Table Discovery Benchmark

Anonymous Author(s)

ABSTRACT

Recently, table discovery has attracted much attention because

ACM Reference Format:
Anonymous Author(s). 2023. Table Discovery Benchmark. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (SIGMOD' 24).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nowadays, the number of open datasets from multiple sources (*e.g.*, governments and companies) keeps growing, which provides large potential opportunity for intelligent data analysis. However, these datasets are always stored in the data lake that are not well-organized like storing in a database with well-defined schemes and indexes. Hence, it is challenging to efficiently and effectively find user-required data considering the data lake scale and the semantics of user requirements. To exploit the enormous value in the data lake, researchers from both industry and academia have implemented a number of data discovery engines [] that retrieve relevant user-specified tables.

In general, discovering related tables from data lakes can be generally categorized into several sub-tasks, namely keyword-based search, joinable table search and unionable table search. The first category [] issues keywords as queries, based on which relevant tables are retrieved. We focus on the latter two categories that issue a table as the query, which have recently attracted much attention because they are widely-used for augmenting more data instances [] or features [], so as to benefit the downstream applications like machine learning based data analysis.

To be specific, given a query table with a user-specified column, joinable table search finds the target tables that can be joined with the column.

Example 1. *As shown in Figure 1(a), given the query table (T_1) and a specified column* `Corporation`*, we can observe that Table T_2 in the data lake is joinable, i.e., the first attribute of T_2 can be joined with the specified column mainly because i). the column names are matching and ii). there exist a number of semantically overlapping cell values (e.g., Apple and Apple Inc.). The joined table can be taken as feature augmentation to improve model performance when the* `Price` *is treated as the predicted column. In addition, although the first attribute of T_3 has a semantically similar column name and contents with the specified column, it is not joinable because there is no overlapping values.*

For unionable table search, given a query table, it aims to find target tables that are unionable with the query from the data lake.

Example 2. *As shown in Table 1(b), given a query table (T_4) for searching unionable tables, we find that T_5 can be unioned with T_4 because they are all about the information of movies and two pairs of attributes are highly correlated (T_4.*`Name` *v.s. T_5.*`Movie` *and T_4.*`Rating` *v.s. T_5.*`Score`*). However, when it comes to T_6, we can observe that there exist three attributes (T_6.*`City`*, T_6.*`Date` *and T_6.*`Rating`*) that can be respectively aligned with attributes (T_4.*`Venue`*, T_4.*`Date` *and T_4.*`Rating`*) of T_4, T_6 is not unionable with the query T_4 because the two tables are totally not semantically relevant (T_4 is about movies while T_6 is about restaurants).*

From the above examples, we can see that table discovery from the data lake is a non-trivial task we should consider (1) the semantic schema similarity of columns, (2) the (semantic) overlappings between columns as well as the (3) contextual information of all columns in a table. What's more, as the data lake is always large-scale, the efficiency is also a challenging problem. Despite its importance and hardness, existing approaches have still not been evaluated sufficiently due to the lack of a comprehensive benchmark. In retrospect, benchmarks have played a significant role in spawning the boom in different research communities, such as TPC benchmarks [CC][[]] for the database community, ImageNet [CC][[]] for computer vision tasks, and GLUE [CC][[]] for natural language processing tasks.

[nan][

- **Intro**
  - **(1) Sec 1 Add a table or several tables to compare the number of tables/queries supported by your benchmark and others'.**
  - **(2) Easy-to-use APIs: maybe you can give examples for designed unified APIs, saying learned from Hugging Face. They can easily compare different methods.**
  - **(3) Leaderboard.**
- **Sec 3 Benchmark design: align with the revised intro**
- **Sec 4 Statistics of datasets, Sec 5 Statistics of queries – they can be merged if each section is short**
- **Annotation process**
- **API design can be pushed before "Analyzing benchmark results"**

]

Existing Benchmarks. TUS [10] first builds a benchmark for table union search, which contains around 5,000 tables (from OpenData) in the data lake, among which 1,000 queries are selected as query tables with ground truth. Santos [8] improves the TUS benchmark by additionally labeling column-to-column relations to consider more semantics, but it is just for table union search with around 10,100 data lake tables and 80 query tables. Josie [13] is a join search benchmark that uses two data lakes. One is the same with TUS, and the other is sampled from the Web Table, with 1,000 query columns respectively. Valentine [9] evaluates multiple schema matching techniques to solve table discovery tasks, using several hundreds of tables to match pairs of tables. A recent Arxiv work [**?**] focuses on evaluating table pretraining methods for table discovery tasks, with the goal of improve downstream machine learning tasks.

Challenges. Overall, benchmarking table discovery in data lake faces several major challenges.

(1) [*Data/Query Coverage*] The effectiveness and efficiency of table discovery largely rely on the characteristics of datasets, like the data lake scale, average column/instance number per table etc. Also,

Table 1: Benchmark Comparison.

Benchmarks	Type	#-Query tables	#-Lake tables	#-Total columns	Size (GB)	API
TUS Small [10]	Union	200	1,530	14,810	1	✗
TUS Large [10]	Union	1,000	5,043	54,923	1.5	✗
Santos Small [8]	Union	50	550	6,322	0.45	✗
Santos Large [8]	Union	80	11,090	123,477	11	✗
Josie [13]	Join	XX	XX	XX	XX	✗
Valentine [9]	XX	XX	XX	XX	XX	✗
LakeBench [?]	Union	XX	XX	XX	XX	✗
OpenData Small	Join & Union	XX	XX	XX	101	✓
OpenData Large	Join & Union	XX	XX	XX	1111.4	✓
WebTable Small	Join & Union	XX	XX	XX	13	✓
WebTable Large	Join & Union	XX	XX	XX	77	✓

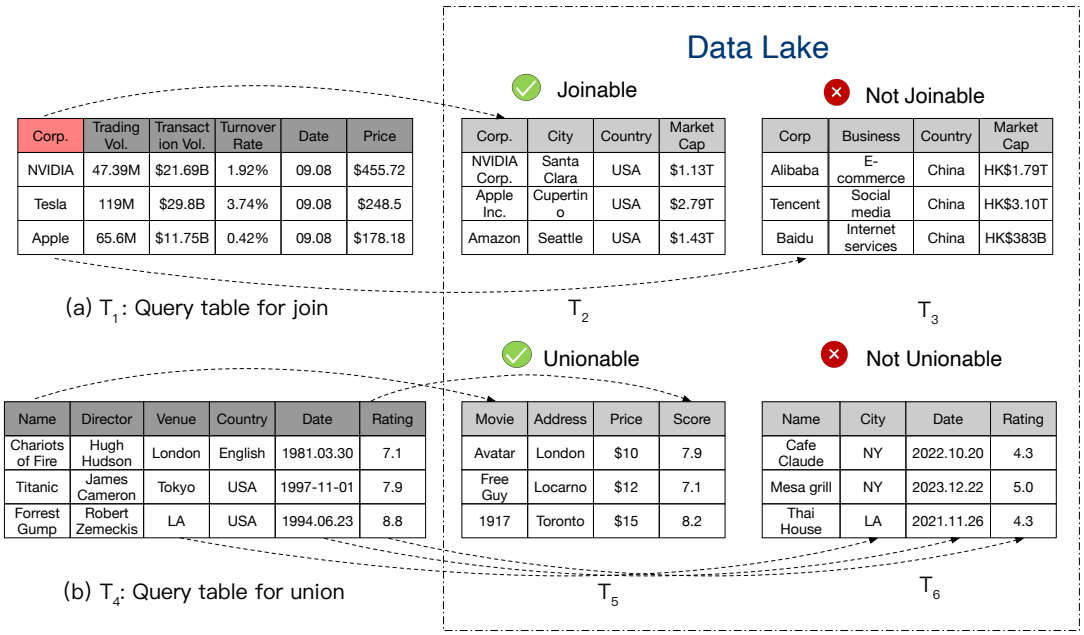

Figure 1: Table Discovery in Data Lake.

different query characteristics also lead to different performance, such as query column/table size, query column type, result size etc.
(2) [*Ground Truth Labeling*] Most existing benchmarks have limited number of queries, mainly because it is rather expensive to label the ground truth (tables that can join/union with query in data lake) for each query table. Hence, it is non-trivial to create sufficient queries with ground truth to evaluate different table search approaches.
(3) [*Solution Coverage*] Recently, several unionable/joinable table search methods have been proposed, but there lacks of a thorough comparison over these methods.
(4) [*User-friendly Interface for Benchmark Usage*] Existing benchmarks just provide datasets, queries and ground truth, which is not very user-friendly if a user wants to run different algorithms over these benchmarks.

To address the above challenges, in this paper, we create a comprehensive benchmark TD-Bench for table search in data lake.

First, we collect 3 TB tables from different sources, including OpenData [] and WebTable [], based on which we evaluate different the table union/join search approaches. These datasets have diverse characteristics. WebTable has a large number of tables (XXX) but each has a small size. In contrast, each table in OpenData is quite large (XXX), but the number is smaller than that of WebTable.

Second, we create sufficient and diverse queries using two approaches. One is to split big tables in data lake into small ones, put these small tables into the data lake and they can be naturally joined or unioned. Hence, when these small tables serve as queries, the ground truth is automatically derived. The other one is that we directly use real tables from the data lake as queries, which indicates that we have to spend much efforts labeling ground truth for

them. To better improve the labeling accuracy and human efforts, we design a candidate generation strategy to improve the recall and implement a labeling platform for this task.

Third, we evaluate various approaches [CC][[]], including techniques like schema matching, local sensitive hash, pre-trained language models etc. for joinable/unionable table search on our benchmark datasets, so as to provide meaningful insights for scalability and accuracy of different approaches on different datasets.

Fourth, we implement an easy-to-use API, based on which a user can conveniently (1) profile the benchmark datasets from different perspectives; (2) issue join/union table search through few lines of python code; (3) compare with multiple state-of-the-art discovery algorithms in one line. [CC]**[Figure ?? shows the functionality of our API.]**

Overall, Table 1 shows the comparison of TD-Bench over existing benchmarks. Based on OpenData [] and WebTable [], we build 4 data lakes (corresponding to the gray rows), over which both join and union search are performed. We can observe that TD-Bench achieves a more comprehensive benchmark due to the much larger number of queries with various characteristics and much larger data lake size such that both the effectiveness and efficiency of different table discovery algorithms are sufficiently evaluated.

To summarize, we make the following contributions.

(1) We build a comprehensive benchmark, TD-Bench, for table discovery in data lake, including large-scale datasets, sufficient and various queries, as well as thorough evaluation.

(2) We collect more than 3 TB real tables from multiple sources, indexing them with different strategies for table search with different approaches.

(3) We create various query tables covering different characteristics, and accurately label ground truth for them.

(4) We evaluate and analyze multiple state-of-the-art joinable/unionable table search approaches using these created queries on the benchmark datasets.

(5) We build a use-friendly API to well support table search in data lakes.

2 BACKGROUND

In this section, we first formally define the two table discovery tasks in a data lake, and the overall architecture to solve them.

2.1 Problem Definition

Table Join Search. Suppose that a data lake $\mathcal{T}$ contains a large set of tables $\mathcal{T} = \{T_1, T_2, ..., T_N\}$, where each table $T_i, i \in [1, N]$ has n_i rows (tuples), m_i columns (attributes) and each cell value is denoted by c_{ij}. Given a query table T_q^J, as well as a specific column C_q^J of T_q^J, table join search is to find the target tables that can be joined with T_q^J on C_q^J. Overall, it can be taken as measuring the relevance score between two columns, denoted by $R(C_q^J, C_t)$, where C_t is a column of a target table $T_t \in \mathcal{T}$. The higher the score, the more likely the two columns can be joined (we can say that the two columns or two tables can be joined interchangebly, *i.e.*, $R(C_q^J, C_t)$ can also be represented as $R(C_q^J, T_t)$. Note that different methods have different criteria to compute the score, like the number of overlaps and/or semantic similarity. The formal definition is as follows.

Definition 1 (Top-K Table Join Search). Given $\mathcal{T}$, a query table T_q^J, the specific column C_q^J and a parameter K, Top-K table join search aims to retrieve a subset $\mathcal{T}_q \subset \mathcal{T}$, $|\mathcal{T}_q| = K$ such that $\forall T \in \mathcal{T}_q$ and $\forall T' \in T \setminus \mathcal{T}_q$, $R(C_q^J, T) > R(C_q^J, T')$.

Remark. Given a target table T_t, there may exist multiple columns that can be joined with T_q. In this way, we take the one with the highest score as the target column, *i.e.*, $C_t^J = \arg\max_{C \in T_t} R(C_q^J, C)$. Also, we just focus on the single join rather than multiple joins like [].

Table Union Search. Given a query table T_q^U, table union search aims at finding the most top-K unionable tables from the data lake $\mathcal{T}$. At a high level, table union search still highly relies on the unionbility of columns, *i.e.*, a pair of unionable tables should have multiple pairs of columns that can be unioned. Similar to the table join search, the column unionbility also considers the overlaps and/or semantics between columns. To be specific, given a target table T_t, we can compute the relevance of each pair of columns, *i.e.*, $R(C, C'), C \in T_q^U, C' \in T_t$, and then a table-level relevance score can be computed as $R(T_q^U, T_t)$.

Definition 2 (Top-K Table Union Search). Given $\mathcal{T}$ and a query table T_q^U, Top-K table union search aims to retrieve a subset $\mathcal{T}_q \subset \mathcal{T}$, $|\mathcal{T}_q| = K$ such that $\forall T \in \mathcal{T}_q$ and $\forall T' \in T \setminus \mathcal{T}_q$, $R(T_q^U, T) > R(T_q^U, T')$.

Remark. One may consider that why we return the top-K results rather than just a set of tables that a table discovery algorithm think as the joinable/unionable results. The reason is that in this way, we have to set a cut-off threshold for the relevance score, which is rather hard to generalize to different table discovery algorithms. Therefore, almost all [] existing works focus on retrieving the top-K results. In this case, another natural problem is that how to set an appropriate K. In real applications,

2.2 Overall Framework of Table Discovery

In this subsection, we will introduce the high-level framework of table discovery, which generally consists of the following modules, namely column modeling, index construction, online table query processing. At a high level, the former two modules are offline, *i.e.*, respectively representing each column in the data lake to a vector and indexing theses columns. Then, when a query table comes, the column(s) of the table is (are) first encoded. Afterwards, with the help of the offline built index, top-K tables with high relevance score are retrieved from the data lake. Next, we respectively illustrate each module as shown in Figure 2.

Column Modeling. As discussed in Section 2.1, column representation plays a significant role in join/union search. Therefore, initially, columns of original tables in $\mathcal{T}$ are always represented as fixed-length vectors, shown as the colorful circles in Figure 2. Mostly, these vectors can be hash codes [] (*e.g.*, generated by the minhash function) or embeddings [] (*e.g.*, generated by pre-trained language model), which are capable of supporting efficient retrieval of columns with high overlapping and/or similar semantics. Besides, there exist solutions [] that just leverage original cell values of each

column to search highly overlapping columns rather than using vectors.

Index Construction. Building upon the column representations, different types of approximate nearest neighbor (ANN) search indexes should be utilized to manage large table repositories seamlessly. Typically, if each column is represented as a fixed-length embedding, local sensitive hashing (LSH) or graph-based index (HNSW) can be utilized to index all the embeddings, so as to enhance the search performance. As an option, inverted index [] can also be employed to accelerate the process of finding highly overlapping columns, where each cell value is mapped with the columns that contain the value. In this case, these colorful circles can be regarded as Column IDs rather than vectors.

Online Table Query Processing. For the online process, when a user issues a table T_q^J for the join search, C_q^J is always first represented as a vector. Then based on the index, a large proportion of columns in the data lake that have low relevance scores with the query are pruned and top-K columns (tables) with high relevance scores, *i.e.*, $R(C_q^J, T_t)$, are returned. When a table T_q^U for the union search is issued, different from the join search, every column should be represented as a vector and searched over the index. Suppose that T_q^U has m columns. For each column $C_i \in T_q^U, i \in [1, m]$, similar to the join search, we retrieve a set of columns with relatively high relevance scores with C_i from the data lake. We use S_i to denote the set of tables that contain the above columns. Then, we compute the union of all retrieved tables, *i.e.*, $\cup_{i=1}^{m} C_i$. Next, for each table $T_t \in \cup_{i=1}^{m} C_i$, the relevance score $R(T_q^U, T_t)$ is computed using techniques like maximum bipartite graph matching, considering the column relevance between the two tables. Finally, top-K tables with the highest relevance scores are returned.

3 BENCHMARK DESIGN

In this section, we first introduce the design goals of TD-Bench benchmark for table discovery tasks, and then present how to generate the TD-Bench benchmark consisting of datasets, queries and ground truth.

3.1 Design Goals

We design the table discovery benchmark by the benchmark design criteria proposed by Jim Gray.

Relevance. The benchmark covers a wide range of table discovery characteristics. First, in terms of the data lake, we incorporate 4 lakes with size ranging from 10GB to 1TB, and with column number ranging from [CC] **[1 million]** to 10 million. Second, in terms of the queries, with the help of much human efforts, we create and label more than 10 thousand table queries that cover various query characteristics, including column semantic, column overlapping, column size, etc.

Scalability. The benchmark involves much larger data lakes than existing data lakes for table discovery. The data lake size should be considered from two aspects. One is the normal total storage size that is highly related to the average table size and the number of tables. The other is the number of total columns because almost all table discovery algorithms build index and search over a large number of columns. TD-Bench involves up to 1 TB data lake and 10 million columns, which is sufficient enough to evaluate the scalability.

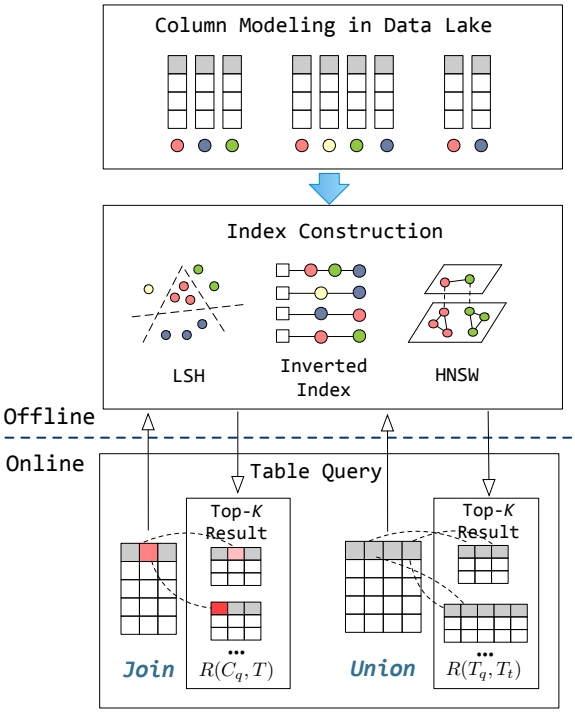

Figure 2: Table Discovery in Data Lake.

Simplicity. The benchmark designs easy-to-use APIs to support table discovery in data lakes. Users can simply leverage few lines of codes to query our benchmark datasets using different algorithms and compare with them. [CC] **[seems limited]**

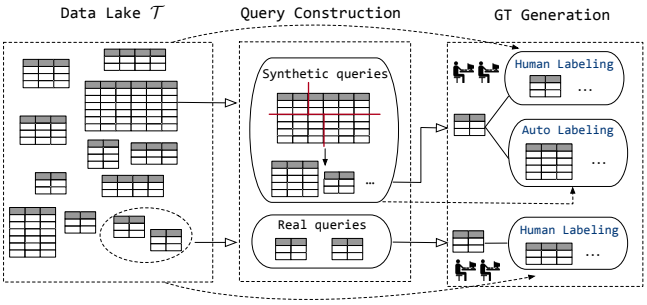

Figure 3: Overview of TD-Bench Construction.

3.2 Benchmark Overview

In this subsection, we overview the pipeline of our benchmark design. Overall, as shown in Figure 3, we first collect the datasets, then construct queries and generate ground truth for these queries. Finally, we implement various table discovery algorithms, analyze their performance from multiple perspectives, and design easy-to-use APIs.

Table 2: Statistics of Data Lakes.

Data Lake	#-Tables	#-Columns	Max/Min/Avg Col. No.	Max/Min/Avg Row No.	Size (GB)
OpenData Small	6374	104,992	514/3/16.5	10,250,220/5/54,454.3	100.03GB
OpenData Large	64,793	1,385,149	6304/3/21.4	66,407,950/5/130,018.3	1.08TB
WebTable Small	2,772,825	17,929,497	25/3/6.5	16,908/5/23.0	13.01GB
WebTable Large	16,684,293	107,891,225	25/3/6.8	16,908/5/23.5	77.05G

Datasets. We build 4 data lakes from OpenData [1] and WebTable [2]. For each data source, we create a small and large data lake respectively. The statistics of them are shown in Table 2. For OpenData [1], we extract 70,546 tables from ^{CC}**[XXX]**, including the union of ^{CC}**[XX]**, ^{CC}**[XX]** and ^{CC}**[XX]**. The characteristic of OpenData is that it contains large tables with numerous rows. For WebTable [2], the original dataset contains 27,783,441 tables, from which we extract 50% into the data lake. The characteristic of WebTable is that it contains a large number of small tables, and thus the total number of columns is very large. As a pre-processing, we remove some very small tables (row number < 5 or column number < 3) and remove the columns with larger then 50% NULL values. Finally, we obtain the OpenData Large and WebTable Large, from which we randomly sample 10% as OpenData Small and WebTable Small, as shown in Table 2.

Query Construction. We use two methods to construct join & union search queries.

Real queries. A direct way is to use real tables as queries, so we extract $Q \subset \mathcal{T}$ as a set of queries. However, in real scenarios, the table search results are likely to be sparse, so in all previous benchmarks, they construct synthetic queries based on big tables, and we conduct this in a similar way as follows.

Synthetic queries. The basic idea is to split large tables into multiple small ones. As shown in Figure 3, generally, we split from two perspectives, namely row-level and column-level splitting. Basically, row-level splitting mainly generates unionable tables, while column-level splitting mostly generates joinable tables. These small tables are then put into the data lake and they can also serve as queries.

Ground Truth Generation. For different query construction methods, the ways of ground truth (GT) generation are different, as shown in Figure 3.

GT generation for real queries. An ideal ground truth generation method is that for each of queries, we ask the humans to label whether all tables in $\mathcal{T}$ can be join/union with the query, which is prohibitively expensive. A natural solution is to leverage a rough search method to retrieve a number of candidate tables that have a relatively high relevance with the query table, *i.e.*, a high recall, and then ask the humans to manually check these candidates.

GT generation for synthetic queries. We can see from Figure 3 that the ground truth of synthetic queries consists of two parts. On the one hand, since each synthetic query is obtained by splitting from a big table, there exist multiple tables splitted from the same table that are joinable/unionable with the query. On the other hand, very likely, there still exist joinable/unionable tables in $\mathcal{T}$ besides the above splitted ones. Therefore, we leverage the above method for real queries to label more ground truth via humans.

Method Evaluation.

^{CC}**[How to category]**
+ ^{CC}**[Join/Union]**
+ ^{CC}**[Schema matching]**
+ Hash-based
+ Inverted index
+ Pre-trained language model (HNSW)

API Design.

4 QUERY & GROUND TRUTH CONSTRUCTION

4.1 Synthetic query generation

In this part, we will discuss how to split large tables and generate the corresponding ground truth.
+ How to choose large tables.
+ How to split.
+ How to generate the ground truth.

4.2 Basic search for candidate generation

In this part, for each synthetic or real query, we design a basic search algorithm to retrieve candidate joinable/unionable tables from the data lake.
+ How to guarantee high recall
+ Details

4.3 Human labeling

In this part, we ask the human experts to label the candidates for each query.
+ Example shown to the experts.
+ Labeling interface.
+ Labeling statistics.

5 APPROACH EVALUATION

In this section, we evaluate existing table discovery approaches in TD-Bench benchmark, and analyze the results. We would like to answer the following questions. (1) What are the concrete solutions of different methods to solve the table discovery problem? (2) How is the scalability of different algorithms, especially on large-scale datasets. (3) Given all queries, what are the precision and recall of different algorithms on the benchmark datasets on average? (4) Given queries with different characteristics, then how about the precision and recall?

5.1 Existing Methods

5.1.1 Table Discovery Approaches. We naturally categorize existing approaches into three categories, namely approaches for join, union, and both of them. Table 3 shows the details of different methods.

Table 3: Table Discovery Methods.

Methods	Task	Index	Embedding	Offline Process		Online Process	
				Time Comp.	Space Comp.	Time Comp.	Space Comp.
Joise [13]	J	Inv. index	✗	$O(\mathcal{K}+\mathcal{R}\log\mathcal{R})$	$O(\mathcal{R})$	$O(\mathcal{L}log\mathcal{L})$	$O(\mathcal{L})$
LSH Ensemble [14]	J	LSH	✗	$O(\mathcal{N})$	$O(\mathcal{N}\times\mathcal{H})$	$O(\mathcal{B})$	$O(\mathcal{B}\times\mathcal{H})$
Pexeso [4]	J	Inv. index	✓	$O(\mathcal{R})$	$O(\mathcal{R})$	$O(\mathcal{A}+\log\mathcal{A}\times\log\mathcal{R})$	$O(\mathcal{A})$
DeepJoin [5]	J	HNSW	✓	$O(\mathcal{N})$	$O(\mathcal{N})$	$O(\log\mathcal{N})$	$O(\mathcal{N})$
TUS [10]	U	LSH	✓	$O(\mathcal{K}+\mathcal{N})$	$O(\mathcal{N}\times\mathcal{H})$	$O(\mathcal{B})$	$O(\mathcal{B}\times\mathcal{H})$
D3L [3]	U	LSH	✓	$O(\mathcal{K}+\mathcal{N})$	$O(\mathcal{N})$	$O(\mathcal{B}\times\mathcal{D})$	$O(\mathcal{B})$
Starmie [6]	U	HNSW	✓	$O(\mathcal{N})$	$O(\mathcal{N})$	$O(\log\mathcal{N})$	$O(\mathcal{N})$
Santos [8]	U	Inv. index	✗	$O(\mathcal{R})$	$O(\mathcal{R}\times\mathcal{N}\overline{\mathcal{T}}^2)$	$O(\mathcal{A}+\mathcal{S})$	$O(\mathcal{A})$
Frt12 [11]	J & U	✗	✗	$O(\mathcal{N})$	$O(\mathcal{N})$	$O(\mid\mathcal{T}\mid\times(\mathcal{B}+\overline{\mathcal{O}})^3)$	$O(\overline{\mathcal{O}}^2)$
InfoGather [12]	J & U	Inv. index	✗	$O(\mathcal{N}^2)$	$O(\mathcal{N}^2)$	$O(\mathcal{B}\times\mathcal{I}\log\mathcal{I})$	$O(\mathcal{I})$
Aurum [7]	J & U	LSH	✓	$O(\mathcal{N})$	$O(\mathcal{N})$	$O(\log\mathcal{N})$	$O(\mathcal{N})$
Pretrain-based Method []	U	HNSW	✓	$O(\mathcal{N})$	$O(\mathcal{N})$	$O(\log\mathcal{N})$	$O(\mathcal{N})$

Table 4: The Meaning of Different Symbols.

Symbol	Meaning
$\mathcal{B}$	Number of columns in the query table T_q
$\mathcal{N}$	The total number of columns in all tables in the data lake $\mathcal{T}$
$\mathcal{K}$	The number of non-distinct cell values in all tables in the data lake $\mathcal{T}$
$\mathcal{R}$	The number of distinct cell values in all tables in the data lake $\mathcal{T}$
$\mid\mathcal{T}\mid$	Number of tables in the data lake $\mathcal{T}$
$\mathcal{X}$	The total number of tuples in all tables in the data lake $\mathcal{T}$
$\mathcal{A}$	The number of non-distinct cell values in query table T_q
$\mathcal{L}$	The length of positing list in Joise
$\mathcal{H}$	The dimension of minhash vector
$\mathcal{P}$	The number of partition in LSH Ensemble
$\mathcal{D}$	The number of neighbors in D3L
$\mathcal{I}$	The number of neighbors in InfoGather
$\overline{\mathcal{O}}$	The average number of tuples in target column

Table Join Search. (1) Joise aims to find joinable tables in data lake via set similarity search. Based on the intuition that two joinable columns have many overlap values, given the query table T_q and query column C_q, Joise considers C_q as a set, and finds the exact top-k columns in the data lake that have high overlap set similarities with C_q. To be specific, overlap set similarity refers to the intersection value between two columns. To achieve this efficiently, Joise builds the inverted index that maps distinct cell values and sets (columns) that contain each corresponding value. Then given a query column, the inverted index can help to identify candidate columns with overlap values in the data lake, a cost model is introduced to quickly eliminate unqualified candidates.
(2) Similar to Joise, LSH Ensemble also considers the set overlap between columns to determine whether two tables can be joined. Different from Joise that computes the exact top-k columns, it estimates the proportion of overlap values (*i.e.*, Jaccard similarity) using an index structure based on MinHash LSH and return the columns with the proportions larger than a threshold, which is built by two stages. First, all columns in the data lake is partitioned disjointly based on their cardinalities. Second, a MinHash LSH index for each partition is constructed and dynamically tuned considering its customized Jaccard similarity threshold. The key contribution is a cost model to conduct an optimal data partitioning for the LSH index, such that the false positive rate could be minimized if an ideal Jaccard similarity filter is used in each partition.
(3) Pexeso considers the semantics of textual columns to capture the column similarity, unlike the above two methods that just consider exact match among cell values of columns. To be specific, it first encodes column values in the data lake into high-dimensional vectors using fastText [], which are then indexed by an inverted index and a hierarchical grid. When a query C_q comes, a block-and-verify strategy is applied to efficiently compute the cosine similarity between vectors.
(4) DeepJoin also considers the column semantics represented by vectors. Different from Pexeso that directly encodes columns into vectors, DeepJoin involves a training process fine-tuning based on DistilBERT [] and MPNet []. This process feeds a pair of columns

into the pre-trained language model, outputs two vectors and computes their cosine similarity, which is used to compute the loss by comparing with the similarity between this pair of columns. Afterwards, columns in the data lake are transformed into vectors that are indexed using HNSW []. When a query C_q comes, it is also transformed into a vector and similar columns that are likely to be joined with it are retrieved though the HNSW index.

Table Union Search. (1) TUS first defines the table union search problem that discovers tables from the data lake that can union with the query table. If two tables have multiple columns (*i.e.*, attributes) with similar domains, they are likely to be unioned. To determine the unionable attributes, three statistical models that respectively consider value overlap , ontology similarity and natural language similarity are incorporated. Then, given any pair of columns, TUS leverages a data-driven method to judiciously choose the best model to describe their unionbility. Also, a TUS benchmark is proposed to validate the algorithm.

(2) D3L also considers the similarity between columns, which are then utilized to union two tables. It considers the column similarity from 5 aspects, namely attribute name, attribute extent, word-embedding of attributes, format representation and domain distribution. Then D3L leverages a weighting strategy that computes a more accurate similarity for columns.

(3) Santos. The above methods mainly consider the semantic similarity of schemes and cell values of columns to determine whether two tables can be joined. Santos further improves the accuracy by incorporating the semantic relationships between pairs of columns. To be specific, it can leverage a KB to annotate a semantic graph for columns within a table. When a query table comes, the graph will be compared with graphs in the data lake such that the column semantics can be well considered. In most scenarios, it is hard for a KB to well cover the values in a real data lake, so Santos proposes a synthesized KB to annotate the column relationships using the knowledge in the data lake.

(4) Starmie focuses on union tables based on large pre-trained language models. To be specific, a lightweight contrastive learning method is applied to train column encoders in an unsupervised way, which captures rich information of contextual semantics. Then, columns in data lake are represented as vectors and the HNSW index is applied to accelerate the query efficiency.

Methods can be used for Join & Union. (1) InfoGather aims at augmenting entities or attributes from a large corpus of HTML tables for a given table, which can be regarded as table union or join search. The basic idea is to first achieve a holistic matching across all tables, and then given a query table, InfoGather finds tables that can be unioned or joined using direct or indirect matching among tables in the data lake.

(2) Aurum can also be utilized to solve both the table join/union search problems. First, the schema of each column is encoded using word embeddings. Then, all columns are organized as a graph that each node corresponds to a column and each edge connects two nodes with similar embeddings. When a query table comes, all columns in the table will also be encoded and hashed to find similar columns in the data lake, and the nearby tables are also retrieved based on the graph. Finally, the join/union scores are computed based on those candidate tables.

(3) Frt12 introduces a framework that captures different relatedness of tables, which are then utilized for union or join. For union, Frt12 considers entity consistency, entity expansion and schema consistency to align two tables. For join, Frt12 considers the coverage of entity sets of two columns.

5.1.2 Evaluation. Metric
Effectiveness
Efficiency

6 API DESIGN

7 RELATED WORK

REFERENCES

[1] [n. d.]. OpenData. https://open.canada.ca/.
[2] [n. d.]. WebTable. https://webdatacommons.org/webtables/.
[3] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 709–720.
[4] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 456–467.
[5] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2022. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *CoRR* abs/2212.07588 (2022).
[6] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. 2023. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. *Proc. VLDB Endow.* 16, 7 (2023), 1726–1739.
[7] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1001–1012.
[8] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2022. SANTOS: Relationship-based Semantic Table Union Search. *CoRR* abs/2209.13589 (2022).
[9] 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 *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 468–479. https://doi.org/10.1109/ICDE51399.2021.00047
[10] 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.
[11] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Y. Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding related tables. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 817–828.
[12] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. InfoGather: entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 97–108.
[13] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 847–864.
[14] 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.