Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning

Suraj Shetiya
UT Arlington
suraj.shetiya@mavs.uta.edu

Saravanan Thirumuruganathan
QCRI, HBKU
sthirumuruganathan@hbku.edu.qa

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Gautam Das
UT Arlington
gdas@cse.uta.edu

ABSTRACT

Accurate selectivity estimation for string predicates is a long-standing research challenge in databases. Supporting pattern matching on strings (such as prefix, substring, and suffix) makes this problem much more challenging, thereby necessitating a dedicated study. Traditional approaches often build pruned summary data structures such as tries followed by selectivity estimation using statistical correlations. However, this produces insufficiently accurate cardinality estimates resulting in the selection of sub-optimal plans by the query optimizer. Recently proposed deep learning based approaches leverage techniques from natural language processing such as embeddings to encode the strings and use it to train a model. While this is an improvement over traditional approaches, there is a large scope for improvement.

We propose Astrid, a framework for string selectivity estimation that synthesizes ideas from traditional and deep learning based approaches. We make two complementary contributions. First, we propose an embedding algorithm that is query-type (prefix, substring, and suffix) and selectivity aware. Consider three strings 'ab', 'abc' and 'abd' whose prefix frequencies are 1000, 800 and 100 respectively. Our approach would ensure that the embedding for 'ab' is closer to 'abc' than 'abd'. Second, we describe how neural language models could be used for selectivity estimation. While they work well for prefix queries, their performance for substring queries is sub-optimal. We modify the objective function of the neural language model so that it could be used for estimating selectivities of pattern matching queries. We also propose a novel and efficient algorithm for optimizing the new objective function. We conduct extensive experiments over benchmark datasets and show that our proposed approaches achieve state-of-the-art results.

PVLDB Reference Format:
Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning. PVLDB, 14(4): 471 - 484, 2021.
doi:10.14778/3436905.3436907

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436907

1 INTRODUCTION

Selectivity estimation is the problem of estimating the proportion of tuples in a relation that satisfy a given query. We consider the problem of accurate selectivity estimation for queries with string predicates such as the SQL LIKE operator. Knowing the selectivity of individual predicates in a query such as "name LIKE '%abc%' AND zipcode LIKE '%123' AND ssn LIKE '%1234'" allows the optimizer to determine which filter to process first. Selectivity estimation over string predicates is much harder as the queries can be based on prefix, substring, suffix or a combination thereof. Inaccurate estimates result in the selection of a poor plan by the query optimizer.

1.1 Prior Approaches and Their Limitations

There has been extensive work on selectivity estimation for string predicate queries.

Trajditional Approaches. The early works build a *summary data structure* for answering string predicate queries. For example, suffix tree [18] is a special type of trie that stores all the suffixes of a given string and can answer queries involving *substrings* including suffixes. Typically, the complete summary data structure is substantially larger than the text used to build it [61] and cannot be stored in memory. Hence, the data structure is pruned by removing all entries whose frequency is less than a given threshold. If the query string q exists in the data structure, then it can be returned precisely. However, if q does not exist, then it is decomposed into possibly overlapping substrings i.e. $q = \alpha\beta$ where both α and β exist in the summary data structure. Then, different statistical approaches [8, 27, 40] can be used to combine the *exact* selectivity of α and β to obtain the *approximate* selectivity of $q = \alpha\beta$.

Prior approaches faced two fundamental issues that limited their performance. First, getting accurate estimates often required an unacceptably large summary data structure even in the current era where the storage is plentiful. Second, the algorithms for estimating the selectivity of infrequent keywords by combining the selectivity of frequent substrings often relied on statistical assumptions that might not be realistic and result in inaccurate selectivity estimates.

Deep Learning (DL) based approaches. A number of recent works such as [20, 22, 35, 36, 79] have tackled the problem of selectivity estimation. However, none of these work are targeted towards string predicates which is a more challenging problem due to its diverse query types. For example, MSCN [36] only tackles equality queries by hashing string literals to an integer. Another approach used in Neo [49] and others [20, 79] is to leverage word embedding techniques from natural language processing (NLP). The strings are encoded as a real-valued vector using the embeddings. Selectivity estimation is then formulated as a learning problem where a DL model is trained using string embeddings and their true selectivity.

One can potentially generate a training dataset containing all prefixes/substrings/suffixes and use it to train a DL model for selectivity estimation. This obviates the need for storing the summary data structure. However, the performance of the model now hinges on the quality of the embeddings. As we shall describe in Section 4, directly using word/character embeddings from NLP provides sub-optimal results. The major reason is that they are not aware of the downstream task of selectivity estimation. As an example, consider three strings 'ab', 'abc' and 'abd' whose prefix frequencies are 1000, 800 and 100 respectively. A naive approach will give similar embeddings for all three strings and result in a large estimation error for 'abd'. The strings 'abc' and 'xyc' can be considered similar for suffix queries which might not be acceptable for substring queries.

1.2 Outline of Technical Results

Recently, deep learning (DL) based approaches have achieved tremendous success in selectivity estimation [20, 36, 59, 79]. However, these works often focus on non-string data types with limited support for string queries through embedding. Selectivity estimation of string predicates with pattern matching is a notoriously challenging problem and requires a dedicated investigation. Our key insight is that revisiting the principles of traditional approaches utilizing deep learning based primitives can alleviate the pain points for these approaches. In this paper, we propose Astrid[1], a generic and extensible approach that synthesizes summary data structure and embeddings/neural language models to yield highly accurate estimates. Combining these requires a number of non-trivial developments. We propose two complementary approaches for building a selectivity estimator that is accurate, compact, and efficient to train and predict. Our proposed approach operates in two phases. During the offline phase, we train a DL model that is then used for predicting the selectivity of queries in the runtime phase.

Selectivity Estimation through Summary Data Structure Embeddings. Our first approach Astrid-Embed revisits the traditional strategy for answering selectivity queries using summary data structures and alleviates the storage issue. Popular data structures such as tries can be dramatically larger than the underlying text itself [61]. Hence, prior approaches prune the dataset to an acceptable size but incur a penalty in accuracy. We train a DL model that *learns* to estimate the selectivity of each of the substrings in the summary data structure. If the model is accurate and compact, then one can use it instead of the verbose summary data structure.

Despite the conceptual simplicity, implementing this idea requires a number of innovations. Since DL models can only process numeric input, we have to learn an embedding to map each string in the summary data structure into a real-valued vector. We cannot directly use pre-trained embeddings such as word2vec [54], fastText [4] and their ilk. These methods are designed for natural language processing and seek to assign similar embeddings to two words if they occur in similar contexts in natural language text, often. However, this approach provides sub-optimal results for selectivity estimation. Instead, we propose novel semantics for an embedding that is based on selectivity and query type (such as prefix, suffix or substring queries). Intuitively, we wish to assign similar embeddings to two words if they satisfy similar queries and have similar frequency. We propose an efficient algorithm to learn such embeddings and use these embeddings to train a DL model for selectivity estimation.

Selectivity Estimation through Neural Language Models. Our second approach is based on the insight that one can modify language models for the purpose of selectivity estimation. Intuitively, a *language model* [32] provides a mechanism to compute the probability of the next word given the previous words. Hence, given a sequence of words $\{w_1, w_2, \ldots, w_m\}$, it can output its probability. We build a language model that operates at the character level [34] instead of the word level. Hence, given the query q as sequence of characters q, the model can be used to estimate the probability that the query sequence is generated. Consider a query q = '*abc%*' that seeks to estimate the number of entries with prefix 'abc'. The selectivity of the query can be computed using the chain rule

$$p(abc) = p(a) \times p(b|a) \times p(c|ab) \tag{1}$$

We train a neural language model to learn the conditional probability distributions in an compact manner. To the best of our knowledge, our work is the first to deploy neural language models for string selectivity estimation. As we shall describe in Section 5, this approach works well for prefix queries. For substring queries, this results in a major underestimate. We modify the objective function of the neural language model so that it can be used for substring selectivity estimation. We also propose an efficient training algorithm for this modified objective function.

2 PRELIMINARIES

Let $\mathcal{A}$ be a finite alphabet. We have a relation $R = \{s_1, s_2, \ldots, s_n\}$ where each $s_i \in \mathcal{A}^*$. There are n strings in the relation.

Queries with String Predicates. SQL supports two wildcard characters % and _ for specifying string patterns. The percent and underscore allows for substitution of one or more characters in a string. A query "LIKE %sam%" matches all strings that contain the substring 'sam'. The query "LIKE s_m" matches all strings that contain three word strings whose first letter is s and last letter is m. One can express a wide variety of queries using these wildcard characters. A *prefix* query matches all strings that start with abc will be specified as "LIKE abc%". The *suffix* query "LIKE %abc" matches all strings that end with abc. The *substring* query "LIKE %abc%" matches all strings that contain the word "abc".

q-grams. Let $s \in R$ be a string of characters whose length is $|s|$. We denote the i-th character of s as $s[i]$ and the substring from its i-th to j-th character as $s[i, j]$. Given a positive value q, a q-gram of s is a set obtained by sliding a window of length q over s. For example, if $q = 2$ and $s = sam$, the q-grams are {sa, am}.

Performance Measures. Let Q be a string predicate query and $Sel(Q)$ represents the size of the strings matched by Q. We use the normalized selectivity between $[0, 1]$ by dividing the result size by n, the number of strings. Let the estimate provided by selectivity estimation algorithm be $Est(Q)$.

[1]Astrid stands for **A**ccurate **Str**ing Selectivity by **D**eep Learning

We use *q-error* for measuring the quality of estimates. Intuitively, q-error describes the factor by which the estimate differs from true selectivity. This metric is widely used for evaluating selectivity estimation approaches [20, 36, 45, 46, 55, 79] and is relevant for applications such as query optimization where the relative ordering is more important [45]. We do not consider the use of relative error due to its asymmetric penalization of estimation error [20, 55] that results in models that systematically under-estimate selectivity. This is due to the fact that the worst possible error for under-estimation is 1 while it is unbounded for over-estimation.

$$\text{q-error} = \max\left(\frac{Sel(Q)}{Est(Q)}, \frac{Est(Q)}{Sel(Q)}\right) \tag{2}$$

3 BACKGROUND

In this section, we provide a concise summary of techniques that are relevant for the development of Astrid.

3.1 Summary Data Structures for String Selectivity Estimation

Traditional approaches for answering string selectivity queries involve two key components: (a) summary data structure that efficiently calculates the selectivities of various strings and (b) statistical techniques to approximate selectivity of input queries when they are not present in the data structure. We focus on suffix trees that is widely used to answer substring/suffix queries. However, our approach is agnostic to any summary data structure that could compute the frequency of all substrings efficiently.

Suffix Tree Primer. A suffix tree [18] is a trie based data structure widely used for estimating string selectivity [10, 28, 40]. A suffix tree for a *single* string s of length $|s|$ has $|s|$ nodes each of which are labeled by substrings of s. No two edges starting from a node share a common prefix. A suffix tree for string s can be built in $\Theta(|s|)$ time. Given a substring t, one can verify if it is present in s in $\Theta(|t|)$ time. One can search for a regular expression in $\Omega(|s|)$ [2].

Pruning Suffix Trees. A generalized suffix tree [18] is built using a set of words S and can represent all suffixes belonging to S along with their frequencies. Figure 1 shows a generalized suffix tree for strings {'sam', 'jim', 'tim', 'time' } with frequencies 2000, 3000, 2000 and 1000 respectively. As the tree could be much larger than the underlying dataset itself, it is common to prune out the nodes whose substring frequency is below a threshold. Since, the vast majority of substrings have very low frequency, this heuristic often provides good reduction in space needed. In Figure 1, we prune all nodes whose frequency is 2000 or lower.

Selectivity Estimation with Pruned Suffix Trees. Given a query string q, we wish to estimate the selectivity as accurately as possible. If q is present in the tree, then we can report the exact selectivity. If q is not present, we need to generate an approximation of the selectivity. There has been extensive work on identifying partial matches of q that exist in the suffix tree and use it to approximate $Sel(q)$. A simple approach decomposes q into disjoint substrings $q_1, q_2, \ldots, q_k$ where each of q_i exists in the suffix tree. Under the independence assumption [40], one could estimate $Est(q) = \prod_{i=1}^{k} Sel(q_i)$. If $q = VLDB$ and $q_1 = VL$ and $q_2 = DB$, then $Est(VLDB) = Sel(VL) \times Sel(DB)$. Another popular approach relies on the Markovian assumption where the probability of q_i is only dependent on its immediately preceding token q_{i-1}. So $Est(q) = Sel(q_1) \times \prod_{i=2}^{k} Sel(q_i|q_{i-1})$. Continuing the above example, $Est(VLDB) = Sel(VL) \times Sel(DB|VL)$ where $Sel(DB|VL)$ returns the selectivity of DB that were preceded by VL. A slightly sophisticated approach from [27] splits q into *possibly overlapping* substrings $q_1, \ldots, q_k$ each of which exist in the suffix tree. We denote by $q_i \oslash q_{i+1}$ the *maximum overlap* (MO) which is the largest suffix of q_i that is also the prefix of q_{i+1}. [27] proposes that,

$$Sel(q) = Sel(q_1) \times \prod_{i=2}^{k} \frac{Sel(q_i)}{Sel(q_{i-1} \oslash q_i)} \tag{3}$$

If $q = VLDB$ and $q_1 = VL$, $q_2 = LD$ and $q_3 = DB$, then $Sel(VLDB) = Sel(VL) \times \frac{Sel(LD)}{Sel(L)} \times \frac{Sel(DB)}{Sel(D)}$. Other methods such as [8] make increasingly sophisticated statistical assumptions with varying effectiveness. We evaluate against all these methods in Section 6.

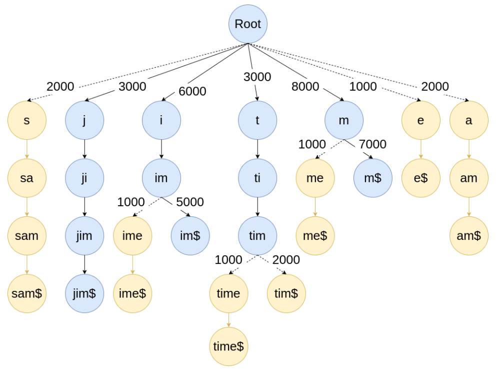

Figure 1: Generalized (uncompressed) Suffix tree for strings {'sam', 'jim', 'tim', 'time' }

3.2 Word Embeddings and Language Models

Embeddings for Strings. Deep learning based approaches for selectivity estimation [20, 49, 79] tackle string selectivity queries through embeddings. Popular embedding approaches include fastText [4], word2vec [54], and many others. They are trained on a large corpus of text such as Wikipedia and output a vector space where each word in the corpus is represented by a real valued vector. By default, each word is represented as a dense 300 dimensional vector (aka embedding). Let $W = \{w_1, w_2, \ldots, w_i, \ldots, w_k\}$ be a sequence of words. Given W and a word w_i, the context is the set of surrounding words $w_{i-C}, \ldots, w_{i+C}$ for some context window of length C. Popular approaches use context to predict w_i or use w_i to predict its context. The former is called continuous bag of words (CBOW) while the latter is called as skip-gram. Algorithms for learning embeddings ensures that the vectors for words that occur in similar context – such as SIGMOD and VLDB – are close to each other in the embedding space.

Language Models. We can abstract language models as providing the functionality to compute the probability of the next word given the preceding words (if any). The language model could compute the probability for $P(w_i|w_1, w_2, \dots, w_{i-1})$. For any given sequence of words, we can estimate its joint probability as

$$P(W) = \prod_{i=1}^{k} P(w_i|w_1, \dots, w_{i-1}) \tag{4}$$

Word embeddings could be considered as simplified language models that could predict a word given a context. An interesting perspective that we later leverage is to view the traditional approaches for selectivity estimation as a space constrained approximation of language models. We provide additional details in Section 5.

Neural Language Models (NLM). Neural networks have become increasingly popular for language modeling. A classical approach first proposed in [3] builds a NLM using three steps. (a) Each word in the vocabulary is associated with an embedding; (b) the joint probability function of word sequences are estimated using the embeddings; and (c) train a DL model to simultaneously learn both the embeddings and the probability function. The language model is trained over a large corpus of text and learns the joint probability distribution. It is not necessary to explicitly storing the counts which obviates the need for summary data structures.

4 SELECTIVITY ESTIMATION THROUGH EMBEDDINGS

In this section, we describe Astrid-Embed that can estimate selectivities of string predicate queries using a task aware embedding.

4.1 Overview of Astrid-Embed

Need for a Hybrid Approach. The traditional and embedding based approaches described in Section 3 have some disadvantages that limit the accuracy of estimations. The traditional approaches pruned the suffix tree to satisfy space constraints and suffer from the resulting approximation that is hard to correct even with sophisticated statistical models. Intuitively, they sought to retrofit a fixed statistical model on top of a pruned suffix tree. DL models have the ability to learn the complicated relationship between substrings and their selectivities. Word embeddings are designed to perform well for NLP tasks and naively using them for selectivity might provide sub-optimal results. For example, two words that co-occur in similar context might have similar embedding. However, in the selectivity setting, these two similar strings could have arbitrarily different selectivities. Using similar embeddings for these words as input to a selectivity estimator will produce poor estimates. The similarity could also be query dependent. One could learn better embeddings by using the summary data structures.

Blended Approach of Astrid-Embed. The key insight of Astrid-Embed is to revisit the principles of traditional approaches and leverage deep learning based primitives to alleviate both their pain points. Intuitively, we train a DL model by feeding it all the substrings from the unpruned suffix tree along with their true selectivities. Once an accurate and compact DL model has been trained, it could be used as an approximation of the entire suffix tree without any need for ad-hoc statistical models as used in the traditional approaches. This hybrid approach can produce dramatically better estimates with limited space.

Components of Astrid-Embed. There are three major components – summary data structure, embedding learner and a selectivity estimator. Our proposed approach (see Figure 2) is very generic and extensible. Each of these components could be replaced with a more sophisticated model producing improved results. The summary data structure is responsible for efficiently computing the selectivities of all the relevant strings from the dataset. This could be query type specific. For example, tries [18] could be used for estimating prefix queries while suffix trees [18] are an apt choice for substring/suffix queries. One could also use any other advanced data structures for this purpose. For ease of exposition, we will focus on suffix trees. The embedding learner uses the summary data structure to learn embeddings that are optimized for the selectivity estimation task. Finally, the selectivity estimator is trained over a dataset of (substring, selectivity) pairs where each substring is encoded using embeddings learned for selectivity estimation.

4.2 Embeddings for Selectivity Estimation

Desiderata. Embeddings for selectivity estimation must satisfy two key requirements. First, we require that two strings are assigned similar embeddings if they are *syntactically* similar and have similar selectivities. Note that this could be query specific. For prefix queries, the two strings 'abcxyz' and 'abcdef' could be considered similar while they will not be for suffix queries. Second, we require that our embeddings operate at the granularity of characters and q-grams. Word based embeddings are not appropriate as the queries could be based on any substring in a given word.

Baseline Approaches. We briefly describe two natural baseline approaches that we empirically evaluate in the experiment section. One could use a character based embedding approach such as fastText [4]. Intuitively, these approaches express the embedding of a word by using the embeddings of q-grams. For example, if $q = 2$, then the string *VLDB* is treated as bag of 2-grams *{BOW V, VL, LD, DB, B EOW}* where *BOW, EOW* are special tokens denoting the beginning and end of a word. Then the embedding for *VLDB* is computed as the *normalized sum* of the embeddings for its q-grams [4]. By default, fastText constructs the q-gram for $q = 3$ to 6. For the selectivity estimation task, one could set q to a different range such as $1-3$. Another approach is based on the observation that the suffix tree could be considered as a graph where each node corresponds to a distinct substring. One could then leverage the techniques for *node* embeddings such as DeepWalk [60]. Both these approaches have some disadvantages. The character based approach – while an improvement – is still not selectivity aware. The graph embedding approach gives even better results but is quite expensive to compute [77] and could be further improved by our techniques.

Formalizing the Embedding Semantics. Let the dataset contain N distinct substrings $\{s_1, s_2, \dots, s_N\}$. Given a token s_i, let C_i be the set of similar tokens while D_i be the set of tokens that are dissimilar to t_i. All strings in C_i are similar to s_i both syntactically and also in selectivity. Our objective is to learn embeddings for each token s_i such that it is similar to substrings C_i but dissimilar to that of D_i. There are now two challenges - (a) for each s_i how can we efficiently retrieve C_i and D_i; (b) how to formalize the objective function that achieves the desired semantics?

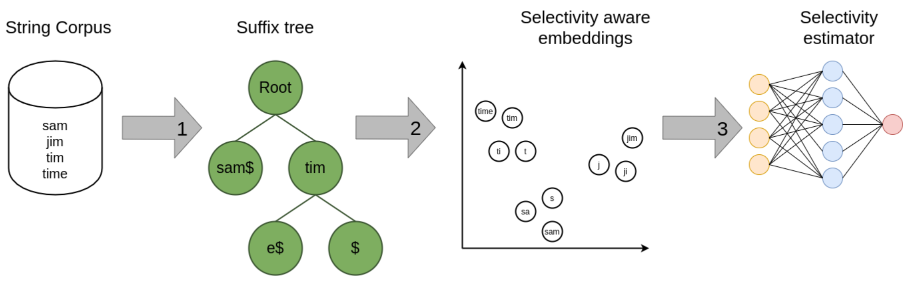

Figure 2: Major components of ASTRID-EMBED

Obtaining Similar and Dissimilar Tokens. Intuitively, we want C_i to be the set of tokens that are similar to s_i in terms of selectivity and query type. The strings 'abcxyz' and 'abcdef' should be considered more similar than the strings 'abcxyz' and 'defghi'. If $Sel(ab) = 1000$ and $Sel(abc) = 900$ and $Sel(abd) = 100$, then the strings 'ab' and 'abc' are considered to be more similar than 'ab' and 'abd'. Given a string s_i, a simple approach would be to design a ranking function that takes both these factors into account and order every other word using this function. In addition to being inefficient, this also creates a new set of problems requiring the design of an appropriate ranking function and a well chosen threshold to identify similar and dissimilar tokens. It is possible to leverage the suffix tree to do a better job. Given a node s_i in the suffix tree, all its children share the same prefix and hence are all *syntactically* similar. Then the children could be ordered based on how similar their frequencies are to s_i. It is possible to generalize this for a larger neighborhood. Suppose that we perform L random walks starting from each node in the suffix tree. Given a node s_i with k children $\{s_{i,1}, \ldots, s_{i,k}\}$, we chose the next node according to:

$$p(s_i, s_j) = \begin{cases} \frac{1}{(k+1)} & if\, s_j \text{ is parent of } s_i \\ \frac{Sel(s_j)}{\sum_{l=1}^{k} Sel(s_{i,l})} \times \frac{k}{k+1} & s_j \in Children(s_i) \end{cases} \tag{5}$$

This transition probability distribution defined in Equation 5 has both the desirable properties. Due to suffix tree properties, tokens that are similar according the query-type are also in the immediate neighborhood of s_i. The transition probability is defined so that the walk from s_i preferentially chooses the child whose selectivity is most similar to that of s_i. Furthermore, it does not need any fixed threshold for cut-off. Figure 3 shows an example.

Improving the Efficiency. This proposed approach has some obvious connections to graph embeddings. However, treating it as a graph embedding problem gives sub-optimal results. Instead, we use the transition probability to quickly compute similar C_i and dissimilar D_i substrings. Due to the stochasticity of Equation 5, one might require multiple random walks from each node that could make this process very inefficient. Other simple tricks such as using 1-hop or 2-hop neighbors of s_i as C_i also produces sub-optimal results. We evaluate these approaches in Section 6.

It is possible to achieve a dramatic speedup by natually treating the suffix tree as a − *tree*. We propose a simple heuristic that lower bounds the probability of reaching between any two nodes and can be computed in *linear* time. Consider two nodes s_i and s_j. The

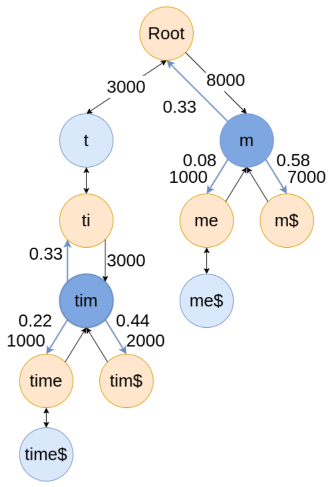

Figure 3: A portion of the suffix Tree from Figure 1 annotated with transition probabilities based on Equation 5.

probability of starting from s_i and reaching s_j using the transition probability defined in Equation 5 provides a notion of similarity between them. Even in a tree, there could be multiple possible walks between s_i and s_j (i.e. in walks both vertices and edges might repeat). Hence, the total probability is the sum of all the probabilities of each of these walks (which could be exponential in number).

One could lower bound this probability by computing the *path* between the two vertices in the shortest number of edges. A path does not allow either vertices or edges to repeat. Since the path is just one of the many other possible walks between the two nodes, the probability of this path is a valid approximation of the true probability. Of course, repeating this for each pair will require $O(N^2)$ time complexity. We further reduce this by using the concept of *landmarks* that has been used to efficiently compute the approximate shortest distance in a graph [1]. Specifically, we identify a set of K nodes in the suffix tree dubbed landmarks $L = \{l_1, l_2, \ldots, l_K\}$. For each node l_k, we can compute the probability of reaching l_k from each of s_i in linear time by performing a BFS starting from l_k. Storing these probabilities requires a linear storage. Given any two pair of vertices s_i and s_j, we compute the bound for probability as

$$\max \{p(s_i, l_k) + p(l_k, s_j) \qquad \forall l_k \in L\}$$

. This could be computed in $O(K)$ time complexity.

Learning Embeddings through Triplet Loss. For each node s_i, let C_i and D_i be the set of similar and dissimilar strings. We wish to learn embeddings for s_i such that the embedding is closer to the substrings in C_i and farther from substrings of D_i. Given a triplet w_i, c_i, d_i where $c_i \in C_i$ and $d_i \in D_i$, the triplet loss [64, 76] could be defined as

$$\max\left(||f(w_i) - f(c_i))||^2 - ||f(w_i) - f(d_i)||^2 + m, 0\right) \quad (6)$$

where $f(s)$ is the embedding for s and m is a positive value called "margin". This formulation will ensure that (a) distance between w_i and d_i is larger than the distance between w_i and c_i and (b) distance between w_i and d_i is larger than m. There are a number of approaches to efficiently learn an embedding under under triplet loss [21, 64, 76]. Using a simpler contrastive loss [19] that minimizes the distances of similar word and maximizes the distance of dissimilar words provides sub-optimal results.

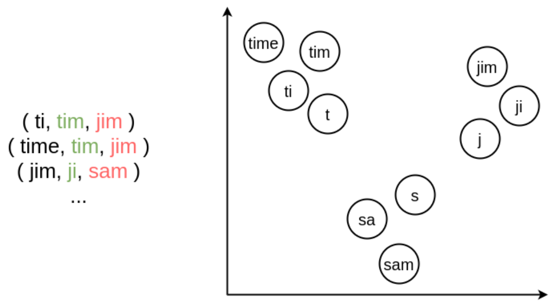

Figure 4: Illustration of Triplet Loss

Heuristics for Triplet Selection. There has been extensive work on identifying good triplets for training [76, 80]. These include selecting based on pairwise relevance scoring [72], top-K pairs within a training batch [73], selecting challenging triplets [24] and so on. As observed in [80], these strategies could result in triplet selection bias that reduces the accuracy of selectivity estimates. We use a distance weighted sampling approach inspired by [76]. First, we apply min-max scaling on the computed probabilities so that minimum value is 0 and the maximum is 1. Then for each node, we set it as an anchor and select positive examples with probability proportional to the normalized value. The negative examples are selected with probability proportional to the inverse value with a weighted clipping to avoid noisy samples [76]. This approach results in better set of triplets than other complex heuristics.

4.3 Selectivity Estimator Using Embeddings

The final step is to train a DL model that accepts a string predicate as an input and outputs its selectivity. The training data consists of a set of substrings from the suffix tree along with their true selectivities. We represent each substring through the embedding learned by triplet loss function. Estimating the selectivity could be considered as a regression problem where the embeddings are the independent features with selectivity being the dependent feature. The relationship between embeddings and the selectivity is complex and is more suited to a DL model. A similar observation has been made in prior works such as [20, 36, 42].

Encoding Selectivities. Each token s_i is associated with the normalized selectivity between 0 to 1. Typically, the distribution of selectivities are very skewed with most of the tokens being infrequent. Directly training the DL model on the skewed data produces sub-optimal results. We reduce the skew by applying log transformation followed by min-max scaling. This has been applied in a number of prior works including [14, 20, 36]. Given a selectivity of 0.00001, the log transformation produces 5. The logarithmic selectivities are much less skewed than the original selectivities. Min-max scaling ensures that log transformed selectivities are rescaled to the range of $[0, 1]$ with the smallest (resp. largest) selectivity being assigned to a value of 0 (resp. 1).

DL Architecture. We evaluated a number of DL architectures and found that a simple fully connected feed forward model with three layers works well. The first layer is the *embedding layer* that looksup the embedding of the input string. The next two layers are used to learn an intermediate representation. The final layer is the softmax layer for producing the probability distribution.

Loss Function. We use q-error metric defined in Section 2 for measuring the accuracy of our model. Specifically, we seek to minimize the mean q-error defined over all substrings in the training set Q.

$$\text{q-error}(Q) = \frac{1}{|Q|}\sum_{i=1}^{|Q|} \max\left(\frac{q_i}{\widehat{q_i}}, \frac{\widehat{q_i}}{q_i}\right) \quad (7)$$

Once the model is trained, we could discard the suffix tree. Given a new query, it could be encoded using the learned embeddings and then passed to the model. The output of the model is then converted to true selectivity by applying inverse of min-max and log scaling.

5 SELECTIVITY ESTIMATION THROUGH NEURAL LANGUAGE MODELS

Next, we develop a selectivity estimator Astrid-NLM by leveraging the connection between selectivity estimation and language modeling. To the best of our knowledge, our work is the first to do so. While neural language models (NLM) can be easily adapted to answer prefix and suffix queries, they require a novel modification to training dubbed *state-reset* for handling substring queries.

5.1 From Suffix Trees to Language Models

Need for an Alternate Approach. A key bottleneck in Astrid-Embed is the need for constructing the suffix tree. While it could be done in time linear to the size of the corpus, the suffix tree itself could be extremely large. It could require as much as 20x storage than the size of the text being indexed [61]. Furthermore, the large number of valid substrings makes building DL models for learning embeddings and selectivity estimation quite expensive. It is desirable to have an alternate approach that can provide accurate selectivity estimates by leveraging the statistical correlation between tokens *without* the need for building suffix trees.

Prior Works as Rudimentary Language Models. We take a fresh look at the traditional approaches for string selectivity estimation. Given a unpruned suffix tree, selectivity estimation is as simple as a lookup. The challenge comes when the suffix tree is pruned. If a query q is not present in the suffix tree, then it is decomposed into smaller substrings $q_1, q_2, \dots, q_k$ that exist in the suffix tree, and $Sel(q)$ is approximated as a function $f(q_1, q_2, \dots)$. In Section 3.1, we described three such functions based on different statistical assumptions – complete independence, Markov independence and maximal overlap based independence.

Recall from Section 3.2 that language models also estimate the probability of a string by decomposing it to shorter substrings. From this perspective, one could think of traditional approaches for selectivity estimation as simple language models that make certain statistical assumptions. When these assumptions hold on the dataset, they give good results. If not, they provide highly inaccurate results. Furthermore, these language models are static, ad-hoc and not *learned* for each dataset. We advocate the use of neural language models as a principled approach to approximate the selectivities. Replacing ad-hoc language models with a neural language model (NLM) brings a number of advantages. We could piggy back on the periodic breakthroughs of the NLP community that seeks to build ever more accurate language models. The language models could be fine-tuned and learned specifically for each datasets for the selectivity estimation task. Most importantly, it is possible to learn language models *without* building the suffix tree.

5.2 Selectivity Estimation Through Neural Language Models

Given a sequence of tokens $q =< q_1, q_2, \dots, q_k >$, language models provide the probability of q as

$$p(q) = p(q_1, q_2, \dots, q_k) = p(q_1) \prod_{i=2}^{k} p(q_i | q_1, \dots, q_{i-1}) \qquad (8)$$

The tokens could either be words or characters. Using a character based language model is more natural for our task. Hence, given a sequence of characters, the language model must provide the probability of that sequence. We can immediately see that the probability is equivalent to the selectivity estimate of the prefix query ‘q%’. By training a language model on the reversed corpus, one could similarly obtain the selectivity for a suffix query ‘%q’. Of course, one could also use language models that can process text in a bi-directional manner. Finally, the estimate provided by Equation 8 is a lower bound of the selectivity for the substring query ‘%q%’.

Training Astrid-NLM**.** We next describe how to adapt the training of neural language models for the task of selectivity estimation. We refer the reader to [32] for further details about training NLM. We are given a bag of strings $R = \{s_1, s_2, \dots, s_n\}$ where each string s_i is a sequence of characters of arbitrary length. We begin by creating a large corpus C of length N by concatenating all s_i. For example, if $R = \{sam, jim, sam\}$, we get $C = \{BOW, s, a, m, EOW, BOW, j, i, m, EOW, BOW, s, a, m, EOW\}$ where BOW, EOW are special tokens indicating beginning and end of the word. Abstractly, $C = \{x^{(1)}, x^{(2)}, \dots, x^{(N)}\}$ where $x^{(i)}$ denotes the i-th sequence in corpus C. The NLM processes C sequentially one character at a time and seeks to accurately estimate the probability distribution for the next character $y^{(t)}$ given the previous characters. For example, the alphabet for R for the running example is $\mathcal{A} = \{a, i, j, m, s\}$. So, given the sequence BOW, s, a, it generates a probability distribution that the next character is one of $\{a, i, j, m, s\}$. A well trained NLM should give a high probability to m than the other characters. It must be penalized otherwise. The loss between the probability distributions for predicted $\hat{y}^{(t)}$ and actual $y^{(t)} = x^{(t)}$ next character is computed using cross-entropy (CE) [32] defined as

$$\mathcal{L}_{NLM}^{(t)} = CE(y^{(t)}, \hat{y}^{(t)}) = - \sum_{j=1}^{|\mathcal{A}|} y_j^{(t)} \log \hat{y}_j^{(t)} \qquad (9)$$

where $\hat{y}_j^{(t)}$ and $y_j^{(t)}$ are the predicted and actual probability that j-th character in the alphabet occurs at position t. The objective function of the NLM [32] is to minimize the cross entropy error over the entire corpus C.

$$\mathcal{L}_{NLM} = \frac{1}{N} \sum_{t=1}^{N} J^{(t)} = -\frac{1}{N} \sum_{t=1}^{N} \sum_{j=1}^{|\mathcal{A}|} y_j^{(t)} \log \hat{y}_j^{(t)} \qquad (10)$$

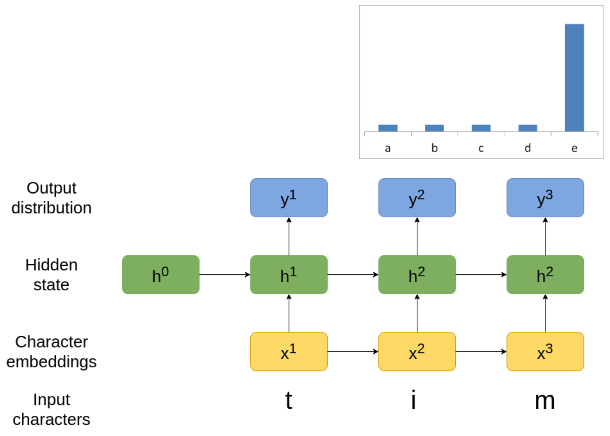

Figure 5: Illustration of a simple Neural Language Model used in Astrid-NLM**. The bar chart provides the probability distribution** $p(\cdot|BOW, t, i, m)$**. The character** e **seems to be most likely one.**

DL Architecture for Astrid-NLM**.** A number of DL architectures such as LSTM [23], GRU [11] have been proposed for neural language modeling. Almost all of these are based on recurrent neural networks that are well suited for processing sequence data with the ability to handle long term dependencies. Intuitively, these models work by processing a character $x^{(t)}$ and outputs a state $h^{(t)}$. This state is used to output the probability distribution for the next character $y^{(t+1)}$. Intuitively, the state $h^{(t)}$ remembers the relevant details about the previously seen characters if any such as BOW, s, a in the above example. This process is then repeated till the end of the corpus C. Concretely,

$$p(q) = p(q_1, q_2, \dots, q_k) = \prod_{i=1}^{k} p(q_i | q_1, \dots, q_{i-1}) \qquad (11)$$

$$= \prod_{i=1}^{k} p(q_i | h_i) \qquad (12)$$

In other words, the state h_i acts as a proxy for the previously seen characters $q_1, \dots, q_{i-1}$. The NLM model then uses h_i to estimate the probability that q_i is generated in the state h_i. Whenever the NLM processes BOW, it is reset to an initial blank state h^0 – an all zero vector. We use Gated Recurrent Unit (GRU) [11]. It is a popular variant of recurrent neural networks that uses gating mechanisms for efficiently learning the language model. GRUs are faster to train than other sophisticated models such as LSTMs and provides comparable accuracy in language modeling [12, 31]. Our proposed approach is agnostic to the specific architecture used. We evaluate the performance of popular architectures in the experiments.

5.3 Accurately Answering Substring Queries

The previously described NLM model can be directly used to estimate the selectivities of prefix queries. However, it produces sub-optimal results for substring queries. To see why, consider our running example, $R = \{sam, jim, sam\}$. Suppose we wish to estimate the selectivity of the substring query '%a%'. While we expect the NLM to produce an estimate around 2/3, it produces a value closer to 0. Specifically, it tries to estimate $p(a) = p(a|h_0)$. Since the NLM did not see any string in R that starts with a, this probability will be estimated to be around 0. The fundamental reason is that NLM processes a string $s_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,k}\}$ from the first character to the last. Hence, it only learns the probabilities corresponding to the prefix queries

$$p(s_{i,1}), \quad p(s_{i,2}|s_{i,1}), \quad \ldots, \quad p(s_{i,k}|s_{i,1}, \ldots, s_{i,k-1})$$

This restriction is natural in a natural language processing setting where the NLM would process the string from the beginning for various tasks in language modeling. It is very unlikely that one has to process a fragment of a sentence. In contrast, this is a very important requirement for answering substring queries. They must be able to estimate the probability of any arbitrary substring in the corpus. To the best of our knowledge, none of the existing NLM could be used for estimating substring selectivities. Given the promising results for prefix queries, we propose a novel adaptation of the training of NLM that alleviates this issue.

Adapting NLM Training for Substring Queries. We wish to retain the ability of NLM to accurately learn selectivities of prefix queries but extend it to substring queries. We observe that the poor performance of substring queries is exclusively due to the fact that the default training of NLM does not try to learn conditional probabilities for substring queries. In our running example, the reason for poor estimate of the substring query '%a%' is that $p(a) = p(a|h_0)$ was not accurately defined. For $R = \{sam, jim, sam\}$, only $p(s|h_0)$ and $p(j|h_0)$ gives non-zero result as they were the only two characters that occur at the beginning of any string.

Our key insight is that the NLM could learn these conditional probabilities through a simple change of the training, which we call *state-reset*. Consider the following change. While processing the corpus C sequentially, we randomly reset h_t to h_0 with some small probability. Let us consider the sequence BOW, s, a, m, EOW. The NLM processes $p(s|h_0)$ and produces a new state h_1. In an unmodified NLM, one would estimate the probability $p(a|h_1)$. However, suppose that our random process resets h_1 to h_0 before processing a. In other words, in this case, we pretend the input sequence was BOW, s, BOW, a, m, EOW. Now, the NLM will try to process $p(a|h_0)$ which will result in a better estimate for the substring query than unmodified NLM.

Concretely, *state-reset* is impacted by a hyperparameter α that controls the probability of resetting the state. Setting $\alpha = 0$ is equivalent to training a classical NLM. Setting a large α results in the a higher likelihood of resets. While this achieves a better performance for substring queries, it degrades the performance of prefix queries and longer substring queries. The concept of *state-reset* could be considered analogous to dropout [65]. Nodes in a feed forward neural networks are randomly dropped with a dropout probability. While dropout makes the training process noisy, it acts as an effective regularizer and improves generalizability [65]. While similar in spirit, our rationale for state reset is to support substring queries. While there has been some work on dropouts for recurrent networks [53, 81], they only apply to non-recurrent connections to avoid impeding the memorization capabilities. In contrast, we explicitly reset the recurrent connections so that the NLM learns how to estimate the selectivity of substrings in the corpus.

6 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the following three questions: (a) How does ASTRID compare against prior approaches from databases and NLP for string selectivity estimation? (b) How much benefit does our proposed modifications of embeddings and neural language models achieve? (c) How can we compare the performance of ASTRID-EMBED and ASTRID-NLM?

6.1 Experimental Setup

Hardware and Platform. All our experiments were performed on a NVidia V100 GPU. The CPU is a quad-core 2.2 GHz machine with 16 GB of RAM. We used PyTorch for building the DL models. The implementation of ASTRID can be found here[2].

Datasets. We conducted our experiments on five benchmark datasets from three different sources (DBLP [75], IMDB [45] and TPC-H) that have been used extensively in the evaluation of prior work on string selectivity. The dataset D-AT uses the article titles from DBLP bibliography and was previously used in [78],[44],[43],[8],[47]. The dataset D-AN is based on the author names for the publications and has been used in [78],[44],[43],[8]. Datasets I-MT and I-AN correspond to movie titles and actor names from IMDB dataset and used in evaluation of [78],[44],[43],[30]. Finally, dataset T-PN is based on part names from TPC-H and has been used in evaluation of [40]. The statistics of these datasets can be found in Table 1.

Algorithms for Selectivity Estimation. We use an embedding size of 64 for ASTRID-EMBED and train it with a batch size of 128. The margin m was set to a value of 0.2. Please refer to Section 4.3 for additional details about the DL model. For ASTRID-NLM, we used a GRU with 2 layers and 650 hidden states that has been previously used for neural language modeling [81]. We then modify it to include a state-reset hyperparameter as $\alpha = 0.1$. The NLM was trained for 40 epochs with a batch size of 128. We used truncated backpropagation for training with 20 time steps. For each dataset, we fixed the space budget to ensure fair comparison. This value is set to be the size of the summary data structure containing only

[2]https://github.com/saravanan-thirumuruganathan/astrid-string-selectivity

Source	Column	Abbrv	#Entries	#Distinct	#Prefixes	#Suffixes	#Substrings
DBLP	Article Titles	D-AT	50K	49.5K	99K	80K	381K
DBLP	Author Names	D-AN	111.1K	69.6K	136K	132K	360K
IMDB	Movie Titles	I-MT	2.52M	1.46M	1.16M	1.21M	3.53M
IMDB	Actor Names	I-AN	4.16M	3.58M	1.71M	1.73M	4.36M
TPC-H	Parts Name	T-PN	200K	200K	22K	21K	453K

Table 1: Statistics for benchmark datasets used for evaluating Astrid

the top-10% most frequent entries. As we shall show later, a lower value resulted in poor performance of the baseline algorithms while a larger value did not materially improve the performance.

Query Workload. We used the appropriate summary data structure to collect the list of prefixes, suffixes and substrings. When training the selectivity estimator, we randomly partitioned the strings and used 50% of them for training. Within the 50% of the training data, we used 10% as a validation dataset to tune the hyperparameters. For evaluation, we generated random positive and negative strings as used in prior work [10, 27, 30, 40, 43, 44, 47, 78]. The number of queries in the testing dataset is the same as the training dataset. Astrid-NLM was trained on the entire dataset and uses the same query workload as Astrid-Embed.

Performance Measures. We used q-error defined in Equation 2 for measuring the estimation quality. A q-error of 1 corresponds to perfect estimate. A q-error of j corresponds to an under- or over-estimate by a factor of j and so on. We display the median and 90-th percentile of q-error. q-error is undefined for negative queries (whose selectivity is 0) as it involves a ratio of estimated and actual selectivities. Hence, we use a smoothed version of q-error where a small $\epsilon = 1e - 2$ is added to both numerator and denominator.

6.2 Comparison Against Baselines

In our first set of experiments, we compare the efficacy of Astrid with popular baseline estimators from traditional and deep learning based approaches. Each of these are widely used approaches with KVI [40] being one of the earliest work to use suffix trees for string selectivity. An improved estimator MO was given by [27] that uses maximum overlap. Both these approaches could under estimate the selectivity. So [8] proposed a CRT estimator to obtain better estimates by finding a shorter string whose frequency is similar to a given string. We evaluated the embedding based approach from Neo [49] and a traditional q-gram based language model [32]. We investigate other embedding approaches in Section 6.3 We ensured that the pruned suffix tree and the models of Astrid and other deep learning based approaches have the space budget.

The overall results can be found in Table 2. Not surprisingly, Astrid outperforms both traditional and DL based baselines. The traditional approaches retrofit an ad-hoc language model to estimate the selectivities from the suffix tree. We can see that the more sophisticated models perform better. KVI is the simplest one and has the least accuracy. MO leverages additional information in the form of maximal overlap that improves the estimates. Finally, CRT leverages short distinctive strings whose selectivity is similar to the query string. This heuristic provides the best accuracy overall. Interestingly, Astrid also outperforms two representative DL based approaches. Current DL selectivity estimators handle strings by computing a task-agnostic embedding. Neo-Embed [49, 56] uses a NLP based approach for embedding. Astrid also outperforms q-gram based non-neural language model. It was implemented using the NLTK's LanguageModel module [57]. We used a frequency based cutoff to ensure that the language model fits the space budget. The selectivity of q-grams are estimated using Kneser-Ney smoothing – one of the most sophisticated and widely used smoothing approach that provides better selectivity estimates [9, 32]. Similar to CRT and MO baselines, the performance of this approach deteriorates for longer queries as their frequency needs to be *interpolated* based on the q-grams whose frequencies are stored. The outperformance of Astrid-Embed over Astrid-NLM is not surprising as Astrid-Embed learns task specific embeddings and then uses a training dataset to learn a supervised selectivity estimator. In contrast, Astrid-NLM uses an unsupervised approach without building the suffix tree.

For the rest of the section, we only report the 90-th percentile of the q-error for different algorithms and datasets due to space limits.

Impact of Space Budget. Table 3 shows how the performance of the selectivity estimation algorithms are affected when the space budget are varied. We set the space budget as a proportion of the size of the summary data structure. By default, the budget is set by pruning the data structure so that only the most frequent 10% of the entries are retained. We vary this number from 1% to 25% for IMDB Actor Names (I-AN) dataset. Not surprisingly, q-error increases when the space budget is reduced and vice versa. The impact is especially notable for the non-neural baselines CRT and q-gram whose 90-th percentile q-error worsens by almost an order of magnitude. However, increasing the budget has comparatively limited impact on the non-neural models that quickly plateaus.

6.3 Evaluating Astrid-Embed

In this subsection, we investigate how the design choices made by Astrid-Embed affects its performance.

Impact of Embedding Size. The most critical hyperparameter for Astrid-Embed is the size of the embedding. The embeddings plays a key role in learning similarity between various substrings based the query type and frequency that is then used in the supervised DL model. A smaller embedding size produces a smaller model at the cost of expressiveness of the representation. In contrast, a larger embedding size allows Astrid-Embed to learn sophisticated correlation patterns and thereby achieve a higher accuracy. Table 4 shows the result of the experiment where we vary the embedding size. As expected, a smaller size produces a steep drop in accuracy. Increasing the embedding size reduces the q-error at a slower rate.

	D-AT		D-AN		I-MT		I-AN		T-PN	
	Median	**90th**	**Median**	**90th**	**Median**	**90th**	**Median**	**90th**	**Median**	**90th**
Astrid-Embed	1.12	4.29	1.28	3.86	1.17	3.98	1.24	3.85	1.32	5.84
Astrid-NLM	1.38	6.95	1.43	7.45	1.34	7.14	1.41	8.4	1.47	8.61
Neo-Embed	1.88	8.9	1.93	9.3	2.1	10.02	1.99	9.8	2.21	10.78
KVI	4.85	13.87	4.44	11.56	4.5	11.9	4.86	10.16	4.9	13.89
MO	3.93	9.3	4.14	10.8	3.9	9.3	4.16	10.84	4.26	10.62
CRT	3.11	8.7	3.81	9.09	2.67	8.8	3.8	10.22	3.9	9.68
q-gram-LM	2.33	7.4	2.88	6.18	3.8	8.32	3.7	9.9	2.68	10.62

Table 2: Comparison of Astrid against traditional and DL based baselines.

	1%	5%	10%	25%
Astrid-Embed	11.4	3.85	2.8	1.9
Astrid-NLM	16.8	8.4	6.1	4.8
CRT	129.1	10.2	8.4	7.1
q-gram LM	79.6	9.9	9.3	7.8

Table 3: Varying Space Budget (I-AN Dataset)

Our goal is to produce a compact model for Astrid-Embed and hence chose the default embedding size as 64 so as to balance the accuracy and model size.

	D-AT	D-AN	I-MT	I-AN	T-PN
32	11.73	8.76	15.77	16.55	9.38
64	4.29	3.86	3.98	3.85	5.84
128	4.08	3.44	3.46	3.54	5.16
256	3.64	3.09	3.38	3.04	4.91

Table 4: Varying size of embeddings

Impact of algorithm used for learning embeddings. In our next experiment, we investigate the impact of the specific algorithm used for learning the embeddings. By default, Astrid-Embed uses triplet loss for learning the embedding. We consider three alternatives. The first is fastText [4], a word embedding based approach that has the capability to express the embedding of a word based on its q-grams. This is especially relevant for selectivity estimation as the queries could be arbitrary substrings. We also compare a classical graph embedding algorithm DeepWalk [60] that learns embedding by performing uniform random walks over the suffix tree. Finally, we also contrasted triplet loss with a simpler contrastive loss [19]. Given a pair of substrings, one could label them as similar or not-similar. Based on this labeling, contrastive loss seeks to learn an embedding such that the Euclidean distance between the embeddings of similar substrings are smaller and dissimilar items to be larger. We used the same value of margin $m = 0.2$ for both the losses. Similarly, we use the same distance based selection strategy to pick triplets and pairs. Prior work such as [76] has shown that it is an effective heuristic for ranking losses.

The result can be found in Table 5. As expected, Astrid-Embed produces the best performance. The contrastive loss based approach produces the next best results. This justifies our effort to produce a blended distance function in Section 4.2 that combines both frequency and query-type similarity. Between the other two baselines, fastText performs better than DeepWalk. This is not surprising as fastText leverages sub-word information for learning embeddings. In contrast, DeepWalk makes random walks that could result in embeddings that are not query-type and frequency aware. We also did a hyperparameter optimization on DeepWalk to vary the random walk size and the context window size. This did not materially improve the results.

	D-AT	D-AN	I-MT	I-AN	T-PN
Astrid-Embed	4.29	3.86	3.98	3.85	5.84
FastText	5.64	6.73	6.59	7.67	7.11
DeepWalk	7.34	6.69	6.86	6.14	7.4
Contrastive Loss	6.98	5.73	6.97	6.1	6.78

Table 5: Varying the algorithm for learning the embeddings of Astrid-Embed

Impact of Triplet Selection Strategy. In this experiment, we evaluate our distance based sampling strategy. We compare it against three representative approaches. The *random* approach chooses the triplet as follows. For every anchor, it randomly picks a positive example from the 3-hop neighborhood. The negative example is then randomly picked from the entire graph. In the *2-hop* approach, we exhaustively consider all its 2-hop neighbors as positive examples and random nodes as negative examples. We also evaluated a *LSH* based approach. Specifically, we used MinHashLSH with Jaccard similarity defined over 3-grams of the string. For each string, we hashed it into a bucket. Then we choose a random string from the same bucket as a positive anchor and a random string from a random bucket as a negative anchor. Note that 2-hop approach could generate a large number of triplets. In contrast, the other approaches are limited by a budget on the number of triplets selected. We can see the results in Table 6. The 2-hop approach produces a better results than the random approach. However, the distance based sampling idea of Astrid-Embed produces a better accuracy while using a dramatically smaller number of triplet examples. The LSH based approach provides slightly worse performance than the 2-hop strategy. The key issue is that it focuses on string similarity but not on the frequency. Hence, two similar strings that have different frequencies could end up in the same LSH bucket.

	D-AT	D-AN	I-MT	I-AN	T-PN
ASTRID-EMBED	4.29	3.86	3.98	3.85	5.84
2-Hop	5.98	5.45	6.4	7.1	7.7
Random	8.5	8.1	8.4	9.93	9.8
LSH	6.1	5.69	6.81	7.07	7.97

Table 6: Varying the triplet selection strategy

Impact of Pruning Threshold. A key bottleneck for ASTRID-EMBED is the size of the suffix tree. This affects both the embedding learning and the training data generation for selectivity estimator. We investigate how pruning the suffix tree impacts ASTRID-EMBED. Given a threshold such as 90%, we remove all nodes that are in the bottom 10% with ties broken randomly. Table 7 shows the results. As the suffix tree is pruned more and more, the q-error slowly trickles up. In resource constrained environments, one can prune the suffix tree, pay a minimal penalty in accuracy while achieving substantial reduction in time taken for embedding learning and training the estimator.

	D-AT	D-AN	I-MT	I-AN	T-PN
ASTRID-EMBED	4.29	3.86	3.98	3.85	5.84
90%	4.33	4.1	4.4	4.8	6.1
75%	4.9	4.59	5.18	5.3	7.6
50%	5.16	5.05	6.71	5.94	8.23

Table 7: Varying the Pruning threshold

Impact of Training Dataset Size. By default, ASTRID-EMBED uses 40% of the suffix tree nodes for training. In this experiment, we show how the performance is affected for shorter trees. For example, in the 10% case, we randomly choose 10% of the suffix tree and use it for training. We can see from Table 8 that one could reduce the training dataset by as much as 10% of the tree and yet get accurate results due to the generalization capabilities of the neural networks.

	D-AT	D-AN	I-MT	I-AN	T-PN
ASTRID-EMBED	4.29	3.86	3.98	3.85	5.84
30%	5.37	4.78	5.45	6.13	6.96
20%	6.2	5.9	6.11	6.61	8.01
10%	6.95	7.1	7.98	8.16	8.6

Table 8: Varying the training data size

6.4 Evaluating Astrid-NLM

In this subsection, we investigate the key design choices of ASTRID-NLM.

Impact of DL Architecture used for Language Modeling. By default, ASTRID uses GRU for learning the NLM. There are other possible architectures of which the most popular is LSTM. We compare the performance of these architectures in Table 9. We ensured that both LSTM and GRU has the same number of hidden *units*. LSTM produces *slightly* better results than GRU. This is mainly due to the fact that LSTM is a more complex architecture that uses 4 gates as against 3 for GRU. GRU requires 25% lesser number of parameters to learn and is as much as 10x faster to train. It also requires less tuning of hyperparameters than LSTM. Our observations also corraborate a number of prior work such as [12, 31] that show that GRU and LSTM produce comparable results for various language modeling tasks. We can see that the RNN architecture provides worse results than either GRU or LSTM. While the performance of RNN is comparable for short queries, the accuracy drops rapidly for longer queries.

	D-AT	D-AN	I-MT	I-AN	T-PN
ASTRID-NLM	6.95	7.45	7.14	8.4	8.61
LSTM	6.19	6.86	6.64	7.46	7.67
RNN	10.54	11.09	11.68	12.9	13.45

Table 9: Varying the NLM architecture

Impact of State-Reset Hyperparameter. Recall that we modify the training process of GRU so that the state is periodically resetted with a probability α. By default, we set $\alpha = 0.1$. Table 10 shows how the results vary for different values of α. A smaller value of α increases the median and 90-th percentile q-error. Almost all of these increase comes from substring query estimates. Setting α to 0 is equivalent to the traditional GRU training that is not conducive for substring queries. Specifically, setting it to a larger value reduces the accuracy of ASTRID-NLM from answering queries involving larger strings. This is due to the fact that $\alpha = 0.9$ results in a significant number of resets that prevent it from learning the selectivities for larger strings.

	D-AT	D-AN	I-MT	I-AN	T-PN
0.05	8.7	9.01	8.11	9.37	9.2
0.1	6.95	7.45	7.14	8.4	8.61
0.5	9.5	9.7	8.2	10.9	10.4
0.9	10.9	10.7	9.29	12.7	12.6

Table 10: Varying state-reset hyperparameter

6.5 Comparing Astrid-Embed and Astrid-NLM

In this subsection, we compare the performance of our two proposed approaches ASTRID-EMBED and ASTRID-NLM. for different query types – prefix, substring and suffix. Table 11 shows the results. As expected, ASTRID-EMBED is not impacted by query type. The process for learning the embeddings and training the selectivity estimator is identical for all query types. In contrast, ASTRID-NLM produces good results for prefix and suffix queries with a slight dip in performance for substring queries. Once again, this is not surprising as neural language models are not trained for answering substring queries. The comparable performance is due to our proposed state-reset idea. Interestingly, ASTRID-NLM is slightly more accurate for suffix queries. This is due to the fact that the suffixes are often more distinct than prefixes. Hence the frequency distribution is comparatively simpler with less skew that is easier to learn for ASTRID-NLM.

	D-AT	D-AN	I-MT	I-AN	T-PN
Astrid-Embed	4.29	3.86	3.98	3.85	5.84
Astrid-NLM	6.95	7.45	7.14	8.4	8.61
Astrid-Embed-PF	4.11	3.69	3.83	3.97	5.25
Astrid-Embed-SF	4.16	3.87	3.91	3.93	5.6
Astrid-Embed-SB	4.43	4.2	4.56	4.62	6.9
Astrid-NLM-PF	6.58	7.52	7.32	8.5	8.8
Astrid-NLM-SF	6.94	7.47	7.36	8.9	8.83
Astrid-NLM-SB	7.49	10.08	9.59	9.74	10.13

Table 11: Impact of Query type on Astrid. PF, SF, SB correspond to prefix, suffix and substring

Model Overhead. The DL models of Astrid are often much smaller than the underlying summary data structures requiring less than 20MB for all datasets. The 90-th percentile time taken for querying is less than 50 milliseconds. There has been extensive work on compressing the models. We use the Distiller [84] library from Intel AI that implements all the state-of-the-art model compression algorithms. This approach reduces the model size without sacrificing accuracy while increasing inference efficiency. The model size after compression is around 12MB. The 90-th percentile inference time dropped to below 40 milliseconds.

Impact of Dataset Size. We use the AOL query log from [25] that has more than 36M queries. We fixed the space budget as 20MB and vary the size of the dataset from 1M queries to 36M. Table 12 shows that Astrid can provide good performance for arbitrarily sized datasets even with a fixed model size. In contrast, q-gram based model shows a steeper decline in performance. This is not surprising as the number of distinct q-grams dramatically increases with larger dataset size. However, the fixed space budget constrains the ability of the model to perform better even with the sophisticated Kneser-Ney smoothing.

	1M	5M	10M	36M
Astrid-Embed	2.4	3.3	4.2	4.8
Astrid-NLM	3.8	5.6	8.7	10.2
q-gram LM	8.8	12.8	16.4	38.6

Table 12: Varying Dataset Size

Model Retraining. Astrid does not currently allow incremental retraining of the models to handle data updates. Astrid-Embed relies on the summary data structure to learn the embeddings. Due to the large size, the data structure is deleted once the model is learned. Retraining the model requires rebuilding it from scratch. A simple approach would be to use a small set of queries for which the true selectivities are known. If Astrid's performance reduces below an acceptable level (e.g. 90-th percentile must be at most 5), then one could retrain the model. There is a need for a sophisticated cost model for understanding the cost-benefit tradeoff.

7 RELATED WORK

Traditional Approaches for String Selectivity Estimation. The earliest effort was from [40] that used suffix trees followed by a number of weighted combinations of individual selectivity. An improved estimator was given by [27]. Both these approaches could under estimate the selectivity. So [8] proposed a CRT estimator to obtain better estimates by finding a shorter string whose frequency is similar to a given string. Two estimators HSol and Vsol using set hashing for estimating selectivities was proposed by [52]. Other major works that tackle similar problems include [10, 28]. There has been a number of work for solving approximate string selectivity such as [30, 43, 44]. Popular approaches for numeric data include sampling [48], histograms [5, 6, 26, 29, 62, 67], wavelets [51], kernel density estimation [17, 33, 37] and graphical models [16, 70].

ML based Approaches for Selectivity Estimation. One of the earliest approaches to use neural networks is [42]. A recent work is [36] that focuses on estimating correlated join selectivities. It proposes a novel set based DL model but focuses mostly on supervised learning. An empirical analysis of various approaches can be found in [59]. Recent works seek to tackle challenging types of queries including group-by [35] and range queries [14, 79]. There has been some promising works on unsupervised approaches for selectivity estimation such as [20, 79]. There is also some promising work on using (deep) learning for approximate query processing such as [22, 63, 68, 74, 82]. Recently, there has been some preliminary work for approximating the edit distance [13] and use it for nearest neighbor search [83]. Unfortunately, there is limited support for string predicate queries.

Deep Learning for Databases. An early work [39] built fast indices by using a mixture of neural networks to effectively learn the distribution of data. Reinforcement learning based techniques have been used for query optimization (and join order enumeration) such as [41, 50, 58, 71]. There has been research efforts to build learned database systems [38] and an end-to-end learned optimizers [49, 66]. DL has also been applied to the problem of data integration [7, 69] and entity resolution in [15].

8 CONCLUSION

Estimating accurate selectivities for prefix, substring and suffix queries is a challenging problem. Our proposed approaches – Astrid-Embed and Astrid-NLM– produces embeddings and neural language models that are tuned for the task of selectivity estimation and achieves state-of-the-art results. There are a number of interesting directions to explore at the intersection of expressive string queries and deep learning concepts such as embeddings and neural language models. For example, the problem of approximate string selectivity that estimates strings whose distance to the query is within a bound is non-trivial to solve. One could also extend character based neural language models with recent architectures for language modeling such as transformers.

ACKNOWLEDGMENTS

The research of Nick Koudas is supported in part by COHESA NOE. The research of Gautam Das was supported in part by grants 2008602, 1745925, 1937143 and 2037433 from the National Science Foundation.

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.

[2] Ricardo A Baeza-Yates and Gaston H Gonnet. 1996. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM (JACM)* 43, 6 (1996), 915–936.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *arXiv preprint arXiv:1607.04606* (2016).

[5] Nicolas Bruno and Surajit Chaudhuri. 2004. Conditional Selectivity for Statistics on Query Expressions. In *SIGMOD*. 311–322. https://doi.org/10.1145/1007568.1007604

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. In *SIGMOD*. 211–222. https://doi.org/10.1145/375663.375686

[7] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. *SIGMOD* (2020).

[8] 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.

[9] Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language* 13, 4 (1999), 359–394.

[10] 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.

[11] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[12] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[13] Xinyan Dai, Xiao Yan, Kaiwen Zhou, Yuxuan Wang, Han Yang, and James Cheng. 2020. Edit Distance Embedding using Convolutional Neural Networks. *arXiv preprint arXiv:2001.11692* (2020).

[14] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *PVLDB* 12, 9 (2019), 1044 – 1057.

[15] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed representations of tuples for entity resolution. *PVLDB* 11, 11 (2018), 1454–1467.

[16] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity estimation using probabilistic models. In *ACM SIGMOD Record*, Vol. 30.

[17] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.* 14, 2 (2005), 137–154. https://doi.org/10.1007/s00778-003-0090-4

[18] Dan Gusfield. 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News* 28, 4 (1997), 41–60.

[19] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 2. IEEE, 1735–1742.

[20] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. *SIGMOD* (2020).

[21] Alexander Hermans, Lucas Beyer, and Bastian Leibe. 2017. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737* (2017).

[22] 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).

[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[24] Elad Hoffer and Nir Ailon. 2015. Deep metric learning using triplet network. In *International Workshop on Similarity-Based Pattern Recognition*. Springer, 84–92.

[25] Jeff Huang. 2020 (accessed Oct 02, 2020). *AOL Query Logs*. https://jeffhuang.com/search_query_logs/

[26] Yannis E. Ioannidis. 2003. The History of Histograms (abridged). In *VLDB*. http://www.vldb.org/conf/2003/papers/S02P01.pdf

[27] HV Jagadish, Olga Kapitskaia, Raymond T Ng, and Divesh Srivastava. 2000. One-dimensional and multi-dimensional substring selectivity estimation. *The VLDB Journal—The International Journal on Very Large Data Bases* 9, 3 (2000), 214–230.

[28] 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.

[29] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. 1998. Optimal Histograms with Quality Guarantees. In *VLDB*. 275–286. http://www.vldb.org/conf/1998/p275.pdf

[30] Liang Jin, Chen Li, and Rares Vernica. 2008. Sepia: estimating selectivities of approximate string predicates in large databases. *The VLDB Journal* 17, 5 (2008), 1213–1229.

[31] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An empirical exploration of recurrent network architectures. In *International conference on machine learning*. 2342–2350.

[32] Dan Jurafsky and James Martin. 2020. *Speech & language processing. 3rd Edition.* Pearson.

[33] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *PVLDB* 10, 13 (2017), 2085–2096. https://doi.org/10.14778/3151106.3151112

[34] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[35] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. [n.d.]. Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue. ([n. d.]).

[36] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *CIDR* (2019).

[37] Flip Korn, Theodore Johnson, and HV Jagadish. 1999. Range selectivity estimation for continuous attributes. In *ssdbm*. IEEE, 244.

[38] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. (2019).

[39] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. ACM, 489–504.

[40] P Krishnan, Jeffrey Scott Vitter, and Bala Iyer. 1996. Estimating alphanumeric selectivity in the presence of wildcards. In *ACM SIGMOD Record*, Vol. 25. ACM, 282–293.

[41] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[42] Seetha Lakshmi and Shaoyu Zhou. 1998. Selectivity estimation in extensible databases-A neural network approach. In *VLDB*. 623–627.

[43] 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.

[44] 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*. ACM, 827–838.

[45] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015).

[46] Viktor Leis, Bernharde Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *CIDR*.

[47] Chen Li, Bin Wang, and Xiaochun Yang. 2007. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams.. In *VLDB*, Vol. 7. Citeseer, 303–314.

[48] Richard J Lipton, Jeffrey F Naughton, and Donovan A Schneider. 1990. *Practical selectivity estimation through adaptive sampling*. Vol. 19. ACM.

[49] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.

[50] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. *arXiv preprint arXiv:1803.00055* (2018).

[51] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-based histograms for selectivity estimation. In *ACM SIGMOD Record*, Vol. 27.

[52] Arturas Mazeika, Michael H Böhlen, Nick Koudas, and Divesh Srivastava. 2007. Estimating the selectivity of approximate string queries. *ACM Transactions on Database Systems (TODS)* 32, 2 (2007), 12.

[53] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

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

[55] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved selectivity estimation by combining knowledge from sampling and synopses. *PVLDB* 11, 9 (2018), 1016–1028.

[56] Parimarjan Negi. 2020 (accessed May 1, 2020). *DB Embedding Tools*. https://github.com/parimarjan/db-embedding-tools

[57] NLTK. 2020 (accessed Oct 02, 2020). *NLTK Language Model*. http://www.nltk.org/api/nltk.lm.html

[58] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. *arXiv preprint arXiv:1803.08604* (2018).

[59] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. *arXiv preprint arXiv:1905.06425* (2019).

[60] 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.

[61] Matthias Petri and Trevor Cohn. 2016. Succinct Data Structures for NLP-at-Scale. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Tutorial Abstracts*. 20–21.

[62] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, Vol. 25.

[63] Fotis Savva, Christos Anagnostopoulos, and Peter Triantafillou. 2020. ML-AQP: Query-Driven Approximate Query Processing based on Machine Learning. *arXiv preprint arXiv:2003.06613* (2020).

[64] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[65] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR* 15, 1 (2014), 1929–1958.

[66] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *arXiv preprint arXiv:1906.02560* (2019).

[67] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. 2002. Dynamic multidimensional histograms. In *SIGMOD*. 428–439. https://doi.org/10.1145/564691.564741

[68] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2020. Approximate Query Processing using Deep Generative Models. *ICDE* (2020).

[69] Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, and AnHai Doan. 2020. Data Curation with Deep Learning. *EDBT* (2020).

[70] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB* 4, 11 (2011), 852–863.

[71] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. A reinforcement learning approach for adaptive query processing. *History* (2008).

[72] Jiang Wang, Yang Song, Thomas Leung, Chuck Rosenberg, Jingbin Wang, James Philbin, Bo Chen, and Ying Wu. 2014. Learning fine-grained image similarity with deep ranking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1386–1393.

[73] Liwei Wang, Yin Li, and Svetlana Lazebnik. 2016. Learning deep structure-preserving image-text embeddings. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5005–5013.

[74] Zhe Wang, Dylan Cashman, Mingwei Li, Jixian Li, Matthew Berger, Joshua A Levine, Remco Chang, and Carlos Scheidegger. 2018. NNCubes: Learned Structures for Visual Data Exploration. *arXiv preprint arXiv:1808.08983* (2018).

[75] 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)*.

[76] Chao-Yuan Wu, R Manmatha, Alexander J Smola, and Philipp Krahenbuhl. 2017. Sampling matters in deep embedding learning. In *Proceedings of the IEEE International Conference on Computer Vision*. 2840–2848.

[77] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 524–537.

[78] Xiaochun Yang, Bin Wang, and Chen Li. 2008. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 353–364.

[79] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2020. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment* 13, 3 (2020), 279–292.

[80] Baosheng Yu, Tongliang Liu, Mingming Gong, Changxing Ding, and Dacheng Tao. 2018. Correcting the triplet selection bias for triplet loss. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 71–87.

[81] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[82] Meifan Zhang and Hongzhi Wang. 2020. LAQP: Learning-based Approximate Query Processing. *arXiv preprint arXiv:2003.02446* (2020).

[83] Xiyuan Zhang, Yang Yuan, and Piotr Indyk. 2019. Neural Embeddings for Nearest Neighbor Search Under Edit Distance.

[84] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural Network Distiller: A Python Package For DNN Compression Research. (October 2019). https://arxiv.org/abs/1910.12232