LPLM: A Neural Language Model for Cardinality Estimation of LIKE-Queries

MEHMET AYTIMUR, University of Konstanz, Germany
SILVAN REINER, University of Konstanz, Germany
LEONARD WÖRTELER, University of Konstanz, Germany
THEODOROS CHONDROGIANNIS, University of Konstanz, Germany
MICHAEL GROSSNIKLAUS, University of Konstanz, Germany

Cardinality estimation is an important step in cost-based database query optimization. The accuracy of the estimates directly affects the ability of an optimizer to identify the most efficient query execution plan correctly. In this paper, we study cardinality estimation of `LIKE`-queries, i.e., queries that use the `LIKE`-operator to match a pattern with wildcards against string-valued attributes. While both traditional and machine-learning-based approaches have been proposed to tackle this problem, we argue that they all suffer from drawbacks. Most importantly, many state-of-the-art approaches are not designed for patterns that contain wildcards in-between characters. Based on past research on neural language models, we introduce the `LIKE`-*Pattern Language Model* (LPLM) that uses a new language and a novel probability distribution function to capture the semantics of *general* `LIKE`-patterns. We also propose a method to generate training data for our model. We demonstrate that our method outperforms state-of-the-art approaches in terms of precision (Q-error), while offering comparable runtime performance and memory requirements.

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, Silvan Reiner, Leonard Wörteler, Theodoros Chondrogiannis, and Michael Grossniklaus. 2024. LPLM: A Neural Language Model for Cardinality Estimation of LIKE-Queries. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 54 (February 2024), 25 pages. https://doi.org/10.1145/3639309

1 INTRODUCTION

In cost-based query optimization, the result size or cardinality of a query or subquery is used to assign a cost to each enumerated query execution plan, which is used in turn to find the plan with the lowest overall cost. Since actual cardinalities are not known at optimization time, a typical query optimizer will estimate them for each operator in the query plan by multiplying the size of the input with the estimated selectivity of the operator [33]. In relational database systems, the problem of cardinality estimation is well-studied [12, 13, 26, 30, 31, 35, 37] and the precision of the

Authors' addresses: Mehmet Aytimur, mehmet.aytimur@uni-kn, University of Konstanz, Universitätsstraße 10, Konstanz, Baden-Württemberg, Germany, 78464; Silvan Reiner, silvan.reiner@uni.kn, University of Konstanz, Universitätsstraße 10, Konstanz, Baden-Württemberg, Germany, 78464; Leonard Wörteler, leonard.woerteler@uni.kn, University of Konstanz, Universitätsstraße 10, Konstanz, Baden-Württemberg, Germany, 78464; Theodoros Chondrogiannis, theodoros.chondrogiannis@uni.kn, University of Konstanz, Universitätsstraße 10, Konstanz, Baden-Württemberg, Germany, 78464; Michael Grossniklaus, michael.grossniklaus@uni.kn, University of Konstanz, Universitätsstraße 10, Konstanz, Baden-Württemberg, Germany, 78464.

ACM 2836-6573/2024/2-ART54
https://doi.org/10.1145/3639309

estimated cardinalities has been shown to have a great impact on the quality of the selected query execution plan in terms of runtime [24].

LIKE-queries filter input tuples by matching a string-valued attribute against a pattern containing two types of wildcards (i.e., % and _). Estimating the result cardinality of such LIKE-queries is a subproblem of cardinality estimation that has received relatively little attention. While both traditional and machine-learning-based approaches exist, they all put restrictions on the structure of the LIKE-patterns that they actually support in order to achieve a practical trade-off between the size of the summary data structure, e.g., a histogram or trained model, and the precision of the cardinality estimation. Either they do not support both types of wildcards, or they limit where wildcards can occur within a LIKE-pattern. For example, several state-of-the-art approaches focus exclusively on prefix, suffix, or substring patterns and do not support LIKE-patterns containing wildcards between characters. Therefore, they cannot estimate the cardinality of LIKE-predicates with a pattern as, for example, `Dav% Ma_er` that matches string values such as `David Maier` and `Dave Mayer`.

In this paper, we study cardinality estimation for queries with predicates that match string-valued attributes against a *general* LIKE-pattern, i.e., a pattern that is not restricted w.r.t. its structure or the permissible types of wildcards. Our approach builds on and extends the ideas of neural language models [3]. We first define a new probability distribution function for our neural language model to predict the next token probability for a given pattern. We also introduce a novel language to capture the syntax and semantics of general LIKE-patterns. Our method first translates a LIKE-pattern into words of our novel language and then utilizes our probability distribution function to estimate the cardinality of the given LIKE-pattern. To train our model, we present a method that generates a set of random LIKE-patterns from a given sequence database. While existing techniques for LIKE-query cardinality estimations may perform better in specific and narrow cases, our experimental evaluation demonstrates that our approach substantially outperforms state-of-the-art approaches in general.

Specifically, we claim the following contributions.

- We introduce the LIKE-*pattern Language Model* (LPLM) that uses a new language and a novel probability distribution function to support general LIKE-patterns (Section 4).
- We propose a method to generate training and test data for our model and show how to estimate the cardinality of LIKE-patterns using LPLM (Section 5).
- Our experimental evaluation demonstrates that LPLM outperforms the state of the art, both in terms of accuracy and types of LIKE-patterns supported (Section 6).

We begin by discussing related work in Section 2, introduce preliminaries in Section 3, and give concluding remarks in Section 7.

2 RELATED WORK

Our work is positioned at the intersection of result cardinality estimation for substring predicates with wildcards and machine-learning-based cardinality estimation.

2.1 Traditional Approaches

Various aspects of cardinality estimation for LIKE-queries in SQL queries have been studied in the literature [1, 2, 5, 14, 18, 21, 22, 25]. The work of Krishnan et al. [18] was the first to study cardinality estimation for substring predicates with wildcards. They proposed an algorithm called KVI that uses pruned suffix trees to estimate the cardinality of substring LIKE-predicates in the presence of wildcards. Since its introduction, though, KVI has been outperformed by many cardinality estimators [5, 14, 25, 34].

Lee et al. [22] proposed the LBS algorithm to estimate the cardinality of an approximate substring query. In the same paper, the authors also present an extension of the LBS algorithm for cardinality estimation of SQL LIKE-predicates. The LBS algorithm builds on the author's previous work [21], which uses minimal base substrings and an N-gram table to estimate the cardinality of a predicate. The LBS is the first algorithm to support LIKE-patterns that contain both wildcards (% and _). Hence, we chose LBS as a first baseline approach to experimentally evaluate our method to the state of the art in terms of support for multiple wildcards.

SPH [1] and its successor P-SPH [2] are two algorithms that estimate the cardinality of LIKE-predicates based on histograms. For example, the more recent P-SPH uses a histogram structure based on frequent positional patterns to estimate the cardinality of LIKE-patterns. Specifically, it mines the complete set of frequent positional patterns from a string-valued column and then constructs a histogram based on the mined frequent positional patterns. During query execution, for a given LIKE-pattern, P-SPH visits all buckets of the histogram and estimates the cardinality of the LIKE-predicate according to the match type (exact match, encapsulated match, and no match) between the pattern and endpoint values of histogram buckets. If the query pattern does not match with any buckets of the histogram, P-SPH returns 10% of the minimum support value used during frequent pattern mining as the cardinality. Both SPH and P-SPH focus on LIKE-patterns that contain a high number of wildcards in-between characters, but only support %-wildcards. Since P-SPH has been shown to outperform LBS for this specific use case, we choose it as a second baseline approach to compare our method to the state of the art in our experimental evaluation.

2.2 Machine-Learning-based Approaches

Recently, deep-learning-based approaches to estimate the result cardinality of predicates have shown promise [9, 10, 16, 17, 27, 36]. Astrid-EMBED [34] and Astrid-NLM [34] are two recent deep-learning-based methods to estimate the selectivity of prefix, suffix, and substring queries, which are, therefore, related to cardinality estimation for (certain types of) LIKE-predicates. Astrid-EMBED estimates result cardinality through word embedding. It consists of an embedding learner and a cardinality estimator. The embedding of a string is learned by using syntactic and cardinality similarities between the string and other strings in the suffix tree. Then, a fully connected neural network learns from the embedded strings and their corresponding cardinalities to estimate the cardinalities prefix, suffix, and substring queries. The second method, Astrid-NLM, adapts a neural language model to estimate result cardinality. Specifically, Astrid-NLM uses a character-based neural language model that processes a pattern one character at a time and computes the probability distribution for the next character given the preceding characters. Then, the probability distributions of the characters are used to estimate the cardinality of prefix, suffix, and substring queries. Since, both methods have been shown to outperform traditional approaches for LIKE-patterns that have prefix, suffix, and substring form, and Astrid-EMBED is superior to Astrid-NLM according to Shetiya et al. [34], we use Astrid-EMBED as the third and final baseline approach to evaluate the performance of our approach w.r.t. prefix, suffix, and substring queries experimentally. Astrid is employed as a substitution for Astrid-EMBED in the following sections of the paper.

While other deep-learning-based approaches exist to estimate the cardinality of string-based queries, they are not closely related to our work. For example, a recent work by Kwon et al. [19] presents a deep-learning-based method called DREAM for cardinality estimation of approximate substring matching. More specifically, the authors aim to estimate the number of strings similar to a query string w.r.t. the edit distance measure. Since the focus of this technique is on string similarity, they do not support cardinality estimation for LIKE-predicate or any patterns that contain wildcards.

2.3 Limitations of Prior Approaches

Traditional approaches are presented with a trade-off between the size of the summary data structure and the precision of the estimate that is often impossible to balance in practice. Getting accurate estimates for LIKE-predicates would require a complete summary data structure, which is impractical as it is typically considerably larger than the original dataset and cannot be entirely stored in memory. In order to reduce memory requirements, the summary data structure, therefore, needs to be pruned by, for example, removing all entries whose frequency is less than a given threshold. However, pruning the summary data structure leads in turn to poor estimates for LIKE-patterns that are not in the summary data structure.

Existing cardinality estimation approaches based on machine learning only offer limited support for general LIKE-patterns. More specifically, these approaches are designed to learn contextual and syntactic similarities between patterns in the form of a sequence of characters, i.e., prefix, suffix, and substring patterns. In contrast, the structure of general LIKE-patterns can be more complex as they may contain wildcards (% and _) in-between the characters of a pattern. Currently, approaches based on neural language models are unable to encode the semantics of these general LIKE-patterns and simply treating % as another character during embedding and estimation will inevitably lead to poor cardinality estimates.

3 PRELIMINARIES AND PROBLEM STATEMENT

Our approach to estimating cardinalities for general LIKE-predicates follows in the tradition of language models and, in particular, neural language models. In this section, we introduce the foundations we build on and end with a formal statement of the problem addressed in this paper.

3.1 Language Models

Let $S = \langle t_1, t_2, \ldots, t_n \rangle$ be a sequence of tokens from a vocabulary $\mathcal{T}$. A token can be either a character or a wildcard, i.e., % or _. We denote the i-th token t_i in S by $S[i]$ and the length n of a sequence S is given by $|S|$. We also denote a subsequence of S from the i-th to and including the j-th token by $S[i:j]$, where $1 \leq i \leq j \leq n$. Furthermore, let $\mathcal{S}$ be the set of all sequences. A *language model* defines a probability distribution over sequences drawn from $\mathcal{S}$ by assigning probabilities such that

$$\forall S \in \mathcal{S} : P(S) \geq 0 \wedge \sum_{S \in \mathcal{S}} P(S) = 1, \tag{1}$$

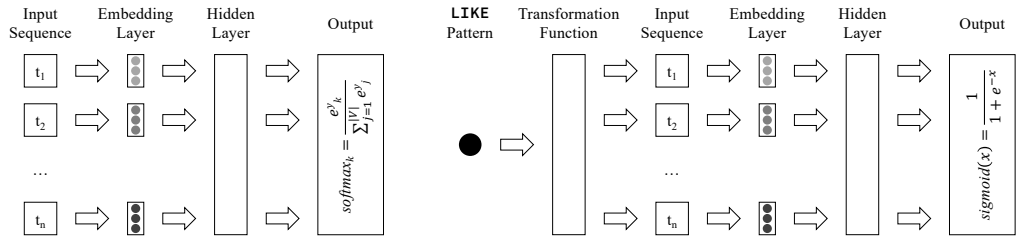
where $P(S)$ is the probability distribution over $\mathcal{S}$. To assign a probability to a given sequence $S = \langle t_1, t_2, \ldots, t_n \rangle$, a language model factorizes the probability as

$$P(S) = P(t_1) \prod_{i=2}^{m} P(t_i | \langle t_1, \ldots, t_{i-1} \rangle).$$

Language models require excessively large amounts of training samples to capture statistical characteristics of the distribution of sequences of tokens in training data. The number of unique tokens in the vocabulary usually rises as the number of training samples increases. A learning algorithm requires at least one example for each relevant combination of tokens from the vocabulary to learn the joint probability distribution of all possible sequences. This requirement causes the problem of the curse of dimensionality [3] for language models during training.

3.2 Neural Language Models

A *neural language model* [3] is a neural-network-based language model that uses distributed representations of input examples to reduce the impact of the curse of dimensionality during



(a) Traditional neural language models (b) LIKE-pattern Language Model

Fig. 1. Comparison of a traditional to our LIKE-pattern Language Model architecture.

learning. A distributed representation of a token t is a vector of non-exclusive features that characterize the token's meaning in the given text. Specifically, given a sequence of tokens $S = \langle t_1, t_2, \ldots, t_n \rangle$, a neural language model assigns each token in S a unique d-dimensional continuous vector as follows

$$\forall t \in S, f_e(t) \rightarrow V_t,$$

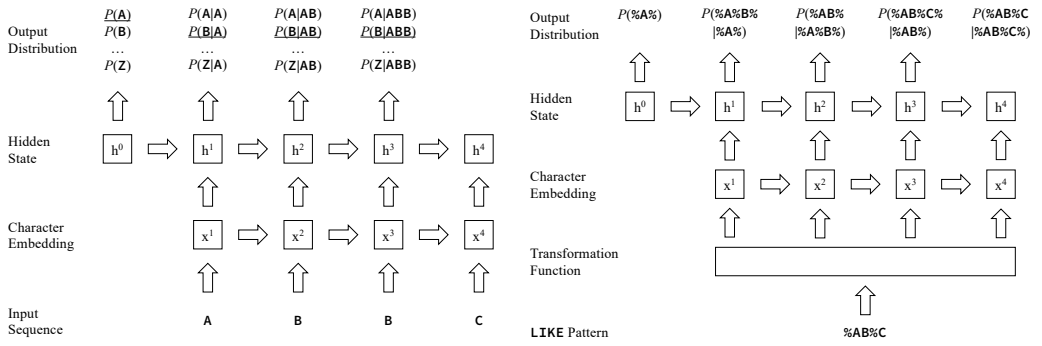
where f_e is an embedding function and V_t is the *embedded feature vector* of t, i.e., a unique d-dimensional continuous vector for token t that contains the learned features of t. Tokens that are semantically close have similar embedded feature vectors. Neural language models train on embedded feature vectors and use the softmax function as an activation function to find the probability of the next token given a sequence of tokens up to that point. The softmax function normalizes the input vector so that all components are in the interval $[0, 1]$ and sum to 1. Last but not least, a distributed representation approach allows the neural language model to generalize well to sequences that are not in the training data by using similarity between embedding vectors. Figure 1a shows the architecture of a traditional neural language model.

3.3 Word Embedding

In natural language processing, word embedding refers to a group of models and feature selection methods that describe the representation of words for text analysis, typically in the form of a real-valued vector that encodes the meaning of the word so that words that are close in the vector space are expected to be similar in meaning. Let $W = w_1, w_2, \ldots, w_n$ be a sequence of words. Given sequence W and word w_i, the meaning of w_i is determined by a set of surrounding words, $w_{i-d}, \ldots, w_{i+d}$, in a sliding window where d represents the length of the window. Bert [7], word2vec [28], doc2vec [20] and FastText [4] are popular word embedding techniques.

3.4 Problem Statement

The SQL `LIKE`-operator selects all tuples in a relation with an attribute value that matches the specified `LIKE`-pattern. For example, the SQL predicate `name LIKE 'Ais'` returns all tuples in a relation with a `name`-value that exactly matches the string `'Ais'`. As defined by ISO/IEC 9075-2:2023 (Section 8.5, General Rule 3.b.ii), the `LIKE`-operator supports the two wildcards `%` and `_`, either independently or in combination. The percent sign matches zero or more characters, whereas the underscore matches exactly one character. For example, the SQL predicate `name LIKE 'Ais%'` matches all tuples with a `name`-value that begins with the prefix `'Ais'`. In contrast, the SQL predicate `name LIKE 'A_s'` matches all tuples that have a `name`-value of length three with the first character being `'A'` and the last being `'s'`.



(a) Traditional neural language models (b) LIKE-pattern language model

Fig. 2. Probability distribution in traditional and advanced neural language model.

Definition 3.1 (Problem Statement). Let $R(A_1, A_2, \ldots, A_n)$ be a relation with attribute names $\{A_1, A_2, \ldots, A_n\}$ and corresponding domains $D_1, D_2, \ldots, D_n$. Without loss of generality, further assume that domain D_i of attribute A_i defines strings s_j over a character set $C = \{c_1, c_2, \ldots, c_m\}$, such that $\forall j : s_j \in C^*$. Let $\mathcal{T} = C \cup \{\%, \_\}$ be a vocabulary of tokens. The problem studied in this paper is to estimate the number of tuples contained in R with an A_i-value that matches a given LIKE-pattern p, i.e., estimate n such that

$$n \approx \left|\sigma_{A_i \texttt{ LIKE } p}(R)\right|,$$

where p is a sequence drawn from the vocabulary $\mathcal{T}$, under the condition that time complexity of the estimation is strictly lower than the time required to evaluate the pattern p over the relation R.

4 LPLM: LIKE-PATTERN LANGUAGE MODEL

Standard language models are designed to determine the probability of a given sequence of characters. Figure 2a shows the probability distributions for each character of the sequence ABBC from the vocabulary $\mathcal{T} = \{\texttt{A}, \ldots, \texttt{Z}\}$ in a standard neural language model. However, in addition to characters, LIKE-patterns may contain wildcards (% and _) that have semantics different from encoding the corresponding character. Since standard neural language models cannot encode these semantics of wildcards, they cannot be applied to cardinality estimation of LIKE-predicates out-of-the-box. As an example, consider the two LIKE-patterns %AB% and %A%B% that are syntactically very similar, but can have arbitrarily different cardinalities. Simply treating % and _ as regular characters in the embedding will inevitably lead to poor cardinality estimates.

Furthermore, directly applying the probability distribution function of standard neural language models to LIKE-patterns leads to problems. On the one hand, if a neural language model treats wildcards as regular characters, the function would simply determine the probability of the next wildcard as the next character in an input sequence. Since wildcards are not in the vocabulary and, therefore, are unknown tokens, a standard neural language model would assign a low probability to a wildcard. Consequently, treating wildcards as a regular characters violates wildcard semantics and leads to inaccurate estimates for LIKE-patterns. On the other hand, if the neural language model was to combine a wildcard with the following regular character of the LIKE-pattern, the function could return a probability distribution with a sum greater than 1, thus violating Equation 1.

Example 4.1. Consider the vocabulary $\mathcal{T} = \{\texttt{A}, \ldots, \texttt{E}\} \cup \{\%, \_\}$ and the LIKE-pattern %AB%C that selects all rows from database given in Table 1 that start with any token followed by AB and end

Table 1. Sample sequence database.

Id	Sequence
1	ABCABE
2	BCACDBE
3	BACDCEDB
4	ACECBE

with C with any number of tokens in-between. Using the approach of combining wildcards with the next regular character, the probability distribution over all possible next tokens given %AB% would be

$$\{ P(\%A\%|\%AB\%), P(\%B\%|\%AB\%), \ldots, P(\%E\%|\%AB\%) \}.$$

The sum of the probabilities of this probability distribution will be greater than 1, if there are tuples in a relation that satisfy multiple conditional probabilities. Therefore, generating conditional probabilities for every token in the vocabulary does not work for patterns that contain wildcards in-between characters.

To overcome these limitations of standard neural language models and to accurately estimate the cardinalities of LIKE-patterns, we propose the LIKE-*Pattern Language Model* (LPLM). First, we propose a new probability distribution function that assigns a probability only to the next likely token, instead of all possible next tokens in the vocabulary. LPLM operates on the character level [15] because word-based neural language models are too coarse-grained to represent LIKE-patterns. Second, we define a new language to embed LIKE-patterns that encodes the semantics of % and _ wildcards. Finally, we adapt the output function of traditional neural language models to our proposed probability distribution function. Figure 1 compares the architectures of a traditional to our method.

4.1 Probability Distribution Function

Our new probability distribution function follows from combining wildcards with the next regular character of a LIKE-pattern. However, instead of returning a set of probabilities for all possible next tokens in the vocabulary, as in traditional language models, our function only returns the probability of the next likely token. The general idea of this approach is as follows. Our probability distribution function always assumes the next token is % in a given sequence of tokens from LIKE-patterns during the computation of the distribution. If the next token is %, it combines it with the character that follows % to determine the character's conditional probability for a given sequence of tokens. If not, it first divides the probability distribution of the next token into two new probabilities according to the type of the next token and computes them.

Algorithm 1 demonstrates the computation of the sequence of probabilities $\mathcal{P}$ for the tokens of a given LIKE-pattern p. The algorithm first checks whether the first token is a wildcard or a regular character and then initializes the sequence of probabilities $\mathcal{P}$ accordingly in Lines 1–8. Specifically, it the first token is a character, then the probability of the character surrounded by % is added to the sequence in Line 2. Otherwise, the probability of the second token in the sequence surrounded by % is added to the sequence in Line 5. If the token is a _, we also append (denoted by $\oplus$) the probability of the first two tokens followed by % given the second token surrounded by % in Line 7. Subsequently, the algorithm examines the intermediate tokens in sequence in Lines 9–20. In Lines 10–12, if the current token $p[i]$ is not a wildcard, we add two conditional probabilities in $\mathcal{P}$ regarding the following sequences: a) the subsequence of p up to $p[i-1]$ followed by $p[i]$ surrounded by %, and b) the subsequence of p up to $p[i]$ followed by %. Otherwise, if $p[i]$ is the

wildcard %, we add a conditional probability to $\mathcal{P}$ regarding the subsequence of p up to $p[i{+}1]$ in Line 16, while if $p[i]$ is the wildcard _, we add two conditional probabilities regarding the subsequence up to $p[i{-}1]$ followed by $p[i{+}1]$ surrounded by %, and the subsequence up to $p[i{+}1]$ followed by % in Lines 18–19. Finally, the algorithm checks the last token of p in Lines 21–25. If the last token is not a wildcard, it adds the probability of p given p followed by %. Otherwise, if the last token is _, it adds the probability of p given the subsequence up to the last token followed by %.

Algorithm 1 Compute Probability Sequence

Input: LIKE-pattern p
Output: sequence of probabilities $\mathcal{P}$

1: **if** $p[1] \notin \{\%, \_\}$ **then** ▷ Examining initial token.
2: $\quad\mathcal{P} \leftarrow \langle P(\%p[1]\%), P(p[1]\% \mid \%p[1]\%)\rangle$
3: $\quad i \leftarrow 2$
4: **else**
5: $\quad\mathcal{P} \leftarrow \langle P(\%p[2]\%)\rangle$
6: $\quad$**if** $p[1] =$ _ **then**
7: $\quad\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p[1]p[2]\% \mid \%p[2]\%)\rangle$
8: $\quad i \leftarrow 3$
9: **while** $i < |p| - 1$ **do** ▷ Examining intermediate tokens.
10: $\quad$**if** $p[i] \notin \{\%, \_\}$ **then**
11: $\quad\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p[1{:}i{-}1]\%p[i]\% \mid p[1{:}i{-}1]\%)\rangle$
12: $\quad\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p[1{:}i]\% \mid p[1{:}i{-}1]\%p[i]\%)\rangle$
13: $\quad\quad i \leftarrow i + 1$
14: $\quad$**else**
15: $\quad\quad$**if** $p[i] =$ % **then**
16: $\quad\quad\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p[1{:}i{+}1]\% \mid p[1{:}i])\rangle$
17: $\quad\quad$**else if** $p[i] =$ _ **then**
18: $\quad\quad\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p[1{:}i{-}1]\%p[i{+}1]\% \mid p[1{:}i{-}1]\%)\rangle$
19: $\quad\quad\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p[1{:}i{+}1]\% \mid p[1{:}i{-}1]\%p[i{+}1]\%\ )\rangle$
20: $\quad\quad i \leftarrow i + 2$
21: **if** $p[|p|] \notin \{\%, \_\}$ **then** ▷ Examining last token.
22: $\quad\mathcal{P} \leftarrow \mathcal{P} \oplus \langle P(p \mid p\%)\rangle$
23: **else if** $p[|p|] =$ _ **then**
24: $\quad\mathcal{P} \leftarrow \mathcal{P} \oplus P(p \mid p[1{:}i{-}1]\%)$
25: **return** $\mathcal{P}$

Example 4.2. Consider the LIKE-pattern p = %AB%C as input to Algorithm 1. Since the first token $p[1]$ is a % wildcard, it is combined with the following regular character A, and probability P(%A%) is appended to $\mathcal{P}$. Since B is a regular character that immediately follows the regular character A, the probability P(%AB%|%A%) is decomposed into two new probabilities P(%A%B%|%A%) and P(%AB%|%A%B%) that are also appended to $\mathcal{P}$. Finally, since the last token of p is not a wildcard, the probability P(%AB%C|%AB%C%) is appended to $\mathcal{P}$. Thus, the final sequence of probabilities for all tokens in the pattern p that is returned by Algorithm 1 is

$$\mathcal{P} = \langle\, P(\%A\%), P(\%A\%B\%|\%A\%), P(\%AB\%|\%A\%B\%),\\ P(\%AB\%C\%|\%AB\%), P(\%AB\%C|\%AB\%C\%)\,\rangle.$$

We then define the probability (selectivity) of a LIKE-pattern as the product of all conditional probabilities of its tokens. For example, the selectivity of the LIKE-pattern %AB%C is

$$P(\%AB\%C) = P(\%A\%) \cdot P(\%A\%B\%|\%A\%) \cdot P(\%AB\%|\%A\%B\%) \cdot P(\%AB\%C\%|\%AB\%) \cdot P(\%AB\%C|\%AB\%C\%).$$

The cardinality of the LIKE-pattern is given by the product of its selectivity and the total number of tuples in the relation.

Note that by adding more constraints to the sequence of probabilities $\mathcal{P}$, the model can identify whether a given LIKE-pattern is a subpattern of another LIKE-pattern and estimate cardinalities accordingly. For instance, our probability distribution function guarantees that the estimated cardinality of the LIKE-pattern %A%B% will always be greater than or equal to the estimated cardinality of %AB%.

4.2 Embedding: The LIKE-Pattern Language

In a traditional neural language model, the length of the output is always the same as the length of the input. However, in LPLM, the number of tokens in the LIKE-pattern p might be different from the length of the corresponding sequence of probabilities $\mathcal{P}$ due to how we encode the semantics of wildcards. Note that the output length of $\mathcal{P}$ depends on the implicit structure and type of the LIKE-pattern p. Therefore, to obtain an input of the same length as the desired output, information about the structure and type of the LIKE-pattern must be explicitly encoded before inputting it into the neural language model. We define a transformation from a LIKE-pattern p into a word w in a new LIKE-*pattern language* that encodes this information.

This transformation must satisfy two requirements to avoid poor estimates. First, it must preserve the semantics of the % and _ wildcards. Second, it must not be pattern-specific. As such, we introduce the function τ that is applied to every token of the LIKE-pattern. If two adjacent tokens are both regular characters, τ encodes that information by placing $\bullet$ after the second token. If two adjacent tokens are a _ wildcard followed by a regular character, τ encodes this information by placing $*$ after the second token. If the last token is a character, τ encodes that information by placing $\diamond$ after the token. If the first token is character, τ encodes that information by placing $\circ$ after the token. Otherwise, τ assumes that there is a % wildcard in-between the characters.

Given a LIKE-pattern p as a sequence drawn from the vocabulary $\mathcal{T} \cup \{\%, \_\}$ and $1 \leq i \leq |p|$, the function τ is given as

$$\tau(p,i) = \begin{cases} p[i] & \text{if } i > 1 \wedge p[i-1] = \% \\ p[i]* & \text{if } i > 1 \wedge p[i-1] = \_ \\ p[i]\diamond & \text{if } i = |p| \wedge p[i] \neq \% \\ p[i]\circ & \text{if } i = 1 \wedge p[i] \neq \% \\ p[i]\bullet & \text{if } p[i] \neq \% \wedge p[i] \neq \_ \\ \epsilon & \text{otherwise} \end{cases}$$

where ϵ is the empty token.

Example 4.3. Continuing from the previous example, consider again the LIKE-pattern p = %AB%C. Applying $\tau(p,i)$ for $1 \leq i \leq |p|$ leads to the following results $\tau(p,1) = \epsilon$, $\tau(p,2) = \mathtt{A}$, $\tau(p,3) = \mathtt{B}\bullet$, $\tau(p,4) = \epsilon$, and $\tau(p,5) = \mathtt{C}\diamond$. The result of the transformation of the LIKE-pattern p is, therefore, $w = \mathtt{AB}\bullet\mathtt{C}\diamond$. Notice that after this transformation the length of w, i.e., $|w| = 5$, matches the length of the desired sequence of probabilities $\mathcal{P}$ (cf. previous example) that the neural language model needs to generate in output. Figure 2b illustrates this process step by step for the LIKE-pattern p. As a

second, more complex example, consider the LIKE-pattern p' = %AB_A%C%D which is transformed by applying τ to w' = AB●A∗CD⋄.

4.3 Activation Function

Traditional neural language models generate probability distributions by applying a softmax function that assigns a probability to each token in the vocabulary given the preceding token sequence. The softmax function enforces the requirement given in Equation 1, i.e., that the sum of the probabilities of the token in the vocabulary for a given preceding token sequence must sum up 1. Our probability distribution function assigns conditional probability only to the next likely token for a given preceding sequence instead of all possible tokens in the vocabulary. Since our probability distribution function does not generate multiple probabilities, the softmax function is not a good classifier for our neural language model. Instead, LPLM uses sigmoid, another popular activation function, as an activation function. The sigmoid function takes a single real value as an input and maps it to the interval $[0, 1]$ to represent the next token's probability. Formally, the sigmoid function is given by

$$sigmoid(x) = \frac{1}{1 + e^{-x}}, \tag{2}$$

where x represents a weighted sum of feature vector for a specific LIKE-pattern in our case.

5 CARDINALITY ESTIMATION USING LPLM

In this section, we describe how cardinalities of LIKE-predicates are estimated using the LIKE-Pattern Language Model (LPLM). Figure 3 shows a high-level overview of the proposed method. The first step is the generation of training data for LPLM. The second step is the transformation of LIKE-patterns into the LIKE-Pattern Language (cf. Section 4.2) and the assignment of a probability to each token in the transformed LIKE-pattern. Finally, the last step is training LPLM using the transformed LIKE-patterns and their corresponding probabilities, and the estimation of the cardinalities of LIKE-patterns.

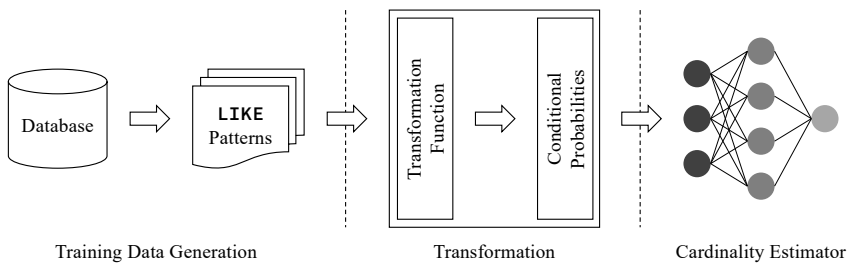


Fig. 3. Summary of our LPLM approach.

5.1 Training Data Generation

To obtain training data for LPLM, we propose a method that generates a subset of all possible LIKE-patterns for a given sequence database. More specifically, we generate n LIKE-patterns from randomly selected rows from a given sequence database SD. We begin by randomly picking a row r from SD. Then, we pick a random set $\mathcal{I}$ of item indexes from r. Next, we iterate over all tokens in r. We replace each token with the % or _ wildcard if the index of the token is in $\mathcal{I}$ and the previous token has not already been replaced by the % wildcard. This way we ensure that the pattern will not contain consecutive % wildcards. The procedure is repeated until n LIKE-patterns

Algorithm 2 Generate Random LIKE-Patterns

Input: sequence database SD, number of patterns n
Output: set of LIKE-patterns P

```
1:  P ← ∅
2:  while |P| < n do
3:      r ← random_row(SD)
4:      I ← random_number_list(1, |r|)
5:      p ← ⟨⟩
6:      for i ← 1 to |r| do
7:          if i ∉ I then
8:              p ← p ⊕ r[i]
9:          if i − 1 ∉ I and (i = |r| ∨ i + 1 ∉ I) then
10:             p ← p ⊕ _
11:         else if p[|p|] ≠ % then
12:             p ← p ⊕ %
13:     P ← P ∪ {p}
14: return P
```

have been generated. Algorithm 2 shows the pseudocode of our method for randomly generating LIKE-patterns.

Example 5.1. Consider the sample database in Table 1. Given the sequence ACECBE, randomly drawn from the sequence database, $n = 3$ and a list of random indexes $[1, 2, 5]$, the output of the algorithm will be the LIKE-pattern %EC_E.

5.2 LIKE-Pattern Transformation

Given a LIKE-pattern p from the output Algorithm 2, we first transform p into a word w in the new LIKE-pattern language. Then, we assign a probability to each token in w. These probabilities are computed by evaluating the corresponding patterns against the sequence database SD. Finally, w is associated with a vector containing the probability of each token.

Example 5.2. Consider the LIKE-pattern p = %AB%C%. The pattern p is first transformed into a word in our LIKE-pattern language resulting in w = AB•C. Then, the vector v of ground truth probabilities based on the sequence database from Table 1 is computed as follows.

$w =$	A	B	•	C
$\mathcal{P} = \langle$	P(%A%)	P(%A%B%|%A%)	P(%AB%|%A%B%)	P(%AB%C%|%AB%) $\rangle$
$v = \langle$	1	1	0.25	1 $\rangle$

As SD contains four sequences, the ground truth cardinality of the the LIKE-pattern p = %AB%C% can then be computed as $1 \cdot 1 \cdot 0.25 \cdot 1 \cdot 4 = 1$. Note that Sequence 1 is only sequence in SD that matches the given LIKE-pattern p.

5.3 Cardinality Estimation

The last step of LPLM is the cardinality estimator. We train the cardinality estimator on all transformed LIKE-patterns together with the corresponding vectors of probabilities. More specifically, our cardinality estimator processes each transformed LIKE-pattern sequentially, one character at a time, and predicts the conditional probability of the following character given the preceding

characters. Training our cardinality estimator can be considered a regression problem where one-hot-encoded LIKE-patterns are the independent features and the vectors of conditional probabilities are the dependent feature.

We use binary cross-entropy to compute the loss between the probability distributions for predicted and actual values for a character in LIKE-pattern p. The binary cross-entropy is defined as

$$L_{BCE} = -\sum_{i=1}^{|w|} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i), \tag{3}$$

where y_i and $\hat{y}_i$ are the predicted and actual probability, respectively, for the character in the transformed LIKE-pattern that occurs at position i and w is the transformation of p. Our proposed method seeks to minimize the average binary cross-entropy over the entire set of LIKE-predicates as

$$z = \frac{1}{N} \sum_{n=1}^{N} L_{BCE}, \tag{4}$$

where N is the number of LIKE-patterns in the training set.

Our neural language model accepts transformed LIKE-patterns as a query and returns the vector of conditional probabilities of the LIKE-pattern as an estimation. In order to estimate the result cardinality of the corresponding LIKE-predicate, we use the product of the conditional probabilities as its selectivity and multiply with the number of rows in the sequence database.

5.4 Deep Learning Architecture

A number of deep learning architectures such as LSTM [11], GRU [6] have been proposed for neural language modeling. We use a Gated Recurrent Unit (GRU) to learn LPLM efficiently. We studied different numbers of layers $(1, 2)$ and hidden units $(128, 256, 512)$ and decided to use a GRU with one layer and 256 hidden states. We also evaluated our model with different batch sizes $(16, 32, 64, 128, 256, 512)$ and various learning rates $(0.1, 0.01, 0.001, 0.0001, 0.00001)$. The model returns its best performance with a batch size of 128 and a learning rate of 0.0001. We used 64 epochs to train our model.

5.5 Handling Data Updates

LPLM is currently not designed to support incremental re-training. LPLM learns the probability distribution of each pattern from transformed LIKE-patterns and the ground truth probabilities. After completing the training phase, the LIKE-pattern samples and the ground truth probabilities are discarded to save space. Hence, the model needs to be re-trained in case of data updates. Naturally, the re-training does not need to happen too often. One could track the accuracy of the estimates, and only when the accuracy deteriorates below a user-defined threshold, re-training occurs. However, incremental re-training of LPLM is out of the scope of this paper and an interesting direction for future work.

6 EXPERIMENTAL EVALUATION

We first examine the efficacy of LPLM compared to both traditional and deep-learning-based state-of-the-art approaches (Section 6.1). In addition, we conduct an ablation study to investigate the effect that different parameters have on the efficacy of LPLM (Section 6.2). Finally, we evaluate the performance LPLM in terms of the end-to-end query runtimes in PostgreSQL 14.5 (Section 6.3). We begin by describing the experimental setup. Our implementation of LPLM is publicly available[1].

[1]https://github.com/dbis-ukon/lplm

Table 2. Statistics for the data sets used in our evaluation.

	DBLP-AN	IMDb-AN	IMDb-MT	TPCH-PN
Total	450,000	550,000	500,000	200,000
Unique	221,784	525,019	343,553	199,997
Min Length	9	10	1	21
Max Length	44	39	79	50
Mean Length	14.61	14.14	14.73	32.75

Table 3. Types of LIKE-queries that are supported by estimation methods

	Exact	Substring	Prefix/Suffix	%	% and _
LPLM	✓	✓	✓	✓	✓
Astrid	–	✓	✓	–	–
P-SPH	–	✓	–	✓	–
LBS	–	✓	–	✓	✓
Postgres	✓	✓	✓	✓	✓

Data Sets. We perform all experiments on four benchmark datasets from three different sources. More specifically, we use author names (DBLP-AN) obtained from DBLP[2], actor names (IMDb-AN) and movie titles (IMDb-MT) obtained from IMDb[3], and part names (TPCH-PT) obtained from TPC-H[4]. All of these data sets have been frequently used in previous evaluations of LIKE-query cardinality estimation [1, 2, 21, 22, 34]. Table 2 reports the characteristics of these four data sets in terms of total and unique number of sequences as well as minimum, maximum, and average length.

Generating Training Data for LPLM. For each data set, we use Algorithm 2 to generate a pair of a training and a validation set for LPLM. Due to the large number of LIKE-patterns, it would be infeasible in terms of time and memory consumption to train LPLM using the whole set of all possible LIKE-patterns. Hence, for each data set, we generate 5 and 1 million random LIKE-patterns in the training and the validation set, respectively.

State-of-the-Art Methods. To evaluate the efficacy of LPLM, we compare it against state-of-the-art techniques that use either a traditional or a deep-learning approach. As outlined in Section 2, most state-of-the-art techniques put restrictions on the structure of supported LIKE-patterns in order to reduce the size of the summary data structure. Table 3 summarizes the capabilities of the chosen baseline approaches and relates them to LPLM.

In what follows, we describe these baseline approaches and how we used them in our experiments in more detail.

- **Astrid** [34] is the deep-learning-based state-of-the-art approach for the cardinality estimation of prefix, suffix, and substring LIKE-patterns that do not contain any wildcard (% and _) in-between characters. Astrid estimates the selectivity of queries based on word embedding. To apply Astrid to estimate the cardinalities of LIKE-patterns that contain % in between characters, we first split each LIKE-pattern into subqueries, i.e., one prefix, one suffix, and one or more substring queries, according to the % wildcard. For example, consider the LIKE-pattern AB%C%D, the subqueries are the prefix query AB%, the substring query %C%, and the suffix query %D. We first compute the selectivity of all subqueries, and then we estimate the

[2]https://dblp.org
[3]https://imdb.com
[4]https://www.tpc.org/tpch/

cardinality of the LIKE-pattern as the product of the selectivities of the subqueries multiplied by the number of rows in the database. If the resulting estimated cardinality is greater than zero but less than one, then we assume the cardinality of the LIKE-pattern is equal to one.

- **P-SPH** [2] is the state-of-the-art approach for the cardinality estimation of substring LIKE-patterns that may contain % wildcards. P-SPH estimates the selectivity of LIKE-patterns using a new type of histogram structure that is based on frequent sequence patterns. To build this histogram for each dataset, we use 2048 buckets. To mine frequent sequence patterns, we use 1.5% of the dataset size as the minimum support threshold. According to the authors of the original paper, these two values represent the optimal configuration for P-SPH. Similar to Astrid, we assume that the cardinality of a LIKE-pattern is equal to one if the estimated cardinality is greater than zero but less than one.
- **LBS** [22] is the state-of-the-art approach for the cardinality estimation of substring LIKE-patterns that may contain both % and _ wildcards. LBS exploits an extended N-gram table as summary structure to store the cardinalities of possible base strings with length up to N. Then, it estimates the cardinality of a query based on the cardinalities of the minimal base strings of the query. We set the $N = 6$ and the maximum edit distance threshold to 1 to exploit an extended set of all N-grams. We prune the N-grams table by removing base strings that have true cardinalities lower than 20. According to the authors, these settings represent the best configuration. Notice, however, that in this configuration, LBS will perform poorly for LIKE-patterns that have a length greater than 6 and true cardinality less than 20.
- **PostgreSQL** is the only state-of-the-art approach for cardinality estimation that supports all types of LIKE-patterns. PostgreSQL's query planner [32] uses column statistics to estimate cardinalities during query planning. These statistics include information about the distribution of values and the presence of patterns in a column. PostgreSQL collects these statistics in the form of histograms. To obtain cardinality estimates for LIKE-patterns, we used histogram-based cardinality estimation of PostgreSQL. We obtain these estimates from the query planner using the EXPLAIN statement and a single table for each dataset.

Query Sets. In order to test the accuracy of our cardinality estimator, we use five benchmarks that differ in terms of the query types that their query set contains. This setup is necessary to ensure fair comparisons in a setting where most state-of-the-art approaches do not support all query types. Table 4 summarizes the statistics of the benchmark query sets used in the evaluation.

In what follows, we explain what query types the five benchmarks contain and how we generated them in more detail.

- The **Astrid Benchmark** is generated using the same approach as Suraj et al. [34] for evaluating Astrid. This query set includes LIKE-queries with patterns that have prefix, suffix, and substring form.
- The **LPLM Benchmark** consists of LIKE-patterns that are generated by Algorithm 2. As the LIKE-patterns are randomly generated, their structure is unrestricted.
- The **P-SPH Benchmark** is generated using the same approach as described by the authors to evaluate P-SPH. Queries in this set have the form $\%s_1\%s_2\% \ldots \%s_n\%$, where s_i is one or more characters. We refer the reader to the work of Aytimur and Çakmak [2] that describes this process in detail.
- The **LBS Benchmark** is generated using the same approach as Lee et al. [22] for evaluating LBS. This query set includes queries in the form of $\%w_0\%$ and $\%w_1\%w_2\%$, where w_i is a word with a length between 5 and 12. More specifically, to generate this query set, a word w with a length between 5 and 12 is randomly drawn. Then, a random number (from 0 to 2) of _ wildcards are inserted at random positions in w.

Table 4. Summary of the statistics of the different query sets used in the experimental evaluation.

Query Set	Dataset	# Queries	# Wildcards			Cardinality		
			Min	**Max**	**Mean**	**Min**	**Max**	**Mean**
Astrid Benchmark	DBLP-AN	474,104	0	0	0	1.0	318,411	27.60
	IMDb-AN	912,302	0	0	0	1.0	459,738	18.78
	IMDb-MT	835,177	0	0	0	1.0	234,970	14.76
	TPCH-PN	969	0	0	0	2038.0	189,605	11976.07
LPLM Benchmark	DBLP-AN	500,000	0	12	1.4	1.0	291,283	137.38
	IMDb-AN	500,000	0	11	1.30	1.0	260,195	139.42
	IMDb-MT	500,000	0	21	2.05	1.0	238,810	219.30
	TPCH-PN	500,000	0	16	2.87	1.0	180,666	347.25
P-SPH Benchmark	DBLP-AN	500,000	0	10	2.25	1.0	372,507	978.0
	IMDb-AN	500,000	0	9	2.12	1.0	459,738	1183.16
	IMDb-MT	500,000	0	18	2.87	1.0	290,809	2072.0
	TPCH-PN	500,000	0	15	3.93	1.0	196,591	4308.01
LBS Benchmark	DBLP-AN	500,000	0	2	1.29	1.0	30,556	79.18
	IMDb-AN	500,000	0	2	1.18	1.0	25,751	103.05
	IMDb-MT	500,000	0	2	1.22	1.0	66,370	70.60
	TPCH-PN	1176	0	2	1.67	10,614	37,025	11209.52

- The **Negative Queries** set consists of queries that do not match any rows in the database, i.e., that have true cardinality zero. We use `LIKE`-patterns that all methods can support from Table 3. Hence, we only generate substring `LIKE`-patterns for the negative query set. To generate such queries, we extend Algorithm 2. Instead of replacing characters from a randomly drawn sequence with wildcards, the extended algorithm first removes the characters in the selected indexes and then shuffles the characters in the remaining sequence. Then, we follow the same procedure in the Algorithm 2 and keep all `LIKE`-patterns that have true cardinalities zero. Each dataset has 10k negative `LIKE`-patterns.

Evaluation Metric. To measure the accuracy of all cardinality estimation approaches, we report the Q-error [29], which measures the factor that the estimated cardinality deviates 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. To use the Q-error as an evaluation metric for the negative queries, we use 1 as the true cardinality. The best Q-error is obtained when estimated cardinality is equal to actual cardinality, and the worst Q-error is unbounded.

Environment. All our experiments were performed on a machine with two NVIDIA RTX 3090 24 GB GPUs, AMD Ryzen Threadripper 3970X CPU and 256 GB of RAM, Ubuntu 20.04 LTS. We used PyTorch 1.10.1 for building the deep-learning models.

6.1 Comparison to the State of the Art

We report results of the experimental comparison of LPLM to Astrid, P-SPH, LBS, and PostgreSQL in terms of estimation accuracy, run-time performance, and model size on all five test benchmarks.

Estimation Accuracy. Table 5 gives the results of our experimental evaluation of the accuracy (Q-error) of LPLM and the state-of-the-art approaches. Learning-based approaches were trained

Table 5. Comparison of estimation accuracy (Q-error) of LPLM against traditional and machine-learning-based baselines.

Query Set	Estimation Method	DBLP-AN					IMDb-AN				
		G. Mean	**Mean**	**Median**	**90th**	**99th**	**G. Mean**	**Mean**	**Median**	**90th**	**99th**
LPLM Benchmark	LPLM	**2.87**	**4.14**	**2.52**	**8.26**	**32.28**	**2.24**	**3.18**	**1.90**	**5.42**	**21.94**
	Postgres	14.17	24.27	22.50	45.0	45.0	27.04	41.35	55.0	55.0	55.0
Astrid Benchmark (All Queries)	LPLM	4.17	6.69	3.98	13.94	41.72	2.90	3.97	2.70	7.30	21.65
	Astrid	**2.48**	**3.73**	**2.20**	**6.02**	**28.88**	**1.95**	**3.10**	**1.65**	**4.37**	**26.57**
	Postgres	17.19	26.42	22.5	45.0	45.0	31.12	41.36	55.0	55.0	55.0
Astrid Benchmark (Substring)	LPLM	4.24	6.80	4.07	14.19	42.35	3.14	4.35	2.97	8.1	23.90
	Astrid	2.50	3.58	2.26	5.80	26.98	1.93	2.94	1.66	4.24	24.42
	P-SPH	224.02	373.98	337.5	675.0	675.0	440.76	602.33	825.0	825.0	825.0
	LBS	**2.01**	**3.09**	**1.14**	**8.0**	**17.0**	**1.52**	**2.16**	**1.0**	**5.0**	**15.0**
	Postgres	17.03	26.32	22.5	45.0	45.0	30.90	41.19	55.0	55.0	55.0
P-SPH Benchmark	LPLM	**2.15**	**3.11**	**1.78**	**5.54**	**22.66**	**1.79**	**2.32**	**1.47**	**4.02**	**14.05**
	Astrid	121.01	956.33	147.0	2306.0	11720.04	126.43	1155.36	177.0	2729.0	14379.01
	P-SPH	9.59	56.80	6.03	168.75	675.0	11.1	94.93	6.07	412.5	825.0
	LBS	5.81	17.60	4.51	42.0	209.0	5.20	17.55	3.90	40.19	228.23
	Postgres	5.88	12.19	5.04	33.0	83.27	6.86	15.35	6.0	55.0	95.07
LBS Benchmark	LPLM	3.26	5.90	2.70	11.42	51.15	3.28	6.02	2.75	11.26	52.73
	LBS	**2.61**	**4.44**	**2.0**	**12.0**	**20.0**	**2.02**	**3.39**	**1.04**	**9.0**	**19.0**
	Postgres	6.2	13.71	6.43	45.0	45.0	10.69	22.53	11.0	55.0	55.0
Full Benchmark	LPLM	**3.05**	**5.47**	**2.48**	**10.97**	**44.74**	**2.60**	**3.99**	**2.22**	**7.28**	**28.22**
	Postgres	9.87	19.05	11.25	45.0	52.04	17.70	32.06	27.5	55.0	55.0

Query Set	Estimation Method	IMDb-MT					TPCH-PN				
		G. Mean	**Mean**	**Median**	**90th**	**99th**	**G. Mean**	**Mean**	**Median**	**90th**	**99th**
LPLM Benchmark	LPLM	**1.24**	**3.27**	**1.94**	**5.24**	**20.25**	**1.51**	**1.68**	**1.38**	**2.30**	**6.26**
	Postgres	24.0	35.72	46.0	46.0	46.0	15.28	32.68	20.0	20.0	439.15
Astrid Benchmark (All Queries)	LPLM	4.02	6.24	3.84	12.50	39.86	**1.09**	**1.10**	**1.02**	**1.40**	**1.80**
	Astrid	**1.96**	**2.80**	**1.66**	**4.39**	**21.83**	255.38	823.39	348.62	1889.37	5410.27
	Postgres	24.68	33.80	46.0	46.0	46.0	1.80	9.14	1.20	2.72	111.52
Astrid Benchmark (Substring)	LPLM	4.55	7.09	4.46	14.08	45.01	1.02	1.02	1.01	1.05	1.09
	Astrid	1.98	2.86	1.67	4.43	23.15	835.17	1349.26	1116.06	2411.94	6244.32
	P-SPH	387.34	545.07	750.0	750.0	750.0	4.18	8.04	3.08	36.10	36.88
	LBS	**1.56**	**2.29**	**1.0**	**5.0**	**16.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	Postgres	24.93	34.05	46.0	46.0	46.0	1.36	1.42	1.30	1.84	2.72
P-SPH Benchmark	LPLM	**1.49**	**1.80**	**1.25**	**2.74**	**9.36**	**1.36**	**1.53**	**1.18**	**2.19**	**6.38**
	Astrid	211.13	2050.51	347.0	5713.0	22193.0	426.53	4289.87	900.0	12535.10	39765.06
	P-SPH	10.07	86.69	5.68	375.0	750.0	10.79	40.75	8.70	150.0	300.0
	LBS	5.44	17.18	4.28	42.0	204.0	5.16	27.85	3.82	31.45	422.89
	Postgres	8.32	21.83	7.56	49.0	176.13	4.91	16.81	2.86	42.65	189.35
LBS Benchmark	LPLM	3.27	5.67	2.81	10.73	44.49	1.94	3.01	1.28	5.79	22.21
	LBS	**2.24**	**3.86**	**1.39**	**11.0**	**19.0**	**1.02**	**1.03**	**1.0**	**1.0**	**1.96**
	Postgres	8.19	17.39	7.66	46.0	46.0	1.38	1.43	1.32	1.84	2.71
Full Benchmark	LPLM	**2.74**	**4.53**	**2.25**	**9.05**	**33.03**	**1.43**	**1.61**	**1.29**	**2.25**	**6.36**
	Postgres	15.35	28.13	23.0	46.0	95.56	8.03	17.26	14.60	20.0	148.65

with queries from the corresponding query set. All approaches were applied to all query sets that they support. We report geometric mean, mean, median, 90th, and 99th percentile errors. We also created an additional query set, Full Benchmark, that contains all queries from all other benchmarks. The only approaches that support all queries in the Full Benchmark are LPLM and PostgreSQL.

LBS outperforms LPLM on the LBS Benchmark. These results are a direct consequence of the characteristics of the queries in this benchmark. Recall that strings with a cardinality less than 20 are pruned from its N-gram table summary structure and that LBS returns an estimated cardinality

of 0 for infrequent LIKE-patterns. Since most queries in the LBS Benchmark have a true cardinality of 1, the Q-error is therefore almost always 1.

Astrid outperforms LPLM on the Astrid Benchmark for the DBLP-AN, IMDb-AN, and IMDb-MT datasets. These results are not surprising as Astrid is specifically designed for the type of queries in this benchmark. Furthermore, most queries in the training and test set have again a true cardinality of 1 with the same consequence as above. In general, we observe that such training and test datasets have a tendency to favor Astrid. However, Astrid's performance drops substantially on the TPCH-PN dataset. The main reason for this degradation is that the part name strings are combinations of only 96 unique words, leading to a low number of unique prefixes, suffixes, and substrings. Hence, Astrid is unable to learn the underlying structure of the TPCH-PN dataset by training on a small number of samples.

On the P-SPH Benchmark, LPLM outperforms all state-of-the-art approaches. Since Astrid is trained on LIKE-patterns that do not contain % wildcards in-between characters, it consistently underestimates the cardinality of such LIKE-patterns. Again, LBS returns an estimated cardinality of 0 for most queries in this set. To estimate the cardinalities of LIKE-patterns that contain % in between sequence of characters, LBS finds the estimated set of tuple ids in the database that match the subqueries of LIKE-patterns and then get the union size of the estimated sets. LIKE-patterns in that group have a higher number of %-wildcards in between characters of LIKE-patterns, which mostly ends up with a low-sized union set. Hence, LBS underestimates the cardinalities of LIKE-patterns in that test benchmark.

Surprisingly, P-SPH is outperformed by both LPLM and LBS on its own benchmark. The reason is that the underlying histogram structure does not accurately approximate the data sets, as most LIKE-patterns do not match any endpoint values.

While PostgreSQL supports all types of LIKE-patterns like our method, it has worse accuracy than LPLM in all cases. PostgreSQL returns a constant number for most of the LIKE-queries, leading to overestimation of LIKE-patterns that have low actual cardinalities.

For the most part, the effect that we have so far observed is a strong dependence between an approach and its way of generating (training and) test queries. Nevertheless, LPLM performs comparably or better than state-of-the-art approaches in the queries they support, and clearly outperforms PostgreSQL, the only state-of-the-art approach that support all queries, in all cases.

Next, we study the estimation accuracy (Q-error) of LPLM and the state-of-the-art approaches depending on the true cardinalities of LIKE-queries. Figure 4 plots accuracy (geometric mean) over four different bins of queries from the Full Benchmark query set. We observe that there is no dependency between the true cardinalities queries and the accuracy of LPLM. The reason for this result is that LPLM is trained on randomly generated LIKE-patterns that provide a more representative summary of a dataset. In contrast to LPLM, most state-of-the-art approaches show a substantial dependency between accuracy and true cardinality. The minimum support threshold value is a key bottleneck for P-SPH. P-SPH uses 6,750 (DBLP-AN), 8,750 (IMDb-AN), 7,500 (IMDb-MT), and 3000 (TPCH-PN) as the minimum support thresholds. If any LIKE-pattern in the bins $[0, 10]$, $[11, 20]$, and $[20, 51]$ matches an endpoint value in the histogram, the estimated cardinality will get a minimum value as minimum support, which is excessively larger than the true cardinalities of the LIKE-queries in that benchmark. However, if a LIKE-pattern in these bins falls into the non-match case, P-SPH returns 10% of the minimum support value as cardinality estimation. Hence, P-SPH always overestimates LIKE-query that have true cardinality less than 10% of minimum support. The accuracy of Astrid decreases as the actual cardinality increases since Astrid is trained on LIKE-patterns that do not contain wildcards and, therefore, have low true cardinalities. Therefore, Astrid cannot capture the underlying structure of LIKE-patterns that have higher cardinalities and contain wildcards. LBS shows its worst performance for LIKE-queries in the bin $[11, 20]$. The reason

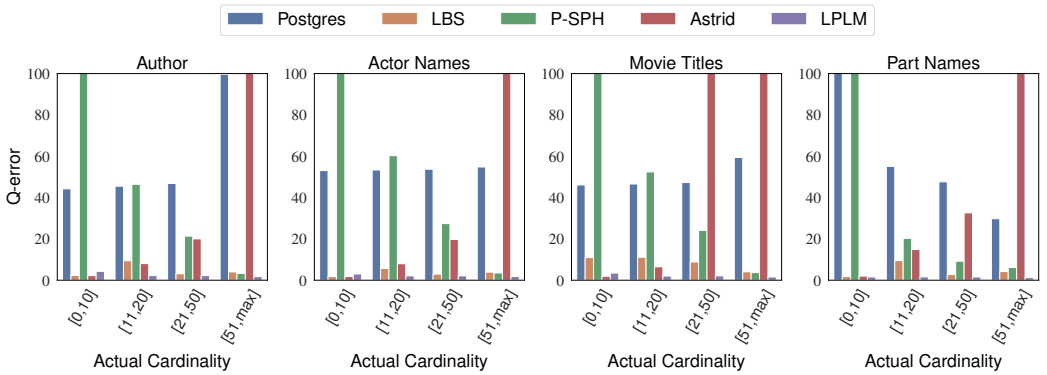


Fig. 4. Geometric mean of Q-errors of different methods for LIKE-patterns varying actual result cardinality.

that LBS fails for these queries is that LBS estimates the cardinality of a LIKE-query based on the cardinalities N-grams in its N-gram table. Since the N-gram table is pruned the keep only N-grams with a cardinality greater than 19, there are few matches between base strings of these LIKE-queries and N-grams in the N-gram table. Therefore, LBS mostly returns an estimated cardinality less than 1 for the LIKE-patterns that have actual cardinality lower than 20. LBS shows its best performance for LIKE-patterns with cardinalities less than 10 because most of the LIKE-queries in the bin $[0, 10]$ have a actual cardinality of 1 which results in a Q-error of 1.

Negative Query Set. In this experiment, we evaluate the performance of LPLM and state-of-art approaches on the negative query set. Table 6 shows the estimation accuracy for all methods for negative LIKE-patterns for all datasets. LBS has the best performance for the negative query set because LBS returns 0 as the estimated cardinality for every query in the negative set. The LBS summary structure selectively retains extended N-grams characterized by true cardinalities equal to or exceeding 20. Consequently, LBS assigns a Q-error of 1 to negative queries, thereby contributing to the best performance outcomes. Astrid has the second-best performance for the negative queries set because most queries in the training have a true cardinality of 1, which works in favor of Astrid for the negative queries set. LPLM learns from the order of tokens and their conditional probabilities. Since the order of tokens and the conditional probabilities of LIKE-patterns in the negative set are not learned by LPLM, it does estimate the cardinality of negative LIKE-patterns as accurately as the cardinality of queries that yield at least one result. PostgreSQL returns a constant number for each query in the negative queries set. P-SPH has the word performance because the underlying histogram structure only contains frequent patterns where all negative LIKE-patterns do not match any endpoint values.

Runtime. We conduct a runtime assessment for the entire pipeline of LPLM and state-of-the-art approaches. While we use a GPU for the training phase of machine-learning-based methods, we only use the CPU for inference in order to get a fair comparison between machine-learning-based and traditional approaches.

Our analysis is divided into three phases for LPLM and Astrid, and in two phases for LBS and P-SPH. The pre-processing phase encompasses the computation of the training dataset for LPLM and Astrid, along with the summary structure for LBS and P-SPH. In the training phase, we report the training time for LPLM and Astrid. In the online phase, we capture the query (estimation) time for all methods. To obtain the runtime for each approach, we randomly select 10, 000 LIKE-queries that each method can support for each data set. The results of our analysis are reported in Table 7.

Table 6. Estimation accuracy of LPLM and state-of-art approaches for negative query set.

Method	Dataset	G.Mean	Mean	Median	90th	99th
LPLM	DBLP-AN	10.1	14.87	9.84	29.21	79.92
	IMDb-AN	4.31	5.64	4.14	10.10	27.60
	IMDb-MT	6.67	9.79	6.49	18.85	58.89
	TPCH-PN	67.28	424.09	71.22	1025.93	5521.59
Astrid	DBLP-AN	1.08	1.08	1.075	1.13	1.24
	IMDb-AN	1.63	1.92	1.48	2.88	8.34
	IMDb-MT	1.96	2.52	1.62	5.40	12.27
	TPCH-PN	10.89	54.19	8.33	38.8	637.72
LBS	DBLP-AN	1.0	1.0	1.0	1.0	1.0
	IMDb-AN	1.0	1.0	1.0	1.0	1.0
	IMDb-MT	1.0	1.0	1.0	1.0	1.0
	TPCH-PN	1.0	1.0	1.0	1.0	1.0
P-SPH	DBLP-AN	675.0	675.0	675.0	675.0	675.0
	IMDb-AN	825.0	825.0	825.0	825.0	825.0
	IMDb-MT	750.0	750.0	750.0	750.0	750.0
	TPCH-PN	300.0	300.0	300.0	300.0	300.0
PostgreSQL	DBLP-AN	45.0	45.0	45.0	45.0	45.0
	IMDb-AN	55.0	55.0	55.0	55.0	55.0
	IMDb-MT	46.0	46.0	46.0	46.0	46.0
	TPCH-PN	20.0	20.0	20.0	20.0	20.0

Table 7. Runtime assessment for the entire pipeline of LPLM and state-of-the-art approaches.

Method	Pre-processing	Training (h)	Query time (ms)
LPLM	11.6 (h)*	6.3	1.5
Astrid	2.0 (min)	5.8	0.4
P-SPH	4.8 (h)	—	1.3
LBS	26.9 (min)	—	0.5
Postgres	—	—	27.0

Astrid demonstrates the fastest pre-processing time because it only precomputes the cardinality of prefixes, suffixes, and sub-strings up to length eight. LBS, which enumerates all N-grams up to length six, has the second fastest pre-processing time. Next comes P-SPH, which precomputes all positional frequent patterns and creates a histogram. LPLM[5] has the highest pre-processing time because it executes the probability distribution for each token in each of the five million `LIKE`-patterns and then precomputes the ground truth for each probability in the distribution. However, note the pre-processing of LPLM time can be further reduced by reducing the sample size. Based on the results reported in Figure 6, the latter will result in only a small increase in the Q-error.

In the training phase, Astrid builds a separate model for prefixes, suffixes, and substrings. Hence, we report the total time for all three models. Astrid has a lower training time than LPLM model because the total number of prefixes, suffixes, and substrings is lower than the number of `LIKE`-patterns. Regarding query time, Astrid has the fastest estimation time (0.4 ms), followed by LBS (0.5 ms) and then P-SPH (1.3 ms). LPLM has the highest running time (1.5 ms), being only slightly slower than the other approaches.

[5]For the pre-processing phase, we used a computing cluster with a slower CPU but by running 1,000 processes in parallel.

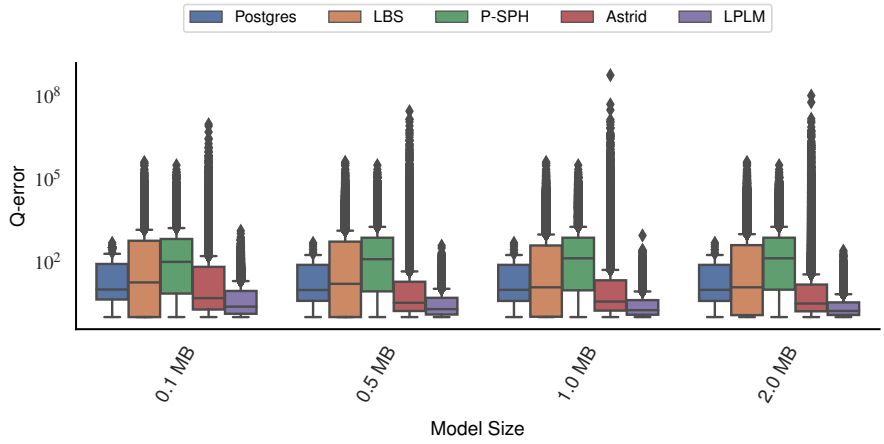


Fig. 5. Q-error of all methods varying space budget.

Impact of Model Size. In this section, we report on the memory requirements for each method in our study. First, we report the average model size on all data sets for the configuration of each method we used in the previous experiments. LPLM requires around 0.93 MB to store its selectivity estimation model. Astrid requires around 0.51 MB to store the embedding and the selectivity estimation model. LBS requires around 260 MB to store the N-gram table. P-SPH requires around 140 kB to store its histogram. Finally, PostgreSQL requires only 1.9 kB to store the most common values and a histogram.

To ensure a fair comparison of all methods, we therefore evaluate the accuracy of all methods w.r.t. different model sizes (0.1, 0.5, 1.0, and 2.0 MB). For LPLM and Astrid, we alter the hyperparameters to adjust the size of the models. For P-SPH, we adjust the number of histogram buckets. For LBS, we adjust the prune threshold. Finally, for Postgres, we modify the statistics target for a specific column in a table. We report the Q-error on 10,000 randomly selected `LIKE`-queries that all methods support for different model sizes in Figure 5. The results are consistent with those reported in Table 5. Specifically, LPLM demonstrates the lowest Q-error, followed by Astrid. Both LPLM and Astrid show an improvement with an increasing model size, particularly between 0.5 and 2.0 MB. PostgreSQL comes third with its accuracy being constant as the memory required by PostgreSQL is so low that it is always well below the given space budget. Finally, LBS and P-SPH profit somewhat from a larger model size, but their accuracy is still clearly worse than that of all other methods.

6.2 Ablation Study

In this section, we conduct an ablation study to examine the effectiveness of our proposed cardinality estimator LPLM. First, we report results for different training data set sizes on the accuracy of LPLM. Second,we evaluate the effectiveness of LPLM for different types of `LIKE`-patterns. Finally, we evaluate the effect of dataset size on estimation accuracy.

6.2.1 ***Impact of Training Data Set Size.*** To study the effect of the training data set size on the accuracy of LPLM, we conduct experiment varying the size of the training data set between 50 thousand and 5 million `LIKE`-patterns for all data sets, and we report the results in Figure 6. We observe an improvement in the performance of LPLM as we increase the size of the training data for all data sets. The improvement is more pronounced between 50 thousand and 1 million `LIKE`-patterns and is even more apparent for the 90th and the 99th percentile. We observe a smaller

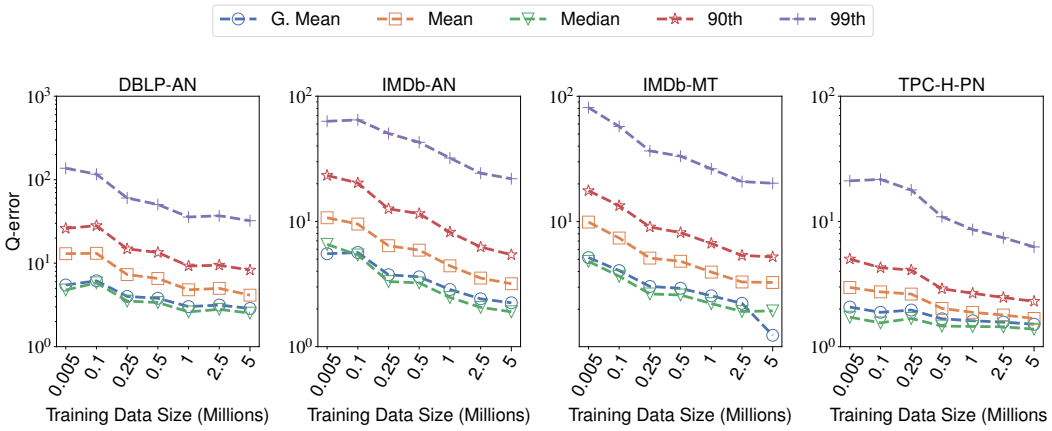


Fig. 6. Geometric mean of Q-errors of LPLM for different sizes of training data set.

improvement in all measures of LPLM between 1 and 5 million LIKE-patterns. Based on these results, we conclude that our methods can produce accurate cardinality estimates, even if the training data set is much smaller than the one we used in the first part of our evaluation.

6.2.2 ***Impact of* LIKE*-Pattern Type.*** Next, we investigate the impact of different types of LIKE-patterns on the performance of LPLM. For this experiment, we classify LIKE-patterns into the following types: exact, substring, prefix-suffix, and patterns that contain at least one wildcard in-between characters (cf. Table 3). Table 8 reports the results of our measurements. LPLM shows the highest accuracy for the exact type of LIKE-patterns because this type includes two constraints to the sequence of probabilities, one at the beginning and one at the end. The difference is more apparent in the 90th and 99th percentile. For the other types of LIKE-patterns, we observe no substantial difference in the accuracy of LPLM. In fact, LPLM yields similar accuracy for all pattern types with the exception of the 99th percentile for queries that contain wildcard(s) in-between characters, where the Q-error is a bit higher.

6.2.3 ***Impact of Data Set Size.*** In this section, we study the impact of the dataset size on the accuracy of LPLM and on model training time. We used all actor names (4.16M) and movie titles (2.52M) from the IMDb benchmark, all author names (3.49M) from the DBLP benchmark, and part names (4M) from the TPC-H benchmark. The average training time for all datasets was 5.6 hours. We report the Q-errors for LPLM in Table 9. By comparing these results with the results for LPLM reported in Table 3 we observe only a slight increase in the Q-errors for the larger data sets. This increase, although not very significant, can be attributed to the larger vocabulary size.

6.3 End-to-End Query Runtimes

In this section, we evaluate the performance LPLM in terms of the end-to-end query runtimes in PostgreSQL 14.5. We used the IMDb dataset and executed the Join Order Benchmark (JOB) [23] workload. To inject cardinalities into PostgreSQL, we extended Han et al.'s patch [8] to accept outside estimations for LIKE and equality predicates on string attributes in addition to the existing support for predicates over categorical and numerical attributes. We trained LPLM on the contents of the columns **movie_companies**.note, **title**.title, **name**.name, and **link_type**.link. We measure

Table 8. Estimation accuracy of LPLM for different types of LIKE-patterns.

Type	Dataset	G.Mean	Mean	Median	90th	99th
Exact	DBLP-AN	2.49	3.03	2.37	5.41	12.08
	IMDb-AN	1.77	1.97	1.65	3.14	6.26
	IMDb-MT	2.01	2.44	1.85	3.93	9.33
	TPCH-PN	1.38	1.39	1.37	1.54	1.74
Substring	DBLP-AN	3.73	5.99	3.41	12.54	38.52
	IMDb-AN	2.94	4.08	2.72	7.64	22.46
	IMDb-MT	3.91	6.19	3.69	12.60	40.51
	TPCH-PN	1.39	1.44	1.34	1.85	3.72
Prefix/ Suffix	DBLP-AN	3.33	5.04	3.02	10.26	31.63
	IMDb-AN	2.46	3.27	2.20	5.83	17.85
	IMDb-MT	2.72	3.80	2.36	7.45	22.10
	TPCH-PN	1.40	1.44	1.37	1.72	2.96
Contains % or _	DBLP-AN	2.72	5.36	2.05	10.21	52.33
	IMDb-AN	2.43	4.23	1.90	7.74	36.91
	IMDb-MT	2.19	3.71	1.68	6.59	30.21
	TPCH-PN	1.44	1.67	1.25	2.43	7.44
Negative	DBLP-AN	11.51	25.82	10.51	43.57	204.29
	IMDb-AN	4.67	13.16	4.0	16.66	110.54
	IMDb-MT	4.87	25.68	3.97	18.71	297.48
	TPCH-PN	4.12	167.86	2.69	20.51	2673.85

Table 9. Estimation accuracy of LPLM for larger datasets.

Method	G. Mean	Mean	Median	90th	99th
DBLP-AN	3.06	5.2	2.40	10.66	48.06
IMDb-AN	2.40	4.23	1.94	6.65	32.15
IMDb-MT	2.34	4.03	1.87	6.75	33.05
TPCH-PN	1.82	2.44	1.51	3.86	16.46

end-to-end runtimes for a total of 61 queries that include LIKE and/or equality predicates on these columns.

In addition to the baseline given by PostgreSQL, we study four different configurations. The first configuration (TrueCard) injects true cardinalities into the PostgreSQL optimizer for *all* types of predicates. The second configuration (TrueCard+PG) uses true cardinalities for string predicates (LIKE and equality) but uses PostgreSQL's own estimations for predicates on categorical and numerical attributes. The third configuration (LPLM+TrueCard) uses our estimations for string predicates but true cardinalities for other predicates. Finally, the fourth configuration (LPLM+PG) uses our estimations for string predicates, but PostgreSQL's estimates for the other types of else. Since PostgreSQL and LPLM are the only two techniques that support general LIKE-patterns (cf. Table 5), we have decided to focus on them in this experiment.

The results of this end-to-end runtime experiment are reported in Tables 10a and 10b. In Table 10a, we report the number of queries out of the 61 total queries that improved and regressed, as well as the respective average improvement and regression. Before we discuss these results in more detail, we need to point out that the PostgreSQL optimizer shows unexpected behavior in this experiment that seems to be inconsistent with the findings of Leis et. al. [23] in the sense that injecting true cardinalities does not yield better plans. Nevertheless, LPLM+PG improves the largest number of

Table 10. Experiments on the Join Order Benchmark.

	Improved		Regressed	
	# Queries	**Avg. Imp.**	**# Queries**	**Avg. Reg.**
PostgreSQL	—	—	—	—
TrueCard	27	11.13%	34	103.23%
TrueCard+PG	33	8.31%	28	41.47%
LPLM+TrueCard	27	10.97%	34	162.48%
LPLM+PG	36	8.34%	25	44.51%

(a) Improved and regressed queries.

	G. Mean	**Mean**	**Median**	**90th**	**99th**	**Total**
PostgreSQL	**0.25**	3.42	0.17	**10.60**	36.49	208.74
TrueCard	0.29	3.46	0.18	14.93	**30.63**	211.34
TrueCard+PG	0.26	3.38	**0.16**	10.85	36.78	206.5
LPLM+TrueCard	0.3	3.48	0.17	14.74	30.71	212.48
LPLM+PG	0.26	**3.36**	0.17	10.90	36.61	**205.64**

(b) Query execution time [s].

Table 11. Estimation accuracy of LPLM and PostgreSQL on the Join Order Benchmark.

Method	**G. Mean**	**Mean**	**Median**	**90th**	**99th**
LPLM	2.48	4.35	1.86	9.8	28.27
Postgres	4.70	32.96	2.61	91.30	451.38

queries in this experiment. Also, as reported in Table 10b, the same configuration improves the average and cumulative runtime of all 61 queries the most. Table 11 reports estimation accuracy (Q-error) for PostgreSQL and LPLM on JOB. Comparing these numbers to the Q-errors reported in Table 5 leads us to the conclusion that the LIKE-predicates in JOB are less challenging than the ones in the query sets proposed together with state-of-the-art methods, further explaining the relatively small total runtime improvement gained by using LPLM+PG on JOB.

We consider these initial end-to-end runtime results to be promising. However, more work is required to (a) study why the PostgreSQL optimizer shows this inconsistent behavior and (b) to establish more comprehensive and challenging benchmarks that enable us to study the impact of cardinality estimation for LIKE-predicates more systematically.

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced the LIKE-*Pattern Language Model*, a novel method for estimating the cardinality of general LIKE-patterns. Our approach employs a new language and a novel probability distribution function to capture the syntax and semantics of general LIKE-patterns. To train our neural language model, we also proposed a method that extracts random LIKE-patterns from randomly selected rows from a sequence database. Through an experimental evaluation on four real-world data sets, we demonstrated that our method outperforms the state of the art in terms of accuracy (Q-error) while supporting more types of LIKE-patterns.

As a next step, our goal is to extend LPLM to support cardinality estimation for regular expressions. We also plan to examine the applicability of our model for estimating the cardinality of approximate string matching queries.

ACKNOWLEDGMENTS

This work is supported by Grant No. GR 4497/5 and Grant No. CH 2464/1 of the Deutsche Forschungsgemeinschaft (DFG). We would also like to thank our project partners at ZHAW Zurich University of Applied Sciences, Kurt Stockinger and Pavel Sulimov, for their valuable feedback on the work presented in this paper.

REFERENCES

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

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

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A Neural Probabilistic Language Model. *Journal of Machine Learning Research* 3 (2003), 1137–1155.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146. https://doi.org/10.1162/tacl_a_00051

[5] Surajit Chaudhuri, Venkatesh Ganti, and L. Gravano. 2004. Selectivity estimation for string predicates: overcoming the underestimation problem. In *Proceedings of the International Conference on Data Engineering (ICDE'04)*. 227–238. https://doi.org/10.1109/ICDE.2004.1319999

[6] 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. https://doi.org/10.3115/v1/d14-1179

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[8] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877* (2021).

[9] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 1035–1050. https://doi.org/10.1145/3318464.3389741

[10] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005. https://doi.org/10.14778/3384345.3384349

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

[12] Yannis Ioannidis. 2003. The History of Histograms (abridged). In *Proceedings of the VLDB Conf.* 19–30. https://doi.org/10.1016/b978-012722442-8/50011-2

[13] Yannis E Ioannidis and Stavros Christodoulakis. 1991. On the propagation of errors in the size of join results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*. 268–277. https://doi.org/10.1145/119995.115835

[14] H. V. Jagadish, Raymond T. Ng, and Divesh Srivastava. 1999. Substring Selectivity Estimation. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '99)*. 249–260. https://doi.org/10.1145/303976.304001

[15] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *Proceedings of the AAAI conference on artificial intelligence*. https://doi.org/10.1609/aaai.v30i1.10362

[16] 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*.

[17] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'19)*.

[18] 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.

https://doi.org/10.1145/233269.233341
[19] Suyong Kwon, Woohwan Jung, and Kyuseok Shim. 2022. Cardinality Estimation of Approximate Substring Queries using Deep Learning. *Proc. VLDB Endow.* 15, 11 (2022), 3145–3157. https://doi.org/10.14778/3551793.3551859
[20] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the International Conference on Machine Learning, (ICML'14)*. 1188–1196.
[21] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. 2007. Extending Q-Grams to Estimate Selectivity of String Matching with Low Edit Distance. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. 195–206.
[22] 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. https://doi.org/10.1145/1516360.1516455
[23] 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.
[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594
[25] Dong Li, Qixu Zhang, Xiaochong Liang, Jida Guan, and Yang Xu. 2015. Selectivity estimation for string predicates based on modified pruned count-suffix tree. (2015), 76–82. https://doi.org/10.1049/cje.2015.01.013
[26] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation for Joins. In *Proc. SIGMOD*. 2121–2124. https://doi.org/10.1145/2882903.2899413
[27] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644
[28] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the International Conference on Learning Representations*, Yoshua Bengio and Yann LeCun (Eds.).
[29] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993. https://doi.org/10.14778/1687627.1687738
[30] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *PVLDB Endow.* 11, 9 (2018), 1016–1028. https://doi.org/10.14778/3213880.3213882
[31] Viswanath Poosala and Yannis E. Ioannidis. 1997. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proceedings og the VLDB Conf.* 486–495.
[32] PostgreSQL 14 Documentation. 2022. Chapter 72: How the Planner Uses Statistics. https://www.postgresql.org/docs/current/planner-stats-details.html
[33] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'79)*. 23–34. https://doi.org/10.1016/b978-0-934613-53-8.50038-8
[34] 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. https://doi.org/10.14778/3436905.3436907
[35] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. *Proceedings of the VLDB Endow.* 8, 12 (2015), 1530–1541. https://doi.org/10.14778/2824032.2824051
[36] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Xin Cao, Yifang Sun, Wei Wang, and Makoto Onizuka. 2020. Monotonic cardinality estimation of similarity selection: A deep learning approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. 1197–1212. https://doi.org/10.1145/3318464.3380570
[37] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. 1525–1539. https://doi.org/10.1145/3183713.3183739

Received July 2023; revised October 2023; accepted November 2023