Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning

Grace Fan
Northeastern University
United States
fan.gr@northeastern.edu

Jin Wang
Megagon Labs
United States
jin@megagon.ai

Yuliang Li
Megagon Labs
United States
yuliang@megagon.ai

Dan Zhang
Megagon Labs
United States
dan_z@megagon.ai

Renée J. Miller
Northeastern University
United States
miller@northeastern.edu

ABSTRACT

Dataset discovery from data lakes is essential in many real application scenarios. In this paper, we propose Starmie, an end-to-end framework for dataset discovery from data lakes (with table union search as the main use case). Our proposed framework features a contrastive learning method to train column encoders from pre-trained language models in a fully unsupervised manner. The column encoder of Starmie captures the rich contextual semantic information within tables by leveraging a contrastive multi-column pre-training strategy. We utilize the cosine similarity between column embedding vectors as the column unionability score and propose a filter-and-verification framework that allows exploring a variety of design choices to compute the unionability score between two tables accordingly. Empirical results on real table benchmarks show that Starmie outperforms the best-known solutions in the effectiveness of table union search by 6.8 in MAP and recall. Moreover, Starmie is the first to employ the HNSW (Hierarchical Navigable Small World) index to accelerate query processing of table union search which provides a 3,000X performance gain over the linear scan baseline and a 400X performance gain over an LSH index (the state-of-the-art solution for data lake indexing).

PVLDB Reference Format:
Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. PVLDB, 16(7): 1726 - 1739, 2023.
doi:10.14778/3587136.3587146

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/megagonlabs/starmie.

1 INTRODUCTION

The growing number of open datasets from governments, academic institutions, and companies have brought new opportunities for innovation, economic growth, and societal benefits. To integrate and analyze such datasets, researchers in both academia and industry have built a number of dataset search engines to support the application of dataset discovery [3, 7, 17, 19, 33, 40, 45]. One popular example is Google's dataset search [3] which provides keyword search on the metadata. However, for open datasets, simple keyword search might suffer from data quality issues of incomplete and inconsistent metadata across different datasets and publishers [1, 16, 41, 42]. Thus it is essential to support *table search* over open datasets, and more generally data lake tables (including private enterprise data lakes), to boost dataset discovery applications, such as finding related tables, domain discovery, and column clustering.

Finding related tables from data lakes [11, 25, 39, 46, 57] has a wide spectrum of real application scenarios. There are two sub-tasks of finding related tables, namely table union search and joinable table search. In this paper, we mainly focus on the problem of *table union search*, which has been recognized as a crucial task in dataset discovery from data lakes [2, 24, 39, 41, 42, 57, 61]. Given a query table and a collection of data lake tables, table union search aims to find all tables that are unionable with the query table. To determine whether two tables are unionable, existing solutions first identify all pairs of unionable columns from the two tables based on column representations, such as bag of tokens or bag of word embeddings. They then devise some mechanism to aggregate the column-level results to compute the table unionability score.

State-of-the-art: Early work on finding unionable tables used table clustering followed by simple syntactic measures such as the difference in column mean string length and cosine similarities to determine if two tables are unionable [4]. Table union search [42] improved on this by applying a rich collection of column representations including syntactic, semantic (leveraging ontologies), and natural language (based on word-embeddings) column representations. Two important innovations of this work were the modeling of data lake context to create an *ensemble unionability score* which models the surprisingness of a score given the score distributions within a data lake and the use of LSH indices to make table union search fast over large data lakes [42]. More recently D^3L [2] added additional column representations based on regular expression matching and SANTOS [24] added to the column representations, representations of binary relationships. In parallel to these search-based approaches, the mighty hammer of deep learning has been applied to the problem of column matching (determining the semantic type of a column) [22, 56]. Since these

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587146

Table A:

Name	Mode of Travel	Purpose	Destination	Day	Month	Year	Expense
Philip Duffy	Air	Regional Meeting	London	10	April	2019	189.06
Jeremy Oppenheim	Taxi	Exchange Visit	Ottawa	30	Jul	2019	8.08
Mark Sedwill	Air	Evening Meal	Bristol	02	September	2019	50

Table B:

Name	Date	Destination	Purpose
Clark	23/07	France	Discuss EU
Gyimah	03/09	Belgium	Build Relations
Harrington	05/08	China	Discuss Productivity

Table C:

Bird Name	Scientific Name	Date	Location
Pine Siskin	Carduelis Pinus	2019	Ottawa
American Robin	Turdus migratorius	2019	Ottawa
Northern Flicker	Colaptes auratus	2019	London

Figure 1: An example of table union search on Open Data.

approaches are supervised, they can only be applied to finding a limited set of semantic types (78 in their experiments), and while not a general solution for unionability in data lakes, they can be used in an offline fashion to find unionable tables containing the types on which they are trained.

However, there are still plenty of opportunities to further improve the performance of table union search. One important issue is to learn sufficient contextual information between columns in tables so as to determine the unionability. This point can be illustrated in the following motivation example.

Example 1.1. Figure 1 shows an example of finding unionable tables. Given the query Table A, existing approaches first find unionable columns. In this example, the column `Destination` in Table A will be deemed more unionable with `Location` from Table C than with `Destination` from Table B. This is because the syntactic similarity score, e.g. overlap and containment Jaccard, between the two `Destination` columns is 0; while the average word embedding of cities (Table A) is also not as close to that of nations (Table B). Similarly, if an ontology is used, Table A and Table C shares the same class while the values in B are in different (though related) classes. Meanwhile, looking at the tables as a whole we observe that Table A is actually irrelevant to Table C. But as existing solutions only look at the pair of single columns when calculating column unionability score, the columns `Year/Date` and `Destination/Location` of the two tables might be wrongly aligned together. Even techniques that look at relationships [24] can be fooled by the value overlap in this relationship and determine the relationship `Year-Destination` in Table A to be unionable with `Date-Location` in Table C. This kind of mistake can be avoided by looking at a table's context, i.e. information carried by other columns within a table. Looking at the table as a whole, a method should be able to recognize that the `Year` in Table A is part of a travel date while in Table C it is the date of discovery of a bird; and `Destination` in Table A refers to the cities to which the officers are traveling; whereas `Location` in Table C is the city where a bird is found.

From the above example, we focus on the following challenges in proposing a new solution. Firstly, it is essential to learn richer semantics of columns based on natural language domain. To this end, we require a more powerful approach to learn the column representation so as to capture richer information instead of relying on simple methods like the average over bag of word embeddings utilized in previous studies [2, 14] or even the similarity of the word embedding distributions [42]. Secondly, we argue that it is crucial to utilize the contextual information within a table to learn the representation of each column, which is ignored by previous studies. Even proposals for capturing relationship semantics do not use contextual information to learn column representations [24]. Finally, due to the large volume of data lake tables, it is also a great challenge to develop a scalable and memory-efficient solution.

We propose Starmie, an end-to-end framework for dataset discovery from data lakes with table union search as the main use case. Starmie uses pre-trained language models (LMs) such as BERT [13] to obtain semantics-aware representations for columns of data lake tables. While pre-trained LMs have been shown to achieve state-of-the-art results in table understanding applications [12, 31, 47], their good performance heavily relies on high-quality labeled training data. For the problem setting of table union search [41, 42], we must come up with a fully unsupervised approach in order to apply pre-trained LMs to such applications, something not yet supported by previous studies. Starmie addresses this issue by leveraging *contrastive representation learning* [10] to learn column representations in a **self-supervised manner**. An innovation of this approach is to assume that two randomly selected columns in a data lake can be used as negative training examples. For positive examples, we propose and use novel data augmentation methods. The framework defines a learning objective that connects the same or similar columns in the representation space while separating distinct columns. As such, Starmie can apply the pre-trained representation model in downstream tasks such as table union search without requiring any labels. We also propose to combine the learning algorithm with a novel multi-column table transformer model to learn *contextualized* column embeddings that model the column semantics depending on not only the column values, but also their context within a table. While a recent study SANTOS [24] can reach a similar goal by employing a knowledge base, our proposed methods can automatically capture such contextual information from tables in an unsupervised manner without relying on any external knowledge or labels.

Based on the proposed column encoders, we use cosine similarity between column embeddings as the column unionability score and develop a bipartite matching based method to calculate the table unionability score. We propose a filter-and-verification framework that enables the use of different indexing and pruning techniques to reduce the number of computations of the expensive bipartite matching. While most previous studies employed LSH index to improve the search performance, we also make use of HNSW (Hierarchical Navigable Small World) index [36] to accelerate query processing. Experimental results show that HNSW can significantly improve the query time while only slightly reducing the MAP/recall scores. Besides table union search, we further conduct two case studies to show that Starmie can also support other dataset discovery applications such as joinable table search and column clustering. We believe these results show great promise in the use of contextualized, self-supervised embeddings for many table understanding tasks.

Our contributions can be summarized as the following.

- We propose Starmie, an end-to-end framework to support dataset discovery over data lakes with table union search as the main use case.
- We develop a contrastive learning framework to learn contextualized column representations for data lake tables without requiring labeled training instances. Starmie achieves an improvement of 6.8% in both MAP and recall compared with the best state-of-the-art method, with a MAP of 99%, a significant margin compared with previous studies.

- We design and implement a filter-and-verification based framework for computing the table-level unionability score which can accommodate multiple design choices of indexing and pruning to accelerate the overall query processing. By leveraging the HNSW index, Starmie achieves up to three orders of magnitude in performance gain for query time relative to the linear scan baseline.
- We conduct an extensive set of experiments over two real world data lake corpora. Experimental results demonstrate that the proposed Starmie framework significantly outperforms existing solutions in effectiveness. It also shows good scalability and memory efficiency.
- We further conduct case studies to show the flexibility and generality of our proposed framework in other dataset discovery applications.

2 OVERVIEW

2.1 Problem definition

A data lake consists of a collection of tables $\mathcal{T}$. Each table $T \in \mathcal{T}$ consists of several columns $\{t_1, \ldots, t_m\}$ where each column t_i can be from different domains. Here m is the number of columns in table T (denoted as $|T| = m$). We will use the notation T to denote both the table and its set of columns if there is no ambiguity. To determine the unionability between two columns, following previous studies, we employ *column encoders* to generate the representations of columns. Then the *column unionability score* can be computed to measure the relevance between those representations. A column encoder $\mathcal{M}$ takes a column t as input and outputs $\mathcal{M}(t)$ as the representation. Given two columns t_i and t_j, the column unionability score is computed as $\mathcal{F}(\mathcal{M}(t_i), \mathcal{M}(t_j))$, where $\mathcal{F}$ is a scoring function between two column representations.

Based on the column unionability scores, we compute the table unionability score between two tables, which is obtained by aggregating the column unionability scores introduced above. Given two tables S and T, we define a table unionability scoring mechanism as $U = \{\mathcal{F}, \mathcal{M}, \mathcal{A}\}$, where $\mathcal{M}$ and $\mathcal{F}$ are the column encoder and scoring function for two column representations, respectively. Here $\mathcal{A}$ is a mechanism to aggregate the column unionability scores between all pairs of columns from the two tables. We will introduce the details of $\mathcal{A}$ later in Section 4.

Following the above discussions, we can formally define the table union search problem as a top-k search problem as Definition 2.1:

Definition 2.1 (Table Union Search). Given a collection of data lake tables $\mathcal{T}$ and a query table S, top-k table union search aims at finding a subset $\mathcal{S} \subseteq \mathcal{T}$ where $|\mathcal{S}| = k$ and $\forall T \in \mathcal{S}$ and $T' \in \mathcal{T} - \mathcal{S}$, we have $U(S, T) \geq U(S, T')$.

2.2 System architecture

Figure 2 shows the overall architecture of Starmie that solves table union search in two stages: offline and online.

During the offline stage, Starmie pre-trains a column representation model that encodes columns of data lake tables into dense high-dimensional vectors (i.e., column embeddings). Then, we apply the trained model to all data lake tables to obtain the column embeddings via model inference. We store the embedding vectors in

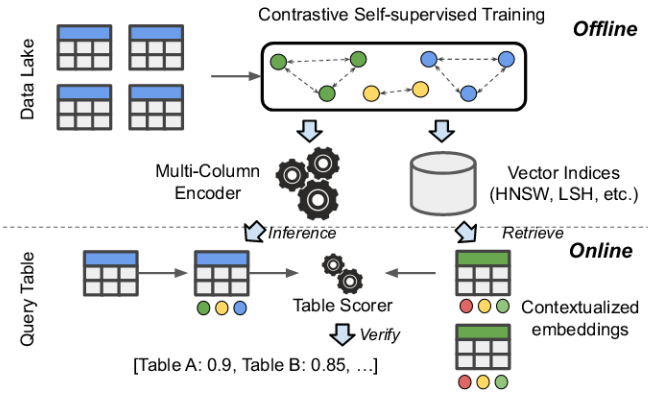


Figure 2: During the offline phase, Starmie **pre-trains a multi-column table encoder using contrastive learning and stores the embeddings of data lake columns in vector indices like HNSW. During online processing,** Starmie **retrieves candidate tables with similar contextualized column embeddings then verifies their table-level unionability scores using column alignment algorithms.**

efficient vector indices for online retrieval. A key challenge for the offline stage is to train high-quality column encoders that capture the semantics of tabular data. In Starmie, we follow a recent trend [12, 31, 47] of table representation learning that encodes tabular data using pre-trained language models (LMs). Pre-trained LMs have achieved state-of-the-art performance on table understanding tasks such as column type and relation type annotation [47]. However, the good performance of pre-trained LMs requires fine-tuning on high-quality labeled datasets, which are always not available in table search applications such as table union search. Using pre-trained LMs off-the-shelf is also problematic as the column embeddings cannot capture (ir-)relevance between columns or the contextual information within tables. To this end, in Section 3, we propose a contrastive learning framework for learning high-dimensional column representations in fully unsupervised manner. We combine the framework with a multi-column table model that captures column semantics from the column values while taking the table context into account. Then we apply the column encoder to all tables to convert each table into a collection of embedding vectors.

During the online stage, given an input query table, we retrieve a set of candidate tables from the vector indices by searching for data lake column embeddings of high column-level similarity with the input columns. Starmie then applies a verification step for checking and ranking the candidates for the top-k tables with the highest table-level unionability scores. The first challenge for the online stage is how to efficiently search for unionable columns. This is not a trivial task due to the massive size of data lakes. We address this challenge by allowing different design choices of state-of-the-art high-dimensional vector indices. Yet another challenge is designing a table unionability function that can effectively aggregate the column unionability scores. As in other studies, we employ weighted bipartite graph matching. To address its limitation of high computation complexity, we introduce a novel algorithm to reduce the number of expensive calls to the exact matching algorithm by deducing lower and upper bounds of the matching score (Section 4).

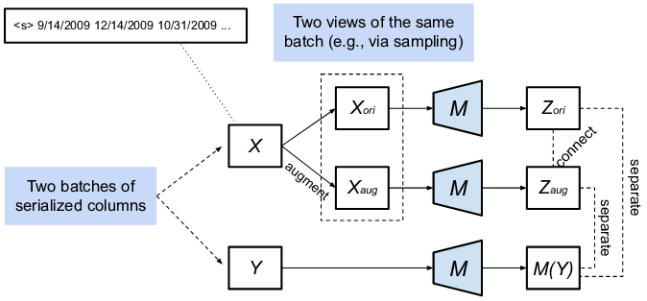


Figure 3: Contrastive learning with single-column input.

3 LEARNING CONTEXTUALIZED COLUMN EMBEDDINGS

We now describe the offline stage for training high-quality column encoders. The encoder pre-processes tables into sequenced inputs and uses a pre-trained LM to encode each column into a high-dimensional vector. Next we describe a novel contrastive learning approach for table encoders in Section 3.2 and generalize it to multi-column encoders for contextualized embeddings in Section 3.3. Finally, we describe the table pre-processing approaches to generate the input for such learning processes in Section 3.4.

3.1 Background

Contrastive learning is a self-supervision approach that learns data representations where similar data items are close while distinct data items are far apart. In this work, we adopt a popular contrastive learning framework SimCLR [10] as the cornerstone. Figure 3 illustrates the high-level idea of the algorithm. The goal is to learn an encoder $\mathcal{M}$ (e.g., a column encoder) that takes a data item (e.g., a column) as input and encodes it into a high-dimensional vector. To train the encoder in a self-supervised manner without labels, SimCLR relies on (1) a data augmentation operator generating semantic-preserving views (in our context this means X_{ori} and X_{aug} that are unionable) of the same data item and (2) a sampling method (e.g., uniform sampling from a large collection) that returns pairs of data items (i.e., X and Y) that are distinct (i.e. non-unionable) with high probability. SimCLR then applies a contrastive loss function that connects the representations of the semantic-preserving (unionable) views meanwhile separating those of the sampled distinct (non-unionable) items. Next, we illustrate how we apply the algorithm for training a single-column encoder.

3.2 Contrastive Learning Framework

The goal is to connect representations of the same or unionable columns in their representation space while separating representations of distinct columns. To achieve the first goal, Algorithm 1 leverages a data augmentation operator op (Line 5). Given a batch of columns $X = \{x_1, \ldots, x_N\}$ where N is the batch size, op transforms X into a semantics-preserving view X_{aug}. We design the augmentation operator to be uniform sampling of the values from the original column. By doing so, we can generate diverse views of the same column while all views preserve the original semantic types. Then $\mathcal{M}$ can encode the batches X (also X_{ori} which is a copy of X) and

Algorithm 1: SimCLR pre-training

Input: A collection D of data lake columns
Variables : Number of training epochs n_epoch;
Data augmentation operator op; Learning rate η
Output: An embedding model $\mathcal{M}$

1 Initialize $\mathcal{M}$ using a pre-trained LM;
2 **for** ep = 1 *to* n_epoch **do**
3 Randomly split D into batches $\{B_1, \ldots B_n\}$;
4 **for** $B \in \{B_1, \ldots B_n\}$ **do**
  /* augment and encode every item */
5 $B_{\text{ori}}, B_{\text{aug}} \leftarrow \text{augment}(B, \text{op})$;
6 $\vec{Z}_{\text{ori}}, \vec{Z}_{\text{aug}} \leftarrow \mathcal{M}(B_{\text{ori}}), \mathcal{M}(B_{\text{aug}})$;
  /* Equation (1) and (2) */
7 $\mathcal{L} \leftarrow \mathcal{L}_{\text{contrast}}(\vec{Z}_{\text{ori}}, \vec{Z}_{\text{aug}})$;
  /* Back-prop to update $\mathcal{M}$ */
8 $\mathcal{M} \leftarrow \text{back-propagate}(\mathcal{M}, \eta, \partial\mathcal{L}/\partial\mathcal{M})$;
9 **return** $\mathcal{M}$;

X_{aug} into column embedding vectors $\vec{Z}_{\text{ori}}$ and $\vec{Z}_{\text{aug}}$ respectively. Note that $\vec{Z}_{\text{ori}}$ and $\vec{Z}_{\text{aug}}$ are both matrices with size N times the dimension of embedding vector (e.g., 768 for BERT).

Next, the algorithm leverages a *contrastive loss function* to connect the semantics-preserving views of columns and separate representations of distinct columns (Line 6). More specifically, let $\vec{Z} = \{\vec{z}_i\}_{1 \le i \le 2N}$ be the concatenation of the two encoded views $\vec{Z}_{\text{ori}}$ and $\vec{Z}_{\text{aug}}$ of batch X introduced above. Here $\vec{z}_i$ is the i-th element of $\vec{Z}_{\text{ori}}$ for $i \le N$ and the $(i-N)$-th element of $\vec{Z}_{\text{aug}}$ for $i > N$. We first define a single-pair loss $\ell(i, j)$ for an element pair $(\vec{z}_i, \vec{z}_j)$ to be Equation 1.

$$\ell(i,j) = -\log \frac{\exp\left(\text{sim}\left(\vec{z}_i, \vec{z}_j\right)/\tau\right)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i, k \neq j]} \exp\left(\text{sim}\left(\vec{z}_i, \vec{z}_k\right)/\tau\right)} \tag{1}$$

where sim is a similarity function such as cosine and τ is a temperature hyper-parameter in the range $(0, 1]$. We fix τ to be 0.07 empirically. Intuitively, by minimizing this loss for a pair $(\vec{z}_i, \vec{z}_j)$ that are views of the same columns, we (i) maximize the similarity score $\text{sim}\left(\vec{z}_i, \vec{z}_j\right)$ in the numerator and (ii) minimize $\vec{z}_i$'s similarities with all the other elements in the denominator.

Next, we can obtain the contrastive loss by averaging all matching pairs shown in Equation 2 (Line 7):

$$\mathcal{L}_{\text{contrast}} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(k, k+N) + \ell(k+N, k)] \tag{2}$$

where each term $\ell(k, k+N)$ and $\ell(k+N, k)$ refers to pairs of views generated from the same column.

3.3 Multi-column Table Encoder

While the method shown in Algorithm 1 learns column representations based on values within a column itself, it cannot take the contextual information of a table into account. For example, the single-column model can understand that a column consisting of values "1997 1998 ..." is a column about years, but depending on the context of other columns present in the same table, the same column can represent "years in which a species of bird was observed

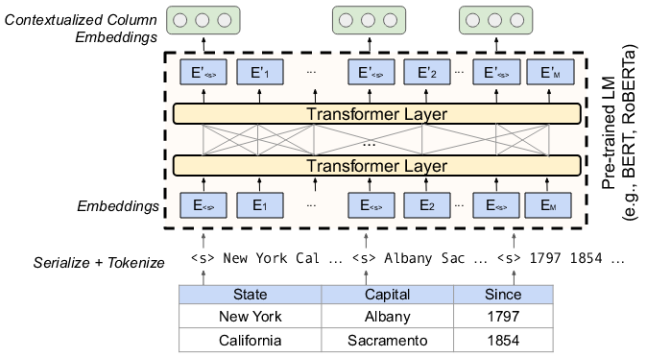


Figure 4: Multi-column table encoder.

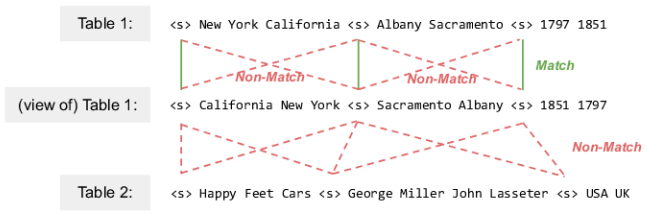


Figure 5: Contrastive learning positive and negative pairs.

in a specific area" or "years of car production" etc. As illustrated in Figure 1, such understanding is important for deciding whether two tables are unionable or not.

To address this problem, Starmie combines contrastive learning with a *multi-column table encoder* illustrated in Figure 4. The model starts with serializing an input table into a string by concatenating cell values from each column. Following the implementation of tokenizers in the HuggingFace library, it also adds a special separator token "<s>" to indicate the start of each column. Next, we feed the sequence as the input to a pre-trained LM such as RoBERTa [35] and use the output vector of the special token in the beginning of the sequence following the paradigm of fine-tuning.

The pre-trained LM first converts the input sequence into a sequence of token embeddings independent of their context then applies 12 or more Transformer layers [48] on top. The self-attention mechanism in the Transformer layers convert the word embeddings into a sequence of *contextualized embeddings*. These vector representations depend not only on the tokens themselves (e.g., "1797") but also their context (e.g., "Albany"). As such, we can extract the representations of the separator tokens (i.e., "<s>") to be the contextualized column embeddings.

To apply contrastive learning using the multi-column model, we adapt the SimCLR algorithm (Algorithm 1) as follows. First, we create the batches of columns (Line 3) by uniformly sampling a batch of tables from all data lake tables and form each batch of columns B using all columns from the sampled tables. To augment the batch B, instead of transforming each column independently, we apply *table-level* augmentation operators such as row sampling and column sampling (Line 5). Note that in the multi-column setting, the augmentation operators produce views of tables with pairs of columns that align with each other.

We summarize the supported augmentation operators in Table 1. While there is a large design space of the operators, we summarize them by the levels (e.g., cell, row, column) of the table to which the operators apply. The cell-level operators are general transformations also used in related tasks such as Entity Matching [31]. The row and column-level operators cover different ways for creating samples of rows/columns. One can also perform more complex transformations by applying multiple operators simultaneously. In our ablation study (see Appendix B.1), we find that the simple column sampling operator (drop_col) provides the best performance.

We then apply the multi-column model on the original and augmented views of tables to obtain the contextualized column embeddings $\vec{Z}_{\text{ori}}$ and $\vec{Z}_{\text{aug}}$ (Line 6) and compute the contrastive loss (Line 7). Note that in the multi-column setting, the positive pairs (for which we maximize the similarity) consist of the aligned pairs of columns generated by the augmentation operators. We minimize the similarity of all other pairs which include (i) pairs of unaligned columns from the same table and (ii) all pairs of columns from two distinct tables. By doing so, the algorithm learns representations that can distinguish columns with the same/different table contexts, thus creating the positive and negative pairs shown in Figure 5. More formally, let P be the set of indices of all aligned pairs of columns in the batch B, we minimize the multi-column contrastive loss shown in Equation 3:

$$\mathcal{L}_{\text{multi-column}} = \frac{1}{2|P|} \sum_{(i,j)\in P} [\ell(i,j) + \ell(j,i)]. \tag{3}$$

3.4 Table Preprocessing

Typical pre-trained LMs like BERT support an input length of at most 512 sub-word tokens, while a column in real-world tables such as those in Open Data may contain thousands or even millions of tokens. To apply the proposed techniques in Section 3.2 and 3.3 on data lake tables, we must preprocess the columns to reduce the input length to fit the token limit of LMs, while preserving their semantics. The procedure is outlined in Algorithm 2, while the full details with design choices (scoring functions, row/column orders, and alignment rules) are in the appendix due to the space limitation.

Algorithm 2 illustrates the steps of table pre-processing. It first assigns an importance score for each cell by first computing the TF-IDF scores of every token in a cell and then averaging the TF-IDF scores of all tokens. Then it ranks the average cell-level scores of rows and then selects the rows to be included in the serialization result. Here we finish this step in a deterministic way: by ranking

Table 1: Data augmentation operators at different levels.

Level	Operators	Description
Cell	drop_cell, drop_token, swap_token, repl_token	Dropping a random cell; Dropping/swapping tokens within cells
Row	sample_row, shuffle_row	Sampling x% (e.g., 50) of rows; Shuffling the row order
Col	drop_col, drop_num_col, shuffle_col	Dropping$X$ (numeric) columns; Shuffling column order

Algorithm 2: Table Preprocessing

Input: A table T; A token scoring function such as TF-IDF TF-IDF$(\cdot)$; The max #tokens m.
Variables :Preprocessing mode $\in$ {"row", "cell", "token"}
Output: The table T' with selected rows, cells, or tokens

foreach *cell* $c \in T$ **do**
  /* Sum over token scores */
  cell_score(c) $\leftarrow \sum_{\text{token } t \in c}$ TF-IDF(c);
foreach *row* $r \in T$ **do**
  /* Sum over cell scores */
  row_score(c) $\leftarrow \sum_{\text{cell } c \in r}$ cell_score(c);
if mode = *"row"* **then**
  return Top-n rows with highest row_score up to length m;
if mode = *"cell"* **then**
  return Top-n cells with highest cell_score for each column up to length $m/|T|$; // $|T|$: number of columns
if mode = *"token"* **then**
  return Top-n tokens with highest TF-IDF for each column up to length $m/|T|$; // $|T|$: number of columns

in the descending order of the importance score, until we reach the token budget for each column.

4 ONLINE QUERY PROCESSING

In this section, we introduce how to find unionable tables based on contextualized column embeddings. We first introduce the table unionability scores and the overall workflow of online query processing in Section 4.1. Then we discuss the design choices for reducing the number of candidates using vector indices and deducing bounds for more efficient verification in Sections 4.2 and 4.3, respectively. Note that the online processing techniques explored here are not limited to any specific column encoders, they are also applicable to other dense-vector column representation methods [22, 56].

4.1 Table-level Matching Score

After training a column encoder $\mathcal{M}$ using techniques from Section 3, we can then obtain the embedding vectors for all columns in data lake tables via model inference. The column unionability score between two columns s and t can be calculated by using cosine similarity as $\mathcal{F}$ between those embedding vectors. Next, we define the function $\mathcal{A}$ for aggregating the column unionability scores to compute the table unionability. Motivated by the idea of c-alignment [42] that aims to find a maximum set of one-to-one alignment between columns in two tables, we propose modeling table unionability as a *weighted bipartite graph matching* problem. More formally, given two tables S and T with m and n columns respectively, we construct a bipartite graph $G = \langle S, T, E \rangle$ where the nodes S and T are the two sets of columns. The edges in E denote the column unionability score between each pair of columns. Then table unionability score $U(S, T)$ can be calculated by finding the maximum bipartite matching of graph G. In order to remove the noise caused by dissimilar pairs of columns, we follow the de-noising strategy from fuzzy string matching [51] by introducing a hyper-parameter τ as the similarity lower bound: given two columns $s \in S$ and $t \in T$, there is an edge $\langle s, t \rangle \in E$ iff $\mathcal{F}(s, t) \geq \tau$.

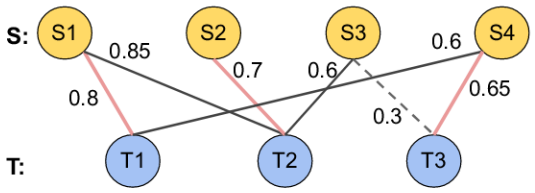


Figure 6: Example of table unionability score via maximum bipartite matching. Solid (red) lines denote the edges belonging to the maximum matching.

Algorithm 3: Online Query Processing

Input: S: the query table; $\mathcal{T}$: the set of data lake tables;
Variables :k: the number of desired results;
  τ: threshold of column unionable score;
Output: $\mathcal{H}$: The top-k unionable tables

Initialize $\mathcal{H}$ and C as $\emptyset$;
for *all columns* $s \in S$ **do**
  $C = C \cup$ findCandidates$(s, \tau, \mathcal{T})$;
for *all tables* $T \in C$ **do**
  if $|\mathcal{H}| < k$ **then**
    Compute Verify(S, T) and add T into $\mathcal{H}$;
  else
    $X \leftarrow$ the score of top element of $\mathcal{H}$;
    if $\boldsymbol{LB}(S, T) > X$ **then**
      Replace the top element of $\mathcal{H}$ with T;
    else if $\boldsymbol{UB}(S, T) \leq X$ **then**
      Discard T;
    else if Verify$(S, T) > X$ **then**
      Replace the top element of $\mathcal{H}$ with T;
return $\mathcal{H}$;

Example 4.1. We show an example of computing the table unionability score in Figure 6. Suppose there are two tables S and T with 4 and 3 columns respectively and the threshold τ for column unionablity score is 0.5. Since the cosine similarity between s_3 and t_3 is 0.3 ($< \tau$), the edge between them is discarded (denoted with a dash line). For the ease of presentation, we omit the remaining dash lines between other nodes in the figure. The maximum bipartite matching of this graph consists of the edges in red (solid lines), which are $\langle s_1, t_1 \rangle$, $\langle s_2, t_2 \rangle$ and $\langle s_4, t_3 \rangle$ with a score of 2.15.

In order to find the tables with top-k highest table unionability scores with the given query table S, a straightforward method is to conduct a linear scan: we use a min-heap with cardinality of k to keep the results of top-k search, then for each table T in the data lake, we directly compute $U(S, T)$; and if the score is higher than the top element of the min-heap, we replace the top element with it and adjust the min-heap accordingly. However, since the time complexity of weighted bipartite matching is $O(n^3 \log n)$, where n is the total number of columns in two tables, it is rather expensive to traverse all tables in a data lake. A scalable solution requires reducing (i) the number of accessed tables and (ii) the computational overhead of verifying each pair of tables.

We propose a filter-and-verification framework to address this issue as illustrated in Algorithm 3. Instead of doing a linear scan over all data lake tables, it employs filter mechanisms to identify a set of candidate tables C for further verification (line: 3). As a result, it can reduce the number of expensive verification operations $\text{Verify}(S, T)$. This is realized by the function findCandidates (Section 4.2). Then for all the candidate tables, we further come up with a pruning mechanism to estimate the lower bound $\mathbf{LB}(S, T)$ and upper bound $\mathbf{UB}(S, T)$ of $U(S, T)$. If the lower bound is larger than the current lowest score, we can directly replace it with the top element without further verification (line: 10). Similarly, if the upper bound is no larger than the current lowest score, we can directly discard it (line: 12). This pruning mechanism is effective since **LB** and **UB** are much more efficient to estimate than the exact verification $\text{Verify}(S, T)$ (Section 4.3).

4.2 Reducing the Number of Candidates

Given a column with its embedding vector, we need to quickly identify tables from the data lake that contain unionable columns, which is realized by the findCandidates function in Algorithm 3. This is a problem of similarity search over high-dimensional vectors. Locality Sensitivity Hashing (LSH) [20] has been used in previous studies of table search to find joinable [61], unionable [42], and related columns [2] in sub-linear time. The basic idea is to use a family of hash functions to map high-dimensional vectors into a number of buckets, where the probability that two vectors are hashed into the same bucket is correlated to the value of a certain similarity metric between them. Following this work, we build a simHash [8] LSH index to estimate the cosine similarity between column embedding vectors. Then for each query column vector s, we can quickly find a set of similar column vectors via an index lookup. Then the candidate set C can be obtained by the union of candidates returned by utilizing each column vector s to query the index. In addition to LSH, we also explore the more recent HNSW [36]. HNSW is a proximity graph with multiple layers where two vertices are linked based on their proximity. It supports fast nearest neighbor search with high recall. We find that HNSW improves the query time by orders of magnitude and thus allows Starmie to support querying over the WDC corpus with 50M tables, which is much larger than the previously supported datasets for table union search.

Since such index structures return approximate instead of exact results, there might be some false negatives in the top-k results. Nevertheless, we find in the experiments that the effectiveness loss caused by the false negatives is within a reasonable range. Meanwhile, the query time can be reduced by one to three orders of magnitude (details in Section 5.3).

4.3 Pruning Mechanism for Verification

Once a candidate table is found, we can reduce the expensive verification cost by quickly computing lower and upper bounds on the unionability score. We first look at how to estimate the upper bound $\mathbf{UB}(S, T)$ between two tables S and T. Recall that in maximum weighted bipartite matching, each column/node in both S and T can be covered by at most 1 edge in the edges of the maximum matching. If we remove this constraint, since nodes can appear in multiple edges, the new optimal matching is easy to compute. Moreover, as it allows edges with greater weights, the total score forms an upper bound of the true table unionability score $U(S, T)$. For the upper bound $\mathbf{UB}(S, T)$, we first sort the edges by their weights in descending order. Then we add edges with the largest weights into the matching in a greedy manner. This process is repeated until all columns in S or T are covered or all edges are used. The time complexity of the above process for calculating $\mathbf{UB}(S, T)$ is $O(|E| \log |E| + n)$, where $|E|$ is the number of edges in G. It is much cheaper to compute than the real table unionability score.

Next, we introduce how to quickly estimate a meaningful lower bound $\mathbf{LB}(S, T)$. For lower bounds, we would like to find a set of edges that do not violate the constraint of bipartite matching, i.e., each column in the two tables is covered by one edge. We can also achieve this goal via a greedy algorithm. Similar to computing the upper bound, we sort the edges by weight in descending order and pick edges with the largest weights. After that, we remove edges that are associated with the columns in the selected edges so as to avoid violations. The termination condition of this process is also the same as that of calculating the upper bound. Since the resulting matching does not necessarily cover all nodes in S or T, the total weight $\mathbf{LB}(S, T)$ is a lower bound of the maximum matching. The time complexity of calculating $\mathbf{LB}(S, T)$ is also $O(|E| \log |E| + n)$.

Example 4.2. We use the example in Figure 6 to illustrate the upper bound computation. Note this example is designed to illustrate the algorithm, not to model the actual distribution of weights in a data lake. We fetch edges in the descending order of weight: $\langle s_1, t_2 \rangle$, $\langle s_1, t_1 \rangle$, $\langle s_2, t_2 \rangle$, and $\langle s_4, t_3 \rangle$. At this point, since all nodes $\{t_1, t_2, t_3\}$ in T are covered, we stop here. The upper bound is $0.85 + 0.8 + 0.7 + 0.65 = 3$, larger than the exact value 2.15.

To compute the lower bound, we start from edge $\langle s_1, t_2 \rangle$ and then remove all edges associated with s_1 and t_2. The remaining edge with maximum weight is $\langle s_4, t_3 \rangle$. After involving this edge into the matching, there is no remaining one and the algorithm stops here. Hence, the lower bound is $0.85 + 0.65 = 1.5$, which is smaller than the exact value 2.15.

5 EXPERIMENTS

We now present an evaluation of Starmie on real-world data lake corpora. First, we show that Starmie achieves new state-of-the-art results on table union search by outperforming the previous best methods by 6.8% in MAP and Recall. Next, our scalability experiments show that Starmie (especially with the HNSW index) achieves significant performance gain (up to 3,000x) while preserving reasonable effectiveness performance. Lastly, we conduct case studies to show that Starmie generalizes to two other dataset discovery applications: column clustering and table discovery for downstream machine learning tasks. We include additional results and discussions in the full technical report [15].

5.1 Experiment Setup

5.1.1 Environment. We implement Starmie in Python using Pytorch and the Hugging Face Transformers library [52]. For contrastive learning, we use RoBERTa [35] as the base language model. We set the hyper-parameters batch size to 64, learning rate to 5e-5,

and max sequence length to 256 across all the experiments. All experiments are run on a server with configurations similar to those of a p4d.24xlarge AWS EC2 machine with 8 A100 GPUs. The server has 2 AMD EPYC 7702 64-Core processors and 1TB RAM.

5.1.2 Datasets. We use five benchmark datasets with statistics detailed in Table 2. Firstly, we evaluate the effectiveness on the first three benchmark datasets, which are subsets of real Open Data. Since accuracy requires manually labeled ground truth, such datasets are not very large. We only use them to conduct the experiments of effectiveness reported in Section 5.2. The SANTOS Small benchmark [24] consists of 550 real data lake tables drawn from 296 Canada, UK, US, and Australian open datasets, and 50 query tables. From Table Union Search [42], there are two available benchmarks: TUS Small and TUS Large. TUS Small benchmark consists of 1,530 data lake tables that are derived from 10 base tables from Canada open data. We also use the larger benchmark, TUS Large, which consists of ~5,000 data lake tables derived from 32 base tables from Canada open data. For these two benchmarks, we randomly select 150 and 100 query tables, respectively, following previous studies [24, 42]. The SANTOS[1] and TUS[2] benchmarks, along with their ground truth of unionable tables, are publicly available.

The last two benchmarks are utilized in efficiency and scalability experiments. The SANTOS Large benchmark contains ~11K raw data lake tables from Canada and UK open data, and 80 query tables. We also run experiments on the WDC web tables corpus [28] which contains 50.8 million relational web tables extracted from the Common Crawl. We randomly select 30 tables as the query.

Table 2: Effectiveness (top) and scalability (bottom) benchmarks.

Benchmark	# Tables	# Cols	Avg # Rows	Size (GB)
SANTOS Small	550	6,322	6,921	0.45
TUS Small	1,530	14,810	4,466	1
TUS Large	5,043	54,923	1,915	1.5
SANTOS Large	11,090	123,477	7,675	11
WDC	50M	250M	14	500

5.1.3 Metrics. For effectiveness, we perform evaluation based on the ground truth from the first three benchmarks. For the TUS benchmarks, the tables are synthetically-partitioned from tables of distinct domains, so the ground truth is created in a generative manner. As for the SANTOS Small benchmark, the tables have been manually-annotated to create a ground truth listing expected unionable tables to each query table. Then we follow previous studies [2, 24, 37, 42] and use the Mean Average Precision at k (MAP@k), Precision at k (P@k) and Recall at k (R@k) to evaluate the effectiveness in returning the top-k results. We compute each score by averaging 5 repeated runs. For efficiency, we measure the average time per query.

[1]https://github.com/northeastern-datalab/santos
[2]https://github.com/RJMillerLab/table-union-search-benchmark

5.1.4 Baselines. For effectiveness experiments, we compare our approach, Starmie, with the following existing approaches.
• D^3L [2] extends Table Union Search [42] for the problem of finding related tables by using table features such as column names, value overlap, and formatting. To compare fairly with Starmie, we omit the column name feature.
• SANTOS [24] proposes an approach that leverages both columns and relationships between columns by using external and self-curated knowledge bases.
• Sherlock [22] is a representation learning method that leverages several column features such as table statistics and word embeddings to learn the embedding vector of a column.
• SATO [56] extends Sherlock by capturing the table context using LDA, and thus performing a form of multi-column prediction.
• SingleCol is our column encoder proposed in Section 3.2 that only uses a single column as the input of the encoder in the training process. This is Starmie without the use of contextual information from Section 3.3.

For efficiency experiments, we aim at exploring the benefits brought by different design choices in the Starmie framework. Thus we compare the performance of 4 methods: basic linear search (Linear), pruning based on estimated bounds (Pruning), search with an LSH index (LSH), and search with an HNSW index (HNSW).

5.1.5 Column encoder settings. We empirically choose the most suitable sampling method (Section 3.4) and augmentation operator (introduced in Section 3.3 and more details in Appendix A). For sampling methods, we find that Starmie achieves the best performance when pre-trained with the cell-level TF-IDF scoring function on the SANTOS Small and TUS Large benchmarks, and with a column-ordered sampling method, alphaHead, that sorts tokens in alphabetical order performs the best, on TUS Small. For augmentation operators, we find that the drop_col operator performs the best on SANTOS Small while drop_cell achieves the best performance on the two TUS benchmarks.

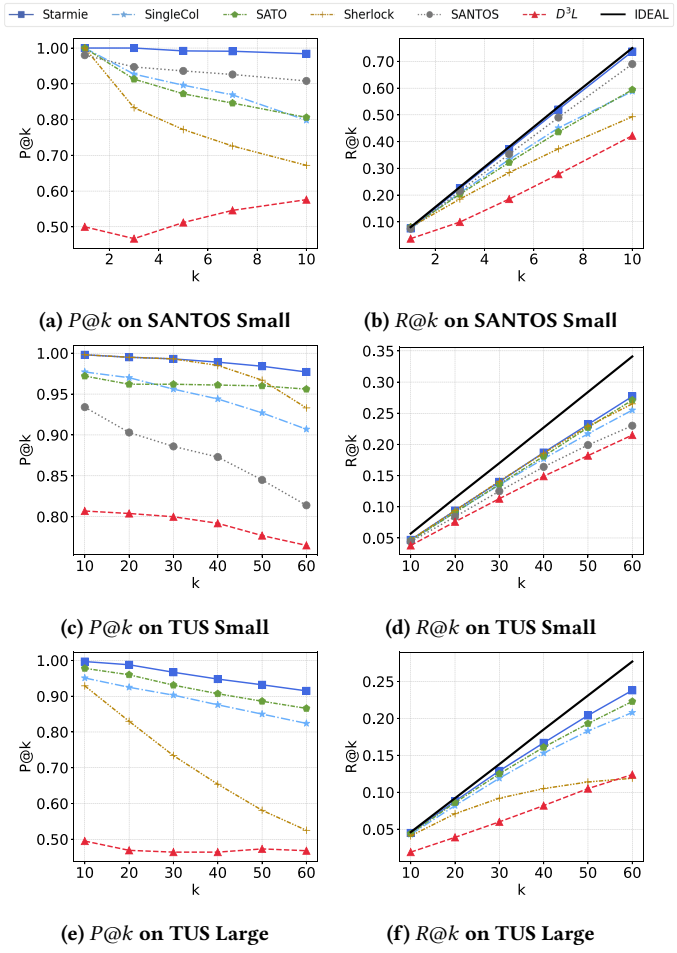
5.2 Results for Effectiveness

Table 3 reports the results of MAP@k and R@k on the three benchmarks for all methods. Note that the results for SANTOS are unavailable for TUS Large because SANTOS, which requires the labeled query table intent columns [24], have not been evaluated on this benchmark due to the absence of annotated intent columns. We run the experiments up to k=10 on SANTOS Small following [24], and up to k=60 on the TUS benchmarks, which is consistent with [42]. Note the recall cannot reach 100% when k is smaller than the number of correct unionable tables from the labeled ground truth as reported in previous studies [24, 42]. Table 3 indicates the maximum recall as IDEAL for each setting.

We can observe that Starmie outperforms the baselines across all three benchmarks. On the SANTOS Small benchmark, Starmie achieves the highest MAP@10 of 99.3% and highest R@10 of 73.7% (which is close to the IDEAL), outperforming SATO, Sherlock, SANTOS, D^3L baselines by large margins of 13%, 27%, 6.8%, and 90% respectively. Also, Starmie outperforms its SingleCol variation by 11%, showing that a multi-column approach is necessary. Similarly, on the TUS Small benchmark, Starmie outperforms the highest-achieving baseline, Sherlock, by 0.7% and SingleCol variation by

Table 3: MAP@k and R@k results on all benchmarks with ground truth, where k=10 for SANTOS Small benchmark and k=60 for the TUS benchmarks. The IDEAL R@k for SANTOS Small is 0.75, IDEAL R@k for TUS Small is 0.341, and IDEAL R@k for TUS Large is 0.277.

Method	SANTOS Small		TUS Small		TUS Large	
	MAP@k	R@k	MAP@k	R@k	MAP@k	R@k
SingleCol	0.891	0.588	0.954	0.255	0.902	0.208
SATO	0.878	0.594	0.966	0.271	0.930	0.223
Sherlock	0.782	0.493	0.984	0.265	0.744	0.119
SANTOS	0.930	0.690	0.885	0.230	-	-
D^3L	0.523	0.422	0.794	0.215	0.484	0.124
Starmie	**0.993**	**0.737**	**0.991**	**0.277**	**0.965**	**0.238**



(a) *P@k* **on SANTOS Small** **(b)** *R@k* **on SANTOS Small**

(c) *P@k* **on TUS Small** **(d)** *R@k* **on TUS Small**

(e) *P@k* **on TUS Large** **(f)** *R@k* **on TUS Large**

Figure 7: *P@k* **and** *R@k* **results on different benchmarks.**

4% in MAP@k. On the TUS Large benchmark, Starmie outperforms SATO by 4% and SingleCol by 7% in MAP@k. Thus, the Starmie approach, by capturing column context and leveraging contrastive learning in pre-training, is very effective in solving the table union search problem.

Figure 7 shows the P@k and R@k of Starmie and the baselines as k increases on all benchmarks. Throughout all values of k, Starmie outperforms all baselines for both P@k and R@k. In Figures 7(b), (d), and (f), Starmie is closest to IDEAL, with R@10 only 1.8% below IDEAL on SANTOS Small, R@60 18.8% below IDEAL on TUS Small, and R@60 14.1% below IDEAL on TUS Large.

To better understand the influence of datasets on the performance of Starmie, we conducted an in-depth analysis to look at its performance for different settings of arity, cardinality, and percentage of numerical columns in query tables. We evenly split the query tables into five groups for each setting. We compare Starmie with alternative representation methods SATO, Sherlock, and SingleCol that also encode columns into high-dimensional vectors. As shown in Figure 8(a)/(c), Starmie consistently outperforms the baselines as the number of columns varies and as the percentage of numeric columns varies. As the number of rows increases (Figure 8(b)), the results of Starmie remain consistently high while the performances of SATO, Sherlock, and SingleCol generally decrease. We believe this is due to our efforts of table preprocessing techniques (Section 3.4). Meanwhile, the performance of SingleCol is much worse than Starmie under all settings, illustrating the importance of contextual information in training the column encoders. The methods have similar trends on TUS Small and TUS Large (Appendix C).

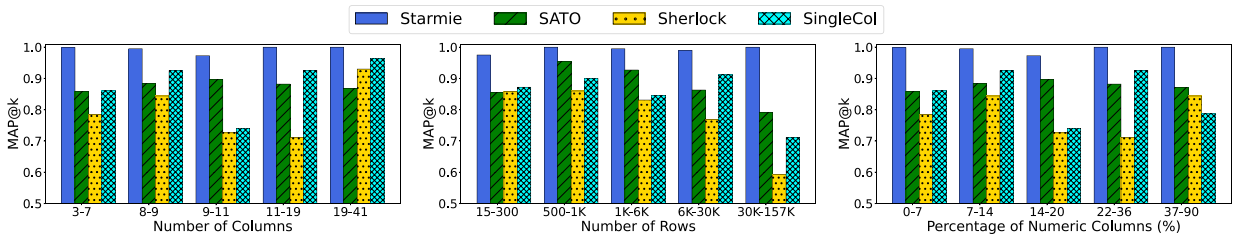
5.3 Scalability

Table 4: Effectiveness of different design choices. The first four methods are for Starmie**.**

Method	MAP@10	P@10	R@10	Query Time (s)
Linear	0.993	0.984	0.737	96
Pruning	0.993	0.984	0.737	61
LSH Index	0.932	0.780	0.580	12
HNSW Index	0.945	0.810	0.606	4
SATO	0.878	0.806	0.594	252
Sherlock	0.782	0.672	0.493	264
SingleCol	0.891	0.798	0.588	108

Impacts on effectiveness. Since some design choices might result in effectiveness loss, we report their results of three evaluation metrics on the SANTOS Small benchmark. As shown in Table 4, we compare Starmie with a basic linear scan with three other design choices (above the horizontal line), as well as baselines SATO, Sherlock, and SingleCol (full experiment results are shown in Appendix C). The main takeaway is that HSNW preserves the effectiveness as much if not better than the LSH index that is widely used in previous studies, while having tremendous speed improvement. This suggests HSNW is a very promising direction for providing real-time search over massive data lakes.

Preprocessing time. Since Starmie requires model pre-training and model inference, in addition to possibly indexing, we provide some insights of such overhead by comparing its preprocessing time with existing systems D^3L and SANTOS that are not based on pre-trained LMs. The preprocessing time of Starmie consists of the following parts: pre-training taking 3.1 hours, model inference taking 4.4 min, and indexing taking 10-30 sec. Meanwhile, D^3L takes 7.6 hours to create four indexes for each column feature and



(a) *MAP@k* **of different # Cols** **(b)** *MAP@k* **of different # Rows** **(c)** *MAP@k* **of different % Num. Cols**

Figure 8: In-depth analysis of Starmie, SATO, Sherlock, **and** SingleCol **as we vary the number of columns, number of rows, and percentage of numerical columns on the SANTOS Small benchmark.**

SANTOS takes 17 hours to create indexes using a knowledge base and the data lake. Thus, pre-training a language model in Starmie does not incur too much overhead compared to existing systems.

Time efficiency. We have observed that the employed design choices can speed up the online query time while sufficiently preserving the effectiveness scores. Next we evaluate the scalability of different design choices. In Figure 9(a), we first evaluate the four variations of Starmie on the SANTOS Large benchmark, as we increase the number of returned unionable tables k from 10 to 60. We then evaluate their query times as the data lake size grows to its full size of ~11K tables / ~120K columns. We also experiment on the WDC benchmark, specifically when the data lake grows to 1M tables / 5M columns (Figure 9(b)) to show the trend of each method , and when the data lake grows to 50M tables / 250M columns (Figure 9(c)). For each method, if a data point's query time does not finish within 24 hours, then we consider it as timeout and omit the result from the corresponding figures.

Throughout all these experiments, we see that the design choice with the HNSW index leads to the best performance. On the SANTOS Large benchmark in Figure 9(a), the k-scalability experiment shows that Pruning is 2X faster than Linear, while LSH index is 20X faster than Linear. Meanwhile, HNSW index, which leads to an average query time of around 300 ms, is 220X faster than Linear and 11X faster than the popular LSH index. As the data lake grows to its full size, there is a steady increase in query time of Linear and Pruning; while that of LSH index and HNSW index remain stable, with the query time of HNSW index remaining around 400 ms. On the WDC benchmark in Figure 9(b), there is a similar trend as the data lake grows to 1M tables. On the full WDC benchmark in Figure 9(c), Linear and Pruning time out after 1M tables, while LSH index times out after having an average query time of 2,520 sec on 10M tables. Meanwhile, the query time for HNSW index stays consistent at around 60 ms as the data lake grows to its full size of 50M tables / 250M columns. The reason is that the hierarchical graph-based structure of HNSW allows it to locate to the nearest neighbors much faster than hash-based indexes [36]. Overall, the design choices explored in this paper, especially HNSW index, show a great improvement in the average query time, even when the data lake grows to an immense size of 50M tables. To the best of our knowledge, the largest dataset that are evaluated by existing solutions of table union search is with only 5,000 tables / 1M columns [42], which has 250 times smaller number of columns.

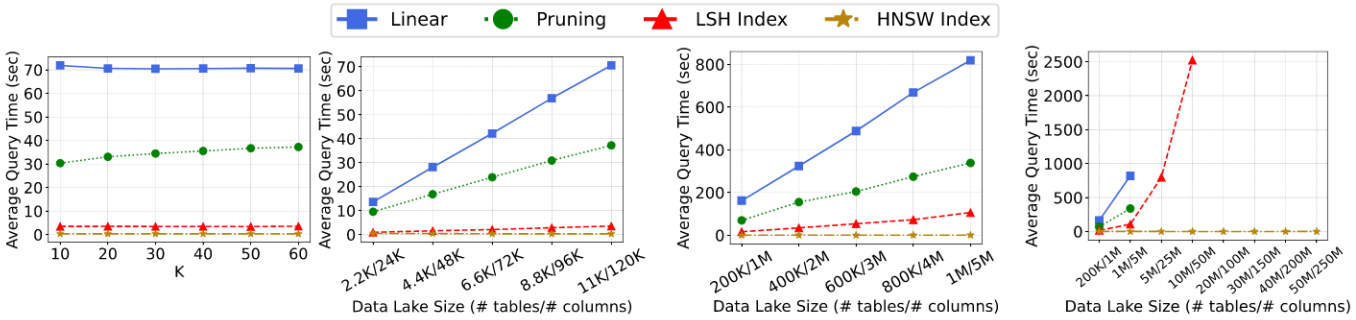
Memory overhead. Lastly, we examine the relative memory overhead of Starmie with different design choices. Specifically, the memory usage of No index (the linear scan and pruning methods from Table 4), LSH index and HNSW index over the data lake of SANTOS Large (11 GB) is 359MB, 733MB and 749 MB, respectively. The results show that Starmie is not only scalable but also memory efficient: its variations take up 3-7% space overhead. The memory saving is mainly due to the condensed vector column representations of Starmie which take up only 3% of the data lake size.

5.4 Data discovery for ML tasks

Next, we conduct a case study to show that Starmie can be applied to another application scenario of dataset discovery, i.e., retrieving relevant tables to improve the performance of downstream ML tasks. For this case study, we consider a subset of 78k WDC tables used in the evaluation of SATO [56], from which we collect all the 4,130 tables of at least 50 rows as the data lake tables. Among these tables, we find that 25 tables of at least 200 rows contain a numeric column called "Rating". These 25 tables contain various types of ratings including those for sportsmen, TV shows, US congress members, etc. From these tables, we construct 25 regression tasks with the goal of training an ML model that predicts "Rating" as the target column. Since the ratings are from different domains, we normalize their values to the range $[0, 1]$. More details about the setting can be found in Appendix D.

For each task, we train a Gradient-Boosted Tree model [9] with all non-target columns as features. We featurize the textual columns using Sentence Transformers [44]. We split each dataset into training and test sets at a ratio of 4:1. Note that the original dataset may not contain informative features. Figure 10 shows such a dataset of US congress members.

To improve the model's performance on these downstream tasks, we leverage Starmie to retrieve relevant tables from the data lake to join with the datasets (i.e., the query tables) to provide additional features. To showcase the effectiveness of Starmie, we use Starmie's contextualized column embeddings to retrieve from the data lake table that contains a column having the highest cosine similarity with a non-target column of the query table. Finally, we augment the query table by performing a left-join with the retrieved table to ensure that the size of the augmented table stays unchanged. We also consider two popular similarity methods for this task, Jaccard



(a) SANTOS Large benchmark (b) WDC (Sample) benchmark (c) WDC (Full) benchmark

Figure 9: Scalability on the SANTOS Large benchmark, a sample of 1M WDC tables, and the full WDC benchmark

and Overlap [14, 60], as baselines by replacing the cosine similarity scores with the corresponding similarity functions.

Table 5 summarizes the results of the 3 evaluated methods. While all 3 methods result in performance improvement (i.e., reduction of MSE), Starmie achieves significantly better overall improvements with a 14.75% MSE reduction, on 15/25 tasks improved, and by an average of 20.64%. By inspecting the retrieved tables, we find that Starmie indeed retrieves qualitatively better candidate tables. As Figure 10 shows, for the same US congress members table, Jaccard similarity retrieves an irrelevant table of dog competitions that also contains a similar "State" column, but the two tables are not semantically relevant. On the other hand, Starmie retrieves a table consisting of the amount of money raised from different interest groups, which is a potentially relevant feature to "Rating". Indeed, by joining with the retrieved table by Starmie, the MSE of the model drops from 0.1598 to 0.1198 (by >25%).

Table 5: Performance gain of data discovery methods on 25 rating prediction tasks from WDC.

	NoJoin	Jaccard	Overlap	Starmie
Avg. MSE	0.0820	0.0753	0.0748	0.0699
Improvement	-	8.23%	8.82%	**14.75%**
#improved	-	13	12	**15**
avg. Improve	-	14.74%	14.05%	**20.64%**

Query Table:

State	Office	District	Name	Party	Rating
AZ	U.S. House	8	Trent Franks	Republican	95.0
TX	U.S. House	3	Sam Johnson	Republican	95.0
OH	U.S. House	4	Jim Jordan	Republican	95.0

Retrieved by Jaccard:

SHOW	State	CITY	DATE	BREED	ENTRY	DOG PTS	BITCH PTS
Tucson Kennel Club	AZ	Tucson	03/28/99	Chinese Cresteds	NaN	NaN	NaN
Fort Bend Kc	TX	Richmond	11/17/96	Chinese Cresteds	NaN	NaN	NaN
Marion Oh Kc	OH	Marion	07/26/98	Chinese Cresteds	12.0	1.0	2.0

Ours:

Name	Party	State	$ From Interest Groups That Supported	$ From Interest Groups That Opposed	Vote
Trent Franks	R	AZ-2	$12,000	$0	Yes
Sam Johnson	R	TX-3	$4,000	$0	Yes
Jim Jordan	R	OH-4	$22,000	$0	Yes

Figure 10: Example tables retrieved by Jaccard vs. Starmie**. By joining the query table with the DL table retrieved by** Starmie**, the MSE for predicting the "Rating" attribute drops from 0.1598 to 0.1195 (vs. 0.1544 when joining with the table retrieved by Jaccard).**

5.5 Case study: Column clustering

Finally we show another application scenario of Starmie in dataset discovery: column clustering. Specifically, we apply Starmie as a column encoder to provide embeddings for clustering all the 119,360 columns from the 78k WDC tables used in the experiments of Sherlock, SATO, and others [22, 47, 56]. These columns are annotated with 78 ground truth semantic types such as population, city, name, etc. The goal of column clustering is to discover clusters of columns that are semantically relevant. The task of semantic type detection has traditionally been solved as a supervised multi-class classification problem which requires significant annotated training data [47]. Starmie provides an *unsupervised* solution. From the contextualized column embeddings, we can construct a similarity graph over all data lake columns as nodes. We can then add undirected edges between all pairs of columns having cosine similarities above a threshold θ (e.g., 0.6). Next the column clusters can be generated via any graph clustering algorithm. We choose the connected component algorithm for efficiency and simplicity.

With Starmie, the clustering algorithm generates 2,297 clusters with an average cluster size of 51.96. We measure the quality of the clusters by the *purity* score, which is the percentage of columns assigned with the same semantic type as the majority ground truth type of each cluster. The discovered clusters are generally of high quality as they achieve a purity score of 51.19 while using baselines such as Sherlock and SATO only achieves 30.5 or 37.36 purity scores when generating a similar number of clusters.

We further inspect the column values within each cluster and find that Starmie discovers clusters of finer-grained semantic types not present in the original 78 types (more results in Appendix E). Table 6 shows 3 such example clusters. The majority types (from the 78 original types) of columns in the 3 clusters are "type", "name", and "artist" respectively. After inspecting the column values, we can interpret the types of the 3 clusters as names of schools, names

Table 6: Column clusters discovered by Starmie. We show the first 3 values from 3 columns of each cluster. The clusters have finer-grained types (e.g., names of schools, grocery stores, song names) than the original ground truth types (e.g., type, name, artist).

Cluster type	1st Column	2nd Column	3rd Column
type	Emerson Elementary School	Choctawhatchee Senior High School	Sumner Academy Of Arts and Science
→	Banneker Elementary School	Fort Walton Beach High School	Wyandotte High School
Names of schools	Silver City Elementary School	Ami Kids Emerald Coast	J C Harmon High School
name	People's Grocery Co-op Exchange	Amazing Grains	Apples Street Market
→	Prairieland Market	BisMan Community Food Cooperative	Bexley Natural Market
Food/grocery stores	The Merc (Community Mercantile)	Bowdon Locker & Grocery	Kent Natural Foods Co-op
artist	I Don't Give A ...	Spoken Intro	New Wave
→	I'm The Kinda	The Court	Up The Cuts
Song names	I U She	Maze	Thrash Unreal

of food/grocery stores, and names of songs. It is difficult to discover such fine-grained types by existing supervised methods.

6 RELATED WORK

6.1 Dataset Discovery

Dataset Discovery has been a hot topic in the data management community. Earlier studies [1, 5, 49] relied on keyword search over web tables to identify essential information. Octopus [4] and InfoGather [54] focused on the problem of schema complement, an important topic in exploring web tables. Aurum [17], S3D [19] and Tableminer+ [38, 58] utilized knowledge bases to identify relationship between datasets. SemProp [18] followed this route by leveraging ontologies and word embeddings, and Leva [59] solved a similar problem with graph neural networks. D^4 [43] addressed the problem of column clustering in data lake tables. Valentine [26] provided resources for evaluating column matching tasks. DomainNet [29] studied the problem of disambiguation in data lakes.

Finding related tables from data lakes is an essential task in dataset discovery. There are two sub-tasks in this application, namely finding joinable tables and table union search [46]. To support finding joinable tables, earlier studies utilized syntactic similarity metrics that are widely used in the applications of string similarity search and join [21, 30, 53]. LSH Ensemble used containment (overlap) [61] as the similarity metric and provided a high-dimensional similarity search based solution. Josie [60] employed overlap over tokens and developed an exact data-optimized solution. PEXESO [14] relied on cosine similarity over word embeddings and proposed indexing techniques to improve performance. The table union search problem has been well explored recently. Ling et al. [34] and Lehmberg et al. [27] illustrated the importance of finding unionable Web tables. Nargesian et al. [42] proposed the first definition and comprehensive solution for the table union search problem in data lakes. Bogatu et al. [2] proposed the D^3L system by dividing columns into different categories. The SANTOS [24] system uses a knowledge base along with binary relationships in the data lake to identify tables that share unionable columns and relationships, and it is the state-of-the-art approach in this field. To the best of our knowledge, our work is the first solution to utilize contrastive learning techniques in table union search.

6.2 Representation Learning for Tables

Recently many efforts use representation learning techniques to address problems related to tabular data. Sherlock [22] and Sato [56] used a supervised feature based approach to learn vector representations for tables and columns. TURL [12] proposed to use a pre-trained language model for web table related tasks and to come up with benchmark datasets for several tasks. And pre-trained language models have been widely applied to different table-related applications, including entity matching [6, 31, 32], column type detection [47, 50], and question answering [23, 55]. Our work follows this line of study and proposes the first solution that employs a pre-trained language model in a fully unsupervised way for the problem of table union search.

7 CONCLUSION AND FUTURE WORK

In this paper, we mainly focused on the problem of table union search, an essential application in dataset discovery from data lakes. We argued that it is crucial to utilize contextual information to determine whether two columns are unionable and proposed Starmie, an end-to-end framework based on contrastive representation learning as the solution. We also developed a multi-column encoder that can capture the contextual information from a table so as to learn contextualized column embeddings. Experimental results on popular benchmark datasets demonstrated that Starmie significantly outperformed existing solutions for table union search.

Our results show the promise of self-supervised contrastive learning in improving the accuracy of table union search, as well as joinable table search, and column clustering – the latter areas we are exploring further. We believe the improved accuracy justifies the use of learning over previous heuristic approaches and the self-supervision will be important to data lakes where labeled training data is expensive to collect and generalize. Our results using the relatively new HNSW index are exciting and important in the development of real-time data lake search solutions.

ACKNOWLEDGMENTS

This work was supported in part by NSF under award numbers IIS-1956096 and IIS-2107248. It was done during Grace's internship at Megagon Labs. We would like to thank Yoshihiko Suhara for his valuable comments on this work.

REFERENCES

[1] Marco D. Adelfio and Hanan Samet. 2013. Schema Extraction for Tabular Data on the Web. *Proc. VLDB Endow.* 6, 6 (2013), 421–432.

[2] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *ICDE*. 709–720.

[3] Dan Brickley, Matthew Burgess, and Natasha F. Noy. 2019. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. In *WWW*. 1365–1375.

[4] Michael J. Cafarella, Alon Y. Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proc. VLDB Endow.* 2, 1 (2009), 1090–1101.

[5] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: exploring the power of tables on the web. *Proc. VLDB Endow.* 1, 1 (2008), 538–549.

[6] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). 1335–1349.

[7] Sonia Castelo, Rémi Rampin, Aécio S. R. Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. 2021. Auctus: A Dataset Search Engine for Data Discovery and Augmentation. *Proc. VLDB Endow.* 14, 12 (2021), 2791–2794.

[8] Moses Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*. 380–388.

[9] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. ACM, 785–794.

[10] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. In *ICML*, Vol. 119. 1597–1607.

[11] Tianji Cong, James Gale, Jason Frantz, H. V. Jagadish, and Çagatay Demiralp. 2023. WarpGate: A Semantic Join Discovery System for Cloud Data Warehouses. In *CIDR*.

[12] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proc. VLDB Endow.* 14, 3 (2020), 307–319.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.

[14] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. In *ICDE*. 456–467.

[15] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. 2022. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. *CoRR* abs/2210.01922 (2022). https://doi.org/10.48550/arXiv.2210.01922

[16] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing Quality to Data Lakes. In *SIGMOD*. 2089–2092.

[17] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *ICDE*. 1001–1012.

[18] Raul Castro Fernandez, Essam Mansour, Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery. In *ICDE*. 989–1000.

[19] Sainyam Galhotra and Udayan Khurana. 2020. Semantic Search over Structured Data. In *CIKM*. 3381–3384.

[20] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. Morgan Kaufmann, 518–529.

[21] Hazar Harmouch, Thorsten Papenbrock, and Felix Naumann. 2021. Relational Header Discovery using Similarity Search in a Table Corpus. In *ICDE*. 444–455.

[22] Madelon Hulsebos, Kevin Zeng Hu, Michiel A. Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César A. Hidalgo. 2019. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *KDD*. 1500–1508.

[23] Hiroshi Iida, Dung Thai, Varun Manjunatha, and Mohit Iyyer. 2021. TABBIE: Pretrained Representations of Tabular Data. In *NAACL-HLT*. 3446–3456.

[24] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2023. SANTOS: Relationship-based Semantic Table Union Search. In *SIGMOD*.

[25] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proc. VLDB Endow.* 16, 4 (2022), 932–945.

[26] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine: Evaluating Matching Techniques for Dataset Discovery. In *ICDE*. 468–479.

[27] Oliver Lehmberg and Christian Bizer. 2017. Stitching Web Tables for Improving Matching Quality. *Proc. VLDB Endow.* 10, 11 (2017), 1502–1513.

[28] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A Large Public Corpus of Web Tables containing Time and Context Metadata. In *WWW (Companion Volume)*. ACM, 75–76.

[29] Aristotelis Leventidis, Laura Di Rocco, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2021. DomainNet: Homograph Detection for Data Lake Disambiguation. In *EDBT*. 13–24.

[30] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *ICDE*. 257–266.

[31] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proc. VLDB Endow.* 14, 1 (2020), 50–60.

[32] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, Jin Wang, Wataru Hirota, and Wang-Chiew Tan. 2021. Deep Entity Matching: Challenges and Opportunities. *ACM J. Data Inf. Qual.* 13, 1 (2021), 1:1–1:17.

[33] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proc. VLDB Endow.* 3, 1 (2010), 1338–1347.

[34] Xiao Ling, Alon Y. Halevy, Fei Wu, and Cong Yu. 2013. Synthesizing Union Tables from the Web. In *IJCAI*. 2677–2683.

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).

[36] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[37] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval*. Cambridge University Press.

[38] Suvodeep Mazumdar and Ziqi Zhang. 2016. Visualizing Semantic Table Annotations with TableMiner+. In *ISWC*, Vol. 1690.

[39] Renée J. Miller. 2018. Open Data Integration. *Proc. VLDB Endow.* 11, 12 (2018), 2130–2139.

[40] Renée J. Miller, Fatemeh Nargesian, Erkang Zhu, Christina Christodoulakis, Ken Q. Pu, and Periklis Andritsos. 2018. Making Open Data Transparent: Data Discovery on Open Data. *IEEE Data Eng. Bull.* 41, 2 (2018), 59–70.

[41] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989.

[42] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (2018), 813–825.

[43] Masayo Ota, Heiko Mueller, Juliana Freire, and Divesh Srivastava. 2020. Data-Driven Domain Discovery for Structured Datasets. *Proc. VLDB Endow.* 13, 7 (2020), 953–965.

[44] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP*. Association for Computational Linguistics, 3980–3990.

[45] Aécio S. R. Santos, Aline Bessa, Christopher Musco, and Juliana Freire. 2022. A Sketch-based Index for Correlated Dataset Search. In *ICDE*. 2928–2941.

[46] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Y. Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding related tables. In *SIGMOD*. 817–828.

[47] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çagatay Demiralp, Chen Chen, and Wang-Chiew Tan. 2022. Annotating Columns with Pre-trained Language Models. In *SIGMOD*. 1493–1503.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008.

[49] Petros Venetis, Alon Y. Halevy, Jayant Madhavan, Marius Pasca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. 2011. Recovering Semantics of Tables on the Web. *Proc. VLDB Endow.* 4, 9 (2011), 528–538.

[50] Daheng Wang, Prashant Shiralkar, Colin Lockard, Binxuan Huang, Xin Luna Dong, and Meng Jiang. 2021. TCN: Table Convolutional Network for Web Table Interpretation. In *WWW*. 4020–4032.

[51] Jin Wang, Chunbin Lin, and Carlo Zaniolo. 2019. MF-Join: Efficient Fuzzy String Similarity Join with Multi-level Filtering. In *ICDE*. 386–397.

[52] Thomas Wolf, Lysandre Debut, Victor Sanh, and et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP*. 38–45.

[53] Jiacheng Wu, Yong Zhang, Jin Wang, Chunbin Lin, Yingjia Fu, and Chunxiao Xing. 2019. Scalable Metric Similarity Join Using MapReduce. In *ICDE*. 1662–1665.

[54] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. InfoGather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*. ACM, 97–108.

[55] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *ACL*. 8413–8426.

[56] Dan Zhang, Yoshihiko Suhara, Jinfeng Li, Madelon Hulsebos, Çagatay Demiralp, and Wang-Chiew Tan. 2020. Sato: Contextual Semantic Type Detection in Tables. *Proc. VLDB Endow.* 13, 11 (2020), 1835–1848.

[57] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *SIGMOD*. 1951–1966.

[58] Ziqi Zhang. 2017. Effective and efficient Semantic Table Interpretation using TableMiner$^+$. *Semantic Web* 8, 6 (2017), 921–957.

[59] Zixuan Zhao and Raul Castro Fernandez. 2022. Leva: Boosting Machine Learning Performance with Relational Embedding Data Augmentation. In *SIGMOD*. 1504–1517.

[60] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864.

[61] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (2016), 1185–1196.