Accelerating Relational Keyword Queries With Embedded Predictive Neural Networks

Limin Ma
Faculty of Science
Ontario Tech University
Oshawa, Canada
limin.ma@ontariotechu.net

Ken Q. Pu
Faculty of Science
Ontario Tech University
Oshawa, Canada
ken.pu@ontariotechu.net

Ying Zhu
Faculty of Business and IT
Ontario Tech University
Oshawa, Canada
ying.zhu@ontariotechu.net

Abstract—Relational keyword queries have proven to be highly effective for information retrieval. The challenge of evaluating keyword queries for relational databases is the performance bottleneck of fuzzy string matching when traditional full-text index structures. We propose a solution to overcome performance bottlenecks by incorporating horizontally partitioned full-text indexes. We rely on a neural network router to optimize the index lookup strategy to minimize index miss rate and thus maximize performance. Using textural features of the user queries, the neural network router supports fuzzy string matching. We evaluated different network architectural designs against real-world datasets. Our experiments demostrates that the neural network router can be self-trained and learn how to optimize index access effectively.

Index Terms—keyword query, neural network, index structures

I. Introduction

II. Partial tuple search

In this section, we formalize keyword queries as partial tuple search. Let R be a relational table. Recall that $\mathrm{attr}(R)$ is the attributes of R, and tuples in R are mappings from $\mathrm{attr}(R)$ to **Values**.

Definition 1 (Labeled values and partial tuple): Let R be a relational table. A labeled value in R is a pair $(l : x)$ where $l \in \mathrm{attr}(R) \cup \{?\}$ and $x \in \mathbf{Values}$. A partial tuple $\vec{x}$ is a set of labeled values: $\vec{x} = \{(l_i, x_i) : i \in I\}$. We define the attributes of a partial tuple $\vec{x}$ as the labels of the partial tuple:

$$\mathrm{attr}(\vec{x}) = \{l_i\}_{i \in I}$$

. A partial tuple is considered complete if $\mathrm{attr}(\vec{x}) = \mathrm{attr}(R)$.

We will write $\vec{x}[l_i]$ to denote the corresponding value x_i.

Note that a partial tuple is a set of values and their respective attribute names from a relational table. However, we allow a special symbol "?" to be in place of the attribute name. The special attribute "?" indicates that the attribute name is unspecified (or unknown). The following example illustrates two partial tuples. The second partial tuple has a wild card "?" as its attribute name:

Example 1:

$$\begin{aligned} \vec{s}_1 &= \{\text{name} : \text{Einstein}\} \\ \vec{s}_2 &= \{\text{name} : \text{Einstein},\ ? : \text{Professor}\} \end{aligned}$$

A keyword query is a partial tuple: $\vec{Q} = \{(l_i, q_i) : i \in I_Q\}$. The values q_i are keyword queries. We want to find a complete tuple $\vec{r} \in R$ that matches $\vec{Q}$ optimally. This requires us to define how to compare the partial tuple $\vec{Q}$ with a complete tuple $\vec{r}$. The guiding principle of comparing the two tuples is to match labeled values from $\vec{Q}$ with those from $\vec{r}$ according to the following:

- Match the labels if they are not the wild card "?".
- Match the values using a fuzzy string matching score.
- Optimize the sum of similarities between labeled values from $\vec{Q}$ and those from $\vec{r}$.

Definition 2 (Partial tuple search): Let $R_1, R_2, R_3, \ldots, R_K$ be K relations. Given a user query that is a partial tuple:

$$\vec{Q} = \{(l_i, q_i) : i \in I_Q\}$$

where $l_i \in attr(R) \cup \{?\}$ and q_i are keyword queries, we want to find a complete tuple $\vec{r} \in R_i,\ i \in \{1, 2, \ldots, K\}$, that maximizes the similarity score between the partial tuple $\vec{Q}$ and full relational query $\vec{r}$.

We use a neural network to optimize the query processing pipeline. Detailed description can be found in Section ??. We define the neural network classifier as:

III. Partial tuple search using full-text search

Traditional full-text indexes support keyword queries over document collections using an inverted index data structure. First the full relational tuples are encoded as documents by some tokenizer. The tokenizer breaks the string values of partial tuples $\{q_i\}$ and full tuple $r \in R$ to tokens. It's the comparison between $\mathbf{tokens}(q_i)$ and $\mathbf{tokens}(r)$ that determines their similarity. To support approximate string matching, the standard approach [3], [4] is to break down strings into their n-grams. An example of a full-text encoding of a tuple is shown in Figure ??.

$$\left[\begin{array}{rcl} \text{Name} & \mapsto & \text{\_\_J \_Ja Jac ack ck\_ k\_\_} \\ \text{Address} & \mapsto & \text{\_\_1 \_10 100 00\_ 0\_\_ \_\_S \_Si Sim imc mco coe oe\_ e\_\_ \_\_S \_St Str tre ree eet et\_ t\_\_} \\ \text{fulltext} & \mapsto & \text{\_\_J \_Ja Jac ack ck\_ k\_\_ \_\_1 \_10 100 00\_ 0\_\_ \_\_S \_Si Sim imc mco coe oe\_ e\_\_ \_\_S} \\ & & \text{\_St Str tre ree eet et\_ t\_\_} \end{array}\right]$$

Fig. 1: Full text encoding of a relational tuple with 3-gram tokenization

IV. Fuzzy keyword search with partitioned indexes

Traditionally, a single full-text index is built on the tuples from all the relations. With the relations $\{R_1, R_2, \dots, R_n\}$, the aggregated index is the full-text index built from the union of the tuples:

$$\mathbf{Index}_{\mathrm{agg}} = \mathrm{InvertedList}\left(\bigcup_{i=1}^{n} R_i\right)$$

Due to the inverted list architecture, the performance bottleneck comes from collisions of multiple documents containing the common tokens. With the total number of document is given by $\sum_{i=1}^{n}|R_i|$, the average number collision of the inverted lists in the aggregate index is estimated as:

$$\mathbf{Collision}_{\mathrm{agg}} = \frac{\sum_{i=1}^{n}|R_i|}{|\mathrm{vocab}|}$$

where $|\mathrm{vocab}|$ is the size of the vocabulary of the distinct tokens. The query evaluate performance is known to be $\mathcal{O}(\mathbf{Collision})$.

When supporting fuzzy string matching, the tokenizer performs 3-gram tokenization, and thus vocab = all 3-grams $\in \mathcal{O}(1)$. Namely, with a fixed alphabet, the size of the vocabular is roughly a constant. This means the query evaluation time complexity is given by:

$$\mathcal{O}(\mathrm{Lookup}(\mathbf{Index}_{\mathrm{agg}}, q)) = \mathcal{O}\left(\sum_{i=1}^{n}|R_i|\right)$$

The query performance, thus, would degrade with increasing number of tuples, and increasing number of relations.

We propose the following strategies to overcome the performance bottleneck of the aggregated index.

- We partition $\mathbf{Index}_{\mathrm{agg}}$ into multiple indexes $\{\mathbf{Index}_j : 1 \le j \le m\}$
- Given a query q, we scan through the partitioned indexes and evaluate $\mathrm{Lookup}(\mathbf{Index}_j, q)$.

The advantages of the partitioned approach are:

- Each index access $\mathrm{Lookup}(\mathbf{Index}_j, q))$ is more efficient due to the reduced number of collisions in the inverted index.
- We have the opportunity of optimizing the sequence of scanning the index partitions. If we can determine the likelihood of that the query q has high similarity with tuples in $\mathbf{Index}_j$, we want to access the partition j before the other partitions.

Algorithm 1 Predictive index lookup

```
Require: $q$: Query, $partitions$: List of Partitions
1: $n \leftarrow$ length of $partitions$
2: $p$: List of float
3: for $i \in [0, n-1]$ do
4:     $p[i] \leftarrow \mathbf{classifier}(q, partition[i])$
5: end for
6: for all $i$ in sorted(range($n$), key=$\lambda$k: $p[k]$, desc) do
7:     $results \leftarrow$ Lookup($partitions[i]$, $q$)
8:     yield $results$
9: end for
```

In Algorithm ??, it evaluates the probability of if q is relevant to the tuples in the ith partition. The probability is computed as a binary classifier

$$\mathbf{classifier}(\mathbf{Index}, \mathbf{Query})$$

V. Design and training of predictive classifiers

Each index partition has an accompanying binary classifier $\mathbf{Classifier}_i : \mathbf{Query} \to [0, 1]$. We use a neural network to learn the classification function. Due to the application scenario, the design constraints are:

- The neural network must be compact in size so that they can be embedded in the database runtime system. Our goal is that the neural network is a small memory addition to the keyword query evaluation engine.
- The inference speed of the neural network incurs a minimal overhead. This means we prefer shallow networks over deeper architectures.

A. Vectorization of tokens and queries

Token representation of queries: given a query $\vec{q} = \{(l_i, x_i) : 1 \le i \le m\}$, we generate the text representation of the query by simply concatenating the text representation of labels and the tokenized query values.

$$\begin{aligned} \mathbf{tokens}(\vec{q}) \quad = \quad & \{l_1\} \oplus \mathbf{tokenize}(x_1) \\ & \oplus\{l_2\} \oplus \mathbf{tokenize}(x_2) \\ & \oplus \dots \{l_m\} \cup \mathbf{tokenize}(x_m) \end{aligned}$$

where $\oplus$ is sequence concatenation.

Integer encoding of queries: next we encode $\mathbf{tokens}(\vec{q})$ using a universal vocabulary $\mathbf{vocab}$. The vocabulary consists of all known tokens, and their respective ordinal integer code. This vocabulary will be built using the existing relational tuples. The construction of the vocabulary is described in subsequent sections. The vocabulary is described as a function from tokens to integers:

$$\mathbf{vocab} : \mathbf{Token} \to \mathbb{N}$$

The integer sequence of a query is given by:

$$\mathbf{sequence}(\vec{q}) = \mathbf{vocab} \circ \mathbf{tokens}(\vec{q}) \in \mathbb{N}^*$$

Embedding vectors of queries: Using a standard embedding layer (Section ??), embed the integer sequence representation of the query to a sequence of latent vectors.

$$\vec{x} = \mathbf{vector}(\vec{q}) = \mathbf{Emb}(\mathbf{sequence}(\vec{q})) \in \mathbb{R}^{|q|\times d}$$

From the vectorized representation $\vec{x}$ of the query, we perform n-way classification by generating a probability distribution in as a vector $\mathbb{R}^n$.

$$\vec{p} = \mathbf{classifier}(\vec{x})$$

B. Architectures for the predictive classifier

MLP based classification.

Given the input of vectorized query representation $\vec{x}$, we first flatten it using global average over the entire sequence length. Then, we process it with a MLP with softmax activation function. The MLP must have n output neurons.

$$\begin{array}{lll} & \vec{x} & (|q|, d) \\ \to & \left(\frac{\sum_{i=1}^{|q|} x_i}{|q|}\right) & (d) \\ \to & \mathbf{MLP}(\text{output-dim} = n) & (n) \\ \to & \mathbf{softmax}(\cdot) & (n) \end{array}$$

Conv1D and LSTM architecture.

We can also utilize sequence based learning, namely Conv2D or LSTM in place of MLP, and use global average to flatten the sequential features for further processing using a MLP.

$$\begin{array}{lll} & \vec{x} & (|q|, d) \\ \to & \textbf{Conv1D or LSTM} & (|q|, d) \\ \to & \textbf{Global Average} & (d) \\ \to & \mathbf{MLP}(\text{output-dim} = n) & (n) \\ \to & \mathbf{softmax}(\cdot) & (n) \end{array}$$

Transformer and MLP Mixer

Transformers have shown to be a superior architecture for many tasks in the domain of MLP and sequence learning. Due to the nature of our problem, we chose to use only a single transformer block so that the model remains small enough to be embedded in the query processing pipeline.

A more recent MLP based architecture is MLP mixer which is a concatenation of two MLP layers separated by a matrix transpose operation

$$\begin{array}{lll} & \vec{x} & (|q|, d) \\ \to & \textbf{Transformer or MLPMixer} & (|q|, d) \\ \to & \textbf{Global Average} & (d) \\ \to & \mathbf{Dense}(\cdot) & (n) \end{array}$$

VI. Self-supevised training

The classifier needs to be trained with data of the following form:

$$(\vec{q}, i)$$

where $\vec{q}$ is a sample query, and i is the relation R_i that contains the best matching tuple of $\vec{q}$.

The training data is generated directly from the relational tuples from the database. For each complete tuple $\vec{r} = \{(l_i, x_i) : i \in I\}$, we formed a query $\vec{q}_r$ by random sampling from the labeled values while masking the labels.

$$\vec{q}_r = \{(?, \mathbf{permute}(x_i)) : i \in \mathbf{sample}_k(I)\}$$

where:

- **permute** : **String** → **String** tokenizes and permutes the input words to query strings.
- **sample** randomly samples k attributes from the tuple.

Together $(\text{permute}, \text{sample}_k)$ allows us to generate a range of different queries from existing tuples in the relations to form the training data.

$$\mathbf{train} = \bigcup_{i=1}^{n} \{(\vec{q}_r, i) : r \in R_i\}$$

The resulting classifier is determined by the choice of three different design choices:

1) architecture design: MLP / Conv / LSTM / Transformer / MLPMixer
2) attribute sampling rate used by **sample**: 100%, 75%, 50% of the attributes are kept to form the query
3) tokenization used by **permute** function: word vs 3-gram tokenizers.

Thus each choice of the classifier is named as

$$\textit{arch\_sample\_tokenizer}$$

VII. Experimental Evaluation

In this section, we will describe the evaluation of the proposed solutions. We will verify the effectiveness and limitations of our solutions, and perform a comparative study of different network architectures. For each network architecture, we will present the benefits and drawbacks, and provide our understanding of the explanation of the observations.

survyear	survmnth	lfsstat	prov	age_12	educ	immig	noc_10	ftptmain	hrlyearn
2019	April	Employed, at work	Ontario	25 to 29 years	Postsecondary certificate or diploma	Non-immigrant	Natural resources, agriculture and related production occupations	Full-time	33.00
2019	April	Not in labour force	British Columbia	70 and over	Bachelors degree	Non-immigrant			
2019	April	Employed, at work	British Columbia	45 to 49 years	High school graduate	Non-immigrant	Business, finance and administration occupations	Full-time	26.00
2019	April	Employed, at work	Ontario	20 to 24 years	Postsecondary certificate or diploma	Non-immigrant	Health occupations	Full-time	37.60
2019	April	Employed, at work	Ontario	70 and over	Some postsecondary	Immigrant, landed more than 10 years earlier	Occupations in art, culture, recreation and sport	Part-time	32.00

TABLE I: 5 samples from the labour force survey dataset "ds01", showing only 10 attributes.

A. Datasets

The datasets we have used to evaluate our system is a collection of survey data from Statistics Canada, which includes one labour force survey, six COVID-19 surveys, one income survey, one community health survey, and one housing survey. They offer many real-world characteristics that pose as challenges to neural network classifiers. Given the intended application scenarios of our research, we felt that it was important to evaluate our work using real-world datasets.

These 10 datasets contain both numerical and textual data with different numbers of attributes and tuples. For example, the labour force survey contains data of the Canadian labour market. It has total 60 attributes related to the job market, such as employment status, industry, status of working full-time or part-time, hourly wage, etc. Table I shows 5 samples with a subset of 10 attributes. The descriptions of the selected 10 attributes are shown in table ??.

B. Evaluation methodology

We evaluate five model architectures, including Multilayer Perceptron (MLP), Long Short-Term Memory (LSTM), 1D Convolution (Conv1D), Transformer and MLP-Mixer. We also exam how misspelled keywords affect the models' top-5 accuracies.

We use two metrics for model evaluation. The first metric is query processing time. We submit queries to partitioned indexes and the aggregate index, and record the response times as evaluation benchmark. More details are given in the subsection ??.

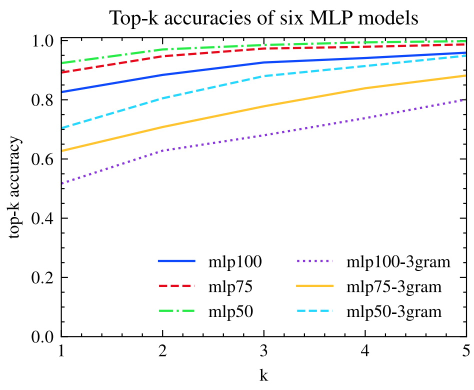
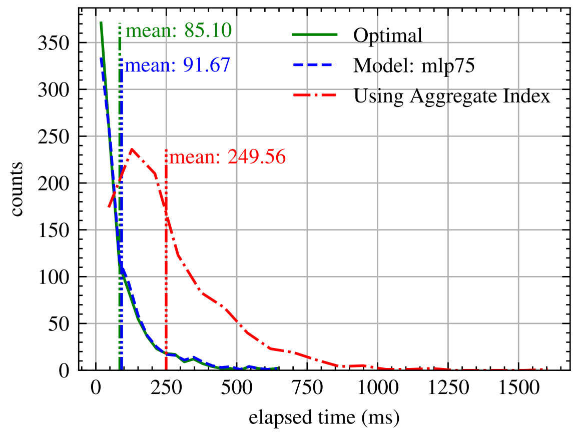
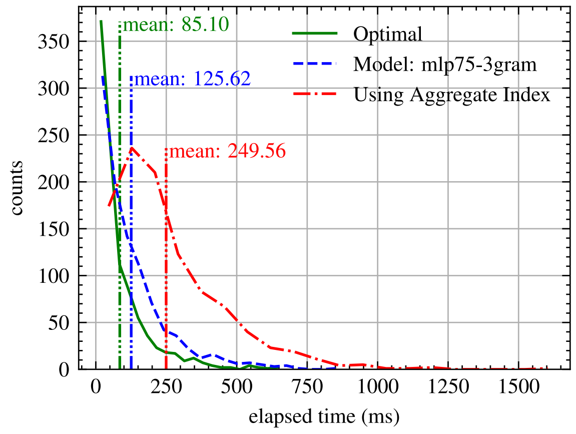


Fig. 2: Predictive accruacy



(a) MLP-75-word



(b) MLP-75-3gram model

Fig. 3: Using MLP-75 with word and 3gram tokenization.

The second metric is top-k accuracy. A model's prediction gives us the ordering of Lucene index access. If the Lucene index that a query belongs to is among the top-k of the model's prediction, we consider the prediction accurate. For example, for a query sampled from ds02, if the predicted access ordering is

idx_ds06, idx_ds10, idx_ds07, idx_ds05, idx_ds02, ...

C. Query workload generation and tokenization

Query workloads are sets of partial tuples. We first randomly sample tuples from data reserved for model evaluation. Then we apply the same normalization process

	Word tokens			3-gram tokens		
Model	parameters	$\frac{\text{model}}{\text{optimal}} \downarrow$	$\frac{\text{model}}{\text{aggr}} \downarrow$	parameters	$\frac{\text{model}}{\text{optimal}} \downarrow$	$\frac{\text{model}}{\text{aggr}} \downarrow$
MLP	3.02M	1.10	0.37	272K	1.49	0.51
LSTM	3.05M	1.52	0.52	305K	1.78	0.61
Conv1D	3.03M	1.12	0.38	277K	1.68	0.57
Transformer	3.09M	1.12	0.38	340K	1.66	0.57
MLP Mixer	3.04M	1.65	0.56	287K	1.65	0.56

TABLE II: Comparison of models under the workload A with the top-3 measurements highlighted.

to them as we do to training datasets. After that we convert them to partial tuples by randomly sampling some attributes from them. We create different workloads by varying the number of attributes sampled.

To create the performance benchmark, we perform search against partitioned indexes and the aggregate index using the constructed partial tuple queries and log the response times. We repeat the process 10 times and compute the mean response times to be used as benchmark for model evaluation. Figure ?? shows the query evaluation performance between three choices:

- Optimal case: finding the right index partition immediately. This is not practically possible as it requires an oracle to perfectly predict which partition to access first. We include the performance for this case as a theoretical upperbound.
- Predictive access pattern based on MLP-75 models.
- Accessing aggregate index without partitioning

One can see that the predictive access pattern significantly outperforms the aggregate index access, and matches the optimal index access when word tokenization is used.

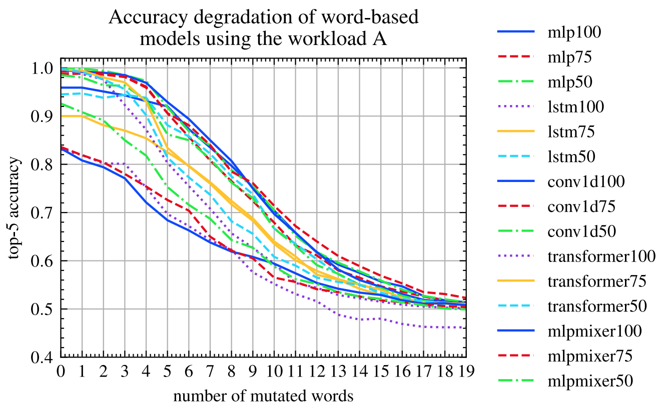
Other architectures (Conv1D, LSTM, Transformers and MLPMixers) have similar performances. The accelerated query performance lies in the fact that the predictive classifers are quite accurate in predicting which index partition is more relevant to the query, even by 3-gram tokenization is used. Figure 3 shows the predictive accuracy of the most relevant partition in the top-k results returned by the classifier.

D. 3-gram based models and noisy queries

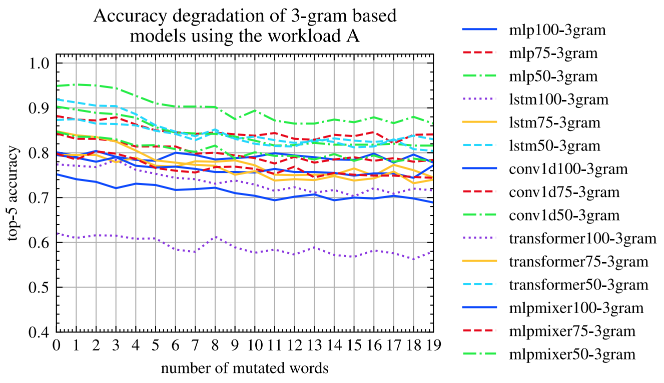
In this experiment, we investigate the impact of misspelled words in queries on the models' performance of top-5 accuracy. We simulate the scenario by replacing a randomly selected character in a word with the special character. The impact of noisy queries on models' performance of top-5 accuracy is shown in Figure4. The performance degradation of word-based models is much faster than that of 3-gram based models. This clearly shows that 3-gram based models are more resilient to query noises.

VIII. Related Work

Keyword search is an active area of research starting with DISCOVER [2], a system for search relational databases with keyword queries. Since then, there have been numerous work [7], extended to fuzzy matching, and



(a) Word-based models



(b) 3-gram based models

Fig. 4: Top-5 accuracy degradation of all models under the workload A.

to graph databases. Neural networks have shown to be effective in learning multidimensional indexes [1], [6]. They have also been applied to learn tree structures for indexing larger-than-memory databases [5].

References

[1] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. Proc. VLDB Endow., -:–, 2020.

[2] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In VLDB'02: Proceedings of the 28th International Conference on Very Large Databases, pages 670–681. Elsevier, 2002.

[3] M Kim, K Whang, and J Lee. n-gram/2l-approximation: a two-level n-gram inverted index structure for approximate string matching. Computer Systems Science and Engineering, 22(6):365, 2007.

[4] Grzegorz Kondrak. N-gram similarity and distance. In String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005. Proceedings 12, pages 115–126. Springer, 2005.

[5] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. Film: A fully learned index for larger-than-memory databases. Proceedings of the VLDB Endowment, 16(3):561–573, 2022.

[6] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pages 985–1000. ACM, 2020.

[7] Jeffrey Xu Yu, Lijun Chang, and Lu Qin. Keyword search in databases. Springer Nature, 2022.