Accelerating Relational Keyword Queries With Embedded 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

***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. Search Algorithm and Neural Network Accelerated Indexing

In this chapter, we define our problem and present our proposed solution.

Problem statement: We want to extend keyword search to relational data using full-text index with the help of an embeddable neural network that optimizes the query processing pipeline. Given a relational database consists of K relations $R_1, R_2, R_3, \ldots, R_K$, the goal is to build a system that takes a partial tuple as user query and can find an optimally matching complete tuple $\vec{r} \in R_i$ ($i \in \{1, 2, \ldots, K\}$) efficiently.

III. Problem definition of partial tuple search

In this section, we provide some definitions related to 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 $\mathbf{Values}$.

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 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 $\vec{r}$.

Towards this end, we define the following similarity measures:

Definition 2 (Similarity of labeled values):

Consider two labeled values: $(l_1 : x_1)$ and (l_2, x_2), where l_1 and l_2 are two labels from $\mathrm{attr}(R) \cup \{?\}$. The similarity between the two labels l_1 and l_2 is given by:

$$\mathrm{sim}(l_1, l_2) = \begin{cases} 1 & \text{if } l_1 = l_2 \text{ and } l_1 \neq ?,\ l_2 \neq ? \\ 0 & \text{if } l_1 \neq l_2 \text{ and } l_1 \neq ?,\ l_2 \neq ? \\ 0.5 & \text{if } l_1 = ? \text{ or } l_2 = ? \end{cases}$$

Thus, whenever l_i is the wildcard " ?", the similarity is 0.5. Otherwise, it is determined by the equality comparison of the two labels.

The similarity between the two values x_1 and x_2 is based on string comparison. We utilize the Jaccard similarity between the 3-grams of x_i:

$$\mathrm{sim}(x_1, x_2) = \mathrm{Jaccard}(\mathrm{3grams}(x_1), \mathrm{3grams}(x_2))$$

Finally, the similarity of the two labeled values is computed as the product of the similarity between labels and similarity between values.

$$\mathrm{sim}((l_1 : x_1), (l_2, x_2)) = \mathrm{sim}(l_1, l_2) \cdot \mathrm{sim}(x_1, x_2)$$

Recall that the user query is a partial tuple $\{(l_i, q_i) : i \in I_Q\}$, and the search results are complete tuples of the form $\{(l_j, v_j) : j \in I_R\}$. We need to generalize the similarity measure in Definition 2 to partial tuples.

Definition 3 (Similarity of partial tuples): Given two tuples: $\vec{q} = \{(l_i, q_i) : i \in I_Q\}$ and $\vec{r} = \{(l_j, v_j) : j \in I_R\}$. We define their similarity based on the optimal matching of labeled values from $\vec{q}$ to $\vec{r}$.

Let $H \subseteq I_Q \times I_R$ be the optimal matching that maximizes the total similarity score. The similarity of two tuples is given as:

$$\mathrm{sim}(\vec{q}, \vec{r}) = \sum_{(i,j)\in H} \mathrm{sim}((l_i, q_i), (l_j, v_j))$$

Note that in order to compute the similarity between a query $\vec{q}$ and a complete tuple $\vec{r}$, we need to first compute the similarities between labeled values from $\vec{q}$ and those from $\vec{r}$. Then we find optimal matches from $\vec{q}$ to $\vec{r}$, and use those to get the total similarity score.

Definition 4 (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 $\mathrm{sim}(\vec{Q}, \vec{r})$.

The process of partial tuple search, illustrated in Figure **??**, consists of encoding partial tuple queries, searching full-text indexes, and find optimal matching among search results for the partial tuple. We provide detailed descriptions about partial tuple search in Section **??**.

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:

Definition 5 (Neural network classifier): Let $R_1, R_2, R_3, \ldots, R_K$ be K relations. The neural network classifier takes a partial tuple query $\vec{q}$ and estimates the probabilities that the search result of $\vec{q}$ belongs to relation R_i, $\forall i \in \{1, 2, \ldots, K\}$.

IV. Partial tuple search using full-text search

Traditional full-text indexes, such as Apache Lucene, support keyword queries over large document collections using an inverted index data structure. In this section, we describe how a full-text index is incorporated as the first stage of the query processing pipeline for partial tuple search.

The advantages of using a full-text index are:

- Flexible and partial string matching between query keywords and document text

my/graphics/tuple_completion.pdf

Fig. 1. Matching a partial tuple $\vec{q}$ to a complete tuple t

- Efficient query processing provided that the inverted document lists are not too long due to collisions.

We first encode complete relational tuples as documents to be stored in the full-text index. The partial tuple query $\vec{q}$ is encoded as a keyword query. It is important to point out that we must specify the same tokenizer to convert relational tuples and query tuples to tokenized terms. We use the 3-gram tokenizer to support fuzzy string matching for the values.

The full-text index evaluates the keyword query, and computes a top-k result set of the best matching documents as relational tuple candidates. These top-k candidates guarantee high similarity between the labeled values in $\vec{q}$ and those in relational tuples.

A. *Encoding of tuples as documents*

Let us denote the tokenizer function as $\mathbf{tokenize}$. A tuple $\vec{r}$ is converted into a document doc as follows:

$$\mathrm{attr}(\mathrm{doc}) = \mathrm{attr}(\vec{r}) \cup \{\texttt{fulltext}\}$$

For each $a \in \mathrm{attr}(\vec{r})$,

$$\mathrm{doc}[a] = \textbf{tokenize}(\vec{r}[a])$$

Also,

$$\mathrm{doc}[\texttt{fulltext}] = \bigcup \left\{ \textbf{tokenize}(\vec{r}[a]) : a \in \mathrm{attr}(\vec{r}) \right\}$$

The special field `fulltext` of doc contains all tokenized terms of the tuple $\vec{r}$.

Example 2: Given a tuple $\vec{r}$ as:

$$(\mathrm{Name} : \mathit{Jack},\ \mathrm{Address} : 100\ \mathit{Simcoe\ Street})$$

Its attributes $\mathrm{attr}(\vec{r})$ are $\{\mathrm{Name}, \mathrm{Address}\}$. Therefore, the document attributes $\mathrm{attr}(doc)$ are $\{\mathrm{Name},\ \mathrm{Address},\ \mathrm{fulltext}\}$.

If we use a standard tokenizer, the result of tokenization will be,

$$\begin{bmatrix} \text{Name} & \mapsto & \text{Jack} \\ \text{Address} & \mapsto & \text{100, Simcoe, Street} \\ \text{fulltext} & \mapsto & \text{Jack, 100, Simcoe, Street} \end{bmatrix}$$

On the other hand, if we use a character-based 3-gram tokenizer with a special padding character "_", the result will be,

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

B. Encoding partial tuple queries as keyword queries

Given a partial tuple $\vec{q}$, we want to encode it as a keyword query such that the search engine can find the documents that correspond to the relevant tuples.

For each labeled value (l_i, x_i) in the partial tuple, we generate a query clause.

- If $l_i \neq ?$, then the query clause is $q_i = l_i : x_i$.
- If $l_i = ?$, then the query clause is $q_i = \texttt{fulltext} : x_i$.

The generated query is:

$$q = q_1 \textbf{ or } q_2 \textbf{ or } \ldots$$

One can see that any complete tuple $\vec{r}$ that satisfy the query $\vec{q}$ will have its encoding document satisfying the keyword query q.

C. From search results to partial tuple completion

For our problem, we need to match the partial tuple $\vec{q}$ with a complete tuple t. This can be done by further post-process the documents in search results returned by the search engine.

Recall that a tuple matching is a function $h : \mathrm{LV}(\vec{q}) \to \mathrm{LV}(t)$, where $\mathrm{LV}(\vec{q})$ and $\mathrm{LV}(t)$ are labeled values of $\vec{q}$ and t, respectively. The top completion of $\vec{q}$ should be a complete tuple that has the highest matching score based on the similarity measure. Since the search engine already returns the top-k candidate documents, we can compute the optimal matching between $\vec{q}$ to each of the candidates in order to rank the candidates with respect to their matching scores.

Finding optimal matching of labeled values from the partial tuple $\vec{q}$ to a candidate tuple t is equivalent to solving the maximum weighted matching of a bipartite graph.

Define the graph G as:

- The vertices are: $V = \mathrm{LV}(t) \cup \mathrm{LV}(\vec{q})$.
- The edges and their respective weights are defined as:

  $$E = \{\langle x_1, x_2, \mathrm{sim}(V(x_1), V(x_2))\rangle : x_1 \in \mathrm{LV}(t) \text{ and } x_2 \in \mathrm{LV}(\vec{q}) \text{ and}$$

  where $L(x_1)$ and $L(x_2)$ are labels of x_1 and x_2, respectively.
- G is the weighted graph (V, E).

The max-weighted matching of G is necessarily a matching h from $\mathrm{LV}(\vec{q})$ to $\mathrm{LV}(t)$. The weight sum of h is the matching score based on the similarity of matched labeled values. It is now that the optimal matching can be found exactly in polynomial time using various algorithms, e.g., Hungarian algorithm [**?**].

V. Optimizing query processing pipeline with neural networks

As described in Section **??**, a potential bottleneck associated with inverted index structures for full-text search is the hashing collision which creates long linked lists at the leaf nodes of the index tree (See Figure **??**).

In our application, the character based 3-gram tokenizer will generate a compact vocabulary consisting of at most $|\mathrm{charset}|^3$ distinct tokens where charset is the total character set. So, when the number of documents grow greater than $|\mathrm{charset}|^3$, the linked lists at the leaf nodes will grow linearly with respect to the number of relational tuples.

The concern is that the full-text index will degrade as the dataset size grows due to the 3-gram tokenization that we employ to support fuzzy string matching. Unfortunately, our experimental evaluation in Section **??** confirms the performance degradation in practice.

In this section, we will describe a method to perform partitioning of the full-text index to overcome the performance bottleneck caused by hash collisions.

Our method utilizes a neural network to optimize the index access order during partial tuple search. The neural network is to be embedded in the overall query processing pipeline, and will be self-supervised based on existing data.

A. Partition of full-text index

An aggregated index is a function as follows:

$$\textbf{index} : \textbf{Query} \to \textbf{List}[\textbf{Tuple}]$$

It indexes all the tuples from every relation.

However, we advocate to partition the tuples based on the relation they below to. Thus, if we have relations $R_1, R_2, R_3, \ldots, R_n$, we will have n indexes $\{\textbf{index}_1$, $\textbf{index}_2$, …, $\textbf{index}_n\}$, each indexing the tuples of a single relation.

$$\textbf{index}_i : \textbf{Query} \to \textbf{List}[\textbf{Tuple}]$$

The search can be done concurrently over all indexes:

```
priority_queue
for index_i in all_indexes {
  spawn index_i.search(q) into priority_queue
}
```

We can also do sequential access of the indexes:

```
for index_i in sorted(all_indexes, q) {
    index_i.search(q) into priority_queue
}
```

In this thesis we focus on the sequential access approach in favour of saving CPU load. The challenge is to sort the indexes dynamically based on the user query $\vec{q}$. The sorting should place relations that can satisfy $\vec{q}$ with higher priority, so that successful matches are found as early as possible.

Thus, our strategy to sort the indexes is based on a sorting key function:

$$S : (\textbf{index}_i, \vec{q}) \to [0,1] \mapsto \textbf{prob}(\text{result}(\vec{q}) \in R_i)$$

The sorting score $S(\textbf{index}_i, \vec{q})$ is the (estimate of the) probability that the search result of $\vec{q}$ belongs to the relation R_i. Since there are only a finite many relations, we can reformulate the scoring function S to as an n-way classification function:

$$\textbf{classify} : \vec{q} \mapsto \begin{bmatrix} S(\textbf{index}_1, \vec{q}) \\ S(\textbf{index}_2, \vec{q}) \\ \vdots \\ S(\textbf{index}_n, \vec{q}) \end{bmatrix}$$

Our objective is to learn the **classify** function using a neural network.

B. Vectorization of queries

The classifier can be trained using a neural network that can be embedded in the query processing pipeline. The design objectives are:

- The neural network must be compact in size so that it incurs minimal performance overhead, and can be embedded in a search system.
- The training of the network must require minimal manual intervention. Thus the neural network must be trained with self-supervision from existing data.

The classification function is:

$$\textbf{classify} : \textbf{Query} \to \mathbb{R}^n \tag{1}$$

where the output of **classify** is the probability distribution over the n partitioned indexes. Elements we presented in Section **??** present several options for the neural network architecture.

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.

$$\textbf{tokens}(\vec{q}) = \{l_1\} \oplus \textbf{tokenize}(x_1) \oplus \{l_2\} \oplus \textbf{tokenize}(x_2) \oplus \ldots \{l_m\} \cup \textbf{toke}$$

where $\oplus$ is sequence concatenation.

Integer encoding of queries: next we encode $\textbf{tokens}(\vec{q})$ using a universal vocabulary **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:

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

The integer sequence of a query is given by:

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

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

$$\textbf{vector}(\vec{q}) = \textbf{Embedding}(\textbf{sequence}(\vec{q})) \in \mathbb{R}^{|q| \times d}$$

At this point, we have many options in mapping the vector sequence to the probability distribution in $\mathbb{R}^n$.

For the remainder of this chapter, we denote the vector sequence representation of $\vec{q}$ as:

$$\vec{x} = \textbf{sequence}(\vec{q}) = (x_1, x_2, \ldots, x_{|q|})$$

where each $x_i \in \mathbb{R}^d$ is the embedding vector of the i-th token in a d-dimensional latent space.

C. Neural network architectures for query classification

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.

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

Recurrent network architecture with LSTM cells.

Since recurrent neural networks (RNN) are specifically designed to process sequence inputs, we can utilize a RNN to map the input $\vec{x}$ to a flattened state vector, which can then be processed by a MLP to compute the output probability distribution.

The advantage of RNN is that it can learn inter-token dependencies in the query at the cost of a bigger model size and higher training cost.

$$\begin{array}{llr} & \vec{x} & (|q|, d) \\ \to & \mathbf{LSTM}(\text{output-state=True}) & (d) \\ \to & \mathbf{MLP}(\text{output-dim} = n) & (n) \\ \to & \mathbf{softmax}(\cdot) & (n) \end{array}$$

1D convolution architecture.

Another standard technique to learn sequential features is to use 1D convolution. The 1D convolution layer (Section **??**) produces a sequence of *feature* vectors that capture the short-range token dependencies within the convolution window size. We then use global averaging to flatten the convolution features for further processing using a MLP.

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

Transformer and MLP Mixer

Transformers have shown to be a superior architecture for many tasks in the domain of MLP and sequence learning. The exact architecture of a transformer block is shown in the implementation section (See Figure **??**). 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. Details of the MLP mixer are shown in Figure **??**.

D. *Unsupervised training of neural network classifiers*

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.

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

Thus, given a database with n relations $R_1, R_2, \ldots, R_n$:

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

VI. Overall query processing pipeline

To summarize the proposed query processing pipeline for partial tuple search, we have the following stages:

1) A partial tuple is entered as user input.
2) The partial tuple is tokenized and converted to a structured keyword query.
3) The neural network classifier sorts the partitioned indexes based on the probability scores.
4) The partitioned indexes are scanned to find top-k complete tuple candidates from the database.
5) The candidates are ranked by max-matching based similarity scores.

Our pipeline requires that partitioned full-text indexes are built from tuples of each relation in offline mode. In addition, the neural network classifier should be trained using sampled tuples from each relation.

In Chapter **??**, we will describe the implementation of each neural network architecture. We will also discuss the experimental evaluation and comparison of the performance gain of each neural network architecture.

VII. Related Work

Early papers: Hristidis and Papakonstantinou [**?**] presented DISCOVER, a system that allowed users to submit keyword queries to relational databases without the knowledge of the underlying database schema. DISCOVER processes keyword queries to generate candidate networks of relations and create execution plans to be submitted to RDBMS, which will return search results. Liu et al. [**?**] proposed an information retrieval (IR) ranking strategy for effective keyword search related to relational databases. The ranking strategy used four normalization factors for computing ranking scores. All answers of a query were ranked based on computed ranking scores, and the top-k answers were returned as results.

Extension to include frequent co-occurring term (FCT): Tao and Yu [**?**] proposed an operator called frequent co-occurring term (FCT) search, and an algorithm that could solve FCT search effectively without using the conventional keyword search methods. The purpose of FCT search is to extract the terms that most accurately represent a set of keywords, i.e., to discover the concepts closely related to the keywords set.

Survey of keyword search: A survey about research on keyword search in relational databases was done by Park and Lee [**?**]. First they listed fundamental characteristics of keyword search in relational databases: an indexing structure, being able to formalize internal queries based on query keywords, being able to correctly constructing candidate answers, and an answer ranking strategy. Then they investigated five research dimensions, including data representation, ranking, efficient processing, query representation, and result presentation. At the end, they pointed out some promising research directions. One of them is efficient top-k query processing. The authors believed that keyword search in relational databases would benefit from advance in top-k query processing techniques.

Interpretation of keywords in query: Zeng et al. [**?**] presented a framework based on keyword query interpretations. It incorporated human feedback to remove keyword vaguenesses and use a ranking model for query interpretation evaluation afterwards.

Top-k recommendation: Meng et al. [**?**] presented an approach to solve typical and semantically related queries to a given query. This can help users to explore their query intentions and improve their query formulation.

Index optimization: Ding et al. [**?**] presented an updatable learned index called ALEX, an in-memory index structure, for index optimization. ALEX addressed practical issues related to various types of workloads with dynamic updates. RadixSpline [**?**], another learned index, tackled the issue of index implementation difficulty. RadixSpline offered quick build using a single pass over data while achieving competitive performance.

Query optimization: Bao [**?**] is a learned query optimization system using reinforcement learning. It can learn from mistakes and adapt to dynamic workloads, data, and schema, thus is capable of applying per-query optimization hints.

Cost models for query processing: Siddiqui1 et al. [**?**] investigated how to learn cost models from cloud workloads for big data systems. The learned cost models could be integrated with existing query optimizers. Such a query optimizer could make accurate cost predictions, which can improve resource efficiency in big data systems.

REFERENCES

[1] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 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] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*, pages 1–5, 2020.

[4] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574, 2006.

[5] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *ACM SIGMOD Record*, 51(1):6–13, 2022.

[6] Xiangfu Meng, Longbing Cao, Xiaoyan Zhang, and Jingyu Shao. Top-k coupled keyword recommendation for relational keyword queries. *Knowledge and Information Systems*, 50:883–916, 2017.

[7] Jaehui Park and Sang-goo Lee. Keyword search in relational databases. *Knowledge and Information Systems*, 26:175–193, 2011.

[8] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 99–113, 2020.

[9] Yufei Tao and Jeffrey Xu Yu. Finding frequent co-occurring terms in relational keyword search. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 839–850, 2009.

[10] Zhong Zeng, Zhifeng Bao, Tok Wang Ling, and Mong Li Lee. isearch: an interpretation based framework for keyword search in relational databases. In *Proceedings of the Third International Workshop on Keyword Search on Structured Data*, pages 3–10, 2012.