Cardinality Estimation of Approximate Substring Queries using Deep Learning

Suyong Kwon
Electrical and Computer Engineering
Seoul National University
sykwon@kdd.snu.ac.kr

Woohwan Jung*
Computer Science and Engineering
Hanyang University
whjung@hanyang.ac.kr

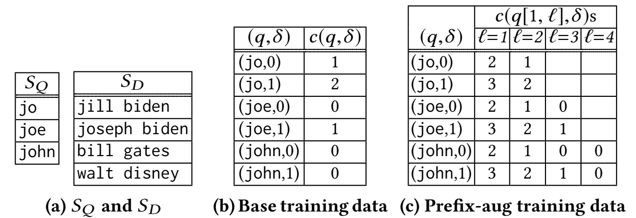
Kyuseok Shim
Electrical and Computer Engineering
Seoul National University
kshim@snu.ac.kr

ABSTRACT

Cardinality estimation of an approximate substring query is an important problem in database systems. Traditional approaches build a summary from the text data and estimate the cardinality using the summary with some statistical assumptions. Since deep learning models can learn underlying complex data patterns effectively, they have been successfully applied and shown to outperform traditional methods for cardinality estimations of queries in database systems. However, since they are not yet applied to approximate substring queries, we investigate a deep learning approach for cardinality estimation of such queries. Although the accuracy of deep learning models tends to improve as the train data size increases, producing a large train data is computationally expensive for cardinality estimation of approximate substring queries. Thus, we develop efficient train data generation algorithms by avoiding unnecessary computations and sharing common computations. We also propose a deep learning model as well as a novel learning method to quickly obtain an accurate deep learning-based estimator. Extensive experiments confirm the superiority of our data generation algorithms and deep learning model with the novel learning method.

PVLDB Reference Format:
Suyong Kwon, Woohwan Jung, and Kyuseok Shim. Cardinality Estimation of Approximate Substring Queries using Deep Learning. PVLDB, 15(11): 3145 - 3157, 2022.
doi:10.14778/3551793.3551859

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sykwon/teddy-dream.



S_Q
jo
joe
john

S_D
jill biden
joseph biden
bill gates
walt disney

(a) S_Q **and** S_D

(q,δ)	$c(q,\delta)$
(jo,0)	1
(jo,1)	2
(joe,0)	0
(joe,1)	1
(john,0)	0
(john,1)	0

(b) Base training data

(q,δ)	$c(q[1,\ell],\delta)$s			
	ℓ=1	ℓ=2	ℓ=3	ℓ=4
(jo,0)	2	1		
(jo,1)	3	2		
(joe,0)	2	1	0	
(joe,1)	3	2	1	
(john,0)	2	1	0	0
(john,1)	3	2	1	0

(c) Prefix-aug training data

Figure 1: Generation of training data ($\delta_M = 1$).

1 INTRODUCTION

Approximate substring search is a fundamental operation in many applications including substring matching [13], query refinement suggestion [24], data cleaning [34], named entity recognition [6], finding DNA subsequences [23] and spell checking [28]. With the explosive growth of data, there has been increasing demand for efficient support of approximate substring queries in database systems. The edit distance between strings is the minimum number of single character edits to transform one string to the other [9] and it is the most widely used distance measure for database applications. The substring edit distance between a query string q and a string s, denoted by $d_{sub}(q, s)$, is then defined as the smallest one among the edit distances between q and all substrings in s [13, 20, 27].

For a query string q, a threshold δ and a string set S_D, the approximate substring query, denoted by (q, δ), retrieves every string $s \in S_D$ with $d_{sub}(q, s) \le \delta$. Consider a query string set S_Q and a string set S_D in Figure 1(a). When q = `joe` in S_Q and $\delta = 1$, since only `joseph biden` in S_D has its substring edit distance with q at most δ, the cardinality of the query (q, δ), denoted by $c(q, \delta)$, is 1. Figure 1(b) shows $c(q, \delta)$ for every pair of $q \in S_Q$ and $\delta \in \{0, 1\}$.

In query optimization, accurate and efficient cardinality estimation is essential to produce an optimal query execution plan. Since many applications require search queries with string predicates, there have been extensive works on cardinality estimation of queries with string predicates [2, 12, 17, 19, 20, 26, 31]. Traditional approaches [12, 17, 20] generally build a *summary* from the text data which contains substrings/patterns with their frequencies. Since the summary can be large, they limit its size by storing only short and frequent substrings/patterns. Whenever a query string is not stored in the summary, it is decomposed into possibly overlapping substrings/patterns stored in the summary and its cardinality is estimated by combining their cardinalities with some *statistical assumptions* [12, 17, 20]. When such assumptions are violated, the accuracy of the estimated cardinality suffers.

Since deep learning models can reflect the underlying patterns and correlations of data, they are shown to outperform traditional methods for cardinality estimations of queries in database systems [15, 16, 26, 31, 35]. However, since they are not yet applied to approximate substring queries, we investigate a deep learning approach for cardinality estimation of such queries. Because the process of training a neural model involves providing the training data, we first describe the training data for cardinality estimation. Given a query string set S_Q and a string set S_D, the *base* training data consists of $(q, \delta, c(q, \delta))$s for every pair of $q \in S_Q$ and $0 \le \delta \le \delta_M$ where

*Corresponding author

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551859

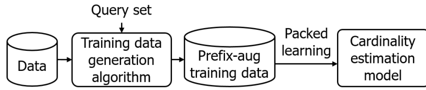


Figure 2: Training a cardinality estimation model.

δ_M is the maximum value of δ. Let $q[1, \ell]$ be the ℓ-length prefix of q. The *prefix-aug* training data has $(q, \delta, c(q[1, 1], \delta), c(q[1, 2], \delta), \ldots, c(q[1, |q|], \delta))$s for every pair of $q \in S_Q$ and $0 \leq \delta \leq \delta_M$. Figure 1(b) and Figure 1(c) are the *base* and *prefix-aug* training datasets for S_Q and S_D in Figure 1(a) with $\delta_M = 1$, respectively.

Since the accuracy of a deep learning model tends to improve as the train data size increases [3], we study how to efficiently generate a large train data. Any pair of a query string q and a data string s such that $d_{sub}(q, s) > \delta_M$ does not affect the cardinality $c(q, \delta)$. Thus, after proposing a lower-bound of $d_{sub}(q, s)$, we develop a dynamic programming algorithm to stop as soon as we notice that $d_{sub}(q, s) > \delta_M$ by using the lower-bound while computing $d_{sub}(q, s)$. On the other hand, if a query string p is a prefix of another query string q, the computation of $d_{sub}(p, s)$ can be shared with that of $d_{sub}(q, s)$ for the same string s. To efficiently generate the train data by using the lower-bound and sharing common computations, we propose the TEDDY (Trie-basED DYnamic programming) and the SODDY (SOrting-baseD DYnamic programming) algorithms.

The cardinality estimation of the queries with string predicates is addressed in [26, 31]. Astrid is proposed in [26] to estimate the cardinality of a substring query. To adapt Astrid to estimate the cardinalities of approximate substring queries, we need to build a model for each different distance threshold δ. However, we still cannot capture the underlying patterns and correlations across queries with different δs. For the cardinality estimation of approximate queries, CardNet [31] can be used to handle approximate *substring* queries. However, since CardNet does not share model parameters across the positions in a query string, it not only requires a large number of model parameters, but also ignores the relationships between the cardinalities of a query string and its prefixes. In addition, it cannot handle a query string longer than a certain length.

To overcome the drawbacks of existing deep learning models, we propose the DREAM (Deep caRdinality Estimation of ApproxiMate substring queries) model which treats a query string as a sequence of characters by adopting the long short-term memory (LSTM) [8]. It takes a query string q and a threshold δ as input, and estimates the cardinality of the query (q, δ). Furthermore, it outputs the estimated cardinalities of all the prefix queries $(q[1, t], \delta)$ together to utilize the *prefix-aug* training data. Using the *prefix-aug* training data improves the estimation accuracy of the model but has additional overheads to train the model with its prefix queries for each query. To reduce such overheads, we propose a novel learning method, called *packed learning*, which enables the model to efficiently learn the cardinality relationships among the prefixes of each query.

To the best of our knowledge, we are the first to consider a deep learning approach to estimate the cardinalities of approximate substring queries. We illustrate how to train a deep cardinality estimation model in Figure 2. After efficiently generating the *prefix-aug* training data, we train the DREAM model with the training data by the *packed learning* method. By conducting extensive experiments on real-life datasets, we show that our training data generation algorithms are much faster than existing algorithms and the DREAM model has the highest accuracy while its required space is the smallest among compared estimators. Moreover, the experimental results confirm that utilizing the *prefix-aug* training data and the *packed learning* for the DREAM model not only improves the estimation accuracy, but also reduces the training time.

2 RELATED WORK

Traditional methods for substring queries: The cardinality estimation of a substring query, which estimates the number of data strings in which a query string occurs as a substring, is studied in [4, 11, 12, 17]. KVI [17] uses a pruned suffix tree as a summary and applies the independence assumption to predict the cardinality by a greedy parsing of a query string without overlapping. MO [12] improves KVI by parsing a query string into possibly overlapping substrings and using the conditional independence assumption based on maximally overlapped substrings. Since both methods suffer from the underestimation problem, CRT [4] finds *short identifying substrings* of a query string with similar cardinalities to the query string and estimates the cardinality as the weighted geometric mean of the cardinalities of identifying substrings. On the other hand, SPH [2] predicts the cardinality of a LIKE query with a histogram built from the sequential patterns for the text column. LBS [20] is the state-of-the-art algorithm for cardinality estimation of approximate substring queries with the edit distance. It generalizes KVI [17] by exploiting an extended N-gram table which stores the cardinalities of possible string patterns of lengths up to N.

Deep learning models for (sub)string queries: The cardinality estimation of the queries with string predicates is addressed in [26] and [31]. The substring query with a string q returns every data string containing the string q. Astrid is proposed in [26] for substring queries (not approximate substring queries). It consists of an embedding learner and a cardinality estimator. The embedding learner expresses the embedding of a string by using the embeddings of q-grams of the string. After building the suffix tree from possible query strings, it annotates each node n with the cardinality of the substring query represented by n. Then, it pre-trains the embeddings of strings by using DeepWalk [25]. From the pre-trained embeddings of strings, Astrid trains the cardinality estimator with every pair of a query string and its cardinality. To adapt Astrid to approximate substring queries, for each distance threshold $\delta \leq \delta_M$, we build a trie from the query strings and training a single model by using the training data. For a substring query (q, δ), we use the output of the model trained for δ to estimate its cardinality. However, the adaptation of Astrid cannot capture the underlying patterns and correlations across queries with different δs.

CardNet [31] is proposed for the cardinality estimation of approximate queries (not substring queries) with distance measures including edit distance. Notice that an approximate query (q, δ) with the edit distance returns every data string s whose edit distance with q is at most δ. CardNet utilizes a single model regardless of distance threshold δ. In other words, its model is trained with training instances $(q, \delta, c(q, \delta))$ for $0 \leq \delta \leq \delta_M$. While CardNet can be used to solve approximate substring queries, it has several limitations. It is an encoder-decoder model where the encoder and

Table 1: List of notations.

Notation	Description
$s[i]$	the character at the i-th position of s
$s[i,j]$	the substring of s from position i to position j
$d(s_1, s_2)$	the edit distance of two strings s_1, s_2
$d_{sub}(q,s)$	the substring edit distance from q to s
$D_{q,s}$	the table for substring edit distance from q to s
S_Q	a query string set
S_D	a data string set
$c(q,\delta)$	the cardinality of a query (q,δ)
δ_M	the maximum possible distance threshold

the decoder use an FNN (fully-connected neural network). Since the FNN encoder does not share model parameters across the positions in a query string, CardNet not only requires a large number of model parameters but also ignores the relationships between the cardinalities of the query string and its prefixes. In addition, it limits the size of an input query string.

Processing approximate substring queries: By executing substring queries on a database, we can generate training data for cardinality estimation models of the database. There have been several works [6, 13, 21, 29, 30] related to processing approximate substring queries with the edit distance. The approximate entity extraction with the edit distance is investigated in [6, 21, 30]. We briefly describe only TASTE [6] since it is the best algorithm among these approaches. Given a threshold δ, after splitting the query string (resp., data string) into non-overlapping substrings, TASTE uses a trie generated the substrings in data strings to find similar data strings to a query string by a lower bound of the substring edit distance based on common substrings. The proposed algorithm in [13] finds the most k similar data strings with a given query string. Notice that all above methods do not share the computation of substring edit distances of a data string and the query strings with the same prefix. The TrieJoin algorithm is proposed in [29] for the string join with edit distance. While it utilizes the subtrie pruning based on edit distance, our TEDDY algorithm applies not only the subtrie pruning based on *substring* edit distance but also the required column pruning and the distance computation sharing.

3 PRELIMINARY

We present the definitions and the state-of-the-art [20] for cardinality estimation of an approximate substring query. We next provide a naive algorithm to generate the training data for neural models. The notations with their descriptions are summarized in Table 1.

3.1 Definitions

Let Σ be a finite alphabet of size $|\Sigma|$. For a string s of Σ^*, we represent the length of s by $|s|$. $s[i]$ with $1 \le i \le |s|$ indicates the character at the i-th position of s. We use $s[i,j]$ with $1 \le i \le j \le |s|$ to denote the substring of s which starts from position i and ends at position j. The empty string is denoted by ϵ.

Substring edit distance: Let $d(s_1, s_2)$ be the edit distance between two strings s_1 and s_2. The substring edit distance $d_{sub}(q,s)$ is the smallest one among the edit distances between q and all substrings in s. That is, $d_{sub}(q,s) = \min_{1 \le i \le j \le |s|} d(q, s[i,j])$. We present the

$S_Q \bowtie_{d_{sub}(q,s) \le \delta_M} S_D$		$d_{sub}(q,s)$	$c(q,\delta)$s	
S_Q	S_D		$c(q,0)$	$c(q,1)$
jo	jill biden	1	X	O
jo	joseph biden	0	O	O
joe	joseph biden	1	X	O

(a) Computing $c(q,\delta)$s

S_Q	$c(q,0)$	$c(q,1)$
jo	1	2
joe	0	1
john	0	0

(b) Count table C

Figure 3: Generation of the count table ($\delta_M = 1$).

dynamic programming algorithm with $O(|q| \cdot |s|)$ time in [27] to compute $d_{sub}(q,s)$. For a query string q and a string s, let us define $D[i,j]$ as the minimum of the edit distance between $q[1,i]$ and every suffix $s[k,j]$ of $s[1,j]$. In other words, we have

$$D[i,j] = \min_{1 \le k \le j} d(q[1,i], s[k,j]) \quad (1)$$

Since the empty string is a substring of any string, $D[0,j] = 0$ for $0 \le j \le |s|$. Furthermore, since we need at least $|q|$ edit (i.e., delete) operations to transform a query string q to ϵ, $D[i,0] = i$ for $0 \le i \le |q|$. In addition, the optimal substructure of the substring edit distance problem gives the following recursive formula

$$D[i,j] = \begin{cases} D[i\text{-}1, j\text{-}1] & \text{if } q[i] = s[j] \\ \min(D[i\text{-}1, j], D[i, j\text{-}1], D[i\text{-}1, j\text{-}1]) + 1 & \text{if } q[i] \ne s[j]. \end{cases} \quad (2)$$

The substring edit distance $d_{sub}(q,s)$ is then computed by

$$d_{sub}(q,s) = \min_{1 \le j \le |s|} \min_{1 \le k \le j} d(q, s[k,j]) = \min_{1 \le j \le |s|} D[|q|, j]. \quad (3)$$

3.2 LBS: The State-of-the-Art Algorithm

LBS [20] is the state-of-the-art algorithm for estimating the cardinality of an approximate substring query. A string b is called a base string of a query (q,δ) if b can be transformed from the query string q, with at most δ edit operations (i.e., $d(b,q) \le \delta$), by modeling the insertion and substitution operations with the wildcard symbol ?. For example, joe? and jo? are base strings of a query (joe, 1).

The LBS algorithm exploits an extended N-gram table as the *summary structure* to store the cardinalities of possible base strings with lengths up to N. It estimates the cardinality of a query (q,δ) based on the cardinalities of the minimal base strings of q. A base string b of the query is minimal if there does not exist any other base string of the query which is a proper substring of b. Since the base strings with lengths larger than N do not exist in the extended N-gram table, the LBS algorithm approximates their cardinalities by using MO [12] and set hashing techniques [5]. While it precisely estimates the cardinalities of queries with short query strings, it may suffer from low estimation accuracy for a query (q,δ) with $|q| > N$. Moreover, since the number of possible substrings appearing in S_D is generally much larger than that of strings in S_D, a large extended N-gram table should be maintained as a *summary structure.*

3.3 Training Data for Deep Learning Models

The *base* training data generated from S_Q and S_D consists of the instances $(q, \delta, c(q,\delta))$ for every pair of $q \in S_Q$ and $0 \le \delta \le \delta_M$. After generating the count table with the columns S_Q, $c(q,0), \ldots,$ $c(q, \delta_M\text{-}1)$ and $c(q,\delta_M)$, for every tuple of the count table, we add $(q, \delta, c(q,\delta))$ with $0 \le \delta \le \delta_M$ to the *base* training data.

The NaiveGen algorithm: We create the count table $C[1..|S_Q|, 0..|\delta_M|]$ where $C[q, \delta]$ represents the number of the matching data strings for a query (q, δ). We next initialize the table with zeroes. Then, we compute $S_Q \bowtie_{d_{sub}(q,s) \le \delta_M} S_D$ by a nested loop join algorithm. For every data string $s \in S_D$, we examine each query string $q \in S_Q$, compute $d_{sub}(q, s)$ and increase $C[q, \delta]$ by one with $d_{sub}(q, s) \le \delta \le \delta_M$. To produce the *base* training data, we add the instances $(q, \delta, c(q, \delta))$ for every tuple of the count table. We refer to this algorithm as the NaiveGen algorithm. Let S_P be the set of all distinct prefixes appearing in S_Q. To produce the *prefix-aug* training data, we use $C[1..|S_P|, 0..\delta_M]$ as the count table and add the instances $(q, \delta, c(q[1, 1], \delta), \ldots, c(q[1, |q|], \delta))$ for every pair of $q \in S_Q$ and $0 \le \delta \le \delta_M$. Let L_{S_Q} and L_{S_D} be the longest strings in S_Q and S_D, respectively. Since computing $d_{sub}(q, s)$ requires $O(|q| \cdot |s|)$ time, it takes $O(L_{S_Q} \cdot |S_Q| \cdot L_{S_D} \cdot |S_D|)$ and $O(L_{S_Q} \cdot |S_P| \cdot L_{S_D} \cdot |S_D|)$ times to produce the *base* and *prefix-aug* training datasets, respectively.

Assume that $\delta_M = 1$. For S_Q and S_D in Figure 1(a), we show the tuples of $S_Q \bowtie_{d_{sub}(q,s) \le \delta_M} S_D$ and their $d_{sub}(q, s)$s in Figure 3(a). Each line also shows whether each column $c(q, \delta)$ is updated (by marking O) or un-updated (by marking X) based on $d_{sub}(q, s)$. Figure 3(b) shows the count table obtained from the table in Figure 3(a). The *base* training data in Figure 1(b) is produced by adding $(q, \delta, c(q, \delta))$ for every tuple of the count table in Figure 3(b). Notice that the *prefix-aug* training data from S_Q and S_D is shown in Figure 1(c).

4 GENERATING TRAINING DATA

We propose the training data generation algorithms that not only avoid the computation of unnecessary entries in D to compute substring edit distances, but also share common distance computations by utilizing the prefix relations across query strings.

4.1 Computation of the Table D

For a pair of a query string q and a data string s, if $d_{sub}(q, s) > \delta_M$, the count $c(q, \delta)$ is not affected and we do not need to compute $d_{sub}(q, s)$. Thus, we present a dynamic programming algorithm that stops as soon as we know that $d_{sub}(q, s) > \delta_M$ by using a lower-bound of $d_{sub}(q, s)$, while computing the table D to get $d_{sub}(q, s)$.

The lower bound of $D[i, j]$: To show that $D[i\text{-}1, j\text{-}1]$ is a lower bound of $D[i, j]$, we provide the following lemma.

Lemma 4.1. *In the table D computed by Equation 2 to get $d_{sub}(q, s)$ for a query string q and a string s, the following statements hold.*

(1) $D[i, j] \ge D[i, j\text{-}1]\text{-}1$ for $0 \le i \le |q|, 1 \le j \le |s|$.
(2) $D[i, j] \ge D[i\text{-}1, j]\text{-}1$ for $1 \le i \le |q|, 0 \le j \le |s|$.
(3) $D[i, j] \ge D[i\text{-}1, j\text{-}1]$ for $1 \le i \le |q|, 1 \le j \le |s|$.

Proof. (1) We denote the concatenation of a string s and a character c by $s \circ c$. For a pair of strings x and y with a character c, we have $d(x, y) \le d(x, y \circ c) + d(y \circ c, y) = d(x, y \circ c) + 1$ by the triangular inequality [1]. Thus, we have $d(q[1, i], s[k, j\text{-}1])\text{-}1 \le d(q[1, i], s[k, j])$ for $1 \le k \le j$. By using this inequality, we get

$$D[i, j] = \min_{1 \le k \le j} d(q[1, i], s[k, j]) \ge \min_{1 \le k \le j} (d(q[1, i], s[k, j\text{-}1])\text{-}1) = D[i, j\text{-}1]\text{-}1.$$

(2) We omit the proof since it is similar to that of Statement (1).

(3) Since we compute $D[i, j]$ by Equation 2, we split the proof into two cases depending on whether $q[i] = s[j]$ or not.

(a) When $q[i] = s[j]$, since $D[i, j] = D[i\text{-}1, j\text{-}1]$ by Equation 2, $D[i, j] \ge D[i\text{-}1, j\text{-}1]$ trivially holds.
(b) When $q[i] \ne s[j]$, we can derive

$$\begin{aligned} D[i, j] &= \min(D[i\text{-}1, j\text{-}1], D[i\text{-}1, j], D[i, j\text{-}1]) + 1 \text{ (by Equation 2)} \\ &\ge \min(D[i\text{-}1, j\text{-}1], D[i\text{-}1, j\text{-}1]\text{-}1, D[i\text{-}1, j\text{-}1]\text{-}1) + 1 \\ &\quad \text{(by using both statements (1) and (2) in this lemma)} \\ &\ge D[i\text{-}1, j\text{-}1]. \end{aligned}$$

■

We next show that $D_{q,s}[i\text{-}1, j\text{-}1]$ is the tightest lower bound of $D_{q,s}[i, j]$ among the three lower bounds derived in Lemma 4.1.

Lemma 4.2. *For the three lower bounds of $D[i, j]$ obtained by Lemma 4.1, we have $D[i\text{-}1, j\text{-}1] \ge \max(D[i\text{-}1, j]\text{-}1, D[i, j\text{-}1]\text{-}1)$.*

We omit the proof of the lemma. Please refer to the extended version [18] of this paper. By Lemma 4.2, if $D[i\text{-}1, j\text{-}1] > \delta_M$, we have $D[i, j] > \delta_M$.

The required column interval: For every row i in the table D, we define the *required column interval* $[J_s(i), J_e(i)]$ in which every column index j such that $D[i, j] \le \delta_M$ is always located.

Definition 4.3 (Required column interval). We define the required column interval of row i in a table D by

$$[J_s(i), J_e(i)] = \begin{cases} [1, |s|] & \text{if } i \le \delta_M + 1, \\ [\min(S(i) \cup \{\infty\}), \max(S(i) \cup \{-\infty\})] & \text{if } i > \delta_M + 1. \end{cases} \quad (4)$$
$$\text{where } S(i) = \{j \mid D[i\text{-}1, j\text{-}1] \le \delta_M, j\text{-}1 \in [J_s(i\text{-}1), J_e(i\text{-}1)]\}$$

Since $S(i)$ in Equation 4 is the set of the indexes j such that $D[i\text{-}1, j\text{-}1] \le \delta_M$, if $D[i, j] \le \delta_M$, j should be in $S(i)$ by Lemma 4.1. Thus, $[J_s(i), J_e(i)]$ contains every column index j such that $D[i, j] \le \delta_M$. When $S(i) = \{\}$, since $[J_s(i), J_e(i)]$ is empty, we set $[J_s(i), J_e(i)]$ to $[\infty, -\infty]$ to indicate that $[J_s(i), J_e(i)]$ is empty. The following lemmas show the properties of the required column interval.

Lemma 4.4. *At row i of the table D computed by Equation 2, for every entry $D[i, j]$ such that $D[i, j] \le \delta_M$, the column index j is always located in the required column interval $[J_s(i), J_e(i)]$.*

Proof. We prove this lemma by induction on i.

(1) For base cases (i.e., $i \le \delta_M + 1$), we have $[J_s(i), J_e(i)] = [1, |s|]$ by Definition 4.3. Since we need to fill only the entries $D[i, j]$ for $1 \le i \le |q|$ and $1 \le j \le |s|$, any index j such that $D[i, j] \le \delta_M$ should be located in $[1, |s|] = [J_s(i), J_e(i)]$.

(2) When $k > \delta_M + 1$, assume inductively that this lemma holds for $i < k$. We shall then prove that this lemma holds for $i = k$. For every index j such that $D[i, j] \le \delta_M$, we have $D[i\text{-}1, j\text{-}1] \le \delta_M$ by Lemma 4.1. In addition, by the inductive hypothesis, we have $j\text{-}1 \in [J_s(i\text{-}1), J_e(i\text{-}1)]$. Thus, we have $j \in S(i)$ by combining both conditions. By this observation, if there exists a column index j such that $D[i, j] \le \delta_M$, we have $S(i) \ne \{\}$. Since $[J_s(i), J_e(i)] \ne [\infty, -\infty]$, we get $[J_s(i), J_e(i)] = [\min(S(i)), \max(S(i))]$ by Definition 4.3. Thus, we have $j \in [J_s(i), J_e(i)]$. When there does not exist any column index j such that $D[i, j] \le \delta_M$, we have $S(i) = \{\}$. Since $[J_s(i), J_e(i)] = [\infty, -\infty]$ by Definition 4.3, this lemma is trivially satisfied. ■

Lemma 4.5. *In the table D computed by Equation 2, $[J_s(i)\text{-}1, J_e(i)\text{-}1]$ is contained by $[J_s(i\text{-}1), J_e(i\text{-}1)]$ for $(\delta_M + 1) < i \le |q|$.*

Proof. Since $D[i\text{-}1, J_s(i)\text{-}1] \le \delta_M$ and $D[i\text{-}1, J_e(i)\text{-}1] \le \delta_M$ by Definition 4.3, and $[J_s(i\text{-}1), J_e(i\text{-}1)]$ contains every column index j such that $D[i\text{-}1, j] \le \delta_M$ by Lemma 4.4, this lemma holds. ■

The required column pruning: By Lemma 4.4, if we compute the entries $D[i, j]$ in each row i with every $j \in [J_s(i), J_e(i)]$, we find all entries $D[i, j]$ such that $D[i, j] \le \delta_M$. Note that $d_{sub}(q, s)$ is the minimum value in row $|q|$ by Equation 3. If $d_{sub}(q, s) \le \delta_M$, after computing row $|q|$ of the table D, we compute $d_{sub}(q, s)$ by

$$d_{sub}(q, s) = \min_{J_s(|q|) \le j \le J_e(|q|)} D[|q|, j]. \quad (5)$$

Otherwise (i.e., $d_{sub}(q, s) > \delta_M$), we stop computing the entries in each row i as soon as we know that $[J_s(i), J_e(i)]$ is empty. We refer to this method as the *required column pruning*. Once the required column interval $[J_s(i), J_e(i)]$ becomes empty, $[J_s(i'), J_e(i')]$ should be empty too for every row i' such that $i < i' \le |q|$ by Lemma 4.5.

When we compute $D[i, j]$, since we already computed $D[i\text{-}1, j]$ for every $j \in [J_s(i\text{-}1), J_e(i\text{-}1)]$ and $[J_s(i\text{-}1), J_e(i\text{-}1)]$ contains $[J_s(i)\text{-}1, J_e(i)\text{-}1]$ by Lemma 4.5, we obtain the following corollary.

Corollary 4.6. *If we compute $D[i, j]$ with every $j \in [J_s(i), J_e(i)]$ only at each row i (in row major order), when computing $D[i, j]$s at row i, $D[i\text{-}1, j]$ is already computed for every $j \in [J_s(i)\text{-}1, J_e(i)\text{-}1]$.*

Computing the table D: If $d_{sub}(q, s) > \delta_M$, since the count $c(q, \delta)$ is not affected, we output ∞ as the value of $d_{sub}(q, s)$. On the other hand, if $d_{sub}(q, s) \le \delta_M$, we output the exact value of $d_{sub}(q, s)$. To do so, for any $D[i, j]$ such that $D[i, j] > \delta_M$, we set $D[i, j] = \infty$.

We next show how to compute $D[i, j]$ with $D[i\text{-}1, j\text{-}1]$, $D[i\text{-}1, j]$ and $D[i, j\text{-}1]$ based on Equation 2.

(1) When $i \le \delta_M + 1$: Since $[J_s(i), J_e(i)] = [1, |s|]$ (i.e., all column indexes) by Definition 4.3, we already have the values of $D[i\text{-}1, j\text{-}1]$, $D[i\text{-}1, j]$ and $D[i, j\text{-}1]$. Thus, we compute $D[i, j]$ by Equation 2.

(2) When $i > \delta_M + 1$: We split into three cases depending on j.

(2-1) When $j = J_s(i)$: By Corollary 4.6, $D[i\text{-}1, J_s(i)\text{-}1]$ and $D[i\text{-}1, J_s(i)]$ are already computed. However, we have not computed $D[i, J_s(i)\text{-}1]$ (by the required column pruning). Since $D[i, J_s(i)\text{-}1] > \delta_M$, we just set $D[i, J_s(i)\text{-}1] = \infty$ and compute $D[i, j]$ by Equation 2.

(2-2) When $j = J_e(i)$: We already have $D[i, J_e(i)\text{-}1]$. We also have $D[i\text{-}1, J_e(i)\text{-}1]$ by Corollary 4.6. Since $J_e(i) \le J_e(i\text{-}1)+1$ by Lemma 4.5, if $J_e(i) \le J_e(i\text{-}1)$, we have already computed $D[i\text{-}1, J_e(i)]$. Thus, we compute $D[i, j]$ by Equation 2. Otherwise (i.e., $J_e(i) = J_e(i\text{-}1) + 1$), $D[i\text{-}1, J_e(i)]$ is not computed and we know that $D[i\text{-}1, J_e(i)] > \delta_M$. Thus, we set $D[i\text{-}1, J_e(i\text{-}1)+1] = \infty$ and compute $D[i, j]$ by Equation 2.

(2-3) When $J_s(i) < j < J_e(i)$: Since $D[i, j\text{-}1]$ has just been computed as well as both $D[i\text{-}1, j\text{-}1]$ and $D[i\text{-}1, j]$ are already computed by Corollary 4.6, we can compute $D[i, j]$ by Equation 2.

Example 4.7. Consider a query string q = `joe biden` and a string s = `joeseph (joe) biden` with $\delta_M = 1$. Figure 4 shows the entries of D computed by considering the required intervals represented by the shaded areas. When $i \le \delta_M + 1$, since the required column intervals are $[1, |s|]$ by Definition 4.3, we compute every entry of row 1 and 2. Note that for any entry larger than δ_M, we set the entry to ∞. Consider the case when $i > \delta_M + 1$. Since $[J_s(3), J_e(3)] = [2, 12]$ and $[J_s(4), J_e(4)] = [3, 13]$ by Definition 4.3, we know that the entries $D[3, 1]$, $D[3, 13]$, $D[4, 2]$ and $D[4, 14]$ are larger than δ_M. Thus, we fill those entries with ∞, and compute the entries in $D[3, 2..12]$ and $D[4, 3..13]$ by Equation 2. After computing the entries in the remaining rows, we compute $d_{sub}(q, s)$ by Equation 5 from the entries in row 9. By the column pruning technique, we compute only 79 entries out of 162 entries in D.

$\delta_M = 1$, ▢ = required column interval

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
i	q \ s	ϵ	j	o	s	e	p	h		(	j	o	e	)		b	i	d	e	n
0	ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	j	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
2	o	2	1	0	1	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞
3	e	3	∞	1	1	1	∞	∞	∞	∞	∞	1	0	1	∞					
4		4		∞	∞	∞	∞	∞	∞	∞	∞	∞	1	1	1	∞				
5	b	5											∞	∞	∞	1	∞			
6	i	6														∞	1	∞		
7	d	7															∞	1	∞	
8	e	8																∞	1	∞
9	n	9																	∞	1

Figure 4: The required column intervals of D.

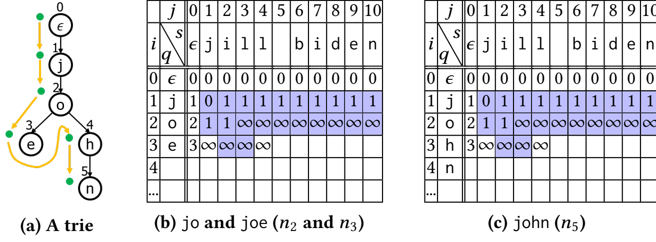
4.2 Sharing Distance Computations

For a query string q and a string s, let $D_{q,s}[i, j]$ be the minimum of edit distances between $q[1, i]$ and all suffixes $s[k, j]$ of $s[1, j]$. If p is a prefix of a query string q, $D_{p,s}$ is a subtable of $D_{q,s}$ (i.e., $D_{q,s}[i, j] = D_{p,s}[i, j]$ for $1 \le i \le |p|$ and $1 \le j \le |s|$). As a result, if we compute $d_{sub}(q, s)$ right after computing $d_{sub}(p, s)$, from the row $|p|$ of $D_{p,s}$, we can compute the rows $(|p|+1)$, $(|p|+2)$, …, $|q|$ in the table $D_{q,s}$. Moreover, for a string s, if p is a common prefix of several query strings in S_Q, we can reuse $D_{p,s}$ to compute $D_{q,s}$ for those query strings q. Thus, for a string s, to compute $D_{q,s}$ with every $q \in S_Q$, we create and use a single global table $D[0..L_{S_Q}, 0..L_{S_D}]$ where L_{S_Q} and L_{S_D} are the maximum lengths of a query string in S_Q and a string in S_D, respectively.

Query ordering scheme: Assume that the elements in the query string set $S_Q = \{q_1, q_2, \ldots, q_{|S_Q|}\}$ are ordered alphabetically. Then, for each string $s \in S_D$, if we compute $d_{sub}(q_\ell, s)$ one by one for $1 \le \ell \le |S_Q|$, we can maximize sharing of computations across all query strings. Let LCP($q_{(\ell\text{-}1)}$, q_ℓ) be the longest common prefix of $q_{(\ell\text{-}1)}$ and q_ℓ. When computing $d_{sub}(q_\ell, s)$, since $D_{q_{\ell-1},s}$ is already stored in the table D, after filling only the entries in D from row $(|\text{LCP}(q_{(\ell\text{-}1)}, q_\ell)| + 1)$ to row $|q_\ell|$, we can get $d_{sub}(q_\ell, s)$ from row $|q_\ell|$ in the table D by Equation 3. We refer to this approach as the *query ordering scheme* and next present its computational overhead.

Lemma 4.8. *Given a string s in S_D, the query ordering scheme computes a single row $|p|$ in the table D for each distinct prefix p of the query strings in S_Q.*

Proof. To compute $d_{sub}(q, s)$ between a string s and every query string q in S_Q by dynamic programming, we have to in fact compute $D_{p,s}$ for all distinct prefixes p appearing in S_Q. For any such a distinct prefix p, to compute $D_{p,s}$, we can compute only row $|p|$ in table $D_{p,s}$ by reusing row $(|p|\text{-}1)$ in the table $D_{p[1..|p|-1],s}$ Furthermore, $D_{p,s}$ remains in the global table D until we examine the last query string with the prefix p in alphabetical order since the query strings with the prefix p always appear contiguously in alphabetical order [7]. Thus, we compute row $|p|$ in table D exactly once for every prefix p appearing in S_Q. ■



	j	0	1	2	3	4	5	6	7	8	9	10
i	$q \backslash s$	ϵ	j	i	l	l		b	i	d	e	n
0	ϵ	0	0	0	0	0	0	0	0	0	0	0
1	j	1	0	1	1	1	1	1	1	1	1	1
2	o	2	1	1	∞	∞	∞	∞	∞	∞	∞	∞
3	e	3	∞	∞	∞	∞						
4												
...												

	j	0	1	2	3	4	5	6	7	8	9	10
i	$q \backslash s$	ϵ	j	i	l	l		b	i	d	e	n
0	ϵ	0	0	0	0	0	0	0	0	0	0	0
1	j	1	0	1	1	1	1	1	1	1	1	1
2	o	2	1	1	∞	∞	∞	∞	∞	∞	∞	∞
3	h	3	∞	∞	∞	∞						
4	n											
...												

(a) A trie **(b)** jo **and** joe **(n_2 and n_3)** **(c)** john **(n_5)**

Figure 5: The global table D while examining query strings.

Reusing required column intervals: Let $p = \text{LCP}(q_{\ell-1}, q_\ell)$. Since the rows $|p|$ of $D_{q_{\ell-1},s}$ and $D_{q_\ell,s}$ are the same as well as the required column interval of row $(|p|+1)$ is determined by row $|p|$ by Definition 4.3, the rows $(|p|+1)$ in $D_{q_{\ell-1},s}$ and $D_{q_\ell,s}$ have identical required column intervals. Thus, we reuse the intermediate computations $D_{p,s}$ and the required column interval $[J_s(|p|+1), J_e(|p|+1)]$ to compute $D_{q,s}$ for all query strings q with the common prefix p.

To efficiently generate a training data by the query ordering scheme, we propose the TEDDY (Trie-basED DYnamic programming) algorithm and the SODDY (SOrting-baseD DYnamic programming) algorithm. Since both algorithms are essentially the same except the use of a sorted list and a trie, we next present the TEDDY algorithm only. The details of the SODDY algorithm including its pseudocode can be found in the extended version [18] of the paper.

4.3 The TEDDY Algorithm

We build a trie from query strings in S_Q and next traverse the trie in preorder to generate a training data.

A trie: It is a tree structure where each path from the root node to a node n_i represents a string $n_i.q$ and every node n_j on the path has a label of a character, denoted by $n_j.c$, in the string. The root node n_0 of a trie is annotated with a special empty character ϵ and the child nodes of a node n_i are alphabetically ordered by their labels.

We build a trie in which every distinct prefix of the query strings corresponds to a unique node in the trie. For example, a trie generated from the three query strings of S_Q in Figure 1(a) is shown in Figure 5(a). The character within the circle at each node is the annotated character at that node. The number above a node is the node id. The shaded arrows represent the order of the nodes visited in the preorder traversal. Note that by traversing the trie in preorder, query strings are visited in the order of jo, joe and john.

Key idea: To produce the *base* training data, we traverse the trie in preorder for each string $s \in S_D$. To store the cardinality $c(q_\ell, \delta)$ for every pair of $q_\ell \in S_Q$ and $0 \le \delta \le \delta_M$, we create the count table $C[1..|S_Q|, 0..\delta_M]$ and initialize its entries with zero. While visiting a node n, if $[J_s(|n.q|+1), J_e(|n.q|+1)]$ is empty, we skip calculating the remaining rows in $D_{q,s}$ for all query strings q represented by the descendants of n since they have $n.q$ as a prefix. Otherwise, since the string represented by n's parent node is a prefix of $n.q$, we simply compute row $|n.q|$ from row $(|n.q|$-1) in the table D. If $n.q$ represents a query string $q_\ell \in S_Q$, we also compute $d_{sub}(q_\ell, s)$ and increase the count table $C[q_\ell, \delta]$ for $d_{sub}(q_\ell, s) \le \delta \le \delta_M$.

Procedure TEDDY(S_Q, S_D, δ_M)
Input: A query string set S_Q, a string set S_D and a maximum threshold δ_M
Output: The training data
begin
1. $L_{S_Q} = \max_{q \in S_Q} |q|$
2. $L_{S_D} = \max_{s \in S_D} |s|$
3. Create $D[0..L_{S_Q}, 0..L_{S_D}]$ and initialize the entries with zero
4. Create $C[1..|S_Q|, 0..\delta_M]$ and initialize the entries with zeros
5. Create the table $J[1..L_{S_Q}]$
6. $T = genTrie(S_Q)$
7. $S = createEmptyStack()$
8. **for each** string $s \in S_D$
9. $J[1].s = 1$
10. $J[1].e = |s|$
11. **for each** child n_c of $T.root$
12. $PUSH(S, n_c)$
13. **while** S is not empty
14. $n = POP(S)$
15. $i = |n.q|$
16. **if** $i > \delta_M+1$ **and** $J[i].s > 0$
17. $D[i, J[i].s\text{-}1] = \infty$
18. **if** $i > \delta_M+1$ **and** $J[i].e\text{-}1 = J[i\text{-}1].e$
19. $D[i\text{-}1, J[i].e] = \infty$
20. Compute row i of D in $[J[i].s, J[i].e]$ by Equation 2
21. Compute $[J_s(i+1), J_e(i+1)]$ by Definition 4.3
22. $J[i+1].s = J_s(i+1)$
23. $J[i+1].e = J_e(i+1)$
24. **if** $n.q$ is a query string $q_\ell \in S_Q$ // row $|q_\ell|$ is computed
25. $dist = \min_{J_s[i] \le j \le J_e[i]} D[i, j]$ // i.e., $d_{sub}(n.q, s)$
26. **for** $\delta = dist$ **to** δ_M
27. $C[q_\ell, \delta] = C[q_\ell, \delta]+1$
28. **if** $[J[i+1].s, J[i+1].e]$ is not empty
29. **for each** child n_c of n
30. $PUSH(S, n_c)$
31. Open training data file
32. **for each** query string $q_\ell \in S_Q$
33. **for** δ from 0 to δ_M
34. Output $(q_\ell, \delta, C[q_\ell, \delta])$

end

Figure 6: The TEDDY algorithm.

The pseudocode: It is presented in Figure 6. We use a table D to store the table $D_{q,s}$ and the count table C (lines 3-4). To store the required column intervals, we also use the table J (line 5). We build a trie from the query strings in S_Q by invoking the *genTrie* procedure and initialize an empty stack S by invoking *createEmptyStack* (lines 6-7). The stack S keeps the nodes to visit next in the preorder traversal. For each string s in S_D, the TEDDY algorithm traverses the trie in preorder to compute $d_{sub}(q, s)$ for every query string q in S_Q. We first initialize $[J_s(1), J_e(1)] = [1, |s|]$ and add the child nodes of the root node to the stack S (lines 9-12). In each iteration of the while loop (lines 13-30), we pop the node n to visit next from the stack S (line 14). To compute $D[i, J_s(i)]$ and $D[i, J_e(i)]$ correctly, as we discussed previously in Section 4.1, we let $D[i, J_s(i)\text{-}1] = D[i\text{-}1, J_e(i)] = \infty$ (lines 16-19). We compute the entries $D[i, j]$ in row i of the table D for every $j \in [J_s(i), J_e(i)]$ (line 20). After computing $[J_s(i+1), J_e(i+1)]$ from row i in the table D based on Definition 4.3, we let $J[i+1].s = J_s(i+1)$ and $J[i+1].e = J_e(i+1)$ (lines 21-23). If the node n represents a query string $q_\ell \in S_Q$, we set *dist* to the value of $d_{sub}(q_\ell, s)$ computed by Equation 5, and increase $C[q_\ell, \delta]$ by one for every $dist \le \delta \le \delta_M$ (lines 24-27). If the required column interval $[J_s(i+1), J_e(i+1)]$ is empty, since $d_{sub}(q, s) > \delta_M$ for every query string q represented by the descendants of n, we do not add the child nodes of n to the stack S. Otherwise, to traverse the descendants of n, we push every child node of n to S (lines 28-30). After we finish

the outer for-loop (lines 8-30), for every query string $q_\ell \in S_Q$, we generate the training data by writing $(q_\ell, \delta, C[q_\ell, \delta])$ to the output file with $0 \le \delta \le \delta_M$.

Example 4.9. Consider S_Q and S_D in Figure 1(a) with $\delta_M = 1$. The TEDDY algorithm first builds a trie T in Figure 5(a). We initialize the count table $C[1..3, 0..1]$ with zeroes. We next create an array J and an empty stack S. Then, we add a child node n_1 of the root node n_0 to S. For a string s = `jill biden`, we traverse the trie in preorder to compute $d_{sub}(q, s)$ for every query string $q \in S_Q$. While traversing n_1 and n_2, since $[J_s(1), J_e(1)] = [J_s(2), J_e(2)] = [1, |s|]$ by Definition 4.3, we compute every entry in both rows 1 and 2 of the table D. Since n_2 represents the first query string `jo`, we compute d_{sub}(`jo,jill biden`) by Equation 5. Because d_{sub}(`jo,jill biden`) = 1, we increase C[`jo`, 1] by one. Among the entries in row 2, since only $D[2, 1]$ and $D[2, 2]$ are at most δ_M, we compute $[J[3].s, J[3].e] = [2, 3]$ by Definition 4.3. Then, we let $J[3].s = 2$ and $J[3].e = 3$. Since $[J[3].s, J[3].e]$ is not empty, we push n_3 and n_4 into the stack S to visit nodes in the preorder traversal. When we next visit n_3, since every entry of row 2 in the table D and $[J[3].s, J[3].e]$(=[2, 3]) are previously computed and stored, we reuse them. Since $[J[3].s, J[3].e] = [2, 3]$, we initialize $D[3, 1] = \infty$ and fill the entries $D[3, 2]$ and $D[3, 3]$ by Equation 2, as shown Figure 5(b). For the query string `joe`, since $D[3, 2] > \delta_M$ and $D[3, 3] > \delta_M$, we conclude that d_{sub}(`joe, jill biden`) $> \delta_M$ and do not update the table C. Finally, we visit n_4 and next n_5, which represents the query string `john`, and compute the table $D_{\texttt{john,jill biden}}$. After filling the entries $D[3, 2]$ and $D[3, 3]$ by Equation 2, since all entries $D[3, j]$ are larger than δ_M with $j \in [J_s(3), J_e(3)]$, we immediately stop the rest of computing $D_{\texttt{john,jill biden}}$. Similarly, by traversing the trie for the remaining data strings, we obtain C in Figure 3(b). The training data generated by adding the training instance $(q, \delta, c(q, \delta))$ from every tuple of the count table is shown in Figure 1(b).

Generating the prefix-aug training data: We can modify the TEDDY algorithm to output $(q_\ell, \delta, c(q_\ell[1, 1], \delta), \ldots, c(q_\ell[1, |q_\ell|], \delta))$ for every pair of $q_\ell \in S_Q$ and $0 \le \delta \le \delta_M$ as follows. Let S_P be the set of all distinct prefixes in S_Q. We use $C[1..|S_P|, 0..\delta_M]$ instead of $C[1..|S_Q|, 0..\delta_M]$ as the count table in line 4 and add the instances $(q, \delta, c(q[1, 1], \delta), \ldots, c(q[1, |q|], \delta))$ for every pair of $q \in S_Q$ and $0 \le \delta \le \delta_M$. Furthermore, we have to compute $d_{sub}(p, s)$ for every distinct prefix $p \in S_P$. Thus, we remove the line 24, which enforces to execute the lines 25-27 to calculate $d_{sub}(q_\ell, s)$ only for $q_\ell \in S_Q$, in the while-loop and replace $(q_\ell, \delta, C[q_\ell, \delta])$ by $(q_\ell, \delta, C[q_\ell[1, 1], \delta], \ldots, C[q_\ell[1, |q_\ell|], \delta]))$ in line 34.

Time complexity: For each string $s \in S_D$, the query ordering scheme accesses query strings in S_Q once. As we discussed previously, since we consider each distinct prefix p in S_P (i.e., the set of all distinct prefixes appearing in S_Q), it takes at most $O(|s|)$ time to compute row $|p|$ in the table D. Note that sorting the query strings takes $O(|S_Q| \cdot \log |S_Q| \cdot L_{S_Q})$ time while building a trie takes $O(|S_P|)$ time. Since examining query strings once by either scanning the sorted query strings or traversing the trie built from query strings takes $O(|s| \cdot |S_P|)$ time, both TEDDY and SODDY algorithms require $O(|S_P| \cdot L_{S_D} \cdot |S_D|)$ time to compute $D_{q,s}$ for every pair of a query string q in S_Q and a data string s in S_D. Note that their time complexities are the same regardless of generating *base* training data or *prefix-aug* training data. Recall that the NaiveGen algorithm takes $O(L_{S_Q} \cdot |S_Q| \cdot L_{S_D} \cdot |S_D|)$ and $O(L_{S_Q} \cdot |S_P| \cdot L_{S_D} \cdot |S_D|)$ times to produce the *base* and *prefix-aug* training datasets, respectively.

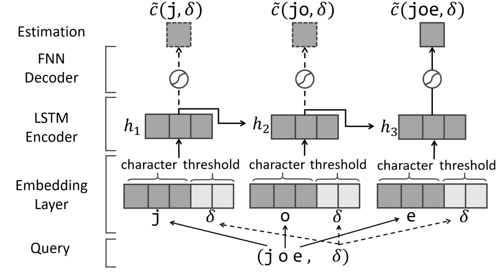


Figure 7: The architecture of DREAM model.

5 NEURAL CARDINALITY ESTIMATION

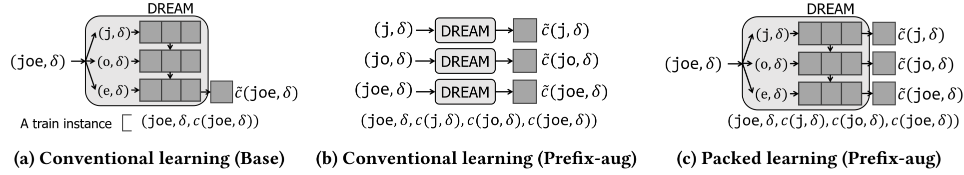
Existing deep learning models used for the cardinality estimation of string predicate queries can be viewed as an encoder-decoder model [26, 31]. The encoder produces a hidden representation of a query, and the decoder outputs the estimated cardinality based on the hidden representation. While the Astrid model in [26] can estimate the cardinality of a substring query, it is difficult to handle the approximate substring queries. As pointed out in Section 2, to adapt the Astrid model to estimate the cardinalities of approximate substring queries (q, δ), we need to build a model for every distance threshold δ with $0 \le \delta \le \delta_M$. On the other hand, the CardNet model in [31] utilizes a feedforward neural network (`FNN`) to construct the encoder as well as the decoder. As addressed in Section 2, since `FNN` does not share model parameters across positions in each query string, it requires a large number of model parameters. Moreover, it cannot handle a query string longer than a certain length.

5.1 Key Ideas of the DREAM model

The DREAM model is an encoder-decoder model that utilizes `LSTM` [8] as an encoder to treat a query string as a sequence of characters as done in [22, 26]. Since `LSTM` is run over each character of a query string, the `LSTM` encoder enables the sharing of model parameters across positions in each query string as well as learning the cardinality relationships across the prefixes of a query.

The architecture of the DREAM model is shown in Figure 7. For a substring query (q, δ), in each step t, we first transform every pair of each character $q[t]$ and δ into a real-valued feature vector. The `LSTM` encoder next generates the hidden representation h_t with the previous hidden representation h_{t-1} and the feature vector of the pair $(q[t], \delta)$. Note that h_t is the hidden representation of the prefix query $(q[1, t], \delta)$. After producing $h_{|q|}$, the `FNN` decoder outputs the estimated cardinality $\tilde{c}(q, \delta)$ of $c(q, \delta)$ by taking $h_{|q|}$ as input. If we want to additionally produce $\tilde{c}(q[1, t], \delta)$ of the prefix query $(q[1, t], \delta)$ in step $1 \le t < |q|$, we apply the `FNN` decoder.

The possible learning methods for the DREAM model are shown in Figure 8. If we use the *base* training data, as illustrated in Figure 8(a), the conventional learning updates the model parameters for each training instance $(q, \delta, c(q, \delta))$. In this case, we learn the cardinality relationships across the queries (q, δ) with $0 \le \delta \le \delta_M$



(a) Conventional learning (Base) **(b) Conventional learning (Prefix-aug)** **(c) Packed learning (Prefix-aug)**

Figure 8: The learning methods for the DREAM model.

but may not learn those across the prefix queries $(q[1,t],\delta)$ with $1 \le t \le |q|$. However, if we use the *prefix-aug* training data, the DREAM model can learn both cardinality relationships. As illustrated in Figure 8(b), for a training instance $(\texttt{joe}, \delta, c(\texttt{j},\delta), c(\texttt{jo},\delta), c(\texttt{joe},\delta))$, the conventional learning trains the DREAM model one by one with the training instances $(\texttt{j}, \delta, c(\texttt{j},\delta))$, $(\texttt{jo}, \delta, c(\texttt{jo},\delta))$ and $(\texttt{joe}, \delta, c(\texttt{joe},\delta))$. Using the *prefix-aug* training data improves the estimation accuracy of the model but incurs additional overheads to train the model with the prefix queries. To reduce such overheads while learning the cardinality relationships among the prefixes of each query, we propose a novel learning method, called *packed learning*, which trains the model once for a *prefix-aug* training instance. As illustrated in Figure 8(c), for a training instance $(\texttt{joe}, \delta, c(\texttt{j},\delta), c(\texttt{jo},\delta), c(\texttt{joe},\delta))$, in every step t, the *packed learning* enforces the DREAM model to output $\tilde{c}(q[1,t],\delta)$ and updates the model parameters to accurately estimate $c(q[1,t],\delta)$.

5.2 The DREAM Model

We present the details of the encoder as well as the decoder.

Embedding layer: When using deep learning models, we usually transform raw inputs into low-dimensional real-valued vectors. Let h_c, h_d and $|\Sigma|$ be the sizes of a character embedding, a distance embedding and the alphabet, respectively. For an approximate substring query (q,δ), the model has a character embedding matrix E^{chr} of type $\mathbb{R}^{h_c \times |\Sigma|}$ and a distance embedding matrix E^{dst} of type $\mathbb{R}^{h_d \times (\delta_M+1)}$ to map each character $q[t]$ and a distance threshold δ into the embedding vectors, denoted by $e^{chr}_{q[t]}$ and e^{dst}_{δ}, respectively. Each embedding vector is obtained by the corresponding column vector of an embedding matrix as

$$e^{chr}_{q[t]} = E^{chr}[q[t]] \quad \text{and} \quad e^{dst}_{\delta} = E^{dst}[\delta].$$

We use $e^{chr}_{q[t]} \oplus e^{dst}_{\delta}$ as the vector representation of the pair $(q[t],\delta)$ where the symbol $\oplus$ represents the concatenation operator.

The LSTM encoder: Since we utilize LSTM as an encoder, we can handle query strings with any length. At each step t, the LSTM encoder receives $e^{chr}_{q[t]} \oplus e^{dst}_{\delta}$ and h_{t-1} as input, and computes the hidden representation h_t of a prefix query $(q[1,t],\delta)$ by

$$h_t = f(e^{chr}_{q[t]} \oplus e^{dst}_{\delta}, h_{t-1})$$

where f is an LSTM cell as a recurrent unit.

The FNN decoder: The FNN encoder takes h_t as input, performs a linear transform and applies an ReLU activation function to ensure that the estimated cardinality is non-negative. It outputs the estimated cardinality $\tilde{c}(q[1,t],\delta)$ by computing

$$\tilde{c}(q[1,t],\delta) = \mathrm{ReLU}(W \cdot h_t + b)$$

where both W and b are the parameters of FNN.

5.3 Learning Methods

After illustrating the conventional learning method, we present the proposed *packed learning* method.

Loss function: We use the mean squared logarithmic error (MSLE) instead of the mean squared error (MSE) as a loss function. The MSLE is a multiplicative error and narrows down the large output space to a smaller one, thereby decreasing the learning difficulty [31]. The squared logarithmic error $\mathcal{L}_{q,\delta}$ between $c(q,\delta)$ and $\tilde{c}(q,\delta)$ is

$$\mathcal{L}_{q,\delta} = (\log(c(q,\delta)) - \log(\tilde{c}(q,\delta)))^2.$$

The MSLE loss of the *base* training data, denoted by $\mathcal{L}_{\texttt{base}}$, is

$$\mathcal{L}_{\texttt{base}} = \frac{1}{|S_Q| \cdot (\delta_M + 1)} \Sigma_{q \in S_Q} \Sigma_{\delta=0}^{\delta_M} \mathcal{L}_{q,\delta}.$$

The MSLE loss of the *prefix-aug* training data, $\mathcal{L}_{\texttt{prefix-aug}}$, is

$$\mathcal{L}_{\texttt{prefix-aug}} = \frac{1}{|S_Q| \cdot (\delta_M + 1)} \Sigma_{q \in S_Q} \Sigma_{\delta=0}^{\delta_M} \frac{1}{|q|} \Sigma_{t=1}^{|q|} \mathcal{L}_{q[1,t],\delta}.$$

Conventional learning: To minimize the losses, we adopt stochastic gradient descent (SGD) by using the Adam optimizer [14]. It trains the DREAM model with a set of training instances, each of which has only a single estimated cardinality. Let $T_{\texttt{base}}$ be the *base* training data and $(q_i, \delta_i, c(q_i,\delta_i))$ be the i-th instance in $T_{\texttt{base}}$. Then, for a mini-batch $I \subset \{i | 1 \le i \le |T_{\texttt{base}}|\}$ of a batch size b, the mini-batch loss on I is $\mathcal{L}_I = (\Sigma_{i \in I} \mathcal{L}_{q_i,\delta_i})/|I|$. On the other hand, let $T_{\texttt{prefix-aug}}$ be the *prefix-aug* training data and $(q_i, \delta_i, c(q_i[1,1],\delta_i), \ldots, c(q_i[1,|q_i|],\delta_i))$ be the i-th instance in $T_{\texttt{prefix-aug}}$. Then, for a mini-batch $I \subset \{(i,j) | 1 \le i \le |T_{\texttt{prefix-aug}}|, 1 \le j \le |q_i|\}$ of a batch size b, the mini-batch loss on I becomes $\mathcal{L}_I = (\Sigma_{(i,j) \in I} \mathcal{L}_{q_i[1,j],\delta_i})/|I|$.

Packed learning: To train the DREAM model with the *prefix-aug* training data efficiently and effectively, the *packed learning* method trains the model once for each *prefix-aug* training instance which contains the cardinalities of all prefix queries of a query. For a mini-batch $I \subset \{i | 1 \le i \le |T_{\texttt{prefix-aug}}|\}$ of a batch size b, the mini-batch loss on I is

$$\mathcal{L}_I = \frac{1}{|I|} \Sigma_{i \in I} \Sigma_{j=1}^{|q_i|} \mathcal{L}_{q_i[1,j],\delta_i}.$$

To minimize the above mini-batch loss, for each query (q_i,δ_i), the *packed learning* method updates the model parameters by using the estimated cardinalities of all prefix queries of (q_i,δ_i), as shown in Figure 8(c). For an instance of the *prefix-aug* training data, the

Table 2: Statistics about the datasets.

Dataset	String Set (S_D)				Query Set (S_Q)				
	$\lvert\Sigma\rvert$	$\lvert S_D\rvert$	ℓ_{avg}	ℓ_{max}	$\lvert S_Q\rvert$	ℓ_{avg}	ℓ_{max}	$\lvert S_P\rvert$	$\frac{\lvert S_P\rvert}{\lvert S_Q\rvert}$
WIKI	3735	1,031,930	98.4	300	170,136	12.1	20	1,072,056	6.3
IMDB	81	1,000,000	19.2	278	464,985	12.4	20	2,494,460	5.4
DBLP	63	50,000	61.6	301	24,764	13.5	20	176,241	7.1
GENE	4	15,400	225.4	469	21,613	13.0	20	136,644	6.3

packed learning and *conventional* learning methods perform the forward-backward pass $O(|q|)$ and $O(|q|^2)$ times, respectively.

Computational efficiency: When LSTM with d units and $O(d^2)$ weights is run over u steps, it performs $O(d^2u)$ arithmetic operations during the forward-backward pass [33]. Thus, for a *prefix-aug* training instance, since the *packed learning* method performs the forward-backward pass $O(|q|)$ times, it requires $O(d^2|q|)$ time to train the DREAM model with a *prefix-aug* training instance. On the other hand, since the conventional learning method performs the forward-backward $O(|q|^2)$ times, it takes $O(d^2|q|^2)$ time to train the DREAM model with a *prefix-aug* training instance.

6 EXPERIMENTS

We empirically evaluate the TEDDY and SODDY algorithms as well as the DREAM model with the *packed learning* method.

Experimental setting: We implement all estimators in python with PyTorch and train them on an NVIDIA RTX 3090 GPU. Moreover, we implement data generation algorithms in C++ and run them on a machine with an Intel i7-8700 3.2GHz CPU.

Datasets: We use three real-life datasets for our performance study.

- WIKI: We create S_D by selecting sentences from 101,873 Wikipedia articles in the DocRED dataset [36] and use entity mentions in the Wikipedia articles as S_Q.
- IMDB: We produce S_D by choosing movie titles from IMDB data in https://datasets.imdbws.com. We generate S_Q by sampling substrings of the strings in S_D.
- DBLP: We use titles of DBLP articles as S_D in [32] and create S_Q by randomly choosing substrings of the strings in S_D.
- GENE: We use the DNA sequences in [37] as S_D and produce S_Q by selecting substrings of the DNA sequences in S_D.

The statistics of the datasets are summarized in Table 2. The average and maximum lengths of a string in a string set are represented by ℓ_{avg} and ℓ_{max}, respectively. Recall that Σ represents the alphabet of a dataset and S_P is the set of all distinct prefixes in S_Q. We limit the maximum query length to 20 and set the default value of δ_M to 3. We produce the queries for every pair of $q \in S_Q$ and δ with $0 \le \delta \le \delta_M$. For each dataset, after generating the training data with S_Q, S_D and δ_M, we put 80%, 10% and 10% of the data in the training set, the validation set and the test set, respectively.

Training data generation: We implement the algorithms below.

- NaiveGen: It is the naive algorithm presented in Section 3.3.
- Qgram: This is a modified NaiveGen algorithm by applying the q-gram pruning technique in [13] to reduce the unnecessary substring edit distance computation.
- TASTE: It refers to a modified NaiveGen algorithm by utilizing the partition-based pruning method in [6]. Note that TASTE is the state-of-the-art algorithm for processing approximate substring matching using edit distance.
- TEDDY: This denotes the TEDDY algorithm in Section 4.3.
- SODDY: It is another variant of the query ordering scheme to scan sorted query strings. See our extended version [18] for details.

Cardinality estimators: We implement and compare the following cardinality estimators for approximate substring queries.

- LBS: This is the state-of-the-art [20] for cardinality estimation of approximate substring queries among the traditional approaches.
- Astrid: This is a neural cardinality estimation model [26] for substring queries. We build a model for each distance threshold δ to support *approximate* substring queries. Since it requires the cardinalities of all prefix queries for each query, we evaluate this model only with the *prefix-aug* training data.
- CardNet: This is the state-of-the-art model [31] for cardinality estimation of approximate queries. We adapt this model to the cardinality estimation problem of approximate *substring* queries.
- DREAM: This is the proposed model presented in Section 5.

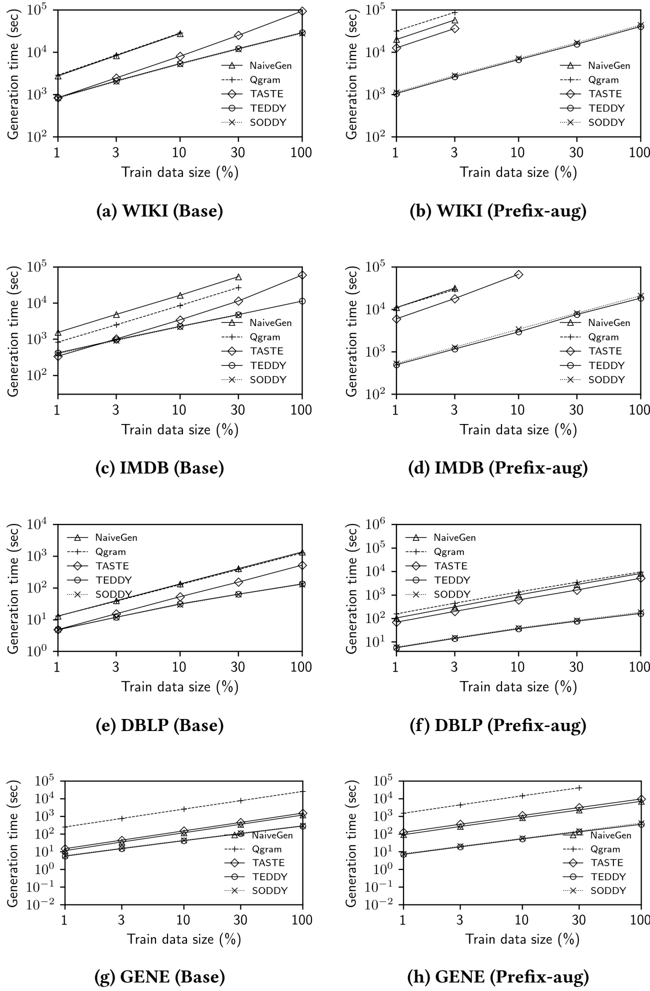
Evaluation measures: We report estimation accuracy by the q-error (i.e., $\max(\frac{estim}{actual}, \frac{actual}{estim})$) [10, 15, 16, 26]. We lower bound the estimated and actual cardinalities at 1 to prevent division by zero. We run all training data generation algorithms and estimators three times, and average the execution times and estimation errors.

Hyperparameters: To set the hyperparameters, we vary the number of hidden units (128, 256, 512, 1024), the batch size (16, 32, 64, 128) and the learning rate (0.1, 0.01, 0.001). Our model with the 512 hidden units, the batch size of 32 and the learning rate of 0.01 performed the best. We set the size of a character embedding h_c to 95 and that of a distance embedding h_d to 5. The numbers of layers in the LSTM encoder and the FNN decoder are 1 and 3, respectively. We use the activation function leaky-ReLU in the FNN decoder. We train our model for up to 100 epochs with early stopping if the validation accuracy has not improved in the last 5 epochs. We also tuned the hyperparameters of compared models. However, the details are in our extended version [18] of this paper due to space limitations.

6.1 Training Data Generation

Varying $|S_Q|$: To produce the *base* and *prefix-aug* training datasets, we run the NaiveGen, Qgram, TASTE, TEDDY and SODDY algorithms while varying the size of the query string set. We produce each query string set by selecting the query strings in S_Q and varying the sampling rate from 1% to 100%. We present the results in Figure 9 but do not report the execution time of an algorithm if it did not finish within 30 hours.

The TEDDY and SODDY algorithms are the best two performers for all datasets. The TASTE algorithm has the third-best performance. Note that the TEDDY and SODDY algorithms are at least 2.5 and 11.3 times faster than the TASTE algorithm when producing the *base* and *prefix-aug* training datasets, respectively. Recall that the NaiveGen algorithm takes $O(L_{S_Q} \cdot |S_Q| \cdot L_{S_D} \cdot |S_D|)$ and $O(L_{S_Q} \cdot |S_P| \cdot L_{S_D} \cdot |S_D|)$ times to generate the *base* and *prefix-aug* training datasets, respectively. Thus, it is more expensive to generate the *prefix-aug* training data. Since the Qgram and TASTE algorithms are implemented by applying the q-gram pruning in



(a) WIKI (Base)

(b) WIKI (Prefix-aug)

(c) IMDB (Base)

(d) IMDB (Prefix-aug)

(e) DBLP (Base)

(f) DBLP (Prefix-aug)

(g) GENE (Base)

(h) GENE (Prefix-aug)

Figure 9: Generating the training data.

Table 3: Generation time (sec) of the training data (100%).

	Base				Prefix-aug			
	WIKI	IMDB	DBLP	GENE	WIKI	IMDB	DBLP	GENE
SODDY	**28204**	**11248**	**131**	302	44216	21093	185	426
TEDDY	29283	11434	135	**288**	**40862**	**18526**	**165**	**361**

[13] and the partition-based pruning in [6] to the NaiveGen algorithm, respectively, the three algorithms exhibit similar behavior. This is also confirmed by the graphs in Figure 9. Moreover, the three algorithms cannot generate even 10% of the *prefix-aug* training data while the TEDDY and SODDY algorithms produce 100% of the *prefix-aug* training data for every dataset.

To compare the TEDDY and SODDY algorithms, we provide their execution times for the sampling rate 100% in Table 3. For the *prefix-aug* training data, the TEDDY algorithm always outperforms the SODDY algorithm by up to 17.9%. On the other hand, the SODDY algorithm is slightly faster than the TEDDY algorithm to produce the *base* training data except for the GENE dataset. Note

Table 4: Ablation study of the TEDDY algorithm.

	DBLP				GENE			
	Filled ($\times 10^{10}$)		Time (sec)		Filled ($\times 10^{10}$)		Time (sec)	
Setting	Base	Prefix	Base	Prefix	Base	Prefix	Base	Prefix
NaiveGen	92.7	547.0	1,367	8,301	87.9	503.1	1,210	7,239
TEDDY-S	49.6	49.6	825	1023	43.4	43.4	559	753
TEDDY-R	36.9	264.0	666	4,222	61.5	391.9	1,019	6,188
TEDDY	**5.7**	**5.7**	**135**	**165**	**17.8**	**17.8**	**288**	**361**

that the subtrie pruning by the TEDDY algorithm is more effective when there are many queries sharing their prefixes. For the *prefix-aug* training data, since there are a lot of query strings sharing a common prefix, the TEDDY algorithm is faster than the SODDY algorithm. Because the TEDDY algorithm is faster than or comparable to the SODDY algorithm, we use the TEDDY algorithm as the representative of query ordering schemes in the rest of the paper.

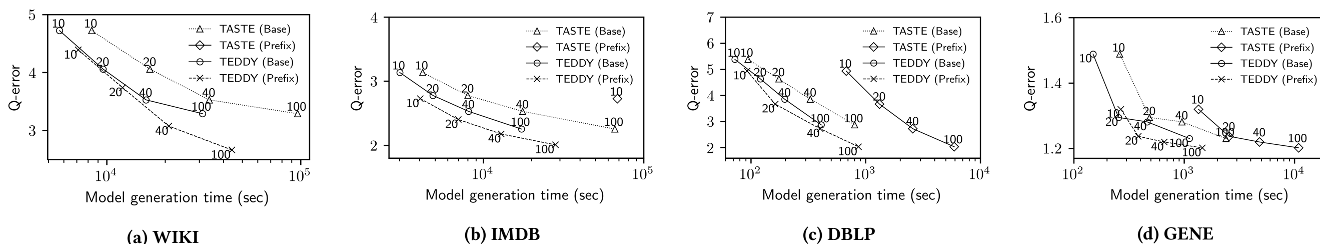
Ablation study: To validate the effectiveness of the required column pruning as well as the distance computation sharing by our TEDDY algorithm, we conduct an ablation study with TEDDY-R and TEDDY-S that denote the TEDDY algorithms with only the required column pruning and only the distance computation sharing, respectively. We report the results with DBLP and GENE datasets only since TEDDY-R and TEDDY-S did not finish for the other datasets within 30 hours. Note that the NaiveGen algorithm utilizes neither the required column pruning nor the distance computation sharing. We show the number of filled entries in the table D and the execution time in Table 4. For the *prefix-aug* training data, since there are a lot of query strings sharing a common prefix, TEDDY-S significantly reduces the number of computed entries in the table D. For instance, TEDDY-S prunes at least 91% of the entries in the table D for DBLP and GENE datasets. As expected, the TEDDY algorithm performs significantly better than TEDDY-S and TEDDY-R.

6.2 Cardinality Estimation

Estimation accuracy: We evaluate the performance of all cardinality estimators with both types of training data. In Table 5, we report the average error, 50th, 90th, 99th and 100th (i.e., max error) percentile errors with the default value of $\delta_M = 3$. We report more percentiles with $1 \le \delta_M \le 5$ in the extended version [18] of this paper. Since the Astrid model can be trained only with *prefix-aug* training data, we cannot report its accuracy for the *base* training data. The DREAM model outperforms all other estimators for all datasets. Since the LBS algorithm leverages only short strings and simple statistical relations, it is insufficient for estimating complicated underlying patterns. On the other hand, since the Astrid model is trained separately for each distance threshold, it cannot capture the relationship between queries that have different thresholds. In addition, since adopting a sequential model is adequate for string queries, the generalization performance of the DREAM model is better than that of the CardNet model. Note that the maximum error (the 100th percentile) is 27.6 times greater than the 99th quantile error on average. Since the maximum errors are much higher than average errors for all estimators, suppressing the maximum error for the DREAM model is an interesting direction for future research. For each neural estimator, training the estimator

Table 5: Accuracy of cardinality estimation.

Type	Estimators	WIKI Avg.	WIKI 50th	WIKI 90th	WIKI 99th	WIKI Max.	IMDB Avg.	IMDB 50th	IMDB 90th	IMDB 99th	IMDB Max.	DBLP Avg.	DBLP 50th	DBLP 90th	DBLP 99th	DBLP Max.	GENE Avg.	GENE 50th	GENE 90th	GENE 99th	GENE Max.
Traditional	LBS	3.750	1.498	7.000	35.0	1418	3.165	**1.353**	6.149	25.7	**528**	3.004	1.321	6.000	26.0	178	2.881	1.311	6.487	23.139	61.0
Base	CardNet	4.267	1.661	5.632	35.3	25899	2.594	1.539	4.477	15.8	1454	3.301	1.670	6.003	24.8	233	1.633	1.217	2.606	6.539	41.3
	DREAM	3.290	1.626	4.627	24.0	9753	2.252	1.464	3.789	12.5	1248	2.869	1.512	4.787	21.2	291	1.230	**1.048**	1.683	3.067	18.7
Prefix-aug	Astrid	6.912	1.827	7.469	47.4	23619	3.559	1.515	4.008	16.6	159887	3.134	1.525	5.877	25.9	204	1.448	1.118	2.065	4.947	65.8
	CardNet	3.358	1.602	4.872	26.2	1914	2.175	1.385	3.604	12.7	2040	2.811	1.435	5.006	20.3	414	1.527	1.147	2.302	6.182	42.3
	DREAM	**2.663**	**1.495**	**4.107**	**18.4**	**1279**	**2.006**	1.392	**3.284**	**10.3**	598	**2.031**	**1.245**	**3.274**	**12.6**	**152**	**1.202**	1.050	**1.563**	**2.931**	**16.9**



(a) WIKI (b) IMDB (c) DBLP (d) GENE

Figure 10: Trade-off between accuracy and time.

Table 6: Effectiveness of the packed learning method.

	WIKI Error	WIKI Time (min)	IMDB Error	IMDB Time (min)	DBLP Error	DBLP Time (min)	GENE Error	GENE Time (min)
Traditional	2.90	316	2.32	1042	**2.09**	60	1.21	149
Packed	**2.72**	**46**	**2.01**	**160**	2.10	**10**	**1.20**	**14**

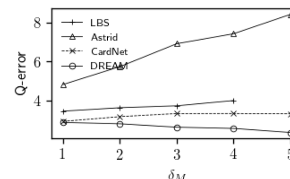


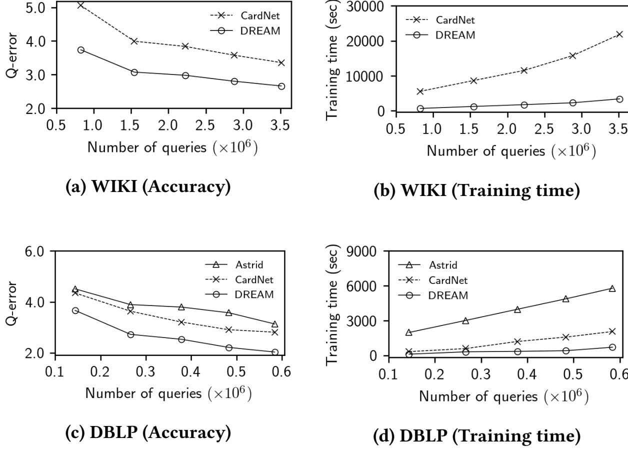
Figure 11: Q-errors of estimators by varying δ_M (WIKI).

with the *prefix-aug* training data is more accurate than training the estimator with the base data. Thus, we report the experimental results only for the *prefix-aug* training data in the rest of the paper.

Trade-off between Accuracy and Time: We plot the accuracy against the model generation time (i.e., the total time to generate the training data as well as train the DREAM model) with varying the size of the query string set S_Q for both training data types in Figure 10. As discussed in Section 6.1, since the TASTE algorithm is the best among the tested algorithms except for the TEDDY and SODDY algorithms, we compare the TEDDY algorithm only with the TASTE algorithm. The number at each point in the graphs denotes the sampling rate (percentage) of the query strings in S_Q. At the same model generation time, the DREAM model is the most accurate when it is trained on the *prefix-aug* data generated by the TEDDY algorithm. Furthermore, with the same sampling rate, the model generation time using the TEDDY algorithm with the *prefix-aug* training data is slower, but the trained model has higher accuracy. On the other hand, for every dataset, using the TEDDY algorithm even with the 40% sample of *prefix-aug* training data has a faster model generation time and a higher estimation accuracy than that with the 100% sample of *base* training data. This confirms the improved trade-off by utilizing the *prefix-aug* training data instead of the *base* training data. Since the TASTE algorithm do not finish within 30 hours for the *prefix-aug* training data of the WIKI dataset, we do not report the result of the TASTE algorithm in Figure 10(a). Note that using the TASTE algorithm with the *prefix-aug* training data instead of the *base* training data worsens the trade-off.

Effectiveness of the Packed Learning: To compare the performance of the conventional learning and packed learning methods for the DREAM model with the *prefix-aug* training data, we present the accuracies and training times of both methods in Table 6. The training time using the *packed learning* method is at least 6 times faster than that of the conventional learning method. The reason is that the conventional learning and packed learning methods take $O(|q|^2 \cdot d^2)$ and $O(|q| \cdot d^2)$ times to train a query, respectively, as discussed in Section 5.3. Meanwhile, the DREAM model trained with the *packed-learning* method is generally more accurate than that trained with the conventional learning method.

Varying δ_M: We also study the performance of all estimators with other maximum distance thresholds δ_M and report the results on the WIKI dataset in Figure 11. Since the estimation time of the LBS algorithm increases exponentially with δ and even a single estimation did not finish within an hour with $\delta_M = 5$, we do not report the result of the LBS algorithm with $\delta_M = 5$. The DREAM model consistently shows the lowest error for $1 \leq \delta_M \leq 5$ and the



(a) WIKI (Accuracy)

(b) WIKI (Training time)

(c) DBLP (Accuracy)

(d) DBLP (Training time)

Figure 12: Accuracy and training time.

Table 7: Sizes of estimators (MB) with $\delta_M = 3$.

	WIKI	IMDB	DBLP	GENE
LBS	89.95	46.56	13.69	27.82
Astrid	169.22	28.72	26.72	19.35
CardNet	50.39	26.21	22.55	10.56
DREAM	**7.88**	**7.84**	**7.83**	**7.81**

error becomes smaller as δ_M increases. In contrast, the error of the Astrid model becomes larger with increasing δ_M becomes larger.

Effects of Training Data Size: Since the LBS algorithm does not require the training data, we conduct the experiment with the neural estimators only by varying the size of training data. Due to the lack of space, the q-errors and the training times of the models on WIKI and DBLP datasets only are shown in Figure 12. Refer to the extended version of the paper [18] for the other datasets. We use the *packed learning* method to train the DREAM model with the *prefix-aug* training data. Since we cannot use the *packed learning* method with the CardNet and Astrid models, we utilize the conventional learning method to train them with the *prefix-aug* training data. Specifically, for a single instance $(q, \delta, (c(q[1,1], \delta), \ldots, c(q[1,|q|], \delta)))$, we train them using multiple instances $(q[1,\ell], \delta, c(q[1,\ell], \delta))$ with $1 \leq \ell \leq |q|$. The DREAM model has the fastest training time as well as the highest accuracy with the same training data size among the deep learning models. The Astrid model takes much more training time than other deep learning models and it finishes within 30 hours only for the DBLP dataset. As expected, with increasing the training data size, the accuracy is better but the training time increases.

Sizes of estimators: We show the sizes of all estimators used in the experiment in Table 7. For every dataset, we set each estimator to have the same size for both types of the training data. Note that the sizes of estimators depend on the datasets. For every dataset, the DREAM model is the most accurate with the smallest size among all estimators. Since the Astrid model consists of a single model for each different distance threshold, it requires a large model size.

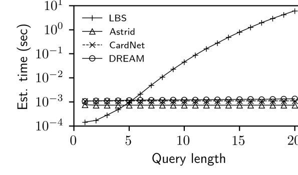


Figure 13: Estimation time of estimators (DBLP).

On the other hand, as discussed in Section 5.1, the DREAM model shares model parameters across positions in each query string while the CardNet model does not. Thus, the size of the CardNet model is larger than that of the DREAM model.

Estimation time: To give an idea of how much the overhead of estimating the cardinality of an approximate substring query is, we report the average estimation time of every estimator on the DBLP dataset in Figure 13. Refer to the extended version [18] of the paper for the other datasets. As the size of the query string increases, the time to estimate the cardinality of the query by the LBS algorithm grows. On the other hand, the estimation times of all neural estimators are almost the same regardless of the query lengths. Note that the Astrid model has the fastest estimation time among the neural models. However, its accuracy is the worst among all neural estimators, as shown in Table 5. Although the LBS algorithm takes the quickest estimation times with short query strings, since it needs to generate many string patterns for a long query string, its estimation time is much longer than deep learning models for long query strings. Thus, it is preferable to use the neural estimators for cardinality estimation of approximate substring queries.

7 CONCLUSION

We studied the cardinality estimation problem of approximate substring queries. We first proposed the TEDDY and SODDY algorithms to quickly generate a training data by using the required column pruning and sharing common computations. We next presented an RNN-based cardinality estimation model, called the DREAM model. In addition, we proposed the packed learning method to efficiently train the DREAM model. The experimental results confirmed that the proposed data generation algorithms are very efficient and the DREAM model using the packed learning method is more accurate than other estimators while it requires the smallest model sizes.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2020R1A2C1003576). Furthermore, it was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2020-0-00857, Development of cloud robot intelligence augmentation, sharing and framework technology to integrate and enhance the intelligence of multiple robots). This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2022-00155885, Artificial Intelligence Convergence Innovation Human Resources Development (Hanyang University ERICA))

REFERENCES

[1] (Accessed June 11, 2021). *Edit distance.* https://en.wikipedia.org/wiki/Edit_distance

[2] Mehmet Aytimur and Ali Cakmak. 2018. Estimating the selectivity of LIKE queries using pattern-based histograms. *Turkish Journal of Electrical Engineering & Computer Sciences* 26, 6 (2018), 3319–3334.

[3] Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics.* 26–33.

[4] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. 2004. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proceedings. 20th International Conference on Data Engineering.* IEEE, 227–238.

[5] Zhiyuan Chen, Nick Koudas, Flip Korn, and Shanmugavelayutham Muthukrishnan. 2000. Selectively estimation for boolean queries. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* 216–225.

[6] Dong Deng, Guoliang Li, and Jianhua Feng. 2012. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In *2012 IEEE 28th International Conference on Data Engineering.* IEEE, 762–773.

[7] Szilárd Zsolt Fazekas and Robert Mercaş. 2021. Clusters of repetition roots: single chains. In *International Conference on Current Trends in Theory and Practice of Informatics.* Springer, 400–409.

[8] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[9] D Gusfield. 1997. Algorithms on strings, trees, and sequences Cambridge University Press. *Cambridge, England* (1997).

[10] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: learn from data, not from queries! *arXiv preprint arXiv:1909.00607* (2019).

[11] HV Jagadish, Olga Kapitskaia, Raymond T Ng, and Divesh Srivastava. 2000. One-dimensional and multi-dimensional substring selectivity estimation. *The VLDB Journal* 9, 3 (2000), 214–230.

[12] HV Jagadish, Raymond T Ng, and Divesh Srivastava. 1999. Substring selectivity estimation. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* 249–260.

[13] Younghoon Kim and Kyuseok Shim. 2013. Efficient top-k algorithms for approximate substring matching. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* 385–396.

[14] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[15] 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 *1st International Workshop on Applied AI for Database Systems and Applications.*

[16] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[17] 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.* 282–293.

[18] Suyong Kwon, Woohwan Jung, and Kyuseok Shim. 2022. *Cardinality Estimation of Approximate Substring Queries using Deep Learning.* Technical Report. Seoul National University, Electrical and Computer Engineering Department. https://github.com/sykwon/vldb-tr/raw/main/pvldb_extended.pdf

[19] 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 Endowment, 195–206.

[20] Hongrae Lee, Raymond T Ng, and Kyuseok Shim. 2009. Approximate substring selectivity estimation. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology.* 827–838.

[21] Guoliang Li, Dong Deng, and Jianhua Feng. 2011. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data.* 529–540.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[23] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM computing surveys (CSUR)* 33, 1 (2001), 31–88.

[24] Marius Pasca. 2004. Acquisition of categorized named entities for web search. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management.* 137–145.

[25] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining.* 701–710.

[26] 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.

[27] Esko Ukkonen. 1985. Finding approximate patterns in strings. *Journal of algorithms* 6, 1 (1985), 132–137.

[28] Rares Vernica and Chen Li. 2009. Efficient top-k algorithms for fuzzy search in string collections. In *Proceedings of the First International Workshop on Keyword Search on Structured Data.* 9–14.

[29] Jiannan Wang, Jianhua Feng, and Guoliang Li. 2010. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1219–1230.

[30] Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. 2009. Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 759–770.

[31] 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 2020 ACM SIGMOD International Conference on Management of Data.* 1197–1212.

[32] Melanie Weis, Felix Naumann, and Franziska Brosy. 2006. A duplicate detection benchmark for XML (and relational) data. In *Proc. of Workshop on Information Quality for Information Systems (IQIS).*

[33] Ronald J Williams and Jing Peng. 1990. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation* 2, 4 (1990), 490–501.

[34] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)* 36, 3 (2011), 1–41.

[35] 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. *arXiv preprint arXiv:1905.04278* (2019).

[36] Yuan Yao, Deming Ye, Peng Li, Xu Han, Yankai Lin, Zhenghao Liu, Zhiyuan Liu, Lixin Huang, Jie Zhou, and Maosong Sun. 2019. DocRED: A Large-Scale Document-Level Relation Extraction Dataset. In *Proceedings of ACL 2019.*

[37] Qiang Yu, Dingbang Wei, and Hongwei Huo. 2018. SamSelect: a sample sequence selection algorithm for quorum planted motif search on large DNA datasets. *BMC bioinformatics* 19, 1 (2018), 1–16.