Searching Data Lakes for Nested and Joined Data

Yi Zhang*
AWS AI Labs
imyi@amazon.com

Peter Baile Chen*
MIT
peterbc@mit.edu

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

ABSTRACT

Exploratory data science is driving new platforms that assist data scientists with everyday tasks, such as integration and wrangling, to assemble training datasets. Such tools take scientists' work-in-progress data as a *search object* (table or JSON) and find relevant supplementary data from an organizational *data lake*, which can be unioned or joined with the current data. Existing data lake search tools find *single*, relational tables to match or join with a search object. Yet many data science applications revolve around hierarchical data, which can only be matched by creating views that simultaneously *join and transform several* tables in the data lake. In this paper, we extend the Juneau data lake search system [46] for this broader class of matches *at scale*. Our contribution is a *general* framework for efficiently merging ranked results to match hierarchical data, leveraging novel techniques for indexing and sketching, and incorporating existing single-table search techniques and ranking functions. We experimentally validate our methods' benefits and broad applicability using real data from data science computational notebooks. Our results indicate that, with different ranking functions, our approach can return the optimal set of views up to 4.8x faster and 43% more related compared to heuristics, and increase the data domain coverage by up to 28%. In a case study to show the utility of our results to data science downstream tasks, we reduce regression error by up to 6.6%, and improve classification accuracy by up to 19.5%.

PVLDB Reference Format:
Yi Zhang, Peter Baile Chen, and Zachary G. Ives. Searching Data Lakes for Nested and Joined Data. PVLDB, 17(11): 3346 - 3359, 2024.
doi:10.14778/3681954.3682005

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

1 INTRODUCTION

Data scientists are increasingly using tools that assist with data exploration [19], machine learning model training, and workflow definition tasks. These scientists search over tables and code from public repositories such as GitHub [44], as well as enterprise or public data lakes [34, 46–48] to find supplemental data that can augment their working dataset(s). In the same spirit as Code Llama [39] or GitHub CoPilot [18] for software engineering, such tools assist programmers with data engineering tasks such as discovery and data augmentation [34, 46], as well as data wrangling and cleaning steps [6, 46]. Such tools complement large language models like GPT, Claude, and Gemini. Unlike LLMs that make suggestions based on their parametric knowledge, search tools do not "hallucinate" results. Moreover, they can incorporate policies for domain-specific data types (e.g., gene sequence similarity) or for preferring one source over another (e.g., based on freshness).

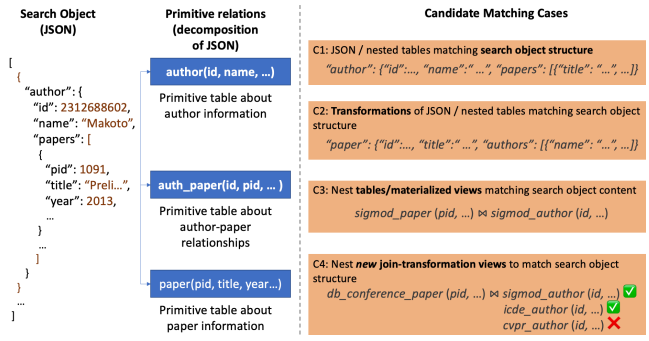

Figure 1: Schema-compliant matches to a JSON search object may come directly or through joins followed by nesting transformations.

The core of such tools is a *search system over data lakes* [13, 32, 34]: Here, the user invokes a query via a *search object* (in existing systems, a table) and the system queries data lake resources for (1) union-compatible tables [13, 33, 46, 48] that can extend training or test datasets with new tuples, (2) joinable tables [16, 46, 47] that integrate data or add machine learning features, or (3) common computational steps found as used in prior workflows [22, 24, 44, 46]. In contrast to web table search [5, 37], the critical challenges of developing such tools lie in understanding the (semantic) compatibility of the schema and detecting the primary-key overlap with the query table [13, 46] . There are two broad classes of existing systems: union-compatible search strives to find tables that *match the search object's schema* while providing novel rows; join-compatible search tends to be non-goal-directed, opportunistically finding tables that appear to join with the existing search table. Recent work [26] shows how to use *full disjunction* to knit the results together to form one output schema.

However, the paradigm of single-table matches to a search object is overly restrictive. Data scientists frequently work with JSON files, nested Dataframes, and composite tabular results (views) that join, pivot, and explode imported tables. Here, to provide the best search results — *the search system must combine subsets of indexed data into new hierarchical or joined data matching the search object structure.* New combinations of indexed tables (i.e., automatically discovered views) better leverage existing data and provide additional insights.

*This work was done while the authors were at the University of Pennsylvania.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682005

This requires that the data lake system can *not only find 1NF tables* but *explore ways of creating query transformations over combinations of indexed tables to match the query.*

Example 1.1. Snapshots of the popular DBLP publications database (dblp.org) have been exported for use in data science tasks, e.g., in notebooks shared on kaggle.com. These snapshots are often subsets of the entire database, taken at different points in time, and their representation may range from flat CSVs[1] to nested JSON[2]. A data lake could also have complementary publication data (e.g., venues not indexed in DBLP) extracted from conference pages or the ACM Digital Library. Now (see Figure 1), if a data scientist takes a JSON snapshot of DBLP and searches for similar data to train a classifier that recognizes CS papers, we would like to return *all* relevant author-publication data (converted to the appropriate structure), including (C1) objects that directly match the nested schema, authors with nested papers; (C2) "pivoted" JSON, in a different structure with semantically compatible content (e.g., papers with nested authors); (C3) nesting of flat tables or views representing the same concepts as the JSON; (C4) nesting of join queries over 1NF relations, matching the JSON schema.

In a similar fashion, a data scientist may take portions of the above data and create an intermediate table for machine learning training, which joins data from different levels of the DBLP hierarchy. At this point, they may want to find matches, which, while "flat," can be matched by joining 1NF data (which, in turn, could have been gathered by decomposing other JSON structures). Unlike prior join-compatible search approaches, this task is *goal-directed* in terms of what joins; unlike prior union-compatible search approaches, this task requires more than single-table matches.

Conventional table search within a data lake introduces two key challenges: identifying semantic and value overlap across thousands to millions of tables, which has generally been addressed using sketch and embedding techniques [16, 45, 47]; and efficiently ranking the results, especially if there are multiple ranking measures [46]. Extending these ideas to heterogeneous and hierarchical *compositions* of tables and JSON/XML results in three additional critical challenges:

(1) Matches to the search object may be *partial*, requiring joins to produce a complete search result. In our example, the JSON can be normalized into three "primitive" relations: *author*, *paper*, *auth_paper*. Individual data objects might contain the information from all these (C1-3 in Figure 1); or might require us to find *other* tables to join, to form a complete match (C4). Efficiently returning the top-k results, when each result has an irregular number of joins, explodes the search space.
(2) Simply querying the primitive relations in top-k order, then joining them, does not produce the top-k answers: intuitively, high-scoring matches to papers may not actually join with high-scoring matches to authors (as per database papers with CVPR authors in Figure 1.) Therefore, the ranking function should measure the compatibility of a composition of primitive relations, and the score should be independent of the order of evaluation.
(3) Automatically and efficiently identifying which relations should join (without introducing large numbers of false positives) is nontrivial – this is aided tremendously by identifying fields that are good keys and also highly indicative of semantic content, e.g., DOIs, SSNs, etc.

[1]https://www.kaggle.com/jakboss/chunk-of-dblp-dataset
[2]https://www.kaggle.com/mathurinache/citation-network-dataset

Searching data lakes for nested and joined data requires novel indexing and pruning strategies. Broadly, as in Figure 1, the problem requires that we decompose the nesting structure of our hierarchical data and find matches to each nesting level (columns in 1NF tables, or in decompositions of other nested tables); then we return a ranked set of queries (views) over these matches as the answers. Note that among the search cases we have discussed, C4 is particularly challenging, as queries need to be created first. To address these problems, we must innovate in how we index content in tables, and perform top-k computations, with the assistance of the indices, across multiple tables while joining them.

A significant body of empirical work has been developed on finding the best features for ranking individual tables considering schema overlap, table overlap, and joinability (see Section 2). In this paper, we develop a general query system that can "plug in" existing individual table ranking strategies (with different degrees of benefits depending on the specific setup), and our focus is on scaling up running times of returning the (new) views over these tables. To facilitate such a query system, we address the following challenges.

Enumerating plans to join and nest data decomposed into 1NF, for a broad class of ranking functions. We *compose* partial matches (1NF primitive tables) from our data lake using nesting and join operations to match search object structure — favoring joins of results that originally came from the same sources. However, our system also considers cross-source joins, which are critical to applications in Linked Open Data and scientific settings with private data referencing entries in public databases.

Incremental maintenance of sketches and profiles. Existing data lake search systems developed *in-memory* sketches and data profiles. To support data compositions in millions of tables, these must be persisted and incrementally maintained.

Automatic index selection. To scale up the exploration of potential joins to produce a match, we develop solutions to determine combinations of fields to index across the data lake, in the form of automatic *data profile* selection.

To sum up, this paper makes the following contributions:

- A new class of data lake search tools over joined and nested data, whose task is to generate unions of nest-join queries over data lake tables to match a search object, with *customizable ranking functions or modules.*
- Identification of key requirements for ranking functions over candidate *views (over multiple tables)*, enabling an efficient *correlated top-k matching algorithm* to rank them.
- *Automatic* selection techniques for *profiling and sketching* millions of tables.
- Experimental validation of the viability and the generality of our methods, over a diverse benchmark suite of real data and notebooks, using state-of-the-art table scoring functions.

2 PRIOR WORK

Semantics of Table Search. The problem of searching a data lake for related tables has generated a significant interest [34].

Value-based table search (keyword search over tables). The earliest table search systems look for matches to words or values *within* tables. They rank table matches by how well the terms represent the tables. This line of work predates the data lake, and ranks tables by TF/IDF-like measures; tables are matched without considering the compatibility between their schemas. Such tools were largely targeted at enterprise networks [3, 15, 16, 20, 21, 43] and/or tables that appear on the web [4, 5, 37].

Unionable table search (table augmentation). Given a search table, these systems use schema alignment techniques to find similar columns. They rank tables based on schema similarity and how many new rows are added [48]. At scale, columns are usually compared using sketch-based techniques including LSH [16, 34, 46], as well as techniques borrowed from schema matching [27] and column or table embeddings [8, 10, 14] with Approximate Nearest Neighbor Search (ANNS). D^3L [2] further combined all these techniques. Beyond those ideas, SANTOS [25] recently proposed to use a knowledge base to discover the semantic relationships between pairs of columns in tables to find unionable tables. There are also efforts considering how data from data science computational notebooks can be leveraged as a data lakes [46], how computation over that data might be reorganized [23], and even how schema constraints might be inferred [41].

Joinable table search. Given a search table, this class of system identifies columns that can be joined to other columns from tables in the data lake, and ranks tables based on the estimated join compatibility [7, 17, 46, 47]. This line of work focuses on tables with matching join columns, but otherwise is not generally directed by a target schema. Similar to union-compatible table search, it is intractable to do pairwise comparisons of columns to find joinable pairs of columns, and therefore LSH and other sketch-based techniques are often used. Recently, DeepJoin [11] proposed an embedding-based retrieval method, which employs a pre-trained language model fine-tuned on equi- or semantic join data for column embedding and uses ANNS, i.e., HSNW [31] for fast retrieval.

The ALITE system [26] explores how to merge and align different (tabular) search results (as a query post-processing), using Full Disjunction as a means of null-padding tables as necessary to achieve union compatibility among the matches. However, since the returned matches are only constructed with tables returned by the query for individual tables, the search space is limited.

Prior efforts match an *individual* table against other "raw" tables. Our work addresses union-compatible *hierarchical* data and investigates techniques for scalably exploring joining and nesting *queries* over primitive tables.

Foundations of our work. As the foundations of our work, we extend an open-source, data lake search system, Juneau [46] [3], which encompasses many of the basic techniques described above. More specifically, the open-source version of Juneau provides three core capabilities, which are integrated into the Jupyter Notebook data science environment.

[3] https://github.com/juneau-project/juneau

Searching: Given a request from the user with a search table S, Juneau searches for other tables that can be combined with S by unioning or joining. It compares S against other tables using a series of "pluggable" metrics, including schema overlap, value overlap, and more. These are combined in a weighted linear fashion, and new metrics can be added, so long as the final function is monotonic with regards to the individual metrics. Juneau works as a general engine for *top-k query processing*, which ranks tables via this similarity score. Juneau's extensibility differentiates it from alternatives in the previous section. Compared to those single-similarity-metric systems [47], Juneau provides significant flexibility and customizability, but at the cost of more expensive search. For our problem which involves multiple matching criteria, this customizability is essential as well — but Juneau's algorithms are not scalable enough.

Indexing and index selection: To scale its search and top-k ranking algorithms to tens of thousands of tables, Juneau relies on additional strategies for indexing. A key capability is *data profile*, a means of identifying and indexing subsets of columns in each table belonging to a particular domain, which often co-occur. Intuitively, a data profile captures the **domain** that identifies semantically meaningful columns (e.g., a country) or combinations of columns (e.g., a country + continent). However, Juneau requires a significant extension to handle millions of tables, as we target.

Loading: As Juneau encounters tables (as they are created or loaded in the Jupyter Notebook environment), it automatically loads these into the data lake, in the process also extending any existing data profiles to consider the data in each new table.

The Juneau system provides a foundation and platform implementation for exploring query-based data lake search — but we also leverage other ideas from the literature.

A key aspect of table matching is the use of sketches for computing column compatibility (both for schema alignment and record linking): LSH ensembles allow us to find overlapping strings [48]; Kolmogorov-Smirnov sketches allow us to compare value distributions for numeric fields [42]. Our platform indexes and stores hierarchical data, including XML and JSON. To enable maximum flexibility during search (where the search object's hierarchy may not exactly match the stored object's hierarchy), we normalize all hierarchical data into 1NF relations, but maintain information about the original structure, as well as the foreign keys required to recompose them into original or pivoted hierarchies. To do this, we adopt ideas developed for storing XML in relations [9, 40]. For XML, this occurs by consulting the XML schema and creating or reusing relations (with appropriate schema elements) at $1:n$ nesting boundaries; then, on-the-fly as the XML is parsed, different components of the XML tree are directly inserted into the appropriate tables. Finally, our work on top-k query processing adapts ideas from Fagin's Threshold Algorithm [12] as well as the J^* algorithm [35] to iteratively explore options.

3 ENUMERATING POSSIBLE QUERIES

Our work explores how to create and rank *join-nest queries* over combinations of tables in a data lake, which can be unioned with a (hierarchical, i.e., non-1NF) search object. Our goal is a general architecture that supports many potential scoring functions (modules) from the literature. In this section, we identify (1) the potential

search space of matching the search object, (2) the necessary *properties that the scoring function must satisfy during join*. We separate our discussion into identifying primitive tables in the data lake that match portions of the schemas in the search object; ranking potential matches by their "value-"add; then merging the results of joining and nesting these tables together (as necessary) to produce a compatible match. For simplicity of exposition, we describe how we handle nested data, but our work can also generalize to "flat" joins (by replacing outerjoins with innerjoins and omitting nesting).

3.1 Data Lake Join-nest Query Generation

When the user searches for a hierarchical data object S, our system explores query expressions, corresponding to the search data in a nested structure, that *augment* S, and finally returns a disjoint union [26, 40] of each query expression's output with S. The detailed definitions of our problem are as follows.

Definition 1 (Possibly-non-1NF Search Data). *Given a (possibly -non-1NF) search data S, we can express it as a search query over 1NF relations denoted as V_q. Denote a nest operation that groups tuples with common values of $\bar{x}$ as $n_{\bar{x}}$, $V_q = n_0(S_{0,1} \bowtie S_{0,2} \bowtie \cdots) \rightarrowtail n_1(S_{1,1} \bowtie S_{1,2} \cdots) \rightarrowtail \cdots n_m(S_{m,1} \bowtie S_{m,2} \bowtie \cdots)$.*

Definition 2 (Data Lake Query Generation Problem.). *Given a (possibly-non-1NF)* search data *S that can be expressed as V_q, and a corpus (data lake) of 1NF* primitive tables *with schema $\Sigma = \{R_1, \ldots, R_n\}$ and data instances $I = \{I_{R_1}, \ldots, I_{R_n}\}$, the data lake combinatorial search problem involves:*

(1) Separate the search data S into separate, 1NF sub-relations $\{S_i\}_{i=1}^m$ by unnesting any nested relations. It means S can be represented as $V_q = n_0(n_1(\cdots(n_m(S_0 \rightarrowtail S_1 \rightarrowtail S_2 \rightarrowtail \cdots \rightarrowtail S_m))))$, where $S_0 = (S_{0,1} \bowtie S_{0,2} \bowtie \cdots)$, $S_1 = (S_{1,1} \bowtie S_{1,2} \bowtie \cdots)$, $\cdots, S_m = (S_{m,1} \bowtie S_{m,2} \bowtie \cdots)$. Each nest operation n_i should take as an argument the attributes of $\bigcup_{j=0}^{i} S_j$.

(2) For each sub-relation S_i, find combinations of primitive table instances from I. For the jth combination that has been found, we denote its table instances as $I_{R_{i,j,0}}, I_{R_{ij,1}}, \cdots$. Along with the associated join keys, we can define a table expression $T_{i,j}$ for each combination, i.e., $T_{i,j} = R_{ij,0} \bowtie R_{ij,1} \bowtie \cdots \approx S_i$. Here, $\approx$ means that $T_{i,j}$ is relevant *to S_i: namely, it has a schema similar to S_i, but I_{R_i} contains a substantial number of rows not present in the instance of S_i.*

(3) Apply an outer union [40] between S and a set of expressions of the form $V_{q,j} = n_{0,j}(n_{1,j}(\cdots(n_{m,j}(T_{0,j} \rightarrowtail T_{1,j} \rightarrowtail \cdots T_{m,j}))))$. Restrict expressions $V_{q,j}$ to those that are relevant to S, namely they have a similar schema to S and yield a substantial number of (nested) rows not present in the instance of S.

Example 3.1. Given the JSON fragment of Figure 1 as our search data S, our system decomposes S into a set of relations: S_0, corresponding to AUTHORS and S_1, corresponding to PAPERS with author IDs (as foreign keys). S can be expressed using the *search query* $V_q = n_{S_0}(S_0 \rightarrowtail S_1)$, where $\rightarrowtail$ represents a left outerjoin between authors and papers-with-author-foreign-keys relations. Adopting the notation of the nested relational algebra, $n_{\bar{x}}$ represents a *nest* operation that groups tuples with common values of $\bar{x}$ and, for each group, associates a nested list of elements created from the remaining columns. In our case, we will nest by all attributes in the schema of S_0, thus nesting S_1 under S_0. Next, referring again to Figure 1, S_0 approximately matches the primitive table in the data lake called *author*; S_1 approximately matches the join expression $paper\_auth \bowtie paper$. We then union $n_{author}(author \rightarrowtail (paper\_auth \bowtie paper)$ to add rows to S. □

From the example, it should be evident that searching for join-nest queries from a data lake can leverage many ideas from standard data lake search for individual tables. We can treat it as a problem of finding matches at each level of a hierarchical search object — then composing these into "appropriate" combinations. However, we additionally need to ensure that we tractably explore combinations of matches that *actually produce join results* and that we develop an effective ranking function (discussed in Section 3.2).

Since some tables in the data lake may be equivalent to joins of others, to restrict the search space to a tractable set of candidate queries, we make two simplifying assumptions in each query expression as we compute its relevance:

- We defer nesting operations to the end, based on the schema — so we only need to enumerate joins and outer joins.
- Note that an outer join expression contains a superset of the corresponding inner join expression. We assume that the cost and cardinality of left outerjoin is closely approximated by the cost and cardinality of an inner join; thus our relevance function treats inner and outer joins as the same.

With these simplifications, we can assume our search query V_q and each of our candidate queries V_j are conjunctive expressions. Over the set of all such expressions, we will develop strategies for exploring all candidates scoring in the top-k results.

Referring back to Definition 2, observe that steps 2 and 3 rely on notions of *schema similarity* and *numbers of rows not present*, to define the utility for each candidate expression. This can be achieved by defining a *scoring function* $rel(s, t)$ as the combination of a schema similarity measure and a "row complementarity" measure between two tables s and t.

3.2 Ranking Candidate Queries

Our goal is a query system that supports a broad array of scoring functions. We assume that the score of each candidate query expression considers at least: (1) how schema-compatible its output is, when compared to the search object; (2) the cardinality of the output of the query expression, which is indicative of how semantically related its base tables are, which also reflects their joinability; (3) the number of new rows that the expression adds to the search object, representing new information.

Suppose our search object S can be separated into sub-relations $\{S_1, \ldots, S_l\}$ that are joined and nested. Our goal is to create join expressions over related tables of each table from $\{S_1, \ldots, S_l\}$, i.e., a *different* series of base tables: $\{R_1, \ldots, R_l\}$, to complement S. Importantly, our scoring function should satisfy two properties.

(1) Algebraically equivalent query expressions should have the same score, so a query optimizer (and precomputed expressions) do not change the score of a match. A join expression should have the same score as a join of the primitive tables.

(2) The scoring function needs to satisfy the monotonicity property of Fagin et al. [12]: namely, that for any two candidate tuples $T(x_1, x_2, x_3, \ldots)$, $T'(x'_1, x'_2, x'_3, \ldots)$, $score(T) \leq score(T')$ if $x_i \leq x'_i$ for every i.

A natural way to define a monotonic scoring function for a conjunctive query is to make it the *weighted sum* of a series of components: by the above criteria, this would include a component establishing how schema-similar each R_i is to each S_i; a component establishing how many new results are in R_i but not in S_i; and a component establishing how effectively each R_i joins with R_{i+1}. Below, we first measure the **joinability** among data lake tables, then introduce the complete scoring function.

3.2.1 ***Joins.*** Some of the data indexed in our data lake come from decomposed JSON or XML — given that our system tracks the original relationships, we could limit our exploration to joins along these relationships. However, important results may also arise by joining *across* data from different sources, e.g., references in Linked Open Data or from local to public scientific databases.

With a large number of tables in the data lake, combinatorial explosion may arise among potential join expressions, resulting in an extremely high number of candidate queries. Inspired by A* search, we seek to iteratively assemble, as separate streams, "promising" subexpressions that start by matching each of sub-relation S_i in the search query V_q. We can combine these subexpressions across the streams until we have a complete candidate expression. Therefore, our metric first scores partial joins, then we will see how they can be combined to measure the joinability among multiple tables.

Traditional top-k enumeration explores how to combine a uniform set of tuples in *streams*. In our setting, we may need to explore in a non-uniform way, since some candidate objects match one primitive table in our search object whereas others may match multiple. We do this by creating a *join subexpression exploration graph* in which nodes are relations, weighted edges represent similarity scores (between a query sub-relation S_i and a primitive table R_i, using some standard measure of union-compatibility as in prior data lakes search systems [46, 48]); or between some table R_i and some other candidate table with which it joins, R_j. (This approach somewhat resembles the J* enumeration of Natsev et al. [35] but is at the relation and not tuple level, and starts by considering table approximate matches.)

Consider a candidate expression V_i, posed over the relations in the data lake Σ, which joins l primitive tables. Each table in the expression, denoted as $R_{j,i}$, may join multiple other tables $R_{k,i}$, each with a different join predicate. Define a helper function $L(R_{j,i}, R_{k,i}) = \{\langle c, c'\rangle | c \in \bar{R}_{j,i}, c' \in \bar{R}_{k,i}\}$, which represents the pairs of columns participating in a single equijoin predicate between $R_{j,i}, R_{k,i}$ from V_i. Building upon this, we introduce a second helper function $P(R_{j,i}) = \{L(R_{j,i}, R_{k,i}) | \langle R_{j,i}, R_{k,i}, L(R_{j,i}, R_{k,i})\rangle \in E_i\}$ which captures the set of all predicates between tables $R_{j,i}, R_{k,i}$. Then for V_i, we define its *view graph* $G(V_i) = \{N_i, E_i, L_i\}$, where:

- Nodes $N_i = \{R_{1,i}, ...R_{l,i}\}$, each of which represents a relation in V_i.
- Edges $E_i = \{\langle R_{j,i}, R_{k,i}, L(R_{j,i}, R_{k,i})\rangle | R_{j,i}, R_{k,i} \in N_i,\ L(R_{j,i}, R_{k,i}) \in L_i\}$ are labeled undirected edges; an edge $\langle R_{j,i}, R_{k,i}, L(R_{j,i}, R_{k,i})\rangle$ indicates that there exists an equijoin predicate $L(R_{j,i}, R_{k,i})$ between two base tables, $R_{j,i}$ and $R_{k,i}$ of view V_i.

To facilitate efficient scoring of join queries, we constrain the **join-related scoring function** to satisfy a series of algebraic equivalences matching the join. For each pair of tables $R_i, R_j \in \Sigma$, we denote $FK(R_i, R_j) = \{\langle c, c'\rangle | c \in \bar{R}_i, c' \in \bar{R}_j, c \to c'\}$. We add a join edge between the relations, with a score based on the *joinability* between two data lake tables. This is based on the logarithm of their (estimated) selectivity when joined on any key-foreign key relationship(s):

$$jscore(R_i, R_j) = \log \frac{|R_i \bowtie_{\Phi_{ij}} R_j|}{|R_i||R_j|} \tag{1}$$

where $\Phi_{ij} = FK(R_i, R_j)$. We extend our pairwise joinability score to measure the joinability of tables from a subgraph of our join subexpression exploration graph, i.e., generalizing the score to more than two tables. Therefore, given $\{R_1, ..., R_l\}$ and the join subexpression exploration graph, we consider the log of the overall selectivity of joining all of the given tables:

$$jscore(R_1, ...R_l) = \log \frac{|R_1 \bowtie R_2 ... \bowtie R_l|}{\prod_{i=1}^{l} |R_i|} \tag{2}$$

where each join may have multiple predicates, with the attributes of multiple relations in the expression. To further simplify the score, we assume attribute independence, as is commonly done in query optimization in the absence of detailed statistics. In this case, we can rewrite the selectivity of a join expression $\bar{R}$ in a form as follows:

$$sel(\bar{R} \bowtie R_b) = sel(\bar{R}) \prod_{\Phi_{a,b}, R_a \in \bar{R}} \frac{|R_a \bowtie_{\Phi_{a,b}} R_b|}{|R_a||R_b|} \tag{3}$$

Here *sel* refers to the selectivity of a join expression, R_a and R_b are tables respectively.

Therefore, our *jscore* can be decomposed as follows with the attribute independence assumption:

$$\begin{aligned} jscore(R_1, \ldots, R_l) &= \log \frac{|R_1 \bowtie_{\Phi_{1,2}} R_2|}{|R_1||R_2|} \frac{|R_1 \bowtie_{\Phi_{1,2}} R_2 \bowtie_{\Phi_{2,3}} R_3|}{|R_1 \bowtie_{\Phi_{1,2}} R_2||R_3|} \\ \cdots \frac{|R_1 \bowtie_{\Phi_{1,2}} \ldots \bowtie_{\Phi_{\xi_l,l}} R_l|}{|R_1 \bowtie_{\Phi_{1,2}} \ldots T_{\xi_l}||R_l|} &= \sum_{\Phi_{\xi_j,j}} \log \frac{|R_{\xi_j} \bowtie_{\Phi_{\xi_j,j}} R_j|}{|R_{\xi_j}||R_j|} \\ = \sum_{\Phi_{\xi_j,j}} jscore(R_{\xi_j}, R_j) \end{aligned} \tag{4}$$

where ξ_l represents the index of any table which R_j is joined with, and $R_{\xi_j} \in \{R_1, ..., R_{j-1}\}$. Note that this decomposition also indicates the *associativity* of our join score.

3.2.2 ***Multi-Table Relatedness.*** Now, we can combine other relatedness factors to finalize our scoring function. Given a search query V_q, assume there is a mapping σ which associates a sub-relation S_i with a table $R_j \in \Sigma$. This mapping will be accompanied by a relatedness function $rel_\sigma(V_q)$ that measures the relatedness between the search query V_q and the tables mapped from the sub-relations of the candidate expression by σ.

The relatedness function $rel_\sigma(V_q)$ is defined between a candidate and the query expression as follows:

$$\sum_{S_i, \sigma(S_i) \in \Sigma} rel\big(S_i, \sigma(S_i)\big) + \sum_{L(S_i, S_j) \in L_q} jscore\big(\sigma(S_i), \sigma(S_j)\big) \tag{5}$$

Therefore, our top-k search problem is to find top-k mappings $\Gamma_k = \{\sigma\}$, such that $\forall \sigma \in \Gamma_k$, $rel_\sigma(V_q) > rel_{\sigma'}(V_q)$, if $\sigma' \notin \Gamma_k$.

4 CORRELATED TOP-K SEARCH

The previous section formalized the search space (the set of conjunctive queries across all combinations of sources) and the scoring function. We then want to use pruning strategies to explore a minimal number of options. As in prior work, we seek to adapt Fagin's Threshold Algorithm [12] (TA) to guide exploration. The standard TA algorithm considers how to explore a set of "streams" that each has scoring sub-components. In our setting, it is natural to consider the matches at each level of the hierarchy to be a separate "stream". The key challenge here is that each of these "streams" is itself a top-k search across many possible tables, considering multiple metrics. The choices made at one level of streams, in turn, interact with the choices made at the next level.

A strawman greedy solution involves tackling this problem at two levels: first, for each sub-relation S_i from the search query V_q, find the top matches. Then, take the top matches to each sub-relation, and find the top-scoring ways of joining these (first, starting with knowledge of foreign keys to other tables, then finding other matches by looking at value overlap). It should be intuitively obvious that the best matches at the local level (which consider factors such as schema overlap and row non-overlap) may not themselves have enough correlation in their join keys to produce meaningful answers, which are supposed to be the best matches at the global level across all sub-relations.

Our proposed solution to the *correlated top-k search* problem builds upon the prior section. Since it is infeasible to enumerate all possible join-nest queries (views), that can correspond to all sub-graphs of the join expression exploration graph introduced in Section 3.2, we propose to incrementally assemble, as separate streams, "promising" subexpressions (of joins), and meanwhile ensure the optimality of the relatedness score (Eq. 5). Specifically, "base" streams are computed by matching data lake tables to each S_i in the search query V_q, using our multiple table-to-table metrics. Intuitively, at the second level, our algorithm starts to combine multiple such streams based on ranked order of cost — in the process, greedily applying any relevant predicates from the join expression. Ultimately, once a combination of base streams is generated that incorporates all base streams, and satisfies all predicates — it becomes a complete candidate query that is emitted if its score is guaranteed to be greater than or equal to the score of any remaining queries to be explored. The idea of incrementally combining streams is similar to J* search [35]. However, our application requires the "base" stream to be read incrementally, because each stream itself is a top-k search across 1NF tables. Our algorithm must minimize accesses to sub-optimal 1NF tables and sub-queries, as each may result in costly computation that can end up not contributing to the final top-k results. Such challenges do not occur in J*.

To present our algorithm, we start with the base case, with only two tables in the search data. Subsequently, we generalize it to multi-table search data by incrementally assembling "base" streams.

4.1 Top-k Search for Two Streams

The basic idea of detecting the top-k join-nest queries for two-stream inputs (i.e., to match a 2-level hierarchy within our search table) extends ideas of the No-Random-Access Algorithm [12]. Here are the notations. For tables $S_i, S_j \in V_q$ (the search object), each table $t' \in \Sigma$ will be ranked based on $rel(S_i, t')$ and $rel(S_j, t')$, where Σ refers to all 1 NF primitive tables in the data lake. For simplicity, we denote the ranked list of Σ based on $rel(S_i, t')$ as Σ'_i (Note Σ'_i is only for notation; we do not actually rank tables in Σ.). We introduce a window size d, which represents that only d tables in each stage will be accessed from Σ'_i and Σ'_j. Note accessing d tables from Σ'_i and Σ'_j are two top-d queries to Σ respectively. Our goal is to find the top-k most related join-nest queries to V_q, meanwhile exploring the fewest 1NF tables and intermediate sub-expressions of join over those 1 NF tables.

Specifically, we first issue two top-d queries to read d tables from Σ'_i and Σ'_j respectively, and try to detect the top-k mapped expressions for V_q based on the join information of those $2d$ tables. To achieve the goal, we try to estimate the relatedness score for all possible expressions (views). For the views that consist of only those $2d$ tables from Σ'_i and Σ'_j, we can directly compute the relatedness score. For possible views that consist of at least one table outside of those $2d$ tables, we estimate their relatedness score. Specifically, if we denote the unseen table as t^*, i.e., $index(\Sigma'_j, t^*) > d$, we compute the *lower bound* of the relatedness score by replacing $rel(S_j, t^*)$ with 0, and compute the *upper bound* by replacing $rel(S_j, t^*)$ with $rel(S_j, t^\circ)$, where $index(\Sigma'_j, t^\circ) = d$. For possible views that consist of only tables that do not belong to those $2d$ tables, they must have a lower relatedness score than other views. Then, we can use the lower and upper bounds as thresholds to find the top-k views. More specifically, we maintain a priority queue of k highest lower bounds. If the k highest lower bound exceeds the upper bound of all other candidate views, we then get the top-k views for the two-table search object. We can use the same proof of the No-Random-Access algorithm [12] to prove the optimality. If we cannot detect the top-k views after accessing all tables that have been read, we then read the next d tables from Σ'_i and Σ'_j respectively, update the corresponding lower and upper bounds for related views, and check if the updated bounds can lead to top-k results. If this effort still fails to detect the top-k expressions, we then read tables again from both lists to continue the exploration. We will keep reading d tables from the input streams stage by stage until we successfully obtain the top-k expressions. Note that in this process, the expressions (views) are constructed based on the join expression exploration graph defined in Section 3.2.1, and the detection, indexing, and maintenance of relations in the graph will be introduced in Section 5.

4.2 Generalizing to Multiple Streams

We then extend the algorithm from the previous section to the general case, considering multiple streams for search objects with more than two tables. Similarly, it starts with reading d tables from each stream through top-d queries. Then it sequentially works on tables from each stream to generate top-d partial expressions, starting from a single relation, building join expressions, until generating complete expressions that can be mapped to all sub-relations from the query. For example, if search query V_q includes S_1, S_2, S_3, we first detect the top-d sub-expressions for $\langle S_1, S_2 \rangle$. Then in the next step, the list of candidate sub-expressions becomes one of the input streams, until the final top-k join-nest queries, here $\langle \langle S_1, S_2 \rangle, S_3 \rangle$, are identified. If there is any intermediate step where we fail to detect the top-d sub-expressions, it means that more tables should be

read from the "base" input streams. We will then read the next d tables from those related streams, use them to update the corresponding lower bounds and upper bounds, until top-d sub-expressions are derived, so that we can use them to continue the detection of the top-k complete expressions.

5 INDEXING EVOLVING DATASETS

A major challenge in supporting standard data lake search is the computational and I/O costs: matching between a single search table and n tables conceptually requires n schema matching operations; and if such schema matching takes into account the data instances, that may involve up to $O(nrk^2)$ computations, where r is the average number of rows and k is the average number of columns. Once we extend this to consider searching for a hierarchical data S that may be comprised of q sub-relations, where new tables are constantly being added to the system — this problem is significantly exacerbated. Thus, we not only need to leverage sketching and indexing techniques from the literature, but also need to extend them to handle scale and dynamicity.

Data profiles. As noted in Juneau [46], when multiple table-similarity metrics are incorporated into data lake search — it becomes essential to rapidly find data that provides effective thresholds for pruning. The intuition behind was to compare S against tables for which we can find good partial matches: namely, on attributes that are likely to appear in semantically similar tables. Therefore, we developed an indexing mechanism for Juneau called *data profile* that captured, for important columns and domains, the sets of tables containing this column. This notion can also be generalized to combinations of columns, e.g., streets and cities.

A table search in Juneau always starts with a match from search table S against any existing data profiles, which typically allows it to start with a "tight" bound for exploring matches. Unfortunately, Juneau relies on a human expert to identify and generate good data profiles, thus only a very small number of these exist. These have relatively easy-to-define patterns: phone numbers within a specific country code, dates, bank account numbers, etc. As new data continues to be added to the data lake, the set of columns most useful as data profiles may gradually shift. We develop innovative techniques to address the problem of *data profile selection* in Section 5.1.

Data sketches for estimating domain overlap. Second, as we compare tables to see if their instances overlap, we want to avoid fetching the actual tables. Zhu et al. [48] and others proposed that, instead we should use sketching and hashing techniques. LSH techniques allow us to estimate value overlap for strings and discrete information [16, 48]. Spoth et al. proposed that numeric attributes can be matched using distributional information [42]. Section 5.2 describes how we extend this work from its initial in-memory versions, to an in-database implementation that can be incrementally updated and can also be incorporated into SQL queries.

5.1 The Data Profile Selection Problem

Data profiles have been proposed to index columns (and associated tables) belonging to a given domain [46]. A *primitive* data profile can be defined for any given semantic type, e.g., a phone number or a bank account number. Unlike an index, it captures a domain, which can be represented as a triple $\langle type, matcher, inx \rangle$, where $type$ is the semantic type, $matcher$ is a function that predicts whether an instance of a table's column conforms to the semantic type, and inx is an index of all columns from tables in the data lake that satisfies the matching function (above some threshold). From this, we can define a lattice of *composite* data profiles, consisting of *combinations* of primitive (or simpler composite) data profiles. For instance, given profiles for first and last names and street addresses, we can build a composite profile for full names, and another for last names with street addresses. Importantly, each composite profile contains a superset of the attributes connected to it at the lower levels of the lattice, but contains a subset of the entries within the index.

Automated data profile generation. Each time our data lake search system is given a new table T to load, we first decompose it into a series of primitive tables, as described earlier in this paper. Subsequently, we match the columns of T against all existing profiles, to determine whether that column belongs to an existing profile. If it does not, we mark it as a candidate for a new (not yet computed) data profile. Periodically, in a background thread the system will go through all candidate data profiles, looking for the most suitable ones.

Identifying primitive data profiles based on value-overlap. Next, we develop a mechanism for identifying common-yet-easy-to-identify data domains that (potentially) represent meaningful semantic types, and are thus suitable for profiling. To do this, we need to track the distributions of each column in the data lake; see which columns seem domain-compatible, contain common representative instances and are suitable to be merged; and repeat. (Note that, in this case, we will be able to detect domain-compatible columns but not identify the name of the domain. It is acceptable in our case, because we use data profiles as indices.)

The basic test for domain compatibility is based on the sketches mentioned earlier in this section: two sketches matching above a threshold are considered to belong to the same domain. As will be described in Section 5.2, we maintain a persistent sketch for each column of a compatible type; once two columns' sketches overlap beyond a threshold, we map them together into the same single-column data profile, which is updated with a composite sketch that is the union of the columns' sketches.

Basic approach to composite data profiles. Composite data (multi-attribute) profiles can be defined using multiple primitive profiles that frequently co-occur. For example, *"street name", "city", "states"* and *"postal code"* can jointly represent a *"U.S. address"*.

Given the lattice structure of composite data profiles, there is a natural synergy with the apriori algorithm used to find frequent itemsets [1]. Leveraging the apriori algorithm, we establish a threshold for the minimum number of matches for a primitive candidate data profile. Next, we build upwards in our lattice, considering all pairwise combinations of profiles that exceed our threshold; and so on, for combinations of three, four, and more profiles.

Simpson's Paradox and the principle of optimality. The statistical phenomenon known as Simpson's Paradox notes that an association between variables may emerge, disappear, or reverse when the population is divided into subpopulations. In our empirical results, we discovered a high overlap between columns *only when those*

columns co-occurred with another column with specific sub-domains. For example, there is an overlap between the street names in Seattle and New York. However, if there is an attribute "postal code" in the same table, the difference of the values in the "postal code" can help us distinguish between a profile of street names in Seattle versus one with street names in New York. The fact we can discover new correlations by increasing the set of attributes means that the principle of optimality, assumed bythe apriori algorithm, is violated in certain cases.

We adopt a simple heuristic. As we iteratively combine primitive candidate profiles based on overlap, we set up two thresholds $\{\tau_1, \tau_2\}$, where $\tau_1 < \tau_2$. For a pair of candidate profiles C_1, C_2, if their similarity $sim(C_1, C_2) > \tau_2$, we combine these candidates into a single primitive data profile. If this condition is unsatisfied, yet $sim(C_1, C_2) > \tau_1$, we then look for a pair of attributes, A_1, A_2, such that if

- A_1 co-occurs in tables with C_1, and A_2 co-occurs with C_2, and
- $sim(A_1, A_2)$ exceeds a third threshold τ_3 ($\tau_3 < \tau_2$), i.e., A_1 and A_2 have substantially overlapping domains

then we instead merge C_1, C_2 into a candidate profile, and similarly for $A_1, A2$, even though they do not separately satisfy the apriori threshold condition. Then we create the composite profile, which *does* satisfy the threshold.

5.2 Incrementally Maintainable Sketches

Many aspects of ranking in table search rely on a test for value overlap between columns. Sketching has shown to provide a good approximation [42, 48] to value-overlap (join cardinality) scores for both numerical and string attributes, while enjoying a significant reduction in runtime versus doing the exact overlap computation. This motivates us to use existing sketching techniques to scale up creating and matching profiles.

Two of the most successful techniques have been shown to be LSHE [48] for string columns, and Kolmogorov-Smirnov (KS) [42] for numerical columns with similar distributions. However, prior work focused on establishing the effectiveness of these measures in relatively restricted, main-memory-only settings that were not incrementally updated. Our context introduces several new challenges that require incremental updates and out-of-memory storage.

First, given the dynamic nature of our data lake, the system must be able to augment sketches with data from new incoming tables on-the-fly. We are able to compute sketches incrementally and store them to speed up similarity score queries.

Incremental LSHE and KS. The LSHE algorithm, based on locality-sensitive hashing and ensembles, consists of distinct hashing and partitioning stages. Hashing can be done independently over each column from each table. However, partitioning is done holistically over all columns in the same sketch. Therefore, we developed a two-stage process in which hashes are computed for each column and stored persistently; and as new columns are added to the sketch, partitioning is re-run over the stored hashes. This sped up our computations by roughly 100 times.

Similar techniques were also applied to the KS algorithm: a histogram of each column/ profile can be computed independently and persisted. Then, partitioning is done over all histograms based on the largest value/ bucket of the histograms. The hashes and histograms computed and stored are then readily available to speed up online queries for similarity scores.

Sampling for KS. Additional optimizations were needed to speed up the construction of histograms for KS. Given that KS attempts to profile the distribution of numeric values, we can reduce the required space and time by computing sketches over a *sample* of values from a column. Therefore, we uniformly sample at random a fixed number of values from each column (in our work: we achieved about 450 × speedup by sampling only 10,000 values). We also found that for many domains, we could drop the least significant digits to get better coarse-level clustering (e.g., postal codes vary widely across cities, and by small amounts within cities). Both optimizations greatly helped improve the running times of numerical profiling, as well as the estimates of overlap. We further optimize querying KS by pruning away partitions which have largest value too far from the query distribution. For instance, a distribution represented by a histogram which has values on the scale of 1000 does not overlap with a distribution represented by a histogram which has largest value of 10, and thus we can prune away the partition of histograms with largest value of 10.

Finally, as we describe in more detail in Section 6, we implemented the sketch creation and value overlap estimation algorithms as user-defined functions in a relational DBMS.

6 SYSTEM IMPLEMENTATION

The algorithms described in the previous sections have all been implemented in an extension to the open-source version of Juneau. Our modifications focused on the middleware layer for indexing and search; and on the relational repository, with services implemented within PostgreSQL.

The Juneau middleware layer, written in Python on Tornado in order to maintain compatibility with the Jupyter backend, is invoked by the user through a Jupyter Notebook web environment. When loading data, it receives a serialized copy of a Pandas dataframe, JSON document, or other data structure. For querying, it receives a serialized copy of the (hierarchical) query table of interest. We replaced significant components of the Juneau code for both functionalities, in accordance with the algorithms described elsewhere in this paper. In addition to incorporating KS and LSHE sketches for measuring value overlap, we leveraged Juneau's existing scoring metrics for schema matching as a key component of finding union-compatible tables [46].

Our work pushes significant functionality into the underlying DBMS layer, rather than the middleware. We load our data into PostgreSQL, which we use to manage and index the tables in our corpus as well as the sketches.

Storing and using sketches in Postgres. We make heavy use of PostgreSQL's support for user-defined functions to integrate sketches. We re-implemented LSHE (the original reference implementation was done in-memory in GO) and other sketches using a combination of C user-defined and PL/pgSQL functions. For each table representing a sketch, heavily-queried columns were indexed. The use of functions and indices substantially reduces costs of loading results from the database, passing them to our middleware layer,

and subsequently executing the algorithms in a high-level language such as Python.

Storing normalized data in Postgres. As described in Section 2, the problem of storing hierarchical data in relations was heavily studied for XML in the early 2000s. We directly leverage those concepts in our implementation for XML data, and further adapt them for JSON. The JSON data model is significantly simpler than that of XML. It represents a composition of *dictionaries*, which are key-value pairs; *lists*, which are sequences of items; and scalars. In principle, JSON data could be shredded along similar lines to XML. However, JSON does not have a native schema language, and we must often *infer* the structure on-the-fly. Moreover, data developers use several conventions that introduce relational-mapping challenges seldom encountered in XML.

Dictionaries represent key-value pairs, but do not particularly differentiate between *keys representing schema columns* and *keys representing entry IDs*. In the common case, the keys represent schema elements, and scalar values can be mapped to individual columns within a relational table. At other times the key of a dictionary may simply represent a unique identifier for the associated value — in other words, a key for each row. Here, our relational schema might simply be the pair $(key, value)$.

Example 6.1. Consider the fragment:

```
{ "A. Rojas": ["paper2", ...]
  "A. Yan": ["paper1", ...],
  "D. Singh": ["paper2", ...]
  "J. Doe": ["paper1", ...],
}
```

Here, the dictionary's keys are not in fact column names in a 5-column relation schema, but rather author names, i.e., belong in a 2-column key-value schema! Thus, an appropriate storage format (if we do not wish to introduce a special token per author) might be $papers(dict\_id, author\_name, index, label)$ where $dict\_id$ represents the node ID given to the dictionary structure itself.

More generally, there may be *multiple* nested dictionaries, perhaps even as siblings in the JSON hierarchy, with similar structure and typing. Storing each dictionary in a separate table would make reconstruction of the JSON unnecessarily complex (since each dictionary would require a separate join). Thus, rather than create a separate table for each nested dictionary, we might create one $key\_strlist(parent\_id, struct\_id, key, index, label)$ table, which handles *many* dictionaries in a generic way. The $struct\_id$ can be used to determine which elements belong to the same dictionary; the $parent\_id$ allows them to be joined to the appropriate parent.

7 EXPERIMENTAL ANALYSIS

In this section, we evaluate our system against a collection of web tables and real data science workflows with hierarchical source datasets. We also compare our system against alternative ranking strategies based on heuristics. We consider four main questions about performance:

(1) Runtime: As we increase the complexity of queries, how does execution time vary in our correlated top-k algorithmic framework?
(2) Generalizability: Can the correlated top-k framework be applied to different individual table ranking modules and achieve improvement in execution time?
(3) Effectiveness: How often would our top-k framework return compatible hierarchical data with new information? How much would the result data quality downgrade (versus optimality as guaranteed by our system) if we instead use heuristics to compute top-k results?
(4) Usability: How would the results returned by our system help with a user's data science tasks?

Table 1: Statistics: Number of Joins v.s. Number of Views

# Joins	1	2	3+
# Views	900	191	35

7.1 Experimental Setup

7.1.1 Workloads, Datasets and Web Tables. We consider two classes of datasets in our data lake: (1) data used in real data science workflows, as found on kaggle.com; and (2) data occurring on the web at much larger scale, as collected in work on web tables [28].

Real data science workflows and derived tables. The data science workflows in our collection are manually analyzed, and are comprised of (1) 102 Jupyter Notebooks with their source data used in [46]; (2) 227 new Jupyter Notebooks with their source data, which is either in JSON format or include multiple CSV tables that naturally map into JSON objects. The new workloads included cover a variety of tasks and data domains. We describe some of them in Table 2. Since we focus on searching for complementary data sources, we identify some specific fields in the data (listed in Table 2), and categorize the data into different domains based on the values of these fields, to support the evaluation. Specifically, the field information enables verifying whether our algorithm and alternatives can return related expressions covering different domains.

To obtain queries and a corpus of tables for our data lake, we re-executed all of these notebooks and loaded all source, intermediate and final tables derived by the notebooks. Whenever we identified collection-valued or hierarchical data objects (i.e., nested list, dictionary) in a running cell, we parsed them and obtained the data. If it was a hierarchical data object, we further identified its base tables from the parsing result, and re-wrote the data object as a view over these base tables. We then stored and indexed all these base tables and other standard tables (Pandas dataframes) in PostgreSQL.

Web tables. It is difficult to scale manually-inspected collections of workflows, but we wanted to test the scalability of our system and introduce more new-yet-common tables. Thus, we further augmented our data lake with web tables collected in prior work, which are available as a large public data set [28].

Overall, we stored and indexed over **2.5 million tables**, among which 12k tables are derived from Jupyter notebooks, with an overall size of approximately around 200GB. For each table stored, we also computed and stored its sketches, and in total the size of the sketches of all tables in the corpus is around 37GB.

Data profiles and indices were created periodically for the tables generated by our notebooks. In the end, we identified 259 individual profiles and 5548 composite data profiles. The size of the sketches

Table 2: Samples from experimental workflows

Task	Dataset	Example Data Complementary Fields (Domains)
Citation network analysis	DBLP citation network[4]	Papers published by *ACM* and *IEEE*
Peek into the Airbnb activity	Airbnb Seattle [5] and Boston[6] Open Data	Airbnb activities in *Seattle* and *Boston*
Explore key education statistics	World Bank: Education Data[7] & GHNP Data[8]	Topics including *Education, Global Health, Nutrition, Population*
Predict flight delays	2015 Flight Delays and Cancellations[9]	Flights depart from *LAX, LAS* and *JFK*
Simulate a specific market strategy	Daily stock market prices[10]	Stocks of listed companies in *NASDAQ, S&P500, NYSE, Forbes 2000*

of data profiles for matching is around 3.6 MB. Examples of data profiles include *neighbourhood*, *longitude*, *country code*, *airline*, etc.

7.1.2 Environment. We conducted experiments on an AWS EC2 `t2.large` node, running PostgreSQL 12 on Ubuntu Linux 20.04. Our middleware layer was implemented in Python 3.8. Our data science IDE was Jupyter Notebook, although for experiments we made direct web service calls to Juneau.

7.1.3 Queries and Evaluation Metrics. We developed queries to study the efficiency and the quality of our results. The queries consist of JSON data objects used in and derived by the notebooks, which in turn can be specified as join-union-nest queries over "shredded" relations as we have described previously in this paper. We detail the queries and evaluation metrics for the key questions.

RQ1 & RQ2: Query Answering Efficiency and its Generalizability over Table Ranking Modules. To study the efficiency and scalability varying query complexity, we divide the queries into groups based on the number of joins required, i.e., queries with 1, 2, and more than 3 joins. Statistics are reported in Table 1. Since we have merged the tables with the same schema as described in Section 6, there are only a few views deeper than 3 joins. We then report the average running time of queries to evaluate the efficiency. Specifically, for each group of queries, we randomly sample 10 views from it as queries to be issued, and compare the average running time with alternatives.

To answer RQ2, we incorporate different individual table ranking modules and report the average query execution time, respectively, to demonstrate the generalizability of our algorithm. We will describe the details in Section 7.2.

RQ3: Effectiveness of Top-Scoring Results. To evaluate the effectiveness of the results returned by our query-answering algorithms, we consider two different types of metrics. The first one is *data domain coverage*, which is obtained by checking whether the returned queries recall data in complementary domains (Example domains are listed in Table 2.). Specifically, we reported the *mean recall* of complementary domains of the results. For example, as shown in Table 2, if our search data is about publications whose publisher is *ACM*, returned results that include publications by *IEEE* will increase the mean recall of complementary domains.

The other metric is *relatedness score*, with respect to the scoring function, of returned queries. Due to the large table corpus size, enumerating all possible combinations of tables to obtain the optimal solutions is infeasible. Therefore, we consider heuristics approaches as baselines. We compare the quality of the results returned by our algorithm against the heuristics baselines, reporting the percentage of improvement of the relatedness score varying k (position). If our algorithm is correctly designed, the scores of queries returned by our algorithms should be higher than those returned by the alternatives.

RQ4: Usability of Top-Scoring Results. To evaluate the usability of the results returned by our algorithm from a data scientist's perspective, we leveraged datasets from kaggle.com, which have been actively used by Kaggle users. We created a set of data science tasks for these datasets, such as classification and regression, and benchmarked the performance on a subset of the corresponding dataset (called source data). Then, we used the source data (typically, a JSON object) as a query to search for unionable views and then combined them with the source data to derive new data for the data science task. We then evaluated the usability of the results by checking whether the data science task performance had been improved. Details will be discussed in the next section.

7.2 Efficiency of Query Answering

To evaluate the efficiency of our correlated top-k algorithm, we conducted the experiments under three different setups corresponding to different individual table ranking modules. The first one is *Native Setup*, where we used the relatedness function proposed in this paper, and applied our data profiling techniques. The other two are *Customized Setups*, where we used other popular individual table ranking modules to show the generalizability of our algorithm.

7.2.1 Native Setup. We use the relatedness function proposed in Section 3.2.2 as our query ranking function, where we can evaluate the full suite of the techniques we have developed, including the correlated top-k algorithm as well as the automatically selected data profiles as indices.

7.2.2 Customized Setups. We consider two different ways to search for individual tables.

$D^3\boldsymbol{L}$: D^3L [11] [2] is based on LSH techniques. As a ranking function, it ensembles a broad set of features, which are all implemented by an individual LSH-based index.

HNSW: In light of recent advances in embedding models [29, 38] and vector databases [36], we experimented with table embeddings plus HNSW [30] as our individual table ranking module. The goal is to show that our computational framework can also be applied to a vector database backend, take advantage of its supported approximate KNN algorithms, and answer the queries efficiently with our correlated top-k algorithm. We use pgvector [12] as our vector database, and a BERT-based embedding model [13] to encode tables.

For customized setups, we cannot exploit data profiles as indices, because they are used in table relatedness computation; but we do take advantage of the correlated top-k algorithm.

7.2.3 Baselines. We set up baselines using the heuristics with existing individual table ranking modules over data lake, such as [2, 38, 46]. Specifically, we break a hierarchical dataset into its

[11] https://github.com/alex-bogatu/d3l/tree/main
[12] https://github.com/pgvector/pgvector
[13] https://www.sbert.net/

constituent relations, separately search for top matches to each, and then rank the top-scoring join expressions over these. More specifically, we compare the running time of the search with (1) our full system (SJ) including the data profiles serving as indices and the multi-stream correlated top-k algorithmic framework (in Native Setup); (2) the multi-stream correlated top-k algorithm *without* using data profiles as indices (NPS); (3) the strawman top-k algorithm introduced in Section 4 that fetches $z \cdot k$ tables from each input stream, and computes the best top-k sets of tables by conducting a Cartesian product of the top-$z \cdot k$ tables from each stream (here we vary z from 2-4, denoted as BL-2, BL-3 and BL-4, respectively). All of our implementations for Native Setup leverage the sketches of Section 5.2 to map columns and tables.

7.2.4 Results. Table 3 reports the runtimes of these methods, across different query complexity classes and setups. Our SJ is faster than alternatives in all cases when searching for top-20 results, and in most cases when searching for top-5. Specifically, our system can get the results 2×, 2.5× and 3× faster than BL-2, BL-3 and BL-4 respectively, when $k = 20$ and queries have 2 joins, and 4×, 14× and 43× faster, when the queries have 3 or more joins. The speedup shows that our algorithmic framework is more scalable than baselines due to its limited number of explorations for combinations. Furthermore, we can observe from the NPS column that leveraging data profiling as indices can consistently bring speed-up, due to its capability of reducing the times of computing mappings among the tables. We conclude that our strategy is nearly as efficient as simpler schemes when we have very simple query expressions, and that it is dramatically faster once the required query expressions are more complex. Similar conclusions can also be observed in Table 3 when using third-party table ranking functions. Leveraging our correlated top-k algorithm speeds up the query process across all of the cases, particularly when query expressions are increasingly more complex.

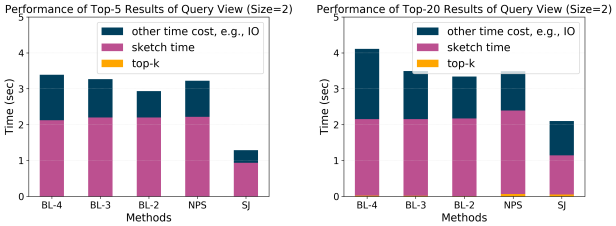

Figure 2: Performance of query views (Size = 2)

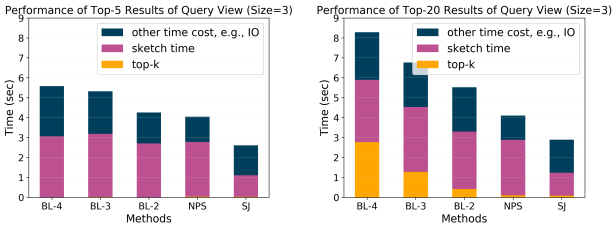

Figure 3: The Performance of Query Views (Size = 3)

Figure 2, 3, 4 further illustrate the contributions of different components to runtimes under Native Setup. Figure 2 shows that

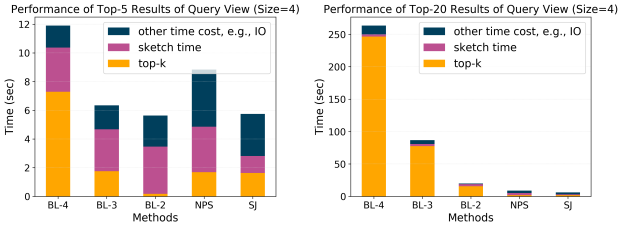

Figure 4: Performance of query views (size ≥ 4)

when the size of the search query is small, most of the runtime is spent on searching for relevant tables (sketch time); when search queries have more joins and need to return more results (Figure 4), most of the runtime will be used to explore the combinations for the top-k results.

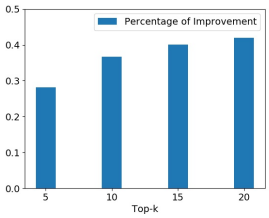

Figure 5: Percentage improvement of SJ, vs BL-2.

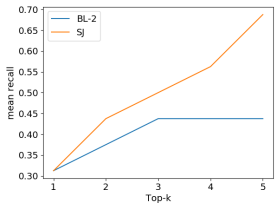

Figure 6: Query mean recall @k of related domains.

7.3 Effectiveness of Query Answering

We next investigate the quality of the results. Since it is infeasible to annotate "gold" queries that should be returned for each search object, we instead evaluate the effectiveness of the results from the following two aspects.

Recall of domains. We study whether our algorithm can find *complementary* (not just overlapping) queries for the search data. To achieve that, for each query listed in Table 2, since we know what domains they are related to, we check the top-5 sets of tables returned by SJ and BL-2 to understand whether they return the complementary data.

Optimality of the results with respect to the scoring function. Since our algorithm is guaranteed to return optimal results, we compared the relatedness score of views returned by our full system (under native setup) to the results returned by the greedy baseline, which is heuristic without any guarantee. Note that here we report BL-2, because BL-3 and BL-4 may already enumerate all relevant combinations, when $k \geq 5$.

We reported the mean recall of related domains we observed for SJ and BL-2 in Figure 6. As shown in the figure, BL-2 tends to return fewer new join combinations since it only focuses on the top ranked tables in each stream. Our system SJ, considers join combinations while simultaneously finding additional matches at each level of the hierarchy.

We visualize the ratio of the boost of the relatedness score result from SJ in Figure 5: clearly, SJ's top results are always much higher-scoring than the tables returned by BL-2, which is due to the fact that BL-2 only explores the combinations of the top-2k tables from each input stream. We observed consistent improvement across using D^{3}L and HNSW under customized setups.

Table 3: Mean time (sec) of returning top-5 & 20 join expressions.

	Native Setup					D^3L Setup				HNSW Setup			
D_2	BL-4	BL-3	BL-2	NPS	SJ	BL-4	BL-3	BL-2	NPS	BL-4	BL-3	BL-2	NPS
5	3.39	3.27	2.94	3.22	**1.84**	11.41	11.02	10.64	**7.47**	26.44	25.23	25.10	**24.48**
20	4.11	3.49	3.34	3.49	**2.29**	11.71	11.49	10.73	**7.75**	25.86	25.43	**25.11**	**25.11**
D_3	BL-4	BL-3	BL-2	NPS	SJ	BL-4	BL-3	BL-2	NPS	BL-4	BL-3	BL-2	NPS
5	5.58	5.32	4.26	4.05	**2.61**	23.03	22.25	22.21	**18.63**	35.66	35.15	35.04	**34.64**
20	8.28	6.77	5.52	4.10	**2.90**	29.98	25.84	25.00	**21.13**	41.94	39.50	36.87	**34.90**
D_{4+}	BL-4	BL-3	BL-2	NPS	SJ	BL-4	BL-3	BL-2	NPS	BL-4	BL-3	BL-2	NPS
5	11.91	6.35	**5.63**	8.83	5.74	28.93	23.94	23.65	**19.95**	47.66	46.73	44.53	**38.08**
20	263.61	86.75	19.90	8.87	**6.22**	nan	130.53	48.34	**20.27**	463.90	180.99	75.91	**39.60**

7.4 Usability of Top-k Results

To understand the usability of the results returned by our top-k algorithm, we conducted case studies using real datasets from kaggle.com, and created data analysis tasks for evaluation. Rather than conducting subjective user studies, we instead focus on demonstrating the value of our results to actual data science tasks. We focused on two types of tasks: regression and classification.

Specifically, for regression, tasks are about predicting flight delays [14] and simulating a specific market strategy [15] (where the benefits can be measured via a loss function). For classification, tasks are about predicting dental benefit utilization level [16], airbnb rating categories[17], various development index of Global Ecological Footprint[18], and the spending level of marketing campaign [19], which can be evaluated by the mean accuracy of predicted labels.

We created a data science task template for each task type: train a model to predict values or labels of interest. We then use a subset of the original dataset from Kaggle as the search data, and append the data derived by returned expressions, equivalent to expanding the original dataset. Then, we can evaluate the benefit of the augmented data by measuring the performance difference of the prediction.

The left part of Table 4 reports, for regression tasks of predicting flight delays and the stock market, the mean absolute error and mean squared error over the (augmented) datasets. Here, $k = 0$ means there are no suggested data augmented to the original search object. As we can observe from the table, the top-1 result notably reduces error, and while benefits are relatively minor as we go to the top-5 results, we still see improvements. The right part of Table 4 reports the mean accuracy of predicting labels for each data science task. It demonstrated that the results returned by our algorithm significantly boosted the classification accuracy.

Since it is difficult to scale the case study due to the required manual effort of designing meaningful data science tasks and performance evaluation for each dataset, we believe an automatic data science task design and evaluation for a given dataset would be an interesting future work.

Table 4: Case Study of Regression and Classification. MAE refers to mean absolute error and MSE refers to mean squared error.

	Rregression				**Classification**			
	Flight Delays		Stock Market		Airbnb Rating	Dental Utilization	Global Eco Footprint	Marketing Campaign
k	MAE	MSE	MAE	MSE	Accuracy			
0	11.26	235.39	1.10	21.378	35.92	71.93	62.16	45.09
1	10.79	227.93	1.09	21.377	40.26	71.27	64.86	51.23
2	10.79	227.93	1.09	21.377	42.00	72.46	62.16	48.88
3	10.79	227.93	1.09	21.377	43.66	74.08	64.86	58.48
4	10.51	227.90	1.03	21.373	43.66	83.65	67.57	59.60
5	10.51	227.90	1.03	21.373	42.93	83.65	67.57	59.60

8 CONCLUSIONS AND FUTURE WORK

This paper developed core techniques for *query-based data lake search*: our system indexes JSON, XML, or Pandas dataframe results, by converting them to a relational form; and conversely, when searching for hierarchical content, it assembles relations into appropriately structured matches to search datasets that are hierarchical. This involved several contributions:

- Novel techniques and scoring functions for efficiently performing *correlated top-k matching* across multiple tables that can be combined to form a hierarchical, join query result.
- Novel *automated* index selection techniques, extending ideas from data profiles, and scaling sketch techniques from prior work to out-of-memory settings.
- Techniques for incorporating indexing, profiling, and sketching techniques into robust open-source DBMSes such as the PostgreSQL system.

Our experimental results demonstrate significant benefits in terms of both quality and efficiency, when compared to direct extensions of the prior state-of-the-art. Compared to the baseline strategy that directly leverages top-k search over individual tables, then combines them — we see significant quality improvements in our answers (as measured by the ranking algorithm). We also obtain up to 40+x speedups in terms of efficiency, even with higher quality. Additionally, our mechanisms for generating automated data profiles provide additional speedup benefits of 30-100%, even when sketches are available in persistent storage which tends to slow things down. Finally, we established that our methods are effective in fairly complex hierarchical documents, yielding fast running times even over JSON queries of depth 4 or higher.

In future work, we hope to develop a more comprehensive benchmark for data lake search, considering both flat and hierarchical data, over a larger sample of data domains. We also hope to develop techniques to automatically tune the various weights in our ranking functions, to return match results that are of most relevance to data scientists performing real tasks.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable feedback, particularly with respect to answer quality and with respect to core functionality. This work was funded in part by NSF grant III-1910108.

[14] https://www.kaggle.com/code/abhishek211119/2015-flight-delays-and-cancellation-prediction
[15] https://www.kaggle.com/datasets/paultimothymooney/stock-market-data
[16] https://www.kaggle.com/datasets/mahdiehhajian/dental-utilization-by-provider
[17] https://www.kaggle.com/datasets/zakariaeyoussefi/barcelona-airbnb-listings-inside-airbnb
[18] https://www.kaggle.com/datasets/jainaru/global-ecological-footprint-2023
[19] https://www.kaggle.com/datasets/rodsaldanha/arketing-campaign

REFERENCES

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proceedings of 20th VLDB Conference*, Vol. 1215. Citeseer, 487–499.

[2] Alex Bogatu, Alvaro AA Fernandes, Norman W Paton, and Nikolaos Konstantinou. 2020. Dataset discovery in data lakes. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 709–720.

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

[4] Michael Cafarella, Alon Halevy, Hongrae Lee, Jayant Madhavan, Cong Yu, Daisy Zhe Wang, and Eugene Wu. 2018. Ten years of Webtables. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2140–2149.

[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. *PVLDB* 1, 1 (2008), 538–549.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System.. In *CIDR*.

[8] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2022. Turl: Table understanding through representation learning. *ACM SIGMOD Record* 51, 1 (2022), 33–40.

[9] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. 1999. Storing Semistructured Data with STORED. In *SIGMOD*. 431–442.

[10] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 456–467.

[11] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2023. DeepJoin: Joinable Table Discovery with Pre-Trained Language Models. *Proc. VLDB Endow.* 16, 10 (jun 2023), 2458–2470. https://doi.org/10.14778/3603581.3603587

[12] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.* 66(4) (June 2003), 614–656.

[13] Grace Fan, Jin Wang, Yuliang Li, and Renée J. Miller. 2023. Table Discovery in Data Lakes: State-of-the-art and Future Directions. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3555041.3589409

[14] 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 (mar 2023), 1726–1739. https://doi.org/10.14778/3587136.3587146

[15] Ju Fan, Meiyu Lu, Beng Chin Ooi, Wang-Chiew Tan, and Meihui Zhang. 2014. A hybrid machine-crowdsourcing system for matching web tables. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 976–987.

[16] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.

[17] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A Cardinality-Based Method for Coupled Estimation of Jaccard Similarity and Containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1190–1201.

[18] GitHub Inc. [n.d.]. Your AI pair programmer. ([n. d.]). https://github.com/features/copilot

[19] B Granger and J Grout. 2016. JupyterLab: Building blocks for interactive computing. *Slides of presentation made at SciPy* (2016).

[20] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 795–806.

[21] Alon Y Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Managing Google's data lake: an overview of the Goods system. *IEEE Data Eng. Bull.* 39, 3 (2016), 5–14.

[22] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 10073–10083.

[23] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[24] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. 2020. Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–12. https://doi.org/10.1145/3313831.3376798

[25] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2023. SANTOS: Relationship-based Semantic Table Union Search. *Proc. ACM Manag. Data* 1, 1, Article 9 (may 2023), 25 pages. https://doi.org/10.1145/3588689

[26] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proc. VLDB Endow.* 16, 4 (dec 2022), 932–945. https://doi.org/10.14778/3574245.3574274

[27] 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 *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 468–479.

[28] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A Large Public Corpus of Web Tables Containing Time and Context Metadata. In *Proceedings of the 25th International Conference Companion on World Wide Web* (Montréal, Québec, Canada) *(WWW '16 Companion)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 75–76. https://doi.org/10.1145/2872518.2889386

[29] Yibin Lei, Liang Ding, Yu Cao, Changtong Zan, Andrew Yates, and Dacheng Tao. 2023. Unsupervised Dense Retrieval with Relevance-Aware Contrastive Pre-Training. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 10932–10940. https://doi.org/10.18653/v1/2023.findings-acl.695

[30] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[31] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (apr 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

[32] 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 (2018), 59–70. https://api.semanticscholar.org/CorpusID:49417541

[33] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2016. Exemplar queries: a new way of searching. *VLDB J.* 25, 6 (2016), 741–765. https://doi.org/10.1007/s00778-016-0429-2

[34] 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. https://doi.org/10.14778/3352063.3352116

[35] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. 2001. Supporting incremental join queries on ranked inputs. In *VLDB*, Vol. 1. 281–290.

[36] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021* (2023).

[37] Rakesh Pimplikar and Sunita Sarawagi. 2012. Answering Table Queries on the Web using Column Keywords. *PVLDB* 5, 10 (2012), 908–919.

[38] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Ellen Tan, Yossef Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez, Jade Copet, Faisal Azhar, Hugo Touvron, Gabriel Synnaeve, Louis Martin, Nicolas Usunier, and Thomas Scialom. [n.d.]. Code Llama: Open Foundation Models for code. https://ai.meta.com/blog/code-llama-large-language-model-coding/.

[40] Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*. 302–304.

[41] Jie Song and Yeye He. 2021. Auto-Validate: Unsupervised Data Validation Using Data-Domain Patterns Inferred from Data Lakes. In *Proceedings of the 2021 International Conference on Management of Data*. 1678–1691.

[42] William Spoth, Poonam Kumari, Oliver Kennedy, and Fatemeh Nargesian. 2020. Loki: Streamlining Integration and Enrichment. *Human in the Loop Data Analytics* (2020).

[43] Petros Venetis, Alon Y Halevy, Jayant Madhavan, Marius Pasca, Warren Shen, Fei Wu, and Gengxin Miao. 2011. Recovering semantics of tables on the web. (2011).

[44] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger,

AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1539–1554. https://doi.org/10.1145/3318464.3389738

[45] Alexandros Zeakis, George Papadakis, Dimitrios Skoutas, and Manolis Koubarakis. 2023. Pre-trained embeddings for entity resolution: an experimental analysis. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2225–2238.

[46] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1951–1966. https://doi.org/10.1145/3318464.3389726

[47] 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*. ACM, 847–864.

[48] 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. https://doi.org/10.14778/2994509.2994534