A Study on Efficient Indexing for Table Search in Data Lakes

Ibraheem Taha[◇]
NKUA, Athena RC, & AAU
Athens, Greece
itaha@di.uoa.gr

Matteo Lissandrini
University of Verona
Verona, Italy
matteo.lissandrini@univr.it

Alkis Simitsis
Athena Research Center
Athens, Greece
alkis@athenarc.gr

Yannis Ioannidis
NKUA & Athena RC
Athens, Greece
yannis@di.uoa.gr

***Abstract*—Data lakes store diverse and large volumes of datasets. One of the core challenges in data lakes is dataset discovery, which involves tasks such as finding related tables, domain discovery, and column clustering. In this paper, we focus on a popular approach for finding related tables in public or private data lakes, namely table search. Given the heterogeneity of the tables in a data lake, recent methods adopt table-representation learning and produce dense vector representations for every row, column, or even cell value. This enables advanced indexing techniques, such as HSNW, LSH, and DiskANN, which implement efficient data-structures to speed-up the core operation of approximate k-NN search in such vector spaces. However, while many indexing techniques have been employed so far, their practical value and effectiveness governed by the tradeoff of accuracy vs. performance have not been explored yet. In this paper, we aim at shedding light on this gap. We start with an overview of state-of-the-art techniques for table search in data lakes that are based on vector-search operations. Then, we present an in-depth analysis of the performances of the k-ANN indexes and techniques they adopt. This allows us to map for the first time the space of alternative implementations for these techniques when applied to data lakes, revealing strengths and weaknesses of each option, and further delineating exciting novel research directions.**

***Index Terms*—Data Exploration; Data Discovery; Data Lakes.**

I. Introduction

A data lake is a data management system storing data in their raw forms, e.g., in CSV files or similar formats. The core promise of a data lake is to allow stakeholders to store their data in a centralized repository, while being able to perform data integration on demand [1]. As a result of their deployment, a new challenge arose: that of data discovery [1].

Data discovery tasks in data lakes took different forms. For example, in Google Dataset Search [2], and similar data catalog systems, users can search a directory of many open-datasets via keyword search. The system retrieves relevant datasets by evaluating the match between the keyword query and the metadata description of each dataset. Unfortunately, metadata are often scarce or inconsistent with the data they are supposed to describe. Further, this method is not able to match keywords against the actual contents of the datasets, e.g., find datasets that mention a set of countries from a provided list.

[◇] Ibraheem Taha is co-affiliated with Athena Research Center, National Kapodistrian University of Athens (a degree awarding institute for Athena Research Center), and Aalborg University.

In the field of machine learning, data discovery is essential when a scientist aims at retrieving data to enrich their training datasets. This corresponds to the task of *data augmentation* of existing datasets. The input is a dataset, usually a table (query table), and the output is a new set of datasets from the data lake (result tables). In machine learning terminology, this implies enriching the known samples with new features or augmenting the training set with more samples having the same set of features. Given a query table to be augmented, these two aims can be mapped to two fundamental data-discovery tasks: search for joinable tables [3]–[7], and search for unionable tables [8]–[10]. *Joinable tables* are tables that share most or all the samples with the given query table but describe attributes (columns) that are not present in the query table. Thus, a joinable table is a table for which, performing the equivalent of a relational join between itself and the query table, allows to obtain a table with approximately the same set of rows as the query table but a wider set of attributes. On the other hand, *unionable tables* are tables that describe tuples from the same domain of those in the query table with approximately the same set of attributes. Hence, performing the equivalent of a relational union operation between the unionable table and the query table would allow to obtain a larger set of tuples.

Finding joinable and unionable tables in a datalake requires an in-depth understanding of the semantics of each table and its contents. Table representation learning techniques aim at identifying matches between different row, column, and cell contents across different tables (e.g., [7], [11], [12]). These techniques are based on either supervised or self-supervised neural networks that are trained to produce vector representation for different table components. The result is a model able to map heterogeneous pieces of information to n-dimensional vectors in the same multi-dimensional space, so that that pieces of data with similar meanings are mapped to vectors that are close in that space. These approaches are aimed at uncovering deeper semantic similarities between values based also on information extracted by pre-trained language models (PLMs).

Consequently, search for joinable and unionable tables translates to comparing the corresponding vectors with operations such as cosine similarity. However, this search is computationally expensive as it requires computing the similarity against millions of vectors. Several research efforts have been proposed to speed up the search process (e.g., [7], [8],

TABLE I: Table Search in Data Lakes

Method	Task	Rep. Learning	ANN Index
Octopus [18]	KS	✗	✗
G.D.S. [2]	KS	✗	✗
Aurum [13]	KS	✗	LSH
LSH-Ensemble [3]	Join	✗	LSH
Juneau [4]	Join	✗	✗
JOSIE [5]	Join	✗	✗
MATE [6]	Join	✗	XASH
DeepJoin [7]	Join	✔	HNSW
D^3L [14]	Union, Join	❙	LSH
Starmie [8]	Union, Join	✔	LSH, HNSW
TUS [9]	Union	❙	LSH
SANTOS [10]	Union	✗	✗
TURL [12]	TU	✔	✗
Sherlock [11]	TU	✔	✗
SATO [19]	TU	✔	✗

(✔):= Fulfilled, (❙):= Partial, (✗):= Missing
(KS):= Keyword Search, (TU):= Table Understanding

[10], [13], [14]), including k-Approximate Nearest Neighbor (k-ANN or simply, ANN) techniques that efficiently find the top-k similar vectors with acceptable effectiveness and efficiency [15]–[17]. Nonetheless, only limited attention has been posed in understanding the actual trade-off between computational cost and resource requirements of these techniques.

Hence, in this paper, we present an overview of the state of the art in table search in data lakes (e.g., table union search), including a recent trend that investigates representation learning techniques for table understanding tasks (e.g., infer column type) (Section II). We also identify bottlenecks when employing such techniques (Section III) and investigate how in-memory ANN indexes speed up data discovery (Section IV). Finally, we present an experimental analysis, in terms of effectiveness and efficiency, of popular k-ANN techniques, such as Locality Sensitive Hashing (LSH) [15], Hierarchical Navigable Small World (HNSW) [16] and the disk-based DiskANN [17] (Section V).

II. Table Search in Data Lakes

The related work on table search in data lakes can be classified as follows: (a) keyword search (KS), (b) joinable and unionable table search (Join, Union), and (c) learning-based techniques aiming at enhancing table understanding (TU) taking into account semantics, annotations, domains, and interrelations of table schemas and content. Table I presents an overview of the state of the art classified based on the core task employed for table search, along with whether the methods presented employ learning techniques and a performance booster, such as an ANN index (we discuss this shortly).

A. Keyword Search

The core task in data discovery is to identify relevant datasets for a given task or information need. The problem has been initially studied in the context of relational databases; e.g., Discover [20], Banks [21], DBXplorer [22], Précis [23]–[25], and Meanks [26]. Later on, the focus shifted to web tables and then to data lakes. Example approaches include Octopus [18], which uses keywords to search over web tables and find the correlation between keywords and cell values of tables. As a result, it returns a group of tables sharing similar schema. Another approach, Google Dataset Search (GDS) [2] focuses on the cleansing and standardization of table metadata, coupled with the construction of relationships (based on tables metadata) among these tables within a knowledge graph. This process enhances the efficiency and accuracy of table retrieval operations. Aurum [13] extracts relationships between datasets and represent them as a knowledge graph. In doing so, it cleanses, standardizes, and infers metadata connections across datasets. In these systems, a user can discover related datasets by using keyword search and navigating the knowledge graph connecting datasets matching the input keyword query.

B. Joinable and Unionable Table Search

With the widespread adoption of data lakes, the use cases have evolved from simple keyword search into more sophisticated scenarios, where the user aims at finding datasets that could *augment* a given dataset or datasets. Data augmentations is particularly useful in data science and machine learning use cases, where it aims at enriching a dataset either with additional features (columns) or samples (records) to improve model performance beyond what is achievable with the original dataset. To deal with this challenge, research has focused on two core tasks: joinable and unionable table search.

Joinable search. One approach to data augmentation would be to enrich query tables with relevant tables in the data lake that can be joined with the query tables. The literature refers to these tables as *joinable tables*. The term hails from the relational join operation between two tables that results into a recordset (i.e., another table) containing a larger number of columns and all or a subset of the records of the two tables.

Related work includes LSH-Ensemble [3] that addresses the challenge of finding joinable tables across multiple columns by handling skewed distributions in column values. In data lakes, skewed distributions arise when some domains are substantially larger than others, leading to a biased search. LSH-Ensemble tackles this by partitioning datasets into buckets based on column value distributions. This strategy reduces the influence of skewed distributions, ensuring a more equitable application of LSH across various columns.

Juneau [4] employs a holistic approach, generating data profiles based on data values (similar to LSH-ensemble), and user interactions (extracted by co-occurrence in existing data-science notebooks) to identify joinable tables. Other approaches include handling the problem as set intersection or using a super key. In particular, JOSIE [5] uses sets intersection between sets representing table columns, which allows to determine table joinability and return top-k joinable tables to a given query table. And MATE [6], presents XASH, which aggregates row values into hash values to construct a super key, facilitating the search for joinable tables across multiple columns.

Unionable search. Another approach to data augmentation would be to enrich query tables by adding more data-points (records) sharing the same set of attributes (columns, features).

The literature refers to such tables as *unionable tables* in a connection the relational union operation. However, in this context, a strict formal definition of unionability is missing, as many systems create their own interpretation of what unionable stands for [1].

TUS [9], one of the early works in this area, establishes union search criteria based on three statistical models encompassing set domains (value-based model), semantic domains (ontology-based model), and natural language domains. To maximize column correlations, TUS assigns three separate scores to columns before computing an aggregate union search score. Another approach, SANTOS [10], adopts a relational methodology to define semantic connections between columns in a table. This approach utilizes both an established knowledge base and a synthetic knowledge base, the latter drawing insights directly from the data lake. SANTOS generates semantic graphs for tables, with columns as nodes and their relationships as edges. When aligning a query table with a target column (intent column) for union, the algorithm assesses unionability by gauging the match strength between columns and their relationships, and the degree of query tree alignment in the semantic graph.

Hybrid search. There are also hybrid approaches that consider both joinable and unionable tables. D^3L [14] identifies joinable and unionable tables using five distinct metrics, including: a word embedding strategy, column name similarity, column values overlap, and format representation similarity, the latter assessed using regular expressions for text columns. Additionally, for numeric columns, the fifth metric involves assessing the similarity of domain distributions. D^3L [14] uses word embedding or for each word in the column, capturing the nuanced semantic context of columns through a world-embedding model (WEM) [27]. The table union score between two tables is determined by calculating the average weighted distance across the five key metrics.

C. Learning-based Table Search

The abovementioned approaches, although effective in several applications, are usually based on exact matches, i.e., two table columns are similar if they contain similar sets of values. This assumption overlooks additional information that could lead to presumably more effective search. For example, they do not consider deeper semantics within columns or the context of relationships between these columns (or tables). Research efforts have attempted to fill this gap by exploring learning techniques to table search. These aim at developing a more subtle understanding of the interrelations and contents among columns and tables, which enables a finer-grained semantic analysis and a more accurate identification of joinable and unionable tables. Initial approaches have shown that deep learning, and particularly the integration of embedded pre-trained language models (PLM), could result into quite promising results for enhancing table understanding in data lakes.

The early works in this area have focused on a diverse set of tasks. For example, TURL [12] proposes a framework that takes into consideration the structure of the table, and thus the relationships between values distributed in rows and columns. Further, it employs a pre-trained language model to encode the values in the cells. TURL is designed to accommodate six distinct applications, including entity linking, column annotation, relation extraction, row population, cell filling, and schema augmentation. Another approach, Sherlock [11], describes a single-column semantic identification framework, taking into account both column values and metadata. It develops four distinct features: global statistics, character distribution, value embeddings, and values themselves. These aggregated features serve as the model's input, allowing it to classify each column into one of 78 predefined semantic categories. Finally, SATO [19] extends Sherlock's capabilities to multiple columns, thus using the inferred relationships among adjacent columns in a given table for more structured interpretations.

Table understanding techniques allow exploiting their representation to determine which columns could be considered as matches in the search for unionable and joinable tables. Example approaches in this area are DeepJoin and Starmie. DeepJoin [7] facilitates the discovery of equi-joinable and semantic-joinable tables within data lakes by employing a pre-trained language model. At model training, the input includes task-specific information, e.g., column metadata. The encoding process transforms columns into dense vectors. Starmie [8], instead, is an integrated framework for unionable table search. It utilizes the pretrained language model RoBERTa [28] to determine similarities between table contents. Operating in an unsupervised mode, Starmie applies multi-column contrastive learning to capture contextual semantic relationships within values in tables. Then, columns from the entire data lake are represented in a unified vector space.

Search using learning techniques involves comparing vector representation of the contents of both the query table and all the tables in the data lake. Due to the vast size of data lakes, this results into searching within a huge representation (vector) space. Hence, the related work proposes techniques to reduce the search space, such as approximate nearest neighbor (ANN) indexes. In DeepJoin, the search for joinable tables is boosted with the Hierarchical Navigational Small World (HNSW) index [16], which improves search by two orders of magnitude compared to other joinable table search solutions. Starmie employs k-NN approximate indexes [16], [29] with promising results for table retrieval using HNSW, for both joinable and unionable table search, along with performant and resource efficient query processing compared to LSH [29].

Although applying learning techniques and leveraging ANN indexing seems a quite promising direction toward scalable table search in data lakes, only a handful of approaches have provided in-depth analysis of these techniques. The preliminary results show potential, but also many limitations as well, especially in terms of resource requirements. In the following, we delve deeper into the matter and closely investigate trade-offs and challenges in employing various ANN techniques.

III. System Architecture for Learning-based Table Search

In this section, we describe first a generic architecture for table search in data lakes based on learning techniques. Then, we identify key challenges related to this process.

A. Table Search Architecture

Figure 1 shows the generic architecture of systems employing learning-based table search. In this architecture, the search revolves around the task of identifying columns with similar data and semantics. It comprises an offline phase and an online phase. In the offline phase, we perform three functions: (a) pre-train a column representation model that encodes the columns of data lake tables into dense, high-dimensional vectors, (b) employ the trained model to all data lake tables to obtain column embeddings, and (c) index the embedding vectors into an appropriate data structure for fast k-ANN search (vector index). In the online phase, we start with an input table (i.e., query table) that we want to augment with extra information and perform two functions: (a) apply the trained model to the query table to get query column embeddings, and (b) use the vector indices to identify data lake column embeddings that have a high similarity with the columns in the query table. Finally, top-k tables are retrieved using column alignment methods and a scoring function. While we focus our description on encoding and indexing table columns, the most common approach across state-of-the-art methods, similar concepts could apply without loss of generality to record, cell values, and table metadata, or combinations of these. Next, we elaborate on the two phases.

Offline training. In the training step, each table is decomposed into different data components (here, columns), which are then normalized and fed to a machine learning model aiming at learning a transformation function to represent each data object into the same high-dimensional vectorial space, such that similar data-objects are assigned similar vectors. Initially, columns are converted into a textual form through *serialization* (sequencing values), *tokenization* (adding textual tokens to the sequences to help the model understand value boundaries), and *augmentation* (creating varied semantics-preserving views of same column for contrastive learning). Note that this process enables multi-column learning, i.e., to consider all table columns collectively. Next, the column texts are used to train a fine-tuned deep learning model, which has been initialized with a pre-trained language model (PLM). The trained model encodes the columns into dense, high-dimensional vectors. For example, state-of-the-art techniques usually produce 768-feature vectors, which is the standard output size of the RoBERTa [28] model.

Offline inference and indexing. After training, we obtain the column embeddings for all tables in the data lake via model inference (dotted blue arrows in Figure 1). Therefore, we obtain a vector encoding a learned representation for each distinct column in all tables. These vectors are supposed to include embedded information about the column semantics as well as table context. After the inference step, the produced

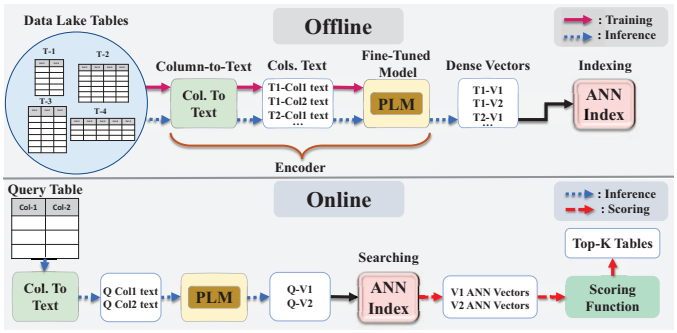


Fig. 1: Learning-based table search in data lakes

vectors are stored (indexed) in an ANN index. The index enables fast search but also offers efficient data representation as it grows linearly with the number of columns across all tables stored in the repository.

Online inference and search. The online phase involves the data retrieval task, i.e., query the vector indices to identify the data lake tables matching the input query table. Initially, the input query table undergoes an inference stage, similarly to the data lake tables in the offline phase (see Figure 1). Thus, the columns of the query table are encoded into vectors. Next, we perform a k nearest neighbor (k-NN) search for vectors representing table columns similar to the given query vector. After retrieving the columns, the final search results are computed via a scoring function combining vector similarities within a column matching algorithm. The choice of the algorithm depends on the search objective, e.g., unionable or joinable table search, and characterizes the compatibility between the query table and potential candidate data lake tables.

B. Challenges in Table Search

Table understanding techniques for table search present two core bottlenecks: training the model required for encoding the data and search across a very large set of vectors. Among these two, identifying the right indexing strategy for the vector search problem is a significant challenge. The main overheads involved relate to execution time and resources required (i.e., memory and disk) for both, index creation and index search.

Indexing high-dimensional vectors is time-consuming and memory-intensive, especially for large data lakes where the number of resulting vectors is significant. To tame memory usage some indexing techniques trade time or accuracy for memory. For example, some techniques load the entire index in main memory and serialize the index only for disk storage. Others employ a disk-based architecture and dynamically load parts of the index in main memory. Moreover, how vectors are stored in the index significantly impacts the index size.

The core search task involves returning the k-Nearest Neighbor (k-NN) vectors of a query vector, using metrics such as Euclidean (ℓ^2) or Cosine distances. This requires linearly scanning each data lake vector, yielding a time complexity of $O(n \cdot m)$, where n and m represent the cardinality of data lake vectors and query vectors, respectively. The complexity increases further when considering the dimensionality of the

vectors. Thus, k-NN search presents a non-trivial overhead as well. To deal with this, several approaches employ approximate nearest neighbor indices to retrieve highly similar vectors. Typical techniques include Locality Sensitive Hashing (LSH) [15] and Hierarchical Navigable Small World (HNSW) [16], which are memory-based solutions. For datasets exceeding memory capacity, disk-based techniques have also been proposed, such as the Disk Approximate Nearest Neighbor (DiskANN) [17] index.

IV. Indexing for Table Search

In this section, we briefly present the key characteristics of the three popular, state-of-the-art methods for ANN vector search: LSH, HNSW, and DiskANN. Here, we focus on the trade-offs involved in selecting these techniques for searching dense vectors, which may represent tables or table components (columns, rows, or cells) in data lakes. Each ANN technique approximates the distance between a query vector and data lake vectors using a specific metric, typically the Cosine distance that is commonly used in dense vector spaces [7]–[9], [14]. This metric measures the cosine of the angle between two vectors, disregarding their magnitudes, making it particularly effective for comparing vectors in multi-dimensional spaces.

A. LSH

Locality-Sensitive Hashing (LSH) [15] is typically used for approximate Jaccard similarity search with boolean vectors, but it can also support vector search for cosine similarity. For doing so, it creates a set of random base vectors (hash functions), where each vector divides the space into sub-spaces. Then, it compares each vector to be indexed via projection to the random base vectors and identifies on which subspace of the base vector it falls. Hence, data lake vectors are transformed into a binary hash space. By projecting the data lake vectors onto the base vectors, LSH creates buckets (hash tables) so that similar vectors would fall into the same bucket. This projection represents the cosine of the angle between the data lake vectors and the base vectors. For querying, the query vectors are handled in the same way. Accuracy and response time depend on the number of base vectors and hash tables; increasing these also increases accuracy at the expense of higher response time. Note that the random projections are independent of data distribution, hence, eliminating the need for update mechanism when indexing more vectors.

The LSH ANN index is memory-based, which speeds up the building and searching phases for datasets fitting the available memory. However, its projection method for finding nearest neighbors is slower compared to the techniques discussed next.

B. HNSW

Hierarchical Navigable Small World (HNSW) [16] builds a graph where each node represent a vector, and edges connect each node to its closest M neighbors. The graph is further organized in layers, where each layer of the graph forms the so-called navigable small world network. Each layer contains nodes at different distances. Thus, vectors are inserted into the index starting from the top layer, containing nodes further away from each other. The data-structure works similarly to a skip-list index. Thus, each vertex to be inserted traverse layers moving to more localized links in lower layers. During insertion, HNSW dynamically updates nearest neighbor sets at each layer and each layer has a probabilistic threshold determining a vector's likelihood of being inserted into that level or kept in the return list during the search phase. The number of potential nearest neighbors is limited by the size of the dynamic list of candidate vertices ef (using the terminology in [16]), which keeps the nearest nodes in a greedy manner. The multilayer structure, combined with a probabilistic approach to graph construction, significantly enhances search efficiency, especially in high-dimensional datasets. Increasing M or the search queue size ef enhances accuracy but decreases speed.

HNSW, an in-memory method, uses a hierarchical structure for faster searches than LSH and needs less memory, enabling more vector indexing. Yet, its memory reliance limits its benefits for datasets larger than available memory.

C. DiskANN

DiskANN [17] is a disk-resident index that overcomes the memory constraints of memory-based algorithms like LSH and HNSW. DiskANN uses a graph-based indexing algorithm (Vamana) to create a graph index stored on disk. Vamana depends on data characteristics, building graphs based on interrelations rather than HNSW's levels. A key parameter in DiskANN is the doubling dimension ρ, defined as the minimum number r where any ball in a space can be covered by at most 2^r smaller balls with half the radius.

DiskANN's storage representation is tailored to disk-based systems and the indexing techniques employed are optimized for SSD access patterns and search latencies, reducing disk I/O. Further, DiskANN employs product quantization for memory-efficient vector compression, allowing a large dataset to be indexed and searched with a small memory footprint. Similar to HNSW, increasing the number of neighbors or the search queue size improves accuracy at the cost of speed.

D. Complexity

To express the computational complexity of the index operators, we consider the following parameters: for a data lake with n vectors of dimension d, the LSH factors are the number of intermediate result vectors v, the count of hash tables $\#tables$ and hash functions per table $\#funcs$. HNSW's key parameters include the maximum graph degree M and the size of the search list ef. DiskANN, shares M and ef with HNSW but adds the graph's diameter α, the doubling dimension ρ, and the ratio of the maximum to minimum distance between vertices, $\Delta = \frac{D_{\max}}{D_{\min}}$.

Time complexity (index creation). At index creation, each vector is individually added to the index, contributing to an $O(n \cdot d)$ complexity for all indexes. LSH's construction involves dot products between the inserted vector and each base vector in hash tables, leading to a complexity of

TABLE II: Benchmarks used in the experiments

Benchmark	Tables	# Tables	# Cols	# Rows
TUS [31]	Data Lake	1405	13200	6.2M
	Query	125	1610	0.5M
SANTOS S [32]	Data Lake	550	6322	3.8M
	Query	50	615	1.1M
SANTOS L [32]	Data Lake	11086	121796	85.5M
	Query	80	1017	1M

$O(n \cdot d \cdot \#funcs \cdot \#tables)$. HNSW is created by connecting graph vertices (vectors) following a logarithmic layer-building normalization, which result in a time complexity of $O(ef \cdot M \cdot d \cdot n \cdot logn)$. DiskANN is created by connecting vertices based on distance and prunes vertices exceeding a certain distance ratio (α). It resembles HNSW in indexing, but it also depends on the dataset $O(ef \cdot M \cdot d \cdot n \cdot log\Delta)$.

Time complexity (search). All three methods have sub-linear search time. LSH's search complexity depends on user settings and the sort of intermediate results for finding top-k similar vectors, hence it is: $O(d \cdot \#funcs \cdot \#tables + v \cdot \log v)$, where the complexity of sorting is $O(v \cdot logv)$. Note that sort is needed as typically the number of retrieved results is larger than k. HNSW's search complexity is $O(logn)$. DiskANN's search relates to vertex proximity within its graph. Its complexity's upper bound is determined by Δ. DiskANN's search phase can locate an $\left(\frac{\alpha+1}{\alpha-1} + \varepsilon\right)$-approximate neighbor within $O\left(\log_\alpha \frac{\Delta}{(\alpha-1)\varepsilon}\right)$ steps [30]. Each step requires no more than $O\left((4\alpha)^\rho \log \Delta\right)$ time, demonstrating that the overall search time is poly-logarithmic w.r.t. Δ under constant ρ. This is because the graph in DiskANN is built relatively to the distance between vertices, consequently, the complexity in the search phase follows this property and relies on Δ.

E. Discussion

Note that HSNW and DiskANN share a limitation regarding updating the index size. Their initialization requires specifying the maximum number of elements they may contain. Should the elements being indexed surpass this predefined limit, the index needs to be rebuild. LSH does not have such a restriction. However, LSH shows worse performance in terms of query response time and memory footprint.

Choosing the right index strategy balances retrieval speed, creation time, memory, scalability, search methods, and and update frequency, tailored to the needs of the application. To shed more light into the capabilities, strengths, and limitations of each approach for effective table search, we present next an evaluation of these indexing techniques.

V. Evaluation of KNN Indexing Methods

In this section, we evaluate the LSH, HNSW and DiskANN indices, utilizing Starmie [8] as a case study for unionable table search. Starmie integrates a deep learning model, built on top of the pre-trained language model RoBERTa [28], to convert table columns into vector representations in a dense embedding space. An Approximate Nearest Neighbor (ANN) index is then employed to facilitate search operations within this vector space, encompassing all column vectors within the data lake's tables. In the following, we aim at answering the following research questions. (1) How does choosing the index affect performance? (2) What is trade-off of choosing different indexes? (3) What is the impact of number of columns (number of dense vectors) on building the index as well as querying it, and thus how well does each index scale?

A. Experimental Setup

Environment. We performed the experiments using Python (3.10.12) on a server with 10 virtual CPUs (Intel Xeon Gold 5318Y) and 48GB DDR4-3200.

Datasets. We use three benchmark datasets derived from a set of Open Data tables, namely TUS [31] and two subsets from the SANTOS [32] dataset, called Small (SANTOS S), and Large (SANTOS L) (see also Table II). Although here we focus on the scalability and performance, it is also fundamental to track the accuracy of results as to examine the trade-off between effectiveness and efficiency. To this end, for TUS and SANTOS S, we have ground truth relevant annotation for evaluating the accuracy of the query answers. The TUS benchmark consists of 1530 tables; we used 125 tables as query tables and 1405 as data lake tables. For SANTOS S we used 50 tables as queries and 550 as data lake tables. Finally, we evaluated the scalability of the approaches with the larger, SANTOS L that comprises more than 11K tables.

We selected these datasets as they are freely available and frequently used for testing data discovery approaches. The interested reader may find more details in our code repo[1].

Implementation. We employed the union table search (including model training), LSH, and HNSW (hnswlib) code and their corresponding parameters leveraging Starmie's codebase [33], and used the diskannpy library of DiskANN [34]. For LSH, we used 8 random vectors and 100 hash tables. For HNSW and DiskANN, we used a max graph degree of 32, with dynamic list sizes of 100 for indexing and 10 for searching, optimizing for graph quality and search speed. DiskANN's indexing on disk was based on the available examples [34].

Metrics. We investigated computational time efficiency and memory disk utilization footprint. Further, given that all indexes perform an approximate search, we consider the table union search as a use case of this study to gauge variation in the quality of the performance. Therefore, for evaluating effects on effectiveness, we use the ground truth annotations of TUS and SANTOS S, and we follow the methodology of Starmie [8] adopting three metrics to evaluate the effect on the quality of the top-k tables retrieved: (a) Mean Average Precision at k (MAP@k), (b) Precision at k (P@k), and (c) Recall at k (R@k).

Pre-Processing. In the pre-processing step, we train our models following the same approach presented in Starmie, then we create the vector representations for the table columns in the data lake. We trained our models on TUS and SANTOS S. For SANTOS L. we used the model trained

[1]Our code repo can be found here: https://github.com/athenarc/table-search

TABLE III: Effectiveness

	TUS [31]			SANTOS S. [32]		
Index	MAP	P@60	R@60	MAP	P@10	R@10
LSH	0.796	0.456	0.147	0.969	0.864	0.645
HNSW	0.795	0.456	0.147	0.970	0.867	0.647
DiskANN	0.812	0.486	0.155	0.974	0.893	0.665
	Ideal R@60:= 0.34			Ideal R@10:= 0.75		

TABLE IV: Data and representation disk usage (in MBs)

Benchmark	Tables	Raw Data	Vectors	Reduction
TUS [31]	Data Lake	933	39	95.9%
	Query	133	5	96.2%
SANTOS S [32]	Data Lake	422	19	95.5%
	Query	122	2	98.4%
SANTOS L [32]	Data Lake	10951	358	96.7%
	Query	137	3	97.8%

on SANTOS S. During indexing, we used Cosine similarity to compute vector similarities in the ANN indexes LSH and HNSW. Currently, the DiskANN code does not support Cosine similarity but only Euclidean distance. However, for ℓ^2 normalized vectors, the squared Euclidean distance is proportional to the cosine distance. Therefore, we normalized all vectors in the data lake, which allow us answer the same k-ANN queries and obtain equivalent results as if we used cosine similarity.

B. Effectiveness

We report the three metrics of effectiveness, MAP@K, P@K, and R@K, on two benchmarks, TUS and SANTOS S in Table III. The results for all k's are very close, therefore, we report the highest k we tested for each benchmark: 60 for TUS and 10 for SANTOS S, which is consistent with the evaluation method used in Starmie. Note that the ideal value for MAP and P@K is 1.0, since both are related to k. R@K depends on the ground truth results: $R@k = \frac{|\hat{V}_Q \cap V_Q|}{|V_Q|}$, where $\hat{V}_Q$, V_Q are retrieved vectors and true vectors, respectively. Therefore, R@K value could be less than 1.0 when the size of true values is more than k.

In terms of effectiveness, the three methods achieve similar results. LSH and HNSW have mostly negligible differences (approximately, 2% to 3%). Interestingly, DiskANN achieves higher precision than LSH and HNSW in both benchmarks. This result is in accordance with the reported experiments, where DiskANN achieves higher recall than HNSW [17].

C. Efficiency

We compare LSH, HNSW, and DiskANN across three dimensions: computational efficiency (measuring running time), memory utilization, and disk space utilization.

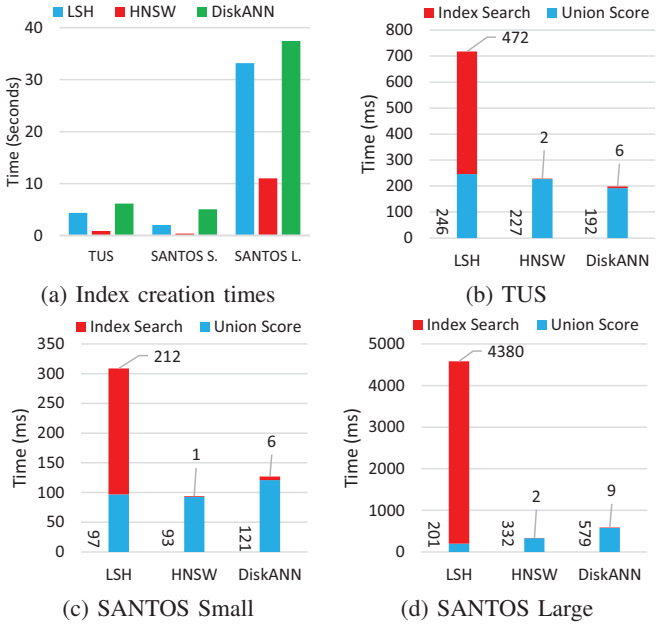
Table representation size. Table IV shows the sizes of the raw data tables and their corresponding vector representations. Here, we report the size of the encoding in the form of 768-dimensional float vectors. Overall, across these dataset we see that the encoded vectors occupy approximately between 3% and 5% of the space occupied by the original raw data. TUS and SANTOS L witness a space reduction ranging between 95.9% and 97.8% of the original table size. For example, for SANTOS S, the data lake tables in their original raw form occupy 422MB, while their vector representation requires only 19MB (i.e., 4.5% of 422MB). This result indicates that encoding tables into dense vectors could significantly reduce the amount of data the index needs to store.

Index creation time. Figure 2(a) shows the time needed for index creation in the three benchmarks. As expected, index creation in DiskANN takes the longest time, as it builds the index on disk. Notably, although LSH is memory-based, it is marginally faster than DiskANN, with HNSW being the fastest for index creation when fitting in memory.

Search and scoring time. Figure 2(b,c,d) shows the efficiency of table union search for the three indexing methods on each benchmark, TUS (b), SANTOS S (c), and SANTOS L (d). The stacked bars in each figure represent the time spent in searching the index for relevant columns (red bar, top) and the time spent in computing the final table score (blue bar, bottom) for each table to which the retrieved columns correspond, to produce the final top-k tables. LSH has slow search time, which increases with the size of the data lake (number of vectors). Hence, LSH does not seem to scale well in production systems that may involve millions of vectors. Overall, HNSW and DiskANN offer fast index retrieval. The small difference in favor of HNSW, which is an efficient, memory-based technique, is due to the more frequent disk visits that DiskANN performs when the data exceeds a predefined threshold. It is worth noting that both methods have quite small search overhead across all datasets tested, indicating promising scaling with increasing data size. For HNSW and DiskANN, scoring is more expensive than search. Although the union scoring function is the same in all indices tested, the scoring times differ. The time to obtain scoring results and top-k unionable tables is affected by the count of columns each index retrieves. Typically, with tables containing multiple columns (denoted as c) and indexes returning t columns per query, the goal is to identify the top-k tables among the $c \cdot t$ related tables. For instance, on average, DiskANN retrieves 27 columns per query for SANTOS L, whereas HNSW and LSH retrieve 17 and 16, respectively.

Memory and disk usage. We used our largest dataset, SANTOS L, to measure the memory and disk requirements for table retrieval in dense vector data lakes. Figure 3a reports the memory and disk usage for the three methods. DiskANN is the most memory efficient method requiring 23MB of memory for vectors of size 358MB (see Table IV). For the same size of vectors, HNSW allocates 406MB and LSH 788MB. This is expected as both HNSW and LSH store the entire index in memory. Therefore, there is an obvious trade-off between memory usage vs execution time, which also determines the choice for the most appropriate index for the task in hand.

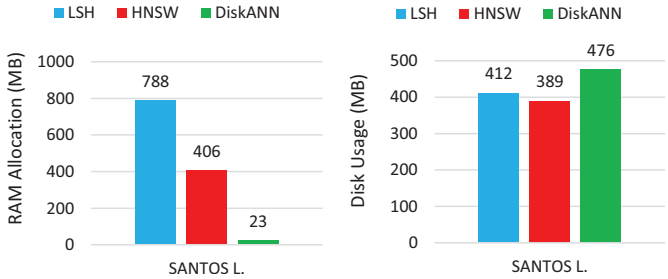
Figure 3b shows disk usage measured as the size of the index (serialized) on disk. DiskANN uses 476MB, adding around 32% overhead to the original vectors size. LSH adds 15%, while HNSW uses the least size 389MB adding 8.7% to the original size. Our experiments indicate linear disk usage in relation to the number of data lake vectors for all indexes.



(a) Index creation times (b) TUS

(c) SANTOS Small (d) SANTOS Large

Fig. 2: (a) Index creation time, and (b-d) Query time



(a) Index memory usage (b) Index size on disk

Fig. 3: Memory allocation and disk usage for SANTOS L

VI. Conclusion

We presented an analysis of data discovery methods, focusing on table representation learning and indices for k-ANN vector similarity search. In particular, we evaluated the HSNW, LSH, and DiskANN indexing techniques, which implement efficient data-structures to speed up the core operation of approximate k-NN search. Our analysis shows that depending on the application needs and available resources, both HNSW and DiskANN could offer competitive solutions that satisfactory balance efficiency and accuracy.

Future research could explore the applicability of these (families of) solutions to billion-scale dense data lakes, and address the challenge of making such indices updatable.

Acknowledgment

This work has received funding from the EU's Horizon 2020 MSCA-ITN programme (grant agreement No 955895).

References

[1] G. Fan, J. Wang, Y. Li, and R. J. Miller, "Table discovery in data lakes: State-of-the-art and future directions," in *SIGMOD*, 2023.
[2] D. Brickley, M. Burgess, and N. Noy, "Google dataset search: Building a search engine for datasets in an open web ecosystem," in *WWW*, 2019.
[3] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller, "Lsh ensemble: Internet-scale domain search," *PVLDB*, 2016.
[4] Y. Zhang and Z. G. Ives, "Finding related tables in data lakes for interactive data science," in *SIGMOD*, 2020.
[5] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller, "Josie: Overlap set similarity search for finding joinable tables in data lakes," in *SIGMOD*, 2019.
[6] M. Esmailoghli, J.-A. Quiané-Ruiz, and Z. Abedjan, "Mate: Multi-attribute table extraction," *PVLDB*, 2022.
[7] Y. Dong, C. Xiao, T. Nozawa, M. Enomoto, and M. Oyamada, "Deepjoin: Joinable table discovery with pre-trained language models," *PVLDB*, 2023.
[8] G. Fan, J. Wang, Y. Li, D. Zhang, and R. J. Miller, "Semantics-aware dataset discovery from data lakes with contextualized column-based representation learning," *PVLDB*, 2023.
[9] F. Nargesian, E. Zhu, K. Q. Pu, and R. J. Miller, "Table union search on open data," *PVLDB*, 2018.
[10] A. Khatiwada, G. Fan, R. Shraga, Z. Chen, W. Gatterbauer, R. J. Miller, and M. Riedewald, "Santos: Relationship-based semantic table union search," *PACMMOD*, 2023.
[11] M. Hulsebos, K. Hu, M. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, c. Demiralp, and C. Hidalgo, "Sherlock: A deep learning approach to semantic data type detection," in *SIGKDD*, 2019.
[12] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu, "Turl: Table understanding through representation learning," *PVLDB*, 2020.
[13] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, "Aurum: A data discovery system," *ICDE*, 2018.
[14] A. Bogatu, A. A. A. Fernandes, N. W. Paton, and N. Konstantinou, "Dataset discovery in data lakes," in *ICDE*, 2020.
[15] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998.
[16] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *TPAMI*, vol. 42, no. 04, 2020.
[17] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," in *NeurIPS*, 2019.
[18] M. J. Cafarella, A. Halevy, and N. Khoussainova, "Data integration for the relational web," *PVLDB*, 2009.
[19] D. Zhang, M. Hulsebos, Y. Suhara, c. Demiralp, J. Li, and W.-C. Tan, "Sato: Contextual semantic type detection in tables," *PVLDB*, 2020.
[20] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *VLDB*. PVLDB, 2002.
[21] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS," in *ICDE*, R. Agrawal and K. R. Dittrich, Eds., 2002.
[22] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A system for keyword-based search over relational databases," in *ICDE*, R. Agrawal and K. R. Dittrich, Eds., 2002.
[23] G. Koutrika, A. Simitsis, and Y. E. Ioannidis, "Précis: The essence of a query answer," in *ICDE*, 2006.
[24] A. Simitsis, G. Koutrika, and Y. E. Ioannidis, "Generalized précis queries for logical database subset creation," in *ICDE*, 2007.
[25] ——, "Précis: from unstructured keywords as queries to structured databases as answers," *VLDB J.*, vol. 17, no. 1, pp. 117–149, 2008.
[26] M. Kargar, A. An, N. Cercone, P. Godfrey, J. Szlichta, and X. Yu, "Meanks: Meaningful keyword search in relational databases with complex schema," in *SIGMOD*, 2014.
[27] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *EACL*, 2017.
[28] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, 2019.
[29] M. Bawa, T. Condie, and P. Ganesan, "Lsh forest: Self-tuning indexes for similarity search," in *WWW*, 2005.
[30] P. Indyk and H. Xu, "Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations," in *NeurIPS*, 2023.
[31] "Tus," 2023, https://github.com/RJMillerLab/table-union-search-benchmark.
[32] "Santos," 2023, https://github.com/northeastern-datalab/santos.
[33] "Starmie," 2023, https://github.com/megagonlabs/starmie.
[34] "Diskann," 2023, https://github.com/Microsoft/DiskANN.