $\Pi$ in Line 17.

Note that we avoid providing a fixed number of patterns that we want to sample as input. The reasoning behind this choice is that there can be no intuition on how to select a meaningful number since there is no way of knowing (or even estimating) how many unique patterns of a specific size

**Algorithm 1** Cardinality Sequence Computation

**Input:** Graph $G = (N, E)$, Pattern $\Pi$
**Output:** Sequence of cardinalities $C$
1: $C \leftarrow \langle\rangle$
2: **for all** $i \in [0, size(\Pi)]$ **do**
3: $\quad C \leftarrow C \circ 0$
4: **for all** $n \in N$ **do**
5: $\quad$ **if** $label(n) = \Pi[0]$ **then**
6: $\quad\quad C[0] \leftarrow C[0] + 1$
7: $\quad\quad$ initialize stack $\mathcal{S}$ with $\{n, 0\}$
8: $\quad\quad$ **while** $\mathcal{S}$ is not empty **do**
9: $\quad\quad\quad \{n_c, i_c\} \leftarrow \mathcal{S}$.pop()
10: $\quad\quad\quad$ **for all** $e = (n, n')$ ∈ outgoing edges of $n_c$ **do**
11: $\quad\quad\quad\quad$ **if** $type(e) = \Pi[i_c{+}1] \vee \Pi[i_c{+}1] =$ '_' **then**
12: $\quad\quad\quad\quad\quad C[i_c{+}1] \leftarrow C[i_c{+}1] + 1$
13: $\quad\quad\quad\quad\quad$ **if** $label(n') = \Pi[i_c{+}2] \vee \Pi[i_c{+}2] =$ '_' **then**
14: $\quad\quad\quad\quad\quad\quad C[i_c{+}2] \leftarrow C[i_c{+}2] + 1$
15: $\quad\quad\quad\quad\quad\quad$ **if** $i_c{+}2 < size(\Pi)$ **then**
16: $\quad\quad\quad\quad\quad\quad\quad \mathcal{S}$.push($\{n', i_c{+}2\}$)
17: **return** $C$

exist in a given graph. Therefore, if a user requests a very large number of patterns for a given graph, the computation of the cardinality sequences would enumerate all paths in the graph, which is too expensive in practice.

Lastly, while the process we described above can be used to generate patterns and compute cardinality sequences under homomorphism semantics, it can easily be extended to generate cardinality sequences under cyphermorphism or isomorphism semantics as well with two modifications. First, the random walk must be performed according to the given semantics. Under cyphermorphism semantics, the random walk must not visit the same edge twice and under isomorphism semantics, the random walk must not visit the same node twice. Second, a check has to be added between Lines 10–11 and Lines 12–13 of Algorithm 1 to ensure that the current path abides by the predefined semantics.

## 6 Experimental Evaluation

We evaluate the efficacy of our approach by comparing it to two traditional methods and two recent state-of-the-art deep learning models. We also present the results of a study where we measure the impact of our cardinality estimates on query optimization.

### 6.1 Experimental Setup

***Data Sets.*** We evaluate our technique on four real-world data sets, i.e., Moreno health[1], SWDF [29], YAGO [46], and a subset of Wikidata [50] and one synthetic data set, i.e., the LDBC Social Network Benchmark 0.3.5 (SNB) [12]. SWDF is a small yet dense real-world data set that models authors, papers, and conferences. The Moreno health data set is a directed graph derived from survey data. Every node represents a student and an edge between two students represents friendship. Edge types are derived from the number of interactions between students. YAGO is a comprehensive cross-domain knowledge graph with information about people, cities, countries, movies, and organizations. Lastly, Wikidata is a collaborative knowledge graph that links structured data across

[1] http://konect.uni-koblenz.de/networks/moreno_health

diverse domains. For SNB, we generated a social network with scale factor 0.1. Table 1 reports the characteristics of these four data sets.

Table 1. Data set characteristics

| Data Set | # Node | # Edges | # Labels | #Types |
|---|---|---|---|---|
| Moreno | 2,539 | 12,969 | N/A | 6 |
| SWDF | 76,711 | 242,256 | N/A | 170 |
| SNB | 432,234 | 2,080,370 | 11 | 15 |
| Yago | 4,295,825 | 12,430,700 | N/A | 37 |
| Wikidata | 4,818,679 | 21,354,359 | N/A | 828 |

***State-of-the-Art Methods***. To evaluate the effectiveness of our approach, we conduct a comparative analysis with both traditional and deep-learning-based state-of-the-art estimators. *GNCE* [41], a recent deep learning-based method, employs knowledge graph embeddings and graph neural networks to achieve accurate cardinality estimation for conjunctive queries in knowledge graphs. *LMKG* [9] is a learned model framework for cardinality estimation in knowledge graphs that learns to estimate the cardinality of commonly used query types, such as star and path queries, and provides algorithms for handling more complex queries like snowflake or tree queries. Yakovets et al. [53] is a traditional approach that functions as a compression mechanism by first enumerating all possible paths up to a specified length. Then, the method constructs a histogram structure to estimate the cardinalities of path queries. In what follows, we will refer to this approach as *Histogram*. CEG [7] is also a traditional compression-based approach that estimates cardinalities using a Markov table, constructed from a set of available queries of the same length, which stores the cardinalities of all sub-paths up to a certain length. However, if the Markov table does not contain any sub-path patterns of a given pattern, CEG cannot provide a cardinality estimate. Mhedhbi and Salihoglu [27] present a hybrid approach that optimizes subgraph queries by combining efficient binary joins with worst-case optimal joins. To estimate the cardinality of a query $Q_k$, it selects a worst-case optimal plan $P$ that computes $Q_k$ through a sequence of extensions $(Q_{j-1}, A_j, l_j)$. In what follows, we refer to this approach as *WCO*.

To compare with these methods, we adopt homomorphism semantics, as do most RDF Stores and SPARQL query processing engines. To evaluate all features of SPACE, though, we also compare our approach to the cardinality estimator of Neo4j, a popular graph database system. Neo4j uses the Property Graph Model and thus supports node labels in addition to edge types, as well as unlabeled node and edge variables in path patterns. For these experiments, we adopt the cyphermorphism semantics of Neo4j.

***Training Data Set and Query Workload***. For each data set, we first enumerate a subset of all possible path patterns up to a length of 13 by conducting 10 million random walks, ensuring that we cover at most 40% of all possible label combinations. Specifically, we initiate 10 million random walks for each graph and terminate the process either if we achieve 40% of the maximum number of possible unique path patterns or if we execute all 10 million walks. After the process terminates, we return the set of collected unique path patterns. Subsequently, we use Algorithm 1 to compute the cardinality sequence for each path pattern in the set. We randomly shuffle the result set, using 90% of the path patterns to train SPACE, GNCE, and LMKG, while the remaining 10% of the paths patterns are kept to test the performance of all approaches. Table 2 shows the number of path patterns containing only edge types in the training and test sets for each data set. For the comparison with Neo4j, we extracted a total of 2,659 path patterns from the SNB data set, which contains both variables with node labels and edge types, as well as non-labeled and non-typed node and edge variables. We then used 2,393 patterns for training and 266 patterns for testing.

Table 2. Number of patterns containing only edge types used for training and testing all data sets.

| **Data Set** | **Type** | **# Patterns** | $size(\Pi)$ | | | | |
|---|---|---|---|---|---|---|---|
| | | | **5** | **7** | **9** | **11** | **13** |
| Moreno | Train | 20146 | 16 | 84 | 468 | 2818 | 16760 |
| | Test | 2240 | 1 | 7 | 51 | 308 | 1873 |
| SWDF | Train | 24173 | 618 | 1974 | 4809 | 7538 | 9234 |
| | Test | 2690 | 67 | 229 | 547 | 799 | 1048 |
| Yago | Train | 23565 | 285 | 1170 | 3118 | 6787 | 12205 |
| | Test | 2623 | 34 | 116 | 334 | 754 | 1385 |
| SNB | Train | 784 | 39 | 84 | 164 | 236 | 261 |
| | Test | 87 | 3 | 8 | 17 | 29 | 30 |
| Wikidata | Train | 192393 | 4905 | 18008 | 38688 | 58409 | 72383 |
| | Test | 21377 | 519 | 2041 | 4325 | 6559 | 7933 |

***Evaluation Metric.*** To study the accuracy of all cardinality estimation approaches, we report the Q-error [28], which quantifies the deviation factor of the estimated cardinality from the true cardinality. The Q-error is defined as

$$max\left(\frac{Card_{est}}{Card_{true}}, \frac{Card_{true}}{Card_{est}}\right),$$

where $Card_{true}$ is the actual number of tuples returned by the query and $Card_{est}$ is the estimated cardinality. The best Q-error is obtained when estimated cardinality is equal to actual cardinality, and the worst Q-error is unbounded.

***Environment.*** The training of all models was conducted on a machine with two NVIDIA RTX 3090 24 GiB GPUs, an AMD Ryzen Threadripper 3970X CPU, and 256 GiB of RAM, running Ubuntu 22.04 LTS. We implement all estimators using Python with PyTorch 2.2.0+cu121. The training overhead and the end-to-end query runtimes were obtained by running Kùzu [13], a graph database system that uses the property graph model and supports a subset of the Cypher query language, on a MacBook Pro with an Apple M1 Pro CPU and 32 GiB of RAM. For efficiency, Algorithm 1 was implemented in C++ and compiled using Apple Clang 16.0.0. Our implementation of SPACE is publicly available[2]

## 6.2 Comparison to the State of the Art

We now report results of our experimental study of SPACE to GNCE, LMKG, Histogram, CEG, and WCO in terms of estimation accuracy, run-time performance, and model size on all five benchmarks.

***Estimation Accuracy.*** Table 3 shows the results of our experimental evaluation of SPACE and the state-of-the-art approaches for cardinality estimation of path pattern matching using homomorphism semantics in terms of accuracy (Q-error). We report geometric mean, mean, median, 90th, and 99th percentile errors. Learning-based approaches and CEG were tested on all benchmarks. In contrast, both Histogram and WCO could only be applied to the Moreno data set. Generating all path patterns up to a specific length for the other data sets using Histogram is computationally too expensive. Similarly, WCO requires a substantial amount of memory to build its catalog. For example, 500 GiB of memory is insufficient to run WCO and create the catalog for the other data sets.

[2]https://github.com/dbis-ukon/SPACE

Table 3. Comparison of Q-error of SPACE against state-of-the-art methods for edge-labeled path patterns.

| Data Set | Estimation Method | $size(\Pi)$ = 7 | | | | | $size(\Pi)$ = 9 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **G. Mean** | **Mean** | **Median** | **90th** | **99th** | **G. Mean** | **Mean** | **Median** | **90th** | **99th** |
| Moreno | SPACE | **1.01** | **1.02** | **1.01** | **1.02** | **1.03** | **1.03** | **1.03** | **1.03** | **1.06** | **1.08** |
| | GNCE | 1.07 | 1.08 | 1.06 | 1.15 | 1.21 | 1.10 | 1.11 | 1.09 | 1.22 | 1.44 |
| | LMKG | 1.07 | 1.08 | 1.05 | 1.16 | 1.18 | 1.05 | 1.06 | 1.04 | 1.13 | 1.17 |
| | Histogram | 2.95 | 4.72 | 2.15 | 9.73 | 18.24 | 3.03 | 4.03 | 2.70 | 6.94 | 18.60 |
| | CEG | — | — | — | — | — | 1.06 | 1.07 | 1.05 | 1.13 | 1.23 |
| | WCO | 1.57 | 1.60 | 1.58 | 1.94 | 2.05 | 2.40 | 2.66 | 2.38 | 4.75 | 6.13 |
| SWDF | SPACE | 1.48 | 11.89 | 1.23 | 2.15 | 13.36 | 1.32 | 1.41 | 1.17 | 2.06 | **4.23** |
| | GNCE | 3.14 | 41.79 | 2.54 | 8.29 | 63.94 | 2.45 | 6.50 | 2.02 | 6.77 | 78.32 |
| | LMKG | 3.82 | 299.35 | 2.75 | 18.32 | 171.59 | 3.87 | 10.02 | 3.16 | 15.16 | 107.76 |
| | CEG | **1.02** | **1.02** | **1.0** | **1.01** | **1.03** | **1.25** | **1.38** | **1.09** | **1.88** | 6.38 |
| SNB | SPACE | **1.39** | **1.44** | **1.35** | **1.95** | **2.37** | 1.77 | 2.40 | 1.22 | 4.0 | 10.41 |
| | GNCE | 34.46 | 320.09 | 52.73 | 767.62 | 1902.85 | 28.0 | 455.58 | 8.44 | 611.59 | 4556.82 |
| | LMKG | 2.28 | 3.0 | 2.16 | 6.0 | 7.50 | 3.14 | 5.94 | 3.04 | 8.28 | 42.58 |
| | CEG | — | — | — | — | — | **1.02** | **1.02** | **1.01** | **1.06** | **1.07** |
| Yago | SPACE | **1.38** | **1.63** | **1.23** | **1.92** | **5.20** | 1.52 | 1.86 | 1.40 | 2.35 | **5.68** |
| | GNCE | 4.29 | 19.57 | 3.27 | 21.41 | 436.42 | 4.38 | 23.33 | 2.95 | 30.98 | 291.49 |
| | LMKG | 8.04 | 86.67 | 6.12 | 86.99 | 638.53 | 7.25 | 33.03 | 5.39 | 53.09 | 528.02 |
| | CEG | — | — | — | — | — | **1.33** | **1.59** | **1.15** | **2.24** | 10.94 |
| Wikidata | SPACE | **1.54** | **4.47** | **1.19** | **2.79** | **51.50** | **1.61** | **5.89** | 1.27 | **3.06** | **33.85** |
| | GNCE | 3.43 | 26.68 | 2.53 | 14.63 | 395.28 | 3.17 | 13.42 | 2.33 | 12.57 | 205.89 |
| | LMKG | 5.95 | 77.32 | 4.16 | 42.13 | 974.74 | 5.61 | 318.07 | 3.89 | 36.29 | 1042.70 |
| | CEG | — | — | — | — | — | 1.72 | 20.43 | **1.16** | 4.84 | 147.33 |

| Data Set | Estimation Method | $size(\Pi)$ = 11 | | | | | $size(\Pi)$ = 13 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **G. Mean** | **Mean** | **Median** | **90th** | **99th** | **G. Mean** | **Mean** | **Median** | **90th** | **99th** |
| Moreno | SPACE | **1.06** | **1.06** | **1.05** | **1.12** | **1.25** | **1.08** | **1.08** | **1.06** | **1.17** | **1.27** |
| | GNCE | 1.06 | 1.06 | 1.05 | 1.12 | 1.20 | 1.08 | 1.09 | 1.07 | 1.19 | 1.32 |
| | LMKG | 1.06 | 1.06 | 1.05 | 1.12 | 1.20 | 1.08 | 1.09 | 1.07 | 1.18 | 1.39 |
| | Histogram | 2.95 | 3.96 | 2.73 | 8.37 | 18.73 | 2.82 | 3.81 | 2.51 | 7.53 | 20.25 |
| | CEG | 1.12 | 1.13 | 1.11 | 1.24 | 1.43 | 1.22 | 1.24 | 1.21 | 1.43 | 1.73 |
| | WCO | 3.15 | 3.59 | 2.99 | 5.85 | 11.43 | 5.45 | 7.39 | 4.83 | 14.80 | 38.45 |
| SWDF | SPACE | **1.29** | **1.36** | **1.16** | **1.87** | **4.59** | **1.35** | **1.54** | **1.19** | **2.03** | **7.07** |
| | GNCE | 2.14 | 3.70 | 1.78 | 5.29 | 19.72 | 2.15 | 7.37 | 1.58 | 5.98 | 86.96 |
| | LMKG | 3.36 | 8.98 | 2.62 | 14.09 | 104.08 | 3.50 | 25.57 | 2.65 | 14.92 | 128.41 |
| | CEG | 1.45 | 1.78 | 1.27 | 2.38 | 7.62 | 1.77 | 2.84 | 1.57 | 3.16 | 14.33 |
| SNB | SPACE | 1.67 | 1.80 | 1.52 | 3.03 | 3.95 | 1.56 | 1.67 | 1.40 | 2.98 | **3.45** |
| | GNCE | 16.41 | 236.71 | 9.86 | 408.47 | 3330.48 | 20.25 | 2538.62 | 11.20 | 593.10 | 50217.27 |
| | LMKG | 3.54 | 6.0 | 3.11 | 9.95 | 35.49 | 3.51 | 4.98 | 3.01 | 8.59 | 56.38 |
| | CEG | **1.06** | **1.06** | **1.02** | **1.16** | **1.58** | **1.23** | **2.70** | **1.04** | **1.47** | 38.37 |
| Yago | SPACE | **1.26** | **1.40** | **1.11** | **2.01** | **2.99** | **1.26** | **1.49** | **1.07** | **2.32** | **4.18** |
| | GNCE | 4.48 | 11.90 | 2.99 | 24.82 | 224.10 | 4.54 | 13.99 | 3.23 | 18.12 | 172.94 |
| | LMKG | 6.90 | 25.03 | 5.16 | 40.89 | 420.73 | 7.07 | 21.42 | 5.03 | 41.81 | 334.32 |
| | CEG | 1.54 | 2.39 | 1.26 | 2.88 | 18.97 | 1.70 | 2.97 | 1.39 | 3.46 | 24.05 |
| Wikidata | SPACE | **1.79** | **10.22** | **1.38** | **3.94** | **42.92** | **2.11** | **14.3** | **1.61** | **5.70** | **64.58** |
| | GNCE | 3.34 | 35.82 | 2.38 | 14.45 | 306.08 | 3.65 | 54.32 | 2.66 | 16.18 | 313.16 |
| | LMKG | 5.74 | 137.66 | 4.01 | 39.73 | 959.22 | 5.77 | 27095.08 | 4.05 | 39.51 | 807.65 |
| | CEG | 2.99 | 173.51 | 1.58 | 22.81 | 1190.45 | 6.57 | 1508.31 | 3.27 | 105.19 | 6496.02 |

We observe that SPACE consistently outperforms LMKG, GNCE, Histogram, and WCO across all data sets and path length variations in terms of cardinality estimation accuracy. The superior performance of SPACE is due to its cardinality-aware embedding and sequence-based learning approach, which effectively captures the complex dependencies between path patterns and their associated cardinalities. However, in certain scenarios, particularly for specific data sets and shorter path lengths, CEG demonstrates better performance than SPACE. Nevertheless, SPACE consistently outperforms CEG for longer path patterns across all data sets and path lengths. Furthermore, CEG fails to return any estimated cardinality for the Moreno, SNB, and Wikidata data sets and path length 7. Recall that CEG cannot estimate the cardinality of a path pattern if none of its sub-paths are present in the Markov table. These cases are indicated by a '—' in Table 3. For the other configurations, CEG fails to estimate cardinalities for a total of 0.7%, 8%, 23%, 26%, and 46% of the queries in the Moreno, Yago, SWDF, SNB, and Wikidata data sets, respectively. In general, CEG tends to fail on data sets that have a large number of unique edge types and node labels. To nevertheless ensure a fair comparison, these queries were excluded from the Q-error shown in Table 3.

Focusing specifically on the SNB data set, SPACE demonstrates exceptional performance among learned models, especially given the comparatively small training data set. The ability of SPACE to achieve lower Q-errors than both LMKG and GNCE underscores the efficacy of its cardinality-aware sequence-based learning mechanism. LMKG and GNCE struggle to capture the underlying structure of the data set when there is limited training data available. With fewer training samples, the test data set can deviate significantly from the training set, making it harder for LMKG and GNCE to generalize. That is not the case for SPACE. The benefit of SPACE is particularly evident in the geometric mean and median errors, where SPACE significantly outperforms LMKG and GNCE. While CEG fails to estimate cardinalities for 23% of the queries in the test set, it otherwise outperforms SPACE across all varying path pattern lengths. The reason CEG performs better on this data set is that SNB has fewer edge types and node labels, as well as a smaller number of unique path patterns. As a result, the Markov table effectively captures the underlying structure of the SNB data set.

Finally, our experiments show that Histogram and WCO perform poorly even for the Moreno data set, on which other methods performed well. Histogram's reliance on predefined histograms, which summarize the data set into fixed bins, can lead to significant information loss. The fixed binning can oversimplify the data representation, obscuring critical differences in cardinalities that arise in more nuanced path patterns. Even though the construction of the histograms requires an expensive enumeration of all path patterns, Histogram demonstrates the lowest accuracy w.r.t. Q-error, rendering the technique impractical for query optimization. WCO relies on a worst-case optimal plan to estimate the cardinality of a given query. Specifically, to estimate the cardinality of $Q_k$, WCO selects a worst-case optimal plan that computes $Q_k$ through a sequence of extensions $(Q_{j-1}, A_j, l_j)$. Then, the estimated cardinality of $Q_k$ is the product of the $\mu(A_j)$ values corresponding to the $(Q_{j-1}, A_j, l_j)$ entries in the catalog. The catalog contains entries for subgraphs with up to $h$ vertices. If the query $Q_k$ contains more than $h$ node labels or edge types, some of the required entries for estimating its cardinality may be missing. As WCO typically uses $h = 3$, this absence of necessary entries explains its degrading performance as the length of path patterns increases from 7 to 13.

***Impact of Model Size.*** To compare memory requirements for learning-based methods, we report the average model size and the number of parameters across all datasets, based on the configurations used in previous experiments. Furthermore, we report the histogram size for Histogram, the Markov table size for CEG, and the catalog size for WCO. Table 4 summarizes the memory requirements and parameter counts for each estimator. SPACE has the largest model size and parameter count among the learned estimators due to its use of an RNN and multiple fully connected layers. However, its memory usage remains under 1.5 MiB, allowing it to capture complex patterns for accurate cardinality estimation. WCO has the highest memory requirement, as it includes all entries extending an at-most 4-vertex subgraph in its catalogue. The histogram method has the lowest memory requirements, using fixed buckets to summarize data distributions, storing start and end indices, and cardinality sums for efficient path aggregation and selectivity estimation. Lastly, CEG stores all sub-paths up to length 3 along with their cardinalities, with the size of the Markov table depending on the number of unique path patterns up to size 3.

Table 4. Model size and number of model parameters

| Data Set | Method | Model size (MiB) | # parameters |
|---|---|---|---|
| Moreno | SPACE | 0.94 | $2.47 \cdot 10^5$ |
| | GNCE | 0.47 | $1.41 \cdot 10^5$ |
| | LMKG | 0.72 | $1.87 \cdot 10^5$ |
| | Histogram | 0.27 | – |
| | CEG | 0.069 | – |
| | WCO | 74.2 | – |
| SWDF | SPACE | 1.40 | $3.67 \cdot 10^5$ |
| | GNCE | 0.47 | $1.39 \cdot 10^5$ |
| | LMKG | 0.72 | $1.87 \cdot 10^5$ |
| | CEG | 0.62 | – |
| SNB | SPACE | 0.97 | $2.53 \cdot 10^5$ |
| | GNCE | 0.47 | $1.24 \cdot 10^5$ |
| | LMKG | 0.72 | $1.88 \cdot 10^5$ |
| | CEG | 0.032 | – |
| Yago | SPACE | 1.03 | $2.70 \cdot 10^5$ |
| | GNCE | 0.47 | $1.36 \cdot 10^5$ |
| | LMKG | 0.73 | $1.93 \cdot 10^5$ |
| | CEG | 0.34 | – |
| Wikidata | SPACE | 2.97 | $7.77 \cdot 10^5$ |
| | GNCE | 0.73 | $1.92 \cdot 10^5$ |
| | LMKG | 0.74 | $1.95 \cdot 10^5$ |
| | CEG | 8.5 | – |

For a fair comparison, we evaluate the accuracy of all learned methods with respect to different model sizes (0.5, 1.0, and 1.5 MiB). For each method, we adjust hyperparameters to meet the specific memory requirements. Figure 4 presents the $Q$-error for all methods across these model sizes, with results consistent with those reported in Table 3. Notably, SPACE achieves the lowest $Q$-error, followed by GNCE. Increasing the model size does not consistently enhance accuracy across all methods. In particular, GNCE exhibits unexpected behavior, with a noticeable decline in accuracy at 1.5 MiB for the SWDF dataset and at 1.0 MiB for the Yago dataset.

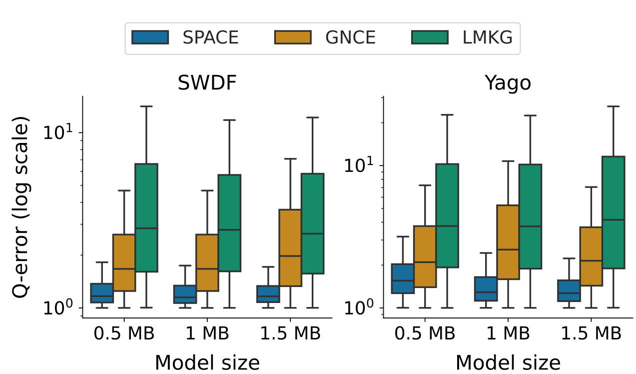


Fig. 4. Q-error of all learned-based methods varying space budget for SWDF and Yago data sets.

**User Queries.** We evaluate the performance of the learned model on a real query workload. We use 224 user queries from the Wikidata dataset, sourced from GNCE [41], which consist of edge type paths of length 2 to 5. Table 5 shows the performance of the learned models for these queries.

SPACE outperforms GNCE and LMKG on user queries, which exhibit characteristics substantially different from the path patterns in the training set. This is due to SPACE's use of cardinality-based embeddings and sequential learning, which better models the evaluation of edge types and node labels, making it more adaptable and generalizable.

Table 5. Q-error of the learned models for user queries.

| Dataset | Method | G. Mean | Mean | Median | 90th | 99th |
|---|---|---|---|---|---|---|
| Wikidata | SPACE | **2.78** | **167.25** | **1.15** | **59.73** | **6376.36** |
| | GNCE | 24.85 | 2058.72 | 10.87 | 1318.58 | 49372.19 |
| | LMKG | 19.35 | 1122.50 | 7.10 | 1215.23 | 21723.16 |

***Comparison with Neo4j.*** Next, we compare SPACE and Neo4j, using cyphermorphism semantics to generate training and test datasets. Table 6 shows the Q-error comparison between SPACE and Neo4j for various path pattern lengths on the SNB dataset. In most cases, SPACE outperforms Neo4j, with lower Q-errors, especially in the mean and 99th percentile. Although Neo4j has lower median Q-error in some cases, the difference is negligible. We do not report model size and runtimes here. First, Neo4j maintains minimal information and, therefore, space overhead is expected to be smaller than the model size of SPACE. Second, we were unable to obtain isolated runtimes for Neo4j's cardinality estimator.

Table 6. Comparison of Q-error of SPACE against state-of-the-art methods for all types of path patterns on SNB.

| $size(\Pi)$ | Method | G. Mean | Mean | Median | 90th | 99th |
|---|---|---|---|---|---|---|
| 7 | SPACE | **1.97** | **2.98** | 1.52 | 7.09 | **16.84** |
| | Neo4j | 2.04 | 13.13 | **1.06** | **6.38** | 256.34 |
| 9 | SPACE | **2.43** | **3.10** | **2.09** | **6.03** | **12.03** |
| | Neo4j | 4.02 | 310.84 | 2.10 | 61.68 | 7288.60 |
| 11 | SPACE | **3.50** | **5.94** | 3.16 | **13.03** | **43.75** |
| | Neo4j | 5.03 | 83.42 | **2.79** | 86.15 | 1597.47 |
| 13 | SPACE | **4.40** | **14.42** | 3.59 | **14.78** | **311.08** |
| | Neo4j | 5.39 | 426.41 | **2.97** | 23.83 | 6822.86 |

**Impact of Training Data Set Size.** To study the impact of training data set size on the accuracy of the learned models, we varied the training data set sizes of SWDF and Yago from $5,000$ to $20,000$ path patterns. Figure 5 plots the results of our experiment. We conclude that SPACE is able to deliver high-quality cardinality estimates even when trained on very small data sets. Notably, even with as few as $5,000$ path patterns, SPACE can estimate accurate cardinalities. The ability of SPACE is particularly important for large graphs, where it is both challenging and resource-intensive to collect training data. In contrast, LMKG and GNCE show degrading performance as the training data set size decreases.

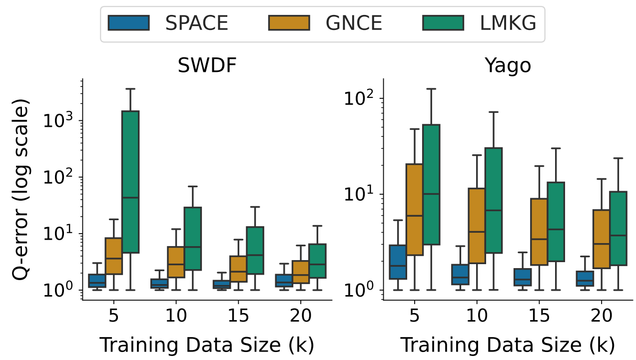


Fig. 5. Q-errors for learned models for different sizes of training data set for SWDF and Yago.

***Runtime.*** We report the average runtime of SPACE and state-of-the-art approaches. While we use a GPU for the training phase of machine-learning-based methods, only the CPU is used for inference to ensure a fair comparison between learned and traditional approaches. For learned models, we include both training and inference times, while for traditional methods, we report the time to create the histogram, Markov table, and catalog, as well as query time. The results are shown in Table 7. LMKG has the shortest training time (0.76 min) due to its efficient use of knowledge graph embeddings. SPACE takes 6.94 minutes for training, as its recurrent neural network captures complex patterns. GNCE has the highest pre-processing time due to its complex graph neural network architecture. Among traditional methods, Histogram requires the most time (40.25 min) due to bucket variance optimization. For query time, LMKG is the fastest (0.009 ms), followed by SPACE (0.37 ms), and GNCE (1.03 ms). WCO has the highest running time (53.21 ms) but the lowest pre-processing time.

Table 7. Average runtime assessment for the entire pipeline of SPACE and state-of-the-art approaches.

| Method | Training time (min) | Inference time (ms) |
|---|---|---|
| SPACE | 6.94 | 0.37 |
| GNCE | 230.13 | 1.03 |
| LMKG | 0.76 | 0.009 |
| **Method** | **Pre-processing time (min)** | **Query time (ms)** |
| Histogram | 40.25 | 7.4 |
| CEG | 0.71 | 0.65 |
| WCO | 0.2 | 53.21 |

**Overhead of Training Data Generation.** To assess training data generation overhead, we perform multiple experiments. First, we evaluate the runtime performance of our path enumeration algorithm (Algorithm 1) across all data sets. This process consists of two phases: the sampling phase that collects path patterns through random walks and the enumeration phase that generates cardinality vectors for these patterns[3]. As shown in Table 8, the dominant pre-processing cost is computing the ground truth cardinality vectors.

[3]The enumeration phase was executed using a compute cluster with a slower CPU, leveraging 360 parallel processes.

Table 8. Sampling and enumeration time for all data sets.

| Dataset | Sampling (m) | Enumeration (h) |
|---|---|---|
| Moreno | 1.15 | 0.007 |
| SWDF | 0.61 | 0.68 |
| SNB | 4.19 | 0.81 |
| Yago | 10.1 | 4.9 |
| Wikidata | 0.84 | 50.96 |

As SPACE takes sub-path pattern cardinalities into account, enumeration can exclude patterns that are sub-path patterns of other patterns without losing accuracy. To demonstrate this fact, we removed such patterns from the training set of the Yago data set, bringing the number of path patterns from 23,565 down to 16,169 and reducing enumeration time from 4.32 to 3.21 hours. Using the the same test set as in Table 2, we report the accuracy of all models for both the full set of patterns and the set excluding sub-path patterns in Table 9. The accuracy of SPACE is stable across both training sets, while the accuracy of GNCE and LMKG declines when sub-path patterns were excluded, thus indicating that SPACE better captures cardinalities of sub-path patterns.

Table 9. Q-error of SPACE and state-of-the-art methods, using training sets with and without sub-paths on Yago.

| Dataset | Method | G. Mean | Mean | Median | 90th | 99th |
|---|---|---|---|---|---|---|
| with sub-paths | SPACE | **1.44** | **1.72** | **1.28** | **2.27** | **7.36** |
| | GNCE | 3.91 | 44.24 | 2.72 | 21.36 | 450.56 |
| | LMKG | 5.42 | 106.57 | 3.99 | 31.55 | 652.76 |
| without sub-paths | SPACE | **1.48** | **1.83** | **1.28** | **2.43** | **8.98** |
| | GNCE | 6.0 | 40.97 | 4.24 | 42.78 | 677.66 |
| | LMKG | 8.39 | 249.37 | 5.46 | 88.43 | 3386.60 |

Last, when used in a real system without having being trained beforehand, all learned approaches suffer from the cold-start problem. While LMKG and GNCE can be trained by executing queries as they arrive using the best plan as determined by the system's, SPACE needs to evaluate every query from the start node variable to the end. This might create an overhead as the start-to-end evaluation plan may not be as good as the one used by the optimizer. To measure this potential overhead, we use the graph database system Kùzu to compute the exact cardinalities required by LMKG, GNCE, and SPACE. We consider the same 2,393 we used to train SPACE for the comparison with Neo4j. We report the results of our experiment on the SNB data set in Table 10. Regarding the best plan and the start-to-end plan of Kùzu, we observe some minor overhead with the start-to-end plan being only slightly worse on average. We believe this overhead is acceptable as SPACE offers better cardinality estimates (see Table 3). Also, as we showed in the previous experiment, SPACE can be trained using fewer patterns without a major effect in its accuracy, thus reducing the overhead.

Table 10. Total and average (in parenthesis) computation time (seconds) of cardinality vectors for SNB.

| Dataset | Kuzu | |
|---|---|---|
| | best plan | start-end |
| SNB | 7,362 (3.08) | 8,753 (3.65) |

## 6.3 In-depth Evaluation

**Impact of Loss Function.** In our method, we use binary cross-entropy (BNCE) as the primary loss function. To assess the impact of different loss functions on SPACE's accuracy, we also tested the Q-Loss [28]. The results are shown in Table 11. We observe that SPACE with BNCE yields more accurate results than with Q-Loss. BNCE handles outliers effectively and provides stable gradients as it penalizes discrepancies between predicted and actual values consistently. In contrast, by emphasizing the maximum ratio between predicted and actual cardinalities, Q-Loss disproportionately penalizes large errors, leading to more instability.

Table 11. Q-error of SPACE with BNCE and Q-Loss.

| Dataset | Method | G. Mean | Mean | Median | 90th | 99th |
|---|---|---|---|---|---|---|
| Yago | BNCE | **1.42** | **1.78** | **1.21** | **2.23** | **7.12** |
| | Q-Loss | 1.45 | 1.80 | 1.24 | 2.38 | 8.02 |
| SWDF | BNCE | **1.25** | **1.45** | **1.14** | **1.16** | **4.72** |
| | Q-Loss | 1.28 | 1.47 | 1.15 | 1.82 | 4.83 |

**Impact of Different Architectures.** We also examine the impact of different architectures on SPACE's accuracy. We implemented a cardinality-based embedding using a transformer, which processes a sequence of path elements through input projection, positional encoding, and multi-head self-attention, with the output passed through fully connected layers to predict cardinality. The results in Table 12 show that the GRU-based architecture outperforms the transformer for path pattern cardinality estimation. While the transformer captures long-range dependencies well, it fails to preserve the step-by-step progression of cardinalities, which is crucial for this task. The GRU handles the sequential dependencies between node labels and edge types better.

Table 12. Q-error of SPACE using RNNs and transformer architectures.

| Dataset | Method | G. Mean | Mean | Median | 90th | 99th |
|---|---|---|---|---|---|---|
| Yago | GRU | **1.42** | **1.78** | **1.25** | **2.23** | **7.12** |
| | Transformer | 19.50 | 708.43 | 13.62 | 399.48 | 9219.24 |
| SWDF | GRU | **1.25** | **1.45** | **1.14** | **1.165** | **4.72** |
| | Transformer | 20.76 | 813.62 | 13.54 | 514.55 | 9415.11 |

**Query Semantics.** We focus on the SNB data set and use the same training and test set as in Table 6. We generate cardinality sequences using homomorphism, cyphermorphism, and isomorphism semantics. The Q-error of SPACE for the different semantics is reported in Table 13. We observe that, regardless of semantics, the Q-error of SPACE is similar. The small difference we observe in the Q-error supports our claim that SPACE can provide good cardinality estimates regardless of the query semantics for path pattern matching used by the system SPACE is implemented on.

Table 13. Q-error of SPACE for all path patterns on SNB using different semantics.

| Semantics | G. Mean | Mean | Median | 90th | 99th |
|---|---|---|---|---|---|
| Homomorphism | 3.28 | 18.06 | 2.39 | 14.14 | 163.02 |
| Cyphermorphism | 3.75 | 10.50 | 2.58 | 20.79 | 117.57 |
| Isomorphism | 3.95 | 17.75 | 3.06 | 18.45 | 161.58 |

### 6.4 Effect on Query Runtime

In the last part of our experimental evaluation, we examine whether the cardinality estimates computed by SPACE can have a positive effect on query runtimes. In cardinality estimation research for relational database systems, experiments measuring end-to-end query runtimes have justifiably become part of the state of the art. Accordingly, benchmarks and a software infrastructure based on PostgreSQL[4] have evolved and became widely adopted in the literature. In the area of graph database systems and, in particular, property graph database systems, a comparable state-of-the-art approach that is commonly used to measure end-to-end runtimes has yet to emerge.

Nevertheless, acknowledging the importance of studying the impact of cardinality estimation on query runtimes, we employ the graph database system Kùzu [13]. Kùzu evaluates graph patterns using a sequence of joins on the node variables.

To do so, we design and develop a simple query optimizer that determines the node variable order for a given path pattern by examining the cardinalities of all its sub-path patterns. For instance, consider the following path pattern.

```
(n:L1)-[r:T]->(n1:L2)-[r1:T]->(n2:L1)-[r2:T]->(n3:L2)
```

We examine all possible sub-path patterns of the given pattern in a similar fashion to Dijkstra's algorithm as shown in Table 14, and estimate their cardinalities. The join order of the execution plan is then defined by the sequence of sub-path patterns for which the sum of the cardinality estimates is minimal.

Table 14. Determination of node variable order.

| **Order** | **Pattern** |
|---|---|
| ⟨n,n1⟩ | (n:L1)-[r:T]->(n1:L2) |
| ⟨n1,n2⟩ | (n1:L2)-[r1:T]->(n2:L1) |
| ⟨n2,n3⟩ | (n2:L1)-[r2:T]->(n3:L2) |
| ⟨n,n1,n2⟩ | (n:L1)-[r:T]->(n1:L2)-[r1:T]->(n2:L1) |
| ⟨n1,n2,n⟩ | (n:L1)-[r:T]->(n1:L2)-[r1:T]->(n2:L1) |
| ... | |

We conducted our experiment using 266 path pattern matching queries on the SNB data set. SPACE was trained on the same set as the one used for the experiments presented in Table 13 and the patterns were selected out of the test set from the same experiment. Following previous work [16], we also include TrueCard, i.e., exact cardinalities, in our experiments. In Figure 6, we report the average runtime of all queries grouped by size. Except for the group of queries of size 9, we observe that SPACE and TrueCard outperform the Kùzu's query optimizer. Also, considering all query patterns, the utilization of SPACE and TrueCard result in faster execution plans. This finding indicates that much of the potential gain that can be achieved by precise cardinality estimation in this simple optimizer is indeed materialized.

Nevertheless, two aspects of the results shown in Figure 6 merit further discussion. First, some experiments indicate that SPACE outperforms TrueCard, which seems to contradict prior research findings in the area of cardinality estimation for relational database systems [25]. We argue that these results can be explained by the relative maturity of query optimizers and execution engines of relational and graph database systems. Second, it should be noted that the simple optimizer that we used in these experiments uses a cost model that only considers cardinalities and therefore operates purely at the logical level of query optimization.

[4] https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark (last accessed: 17/10/2024)

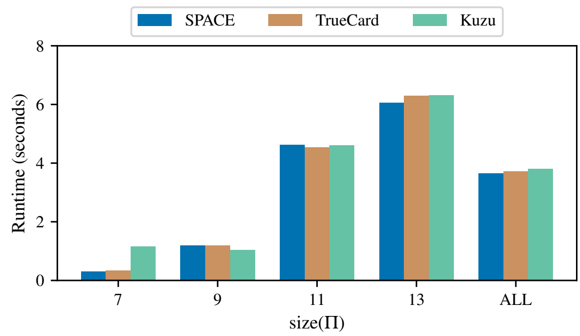


Fig. 6. Query processing runtimes for SNB using Kùzu.

While designing and developing a complete cost-based query optimizer for a property graph database system such as Kùzu is outside the scope of this paper and the objective of our future work, for the time being, we are satisfied by the fact that TrueCard and SPACE yield similar runtimes that provide an improvement over Kùzu's own technique.

## 7 Integrating SPACE into Existing Graph Database Systems

Throughout this paper, we have kept the description of our approach and the setting in which it can be applied very general. We made the deliberate choice to avoid designing our approach for a specific graph database system or graph data model. Therefore, certain adaptations or extensions would be required if one implemented our approach to integrate it into a real-world graph database system. Also, SPACE would have to be combined with some other approach that supports cardinality estimation for star-shaped patterns, e.g., Characteristic Sets [32] to support query optimization for more complex patterns, e.g., tree-shaped patterns and patterns with cycles. In this section, we outline some of these extensions and adaptations that would need to be made to our approach to deploy it to a production environment.

### 7.1 Property Graph Database Systems

SPACE supports some features of the Property Graph Model [1], i.e., node labels, edge types, non-labeled node variables, and non-typed edge variables in a pattern. In what follows, we outline the necessary extensions to use SPACE on property graphs.

First, for the training and the cardinality estimation phase of SPACE, we have considered paths where all edges are forward ones, e.g, `(:L1)-[:R]->(:L2)`. However, a path expression on property graphs may also contain edges in the opposite direction, e.g., `(:L1)-[:R]->(:L2)<-[:R2]-(:L3)`. To support such patterns, during the pattern sampling phase, one needs to treat the graph as undirected, i.e., expand both outgoing and incoming edges during the random walk (see Line 10 of Algorithm 1) and couple the edge direction with the edge type. In other words, for every edge type $t \in T$, one should consider two variants $\overrightarrow{t}$ and $\overleftarrow{t}$.

Second, a node in a property graph may have more than one label. To handle multiple labels per node, one has to mine patterns while considering all combinations of labels per node in a pattern. For example, assume that the random walk executed during the sampling phase of SPACE generates the pattern `(:L1:L2)-[:T]->(:L3)`. A cardinality sequence is computed for the pattern with all labels, treating the pair of labels `:L1:L2` as a single entity. However, it would be advisable to also compute cardinality sequences considering subsets of the labels, i.e, compute the cardinality sequences for patterns `(:L1)-[:T]->(:L3)` and `(:L2)-[:T]->(:L3)` as well.

Third, query languages such as Cypher [14] allow expressions of paths with variable length size. For instance, the expression `(:L1)-[:T*1..3]->(:L2)` matches paths that start with a node that has the label `:L1`, end with a node that has the label `:L2`, and have at least one and up to three edges of type `:T`. SPACE can estimate the cardinality of such a pattern by breaking it down into fixed-length patterns and summing all estimates. For instance, for the aforementioned pattern, SPACE would estimate the cardinalities of `(:L1)-[:T]->(:L2)`, `(:L1)-[:T]->()-[:T]->(:L2)`, and `(:L1)-[:T]->()-[:T]->()-[:T]->(:L2)` and then sum up all three estimates. Naturally, this approach does not work for path patterns such as `(:L1)-[:T*2..]->(:L2)`, where no upper limit to the size of the path pattern is specified. The best strategy to evaluate such queries is usually to match the starting and the end node variables first and then execute multiple depth-first or bidirectional searches. Since the execution strategy is straightforward, there is no need for cardinality estimation and query optimization.

Last, node and relationship properties are another feature of property graphs that SPACE could potentially support. Since each property is a key-value pair, SPACE could treat all combinations of key-value pairs in the same way it treats labels and types. However, since the number of distinct key-value pairs is typically very large, the search space of potential path patterns that SPACE has to sample from also becomes very large. Furthermore, most properties are highly selective. Therefore, if a property is included in a graph pattern, a query optimizer would most likely begin the evaluation of the pattern from the node/relationship for which the property value is defined. The only case where it makes sense to consider a property in the training set is if its selectivity is very low, thus resembling the function of a label or a type. Otherwise, traditional approaches like histograms [18] will be more suitable.

### 7.2 RDF Stores

RDF stores [52] and query languages for knowledge graphs like SPARQL [35] do not support node labels per se. While predicates of triples function in the same fashion as edge types, subjects or objects are usually node identifiers, literals, or variables that are expected to match specific nodes. The equivalent functionality to node labels in knowledge graphs is achieved by using the standardized predicate `rdf:type`. However, the fact that one or more node variables in a pattern may have `rdf:type` edges to other nodes, means that such a pattern is not a path pattern. To circumvent this issue and utilize the full functionality of SPACE on knowledge graphs, during both the training and the cardinality estimation phase triples with `rdf:type` predicates need special considerations. Specifically, the objects of such triples need to be treated automatically as labels of the nodes (or node variables) that are subjects in the same triple.

## 8 Conclusion and Future Work

In this paper, we introduced *Sequence-based Path Pattern Cardinality Estimation* (SPACE), a learning-based approach for cardinality estimation of path patterns in graph databases. Our method encodes path patterns as sequences of node labels and edge types, incorporating cardinalities during the learning process for each sub-path pattern. This approach effectively captures not only the structural and syntactical relationships between path patterns but also the evolution of cardinality over the sequences of the labels and types in the pattern, resulting in more accurate cardinality estimations. To train our model, we propose a sampling method to extract path patterns from a given graph, along with an algorithm to compute the ground-truth cardinality sequences for the extracted patterns. Through extensive experimental evaluation on real-world graph data sets, we demonstrate that SPACE significantly improves the accuracy of cardinality estimation compared to state-of-the-art techniques while being flexible enough to support diverse graph data models and query semantics. In the future, we plan to investigate the applicability of SPACE in the presence of updates over the graph. More specifically, we plan to investigate the periodical retraining of our model and incremental learning techniques to maintain an already trained model.

## Acknowledgments

This work is supported by Grant No. GR 4497/5 and Grant No. CH 2464/1 of the Deutsche Forschungsgemeinschaft (DFG).

## References

[1] Renzo Angles. 2018. The Property Graph Database Model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018 (CEUR Workshop Proceedings, Vol. 2100).* CEUR-WS.org. https://ceur-ws.org/Vol-2100/paper26.pdf

[2] Mehmet Aytimur and Ali Çakmak. 2018. Estimating the selectivity of LIKE queries using pattern-based histograms. *Turkish Journal of Electrical Engineering & Computer Sciences* 26 (2018), 3319–3334. doi:10.3906/elk-1806-96

[3] Mehmet Aytimur and Ali Çakmak. 2021. Using positional sequence patterns to estimate the selectivity of SQL LIKE queries. *Expert Systems with Applications* 165 (2021), 113762. doi:10.1016/j.eswa.2020.113762

[4] Mehmet Aytimur, Silvan Reiner, Leonard Wörteler, Theodoros Chondrogiannis, and Michael Grossniklaus. 2024. LPLM: A Neural Language Model for Cardinality Estimation of LIKE-Queries. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–25. doi:10.1145/3639309

[5] Andreas Björklund, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. 2021. Approximate counting of k-paths: Deterministic and in polynomial space, In 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019). *ACM Trans. Algorithms* 17, 3, 26:1–26:44. doi:10.1145/3461477

[6] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *Proceedings of the VLDB Endowment* 11, 2 (2017), 149–161. doi:10.14778/3149193.3149196

[7] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. 2022. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1533–1545. doi:10.14778/3529337.3529339

[8] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'14).* 1724–1734. doi:10.3115/v1/d14-1179

[9] Angjela Davitkova, Damjan Gjurovski, and Sebastian Michel. 2022. LMKG: Learned Models for Cardinality Estimation in Knowledge Graphs. *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022* (2022), 2:169–2:182. doi:10.48786/EDBT.2022.07

[10] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057. doi:10.14778/3329772.3329780

[11] Jeffrey Elman. 1998. Generalization, simple recurrent networks, and the emergence of structure. In *Proceedings of the twentieth annual conference of the Cognitive Science Society*. Mahwah, NJ: Lawrence Erlbaum Associates, 6.

[12] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* 619–630. doi:10.1145/2723372.2742786

[13] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU graph database management system. In *13th Conference on Innovative Data Systems Research, CIDR.* https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data.* 1433–1445. doi:10.1145/3183713.3190657

[15] Andrey Gubichev. 2015. *Query Processing and Optimization in Graph Databases.* Ph. D. Dissertation. Technische Universität München.

[16] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proceedings of the VLDB Endowment* 15, 4 (2021), 752–765. doi:10.14778/3503585.3503586

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780. doi:10.1162/neco.1997.9.8.1735

[18] Yannis E Ioannidis and Stavros Christodoulakis. 1993. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems (TODS)* 18, 4 (1993), 709–748. doi:10.1145/169725.169708

[19] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating filtered group-by queries is hard: Deep learning to the rescue. In *Proceedings of the International Workshop on Applied AI for Database Systems and Applications*. https://drive.google.com/file/d/1Yt2w7_8UKFxsgcnWE8BjnFwhw509lyyY/view

[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR*. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[21] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. doi:10.1145/3183713.3196909

[22] P. Krishnan, Jeffrey Scott Vitter, and Bala Iyer. 1996. Estimating Alphanumeric Selectivity in the Presence of Wildcards. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*. 282–293. doi:10.1145/233269.233341

[23] Seetha Lakshmi and Shaoyu Zhou. 1998. Selectivity estimation in extensible databases-a neural network approach. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Morgan Kaufmann, 623–627. http://www.vldb.org/conf/1998/p623.pdf

[24] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. 2009. Approximate Substring Selectivity Estimation. In *Proceedings of the International Conference on Extending Database Technology (EDBT '09)*. 827–838. doi:10.1145/1516360.1516455

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. doi:10.14778/2850583.2850594

[26] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation for Joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 2121–2124. doi:10.1145/2882903.2899413

[27] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704. doi:10.14778/3342263.3342643

[28] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993. doi:10.14778/1687627.1687738

[29] Knud Möller, Tom Heath, Siegfried Handschuh, and John Domingue. 2007. Recipes for semantic web dog food—the ESWC and ISWC metadata projects. In *International Semantic Web Conference*. 802–815. doi:10.1007/978-3-540-76298-0_58

[30] M Muralikrishna and DJ DeWitt. 1988. Equidepth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*. 28–36.

[31] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust query driven cardinality estimation under changing workloads. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1520–1533. doi:10.14778/3583140.3583164

[32] Thomas Neumann and Guido Moerkotte. 2011. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *2011 IEEE 27th International Conference on Data Engineering*. 984–994. doi:10.1109/ICDE.2011.5767868

[33] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659. doi:10.14778/1453856.1453927

[34] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1099–1114. doi:10.1145/3318464.3389702

[35] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 1–45. doi:10.1145/1567274.1567278

[36] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. *SIGMOD Rec.* 25, 2 (June 1996), 294–305. doi:10.1145/235968.233342

[37] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proceedings of the VLDB Endowment* 17, 4 (2023), 740–752. doi:10.14778/3636218.3636229

[38] Petar Ristoski and Heiko Paulheim. 2016. Rdf2vec: Rdf graph embeddings for data mining. In *International semantic web conference*. Springer, 498–514. doi:10.1007/978-3-319-46523-4_30

[39] David E Rumelhart, James L McClelland, PDP Research Group, et al. 1986. *Parallel distributed processing, volume 1: Explorations in the microstructure of cognition: Foundations*. The MIT press.

[40] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80. doi:10.1109/TNN.2008.2005605

[41] Tim Schwabe and Maribel Acosta. 2024. Cardinality Estimation over Knowledge Graphs with Embeddings and Graph Neural Networks. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26. doi:10.1145/3639299

[42] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning. *Proceedings of the VLDB Endowment* 14, 4 (2020), 471–484. doi:10.14778/3436905.3436907

[43] E. Patrick Shironoshita, Michael T. Ryan, and Mansur R. Kabuka. 2007. Cardinality estimation for the optimization of queries on ontologies. *ACM SIGMOD Record* 36, 2 (2007), 13–18. doi:10.1145/1328854.1328856

[44] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, 1043–1052. doi:10.1145/3178876.3186003

[45] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 595–604. doi:10.1145/1367497.1367578

[46] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2008. Yago: A large ontology from wikipedia and wordnet. *Journal of Web Semantics* 6, 3 (2008), 203–217. doi:10.1016/J.WEBSEM.2008.06.001

[47] Hien To, Kuorong Chiang, and Cyrus Shahabi. 2013. Entropy-based histograms for selectivity estimation. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 1939–1948. doi:10.1145/2505515.2505756

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. (2017), 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[49] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P Chakkappen. 2015. Join size estimation subject to filter conditions. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1530–1541. doi:10.14778/2824032.2824051

[50] Xiaozhi Wang, Tianyu Gao, Zhaocheng Zhu, Zhengyan Zhang, Zhiyuan Liu, Juanzi Li, and Jian Tang. 2021. KEPLER: A unified model for knowledge embedding and pre-trained language representation. *Transactions of the Association for Computational Linguistics* 9 (2021), 176–194. doi:10.1162/TACL_A_00360

[51] Leonard Wörteler, Moritz Renftle, Theodoros Chondrogiannis, and Michael Grossniklaus. 2022. Cardinality Estimation using Label Probability Propagation for Subgraph Matching in Property Graph Databases. *Proceedings of the 25th International Conference on Extending Database Technology, EDBT* (2022), 2:285–2:297. doi:10.48786/EDBT.2022.16

[52] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF data storage and query processing schemes: A survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–36. doi:10.1145/3177850

[53] Nikolay Yakovets, Li Wang, George HL Fletcher, Craig Taverner, and Alexandra Poulovassilis. 2018. Histogram domain ordering for path selectivity estimation. In *EDBT/ICDT 2018 Joint Conference 21st International Conference on Extending Database Technology*. OpenProceedings. org, 493–496. doi:10.5441/002/EDBT.2018.55

[54] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proceedings of the VLDB Endowment* 13, 3 (Nov. 2019), 279–292. doi:10.14778/3368289.3368294

[55] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. Prone: Fast and scalable network representation learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. ijcai.org, 4278–4284. doi:10.24963/IJCAI.2019/594

[56] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A learned sketch for subgraph counting. In *Proceedings of the 2021 International Conference on Management of Data*. 2142–2155. doi:10.1145/3448016.3457289


Received October 2024; revised January 2025; accepted February 2025