Searching Data Lakes for Nested and Joined Data

Anonymous Author(s)

ABSTRACT

Data science is driving a new class of data management systems that assist data scientists with common tasks — e.g., by providing tools that take an existing table, and find relevant supplementary (joinable or unionable) data from an organizational data lake. Existing data lake search tools return matches to relational tables. Instead, data scientists often have non-normalized data (e.g., JSON, nested Python or R dataframes) — which does not directly match tables in the data lake, but might match combinations of data from such tables. We propose System J, a new data lake search platform that searches over *nested, joined* views of content in the data lake. The complexities of handling such queries require System J to develop novel techniques for indexing, sketching, and ranked-results merging. We experimentally validate the benefits of our methods using real data from data science computational notebooks.

ACM Reference Format:
Anonymous Author(s). 2022. Searching Data Lakes for Nested and Joined Data. In *SIGMOD '22: ACM SIGMOD Conference on Management of Data, Philadelphia, PA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Much like programmers with integrated development environments, data scientists benefit from integrated data management tools that accelerate their workflow: data search tools that discover useful datasets [21], visualization recommendation tools to help present data [19], and data validators to check that a data processing workflow is functioning correctly [25]. Such tools commonly make use of the tables and code from public repositories such as GitHub [28], as well as enterprise or public data lakes [21, 29–31] to find datasets that augment the data at the user's fingertips. This has driven new research in *searching across data lakes*. Here, the user typically selects a given table and searches for (1) additional union-compatible tables [20, 29, 31] useful for expanding training or test datasets, (2) for joinable tables [29, 30] useful in data integration tasks or for adding machine learning features, or (3) for computational steps in a notebook or workflow [15, 17, 28, 29] that can be used to extract key results.

A key limitation of these tools is that they are focused on searching for individual, "flat" tables. Yet data scientists often work with non-normalized datasets: JSON files, nested Python or R dataframes, and so on. If they attempt to use a data lake search tool to find additional data, they will often be disappointed about what is returned — even when, in principle, there exist tables that might be joined or nested to produce useful results.

Example 1.1. Consider the popular DBLP publications database (dblp.org). This database has been exported in a variety of different formats for use in data science tasks, as can be quickly seen by searching for DBLP notebooks on kaggle.com. One version [1] is in CSV format and represents a series of $(author, title, year, journal, \ldots)$ entries. Another appears in JSON format[2], with papers as the central entities and authors nested underneath.

Fundamentally, these represent different views (in a language such as the nested relational algebra [23]) over two (versioned) instances of the same dataset. If a data scientist is working with data in one of these representations, they should be able to search the data lake for additional data — which would be combined and structured with the same level of (de-)normalization. Naturally, the more complex result structures can be produced with a simple join (or nesting join) query over the "raw" 1NF form of the data.

Yet, existing data lake search tools only match one table at a time, and neither consider how to capture and index hierarchical data in the lake, nor how to match joined or nested tables against indexed data. In this paper we take a first step towards addressing the challenges in this space, addressing the problem of searching over the space of *combined tables* in a data lake: namely, we consider potential ways of generating matches to searches by joining or nesting tables in the lake. We assume that as hierarchical data is first indexed in the data, it is converted relational form; and that such normalized relations are combined at query-time to produce results matching a search. We make the following contributions:

- We develop storage strategies for hierarchical data (JSON, XML, dataframes), by storing the normalized base data and transformations over that data.
- We extend and automate prior techniques for *profiling and sketching* data, to index and scale data lake search to tens of thousands of tables.
- We develop novel techniques and scoring functions for efficiently doing *correlated top-k matching* across multiple tables that can be combined to form a hierarchical, join query result.
- We experimentally demonstrate the viability of our methods, over a diverse benchmark suite of real data and notebooks.

This paper is organized as follows. Section 2 reviews prior work on data lake search. We provide an overview of the approach, including how we represent tables in the data lake, in Section 3. Section 4 describes how we use indexing structures and sketches to address the problems of finding related columns at scale; Section 5 explains how such information is used to provide *correlated top-k search* capabilities that allow us to join multiple result streams. We describe the System J[3] implementation in Section 6 and conduct

SIGMOD 22, June 12–17, 2022, Philadelphia, PA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

[1] https://www.kaggle.com/jakboss/chunk-of-dblp-dataset
[2] https://www.kaggle.com/mathurinache/citation-network-dataset
[3] System name changed for anonymity.

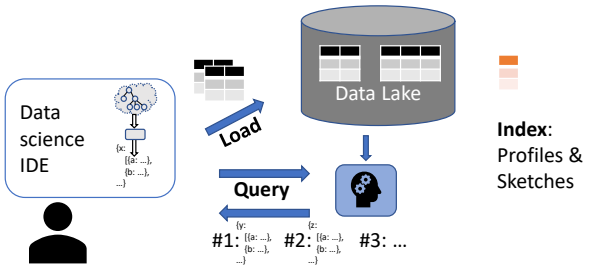

Figure 1: System J interacts with a data scientist's integrated development environment to provide data loading, indexing, and query services. Loading hierarchical data results in a conversion to relational form within the data lake. Querying the data lake with a relational or hierarchical data collection will result in returned results that match the structure and form of the provided dataset.

an experimental evaluation in Section 7. Finally, we conclude and describe future work in Section 8.

2 PRIOR WORK

The problem of finding related tables in a data lake has become an important one in the database and data science literature, as search tools must handle scale, relatedness, and freshness [21]. Early building blocks for large-scale table search predate the data lake, and generally query over tables for matches to keywords and phrases. Such tools were largely targeted at enterprise networks [2, 9, 10, 13, 14, 27] and/or tables that appear on the web [3, 4, 22].

More recent work assumes the user queries "by example" with a table rather than a set of keywords; and it may discover either unionable [7, 31] (sometimes called "data augmentation") or joinable [5, 11, 29, 30] tables within a data lake. Typically these must handle issues at scale: it becomes computationally intractable to do pairwise comparisons of columns between a "search table" and all tables in the data lake. Thus, LSH and other sketch-based techniques [10, 21, 29], as well as techniques borrowed from schema matching [18] have been investigated for determining which columns overlap. Recent efforts also consider how data from data science computational notebooks can be leveraged as a data lake [29], how computation over that data might be reorganized [16], and even how schema constraints might be inferred [25].

These prior efforts focus on taking a table and matching against "raw" tables, whereas our work on System J considers searching based on results that match *joined* or *nested* tables from our data lake. To map between relational and hierarchical data, we adapt ideas that were developed for storing XML in relations [6, 24]. To the best of our knowledge, no other system attempts to discover how to join or nest existing data lake tables; the most closely related work is Constance [12], which exploits (provided) semantic mappings to mediated schemas to enable query reformulation.

3 SEARCHING FOR COMBINED TABLES

Our work, implemented in System J, targets storing, indexing, and searching for both relational and nested (hierarchical) data from data lakes — relations, JSON, XML, and nested Pandas dataframes.

As illustrated in Figure 1, System J interacts with a data scientist's integrated development environment (IDE) — for example, with JupyterLab. As a data scientist loads external data into a form that can be processed, e.g., as a Pandas or Spark dataframe, System J can *load* this data into its data lake. Additionally, certain distinguishing fields may be *indexed* via *profiles* [29] and sketches. Finally, System J allows the user to select a *search data set* or dataframe as an example instance, and to *query* for additional data to *augment* the data set. This should return results with new content in a similar form, returned in ranked order based on how similar in structure and semantics they are to the original search data set, as well as how much additional data they provide. We describe each of these tasks in more detail.

3.1 Loading, Indexing, and Searching

Loading: As a data scientist loads, creates, or manipulates a flat or nested data resource, System J interacts with their development environment (e.g., a Jupyter notebook [29]) to capture and index that tabular or nested data resource. Our data lake storage system leverages a relational back-end, like prior data lake search systems. System J "shreds" the new data into relational tables as necessary [6, 24], and then records auxiliary metadata capturing the key-foreign key and nesting relationships that were represented in the original data. Such auxiliary information can be considered a primitive form of a view (limited to join, union, and nest [23] operations over the "shredded" tables).

Indexing: In order to scale the search and top-k ranking algorithm to large numbers of tables (and their combinations) in the data lake, we must develop techniques to (1) identify and pre-index important fields and sets of fields, such that we can use these to focus our search for matching columns in the data lake; (2) incorporate the latest techniques from locality-sensitive hashing and sketching to identify overlapping columns; (3) develop measures to predict how many *new* as well as overlapping values exist between columns in two different relations.

Searching: A data scientist may select some data resource D and search for other related data to *augment* D. If D is a flat relation, we will search for new, non-duplicate rows that have very similar schemas. If D is the result of a workflow operation that joins input tables, we will search for tables similar to each of the inputs, which add new joint tuples that complement existing ones. If D is a hierarchical dataset, i.e., a forest, we will consider how to add new values at different levels in the hierarchy: this may consist of adding new child elements with the same parents (e.g., for DBLP, this may involve adding new publications for existing authors) and/or adding new top-level parent elements (e.g., adding new authors, also with nested publications to match the structure of existing trees). To actually create this result, we must *search for matches at different levels of a hierarchical dataset*, and also consider how effectively they join — returning ranked matches according to a relatedness function. To achieve the necessary scale, we must develop a novel variant of the top-k ranking algorithm, which can merge multiple streams.

Of course, the quality of search results in System J is highly dependent on the ranking function. In this paper we use basic metrics established in prior work [29]. However, as we describe

later, our platform is very general, and a number of different metrics could be "plugged into" the System J framework in order to capture domain knowledge or algorithmic innovations. We describe our approach to storage in Section 3.2, and then present techniques for indexing important columns in Section 4. Section 5 then describes how we develop a cost model and adapt top-k search to consider combinations of tables in ranked fashion.

3.2 Storing Non-Normalized Data

The problem of storing hierarchical data in relations was heavily studied during the early 2000s, as techniques were developed to store XML in relational DBMSs, and to subsequently reconstruct the XML results. For System J we store both XML and JSON, and subtle differences in the data model and its conventions require us to develop innovative techniques for the latter.

3.2.1 XML. The XML data model intermingles sequential data (in the form of *elements* with open- and close-tags) and unordered key-value pairs (in the form of a small number of *attributes* embedded within open-tags); data is always stored in text nodes. Elements do not necessarily have a unique attribute differentiating them by identity; but given that each has a unique position in the XML document, when we store an element we can assign a unique ID to it based on its byte offset within the file.

The problem of storing XML in relations, often called "shredding' the XML, is well-studied [24], so we sketch the basic approach here. Assume we have XML data with a schema. We evaluate the element nesting structure to determine *repeating child elements*, and split the XML document into parent and child relations: the first captures the parent structure and its elements/attributes/text nodes, and the second captures the (repeating) child structure and its elements/attributes/text nodes. Finally, we add a foreign key between the child table and parent table. If we assigned an ID to every element based on its position, this ID also can be used to establish a relative ordering among the elements.

Example 3.1. Given a fragment:

```
<dblp>
  <article>
   <title label="paper1">First paper</title>
   <authors>
    <author>J. Doe</author><author>A. Yan</author>
   </authors>
  </article>
  <article>
   <title label="paper2">Second paper</title>
   <authors>
    <author>A. Rojas</author><author>D. Singh</author>
   </authors>
  </article>
</dblp>
```

Following standard practice, we would split the table into three tables: $instances(doc\_id)$ (which would have a single row for each document conforming to this schema); $articles(doc\_id, article\_id, label, title)$; $authors(article\_id, author\_id, name)$. We would also record the foreign key-key relationships between the fields in *author* and *articles*, and between fields in *articles* and *instances*. The original XML could be reconstructed by computing joins between *instances*, *article*, and *authors*; then applying a nest operator to group elements under common, merged parents; and finally adding appropriate element tags and attribute names.

XML storage is performed first by consulting the schema and creating the appropriate relations; then, on-the-fly as the XML is parsed, different components of the XML tree are directly inserted into the appropriate tables.

3.2.2 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 a wide variety of conventions, which introduce 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 3.2. Consider the fragment:

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

Here, the keys of the dictionary are not in fact column names, equivalent to a 5-column relation schema, but rather keys representing authors, equivalent to 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 generically handles *many* dictionaries in a generic way. The $dict_id$ can be used to determine which elements belong to the same dictionary; the $parent\_id$ allows them to be joined to the appropriate parent structure.

Algorithm 1 provides detailed pseudocode for how we handle a JSON structure (non-scalar JSON object). When we create multiple relations representing list elements that have a common schema, we coalesce them into a single relation. Additionally, when we create tables representing dictionaries within a structure — if these tables result in small and sparse tables, we "pivot" column names into attribute values, and add multiple rows per dictionary.

3.2.3 Other Types. Many data scientists use Pandas or Apache Spark dataframes to manipulate their data; dataframes are essentially ordered multiset relations, and support hierarchical content.

Algorithm 1 Storing JSON

```
Function StoreJSON:
Require: T: JSON, c: table, L: lake, d: depth
  if T is a list then
    Create a blank tuple x matching c's schema, set parent_id
    to the current tuple in c, and set struct_id to a unique ID
    representing the list
    for i in T do
      if i is scalar then
        Add i as a column to c
      else
        Let t := new table added to L
        Add parent_id to t and link it as a foreign key to c
        StoreJSON(i, t, L, d+1)
        If t shares the same schema as another table t2 in L, merge
        t with t2
      end if
    end for
    for i in T do
      if i is scalar then
        Set x[i] := value(i)
      end if
    end for
    Append x to relation c
  else if T is a dictionary then
    Create a blank tuple x matching c's schema, set parent_id
    to the current tuple in c, and set struct_id to a unique ID
    representing the dictionary
    for i in keys(T) do
      if i is scalar then
        Add i as a column to c
      else
        Let t := new table with name i added to L
        Add parent_id to t and link it as a foreign key to c
        StoreJSON(i, t, L, d+1)
      end if
    end for
  end if
  for any child relations s of c with < τ non-null columns per row
  do
    Create or reuse table t'(parent_id, struct_id, key, value)
    for each tuple x in t_s do
      for each non-null column-value combination (k, v) in x do
        create a row in t' copying the parent_id and tuple_id,
        and set key = k and value = v.
      end for
    end for
    Delete t_s from the data lake L
  end for
```

If the dataframes are flat, System J simply stores them as relations with an additional field capturing the index order. If they included nested structure, we leverage the same techniques as for JSON.

3.3 Constructing Structured Output

Above we described how System J transforms from hierarchical data to relations, when content is loaded into its managed data lake. Conversely, given a search dataset S, we must find matches to the different "shredded" components of S and construct a transformation that combines them in a similar structure to S.

To do this, we first identify the different relations corresponding to our search dataset, using the same methods as in the prior section; accompanied by a data transformation (join-union-nest query, where the join may be a left outerjoin) corresponding to how those relations should be combined.

Example 3.3. Given the XML fragment of Example 3.1 as a search dataset, System J will break the XML into *instances*, *articles* and *authors* relations; and also define a query $n_{instance}(instances ⟕ n_{articles}(articles ⟕ authors))$, where ⟕ represents a left outerjoin between the parent elements and their children; and following the nested relational algebra, $n_{\bar{x}}$ represents a nesting operation that groups tuples with common values of $\bar{x}$ and produces a nested (non-aggregated, non-1NF) list of elements from the remaining columns.

More generally, if an XML or JSON tree structure has multiple sibling sub-tables, we will have nesting operations over unions of left outerjoin expressions.

Scaling up search for combined tables. Conceptually, System J must run *multiple* top-k searches at the same time (one per "shredded" relation from the search dataset) while also considering how well the tables join and what new results they add: this is in essence a top-k ranked join query, posed over a stream of top-k searches for matching relations. The top-k search problem over data lakes is already expensive in terms of computation and I/O, and this further challenges our ability to scale performance. We tackle this in a multifaceted way.

To improve the ability to find relevant tables that might be combined to produce a match to the search dataset — we fully develop an idea proposed in prior work [29], called a data *profile*. Intuitively, a data profile represents a set of matching algorithms that identify semantically meaningful columns (e.g., a country) or combinations of columns; System J would pre-index all matches to a profile in the data lake, enabling searches to directly look up matches to these columns. Prior work assumed that data profiles were predefined, and that matchers were developed as detection algorithms written by a data expert. In Section 4 we consider how to automatically discover data profiles, suggest them to a data administrator, and build them as needed. Our work uses sketch-based techniques to identify whether a given column matches a data profile.

Additionally, to produce results that combine multiple source tables, we must go beyond the top-k ranking algorithms used in prior data lake search tools. In Section 5 we show how we create a ranking model for joined query results, and use it to drive a new algorithm for doing correlated top-k search across multiple "streams" of source relations.

4 INDEXING THE DATA LAKE

A major challenge in supporting traditional 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 $O(rk^2)$ 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 dataset D that may be comprised of q sub-relations, this problem is only exacerbated.

We seek to reduce this dramatically, by leveraging and extending two main methods that have been proposed in the literature.

First, given that we are performing a top-k search over possible matches — it becomes important to establish an effective pruning *threshold* to prioritize our search over promising candidate results. This suggests that we should start by matching D against tables for which we can find good partial-matches: namely, on attributes that are likely to appear in most tables that will match. This basic intuition led to the proposal of data *profiles* [29], which indexed tables in the data lake if they contained columns of sets of columns of particular significance to a domain. Suppose dataset D contains an important attribute like a social security number — if we pre-indexed all other tables containing social security numbers, we might start by matching D against those tables, and then find all table matches scoring above the thresholds established here. Unfortunately, a limitation of the prior work was that data profiles needed to be selected by the system developer, and hard-coded detection algorithms needed to be introduced. Our contribution in Section 4.1 is to develop an offline technique to identify attributes or combinations of attributes suitable for data profiles.

Second, as we compare tables to see if their instances overlap, we want to avoid fetching the actual tables. Instead, we can approximate column overlap through the use of sketching and hashing techniques. Prior work leveraged hashing and locality sensitive hashing to estimate exact-value overlap for strings and discrete information [10, 31], and distributional information to estimate whether two columns with continuous values belong to the same rough value distribution [26]. Section 4.2 describes how we extend this work from its initial in-memory versions, to an in-database implementation that can also be incorporated into SQL queries.

We describe our work on these two problems assuming, for the moment, that we are focused on matching top-k results for a single table. In the next section we will discuss how to generalize this to multiple tables in a hierarchy.

4.1 Using and Selecting Data Profiles

Data profiles have been proposed to index columns (and associated tables) belonging to a given domain [29].

A *primitive* data profile can be defined for any given semantic type, e.g., a phone number or a bank account number. It is a triple $< type, matcher, inx >$ where $type$ is the semantic type, $matcher$ is a function that predicts whether an instance of a table conforms to the semantic type, and inx is an index of all tables in the data lake that satisfy 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.

Data profiles accelerate search because we can take a new search dataset and match it against all of the profiles' matchers. Now we take the satisfied matchers and combine them to get the highest match points within the lattice. We look up the index entries corresponding to these profiles, and use these as the initial candidates for top-k queries.

Automated data profile generation. Prior work [29] established that a set of predefined data profiles (for names, addresses, etc.) could be highly effective in pruning the search space of candidate tables in a data lake. However, it did not answer the question of how a user or data lake administrator could scale this up to take into account the unique datatypes in their domain.

Unfortunately, our task of matching and combining *multiple* relations to a given dataset essentially requires a strong set of data profiles to obtain effective performance. Thus, in this work, we develop novel techniques to detect attributes and attribute combinations suitable for data profiles, and *automatically* generate profiles.

Specifically, as the data scientist loads new data into their IDE, we match the columns of each new dataset against all existing profiles, to determine whether the column belongs to an existing profile. If it does not, we mark it as a candidate for a new data profile. Periodically, in a background thread System J will go through all candidate data profiles, looking for the most suitable ones.

Primitive data profiles. Prior work has focused on data profiles that have relatively easy-to-define patterns, as identified in advance by an expert: phone numbers within a specific country code, dates, bank account numbers, etc. Instead, our goal is to define 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 and are suitable to be merged; and repeat.

The basic test for domain compatibility is based on sketches, as defined in more detail in Section 4.2. We maintain a sketch for each column of a compatible type; if two columns' sketches overlap beyond a threshold, we map them together, and create a composite sketch by unioning together the columns' sketches.

Basic approach to composite data profiles. Composite data 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.

Non-monotonicity. In fact, empirical validation of our strategy demonstrated a nonmonotonicity property: sometimes, we discovered a higher overlap between columns *only when those column*

were adjacent to another column with common domain. For example, there is overlap between the street names in Seattle and New York[4] but 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.

Thus, 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 the 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 cooccurs in tables with C_1, and A_2 cooccurs with C_2, and
- $sim(A_1, A_2)$ exceeds a threshold τ_3, i.e., A_1 and A_2 have substantially overlapping domains, and
- $sim(C_1, C_2) > \tau_2$

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.

4.2 Sketching

Many aspects of System J must efficiently test for value overlap between columns. Sketching has shown to provide a good approximation [26, 31] 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 [31] for string columns, and KS [26] for numerical columns. However, prior work focused on establishing the effectiveness of these measures in relatively restricted, main-memory-only settings.

In order to make them more suitable for our context of dynamic data lake and much larger tables, we developed new techniques for incorporating these sketches into a larger-than-memory, DBMS-backed implementation.

First, we are able to compute sketches incrementally and store them to speed up similarity score queries. Given the dynamic nature of our data lake, System J must be able to add sketches for new incoming tables on-the-fly.

For the LSHE algorithm, which is based on locality-sensitive hashing and ensembles, there are both hashing and partitioning stages. Hashing can be done independently over each column. However, partitioning is done holistically over all instances that are being sketched. 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 incrementally. The hashes and histograms calculated and stored can then be made readily available to speed up online queries for similarity scores.

Additional optimization were done to speed up constructing histograms for KS. Given that KS attempts to profile the distribution of numeric values, we can reduce the space and time complexity of the algorithm by computing the sketch over a sample of the columnar data. We randomly sample a fixed number of items from the column (in our work: 10,000 values which sped up the computation by around 450 times). 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 tricks greatly helped improving the running times of numerical profiling, as well as the estimates of overlap. 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.

5 CORRELATED TOP-K SEARCH

The core problem we tackle in this paper is how to return top-k matches to a join-union-nest query that *combines* the results from multiple data lake tables. As with most ranked query systems, we employ a variation of the standard Threshold Algorithm [8], in this case used to rank candidate join-union-nest queries.

The scoring function used for the Threshold Algorithm must satisfy a key *monotonicity property*: if the overall function is $t(x_1, \ldots x_m)$ then $t(x_1, \ldots, x_m) \leq t(x'_1, \ldots, x'_m)$ whenever $x_i \leq x'_i$ for every i. Space limitations preclude a full description of the Threshold Algorithm, but we adopt a variation specifically of the No Random Access Algorithm.

Intuitively, this algorithm operates over multiple "streams" that index candidate results in descending order by one of the metrics x_i. Each time it reads from one of the streams, say i, this establishes a new minimum value for that score component — say, replacing value x'_i with x_i. At the same time, fetching this result also establishes new values for the remaining score components. The moment we retrieve (or compute) a new result whose total score exceeds the *threshold* established by looking at the *most recent* value consumed from each stream — we know we have an answer whose score exceeds any result not-yet-seen, and we can emit the result.

Suppose we want a generic scoring function for candidate join-union-nest queries over data lake tables, which represents the sum of several different scoring components. For simplicity we will refer to each query as a candidate *view*. The score of each view should consider at least the following factors: (1) how schema-compatible its output is, when compared to the search dataset; (2) the cardinality of the output of the view, which is indicative of how semantically related the base tables are; (3) the number of new rows the view adds to our search dataset, representing new information.

Suppose our search dataset D can be expressed as a series of base tables $d_1, \ldots, d_q$, which are to be joined and/or unioned together via a join-union-nest *query view* $V_q(d_1, d_q)$. Similarly, each view V is made up of the same join-union-nest query expression over a *different* series of base tables: $V = Q(t_1, \ldots, t_q)$. In fact, we argue it is appropriate to ignore the nest operations with regards to scoring: the nest operator affects the presentation of the data, but the *value*

[4]Note regarding anonymity: the authors have no direct connection to either Seattle or New York.

of each new query can be estimated by many how (non-duplicate) tuples it produces prior to the nesting operation.

Moreover, any union-join query can be broken into DNF, i.e., a union of conjunctive queries, and we propose that each conjunctive query can be scored independently. Thus, for simplicity of presentation, we will assume for the remainder of this section that our view query and our candidate views are conjunctive expressions.

A natural, yet still general, way to define a monotonic scoring function for a conjunctive queries is to make it the *weighted sum* of a series of components: a score establishing how schema-similar each d_i is to each t_i; another score establishing how many new results are in t_i but not in d_i; another score establishing how effectively each d_i joins with d_{i+1}.

Building upon this intuition, we develop a method to produce top-k join queries by *incrementally starting from single relations and building join expressions*, so long as these join expressions exceed a threshold; and combining these expressions to produce views – which we emit once they exceed a threshold.

5.1 Ranking Partial Joins

With a large number of tables in the data lake, there may be an extremely high number of candidate join expressions that could result in a suitable view. Rather than enumerate full join expressions immediately, our goal is to instead allow the exploration of these candidates, by incrementally assembling, as separate streams, "promising" subexpressions that start by matching each of the tables d_i in the search dataset our query view V_q. We can combine these subexpressions across the streams until we have a complete view, and use that to establish a threshold on all possible query expressions.

We do this by creating a *graph* in which nodes are relations, weighted edges represent similarity scores (between a search table d_i and some other table t_i, using some standard measure of union-compatibility as in prior data lakes search systems [29, 31]); or between some table t_i and some other candidate table with which it joins, t_j. The Threshold Algorithm receives a stream of results for each d_i, describing the join expressions that can be generated starting with matches to d_i, in decreasing order of score.

We denote the tables in the data lake as $\Sigma = \{T_i\}_{i=1}^{n}$. The views defined on Σ, are denoted as $V = \{V_i\}_{i=1}^{m}$.

For convenience, let us define a helper function $L(T_j^i, T_k^i) = \{\langle c, c'\rangle | c \in \bar{T}_j^i, c' \in \bar{T}_k^i\}$, which represents the pairs of the columns participating in an equijoin predicate between two tables T_j^i, T_k^i. Building upon this, we introduce a second helper function $P(T_j^i) = \{L(T_j^i, T_k^i) | \langle T_j^i, T_k^i, L(T_j^i, T_k^i)\rangle \in E_i\}$ which captures the predicates between tables T_j^i, T_k^i.

Then for each view $V_i \in V$, we define its view graph as $G(V_i) = \{N_i, E_i, L_i\}$, where:

- $N_i = \{T_1^i, ...T_l^i | T_1^i, ...T_l^i \in \Sigma\}$ are the nodes of the graph, each representing a base table of V_i.
- $E_i = \left\{\left\langle T_j^i, T_k^i, L(T_j^i, T_k^i)\right\rangle | T_j^i, T_k^i \in N_i, L(T_j^i, T_k^i) \in L_i\right\}$ are labeled undirected edges of $G(V_i)$, where the presence of an edge $\left\langle T_j^i, T_k^i, L(T_j^i, T_k^i)\right\rangle$ indicates that there exists an equijoin predicate $L(T_j^i, T_k^i)$ between base table T_j^i and another base table T_k^i of view V_i.

In addition to the views, we define key-foreign key constraints among the tables in the data lake. For each pair of tables $T_i, T_j \in \Sigma$, we denote $FK(T_i, T_j) = \{\langle c, c'\rangle | c \in \bar{T}_i, c' \in \bar{T}_j, c \rightarrow c'\}$.

Now let us consider how we might measure the suitability of mapping from some base table of our view query V_q to some other table in the data lake. Given a view query V_q, assume mapping σ which associates table $T_i^q \in N_q$ with a table $T_j \in \Sigma$. This mapping will be accompanied by a relatedness function $rel_\sigma(V_q)$ that measures the relatedness between the query view V_q and the tables mapped to from the base tables of the view by σ.

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

$$\sum_{T_i^q \in N_q} rel\big(T_i^q, \sigma(T_i^q)\big) + \sum_{L(T_i^q, T_j^q) \in L_q} jscore\big(\sigma(T_i^q), \sigma(T_j^q)\big) \tag{1}$$

and,

$$jscore\big(\sigma(T_i^q), \sigma(T_j^q)\big) = \log \frac{|\sigma(T_i^q) \bowtie_{\Phi_{ij}^q} \sigma(T_j^q)|}{|\sigma(T_i^q)||\sigma(T_j^q)|} \tag{2}$$

where $\Phi_{ij}^q = FK\big(\sigma(T_i^q), \sigma(T_j^q)\big)$.

Therefore, our correlated 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 $\sigma' \notin \Gamma_k$.

The intuition of *jscore* here is to measure the correlation among the tables, which σ maps to. Our basic idea is measuring the *selectivity* of joining all of the tables that are mapped to. Concretely, we define the selectivity of the join as a normalized join cardinality of $\{T_1, ..., T_l\}$ as follows:

$$jscore(T_1, ...T_l) = \log \frac{|T_1 \bowtie_{\Phi_{1,2}} T_2 ... \bowtie_{\Phi_{\xi_l,l}} T_l|}{\prod_{i=1}^{l} |T_i|} \tag{3}$$

where ξ_i represents the table which T_i is joined with.

Here, we introduce the attribute value independence assumption. Under this assumption, all columns are treated as if independent of each other. Therefore, we can derive the following results:

$$\frac{|T_1 \bowtie_{\Phi_{1,2}} \dots \bowtie_{\Phi_{\xi_j,j}} T_j|}{|T_1 \bowtie_{\Phi_{1,2}} \dots T_{\xi_j}|} = \frac{|T_{\xi_j} \bowtie_{\Phi_{\xi_j,j}} T_j|}{|T_{\xi_j}|} \tag{4}$$

Therefore, our *jscore* has associativity as follows:

$$\begin{aligned} jscore(T_1, \dots, T_l) &= \log \frac{|T_1 \bowtie_{\Phi_{1,2}} T_2|}{|T_1||T_2|} \frac{|T_1 \bowtie_{\Phi_{1,2}} T_2 \bowtie_{\Phi_{2,3}} T_3|}{|T_1 \bowtie_{\Phi_{1,2}} T_2||T_3|} \\ &\quad \dots \frac{|T_1 \bowtie_{\Phi_{1,2}} \dots \bowtie_{\Phi_{\xi_l,l}} T_l|}{|T_1 \bowtie_{\Phi_{1,2}} \dots T_{\xi_l}||T_l|} \\ &= \log \frac{|T_1 \bowtie_{\Phi_{1,2}} T_2|}{|T_1||T_2|} \dots \frac{|T_{\xi_l} \bowtie_{\Phi_{\xi_l,l}} T_l|}{|T_{\xi_l}||T_l|} = \sum_j \log \frac{|T_{\xi_j} \bowtie_{\Phi_{\xi_j,j}} T_j|}{|T_{\xi_j}||T_j|} \\ &= \sum_j jscore(T_{\xi_j}, T_j) \end{aligned} \tag{5}$$

Given these preliminaries, we can now develop a top-k algorithm that reads from multiple correlated streams of view subexpressions, composes these, and emits views in ranked order.

5.2 Correlated Top-k Search

Given the relatedness function and the associativity of the join score, our problem is to detect the top$-k$ mappings Γ_k. However, enumerating all possible mappings is infeasible, given the fact that each base table in the query view has a large number of candidate matching tables. Therefore, we propose a multi-way top$-k$ framework, where we treat the candidate tables of each base table in the query view as a data stream, and sliding windows will be specified to read the tables from the stream when it is necessary, so that we can explore the best mappings in a limited number of combinations without reading the whole stream.

In the following text, we first describe how to detect the top$-k$ mappings for two streams and then we generalize it to handle multiple streams, based on which we can detect the complete top$-k$ mappings.

5.2.1 Correlated top$-k$ search for two streams. The basic idea of detecting the top$-k$ mappings for two stream inputs follows the idea of No-Random-Access Algorithm [].

Concretely, for each query base table $T_i^q \in N_q$, tables in Σ are sorted based on $rel(T_i^q, T')$ where $T' \in \Sigma$, therefore, we denote the sorted list of Σ as Σ'_i. Furthermore, we denote d as the window size, which means in each stage, d tables will be accessed in Σ'_i.

Then, in the first stage, we read d inputs from Σ'_i and Σ'_j respectively, and try to detect the top$-k$ mappings for T_i^q and T_j^q. Therefore, we first do sorted access to inputs read from both lists, and compute the relatedness score for the possible mappings related to those tables. For each possible mapping with an unseen table (tables in the sorted list that have not been read.), i.e., $\sigma(T_j^q) = T^*, index(\Sigma'_j, T^*) > d$, we compute the lower bound by replacing $rel(T_j^q, T^*)$ with 0, and compute the upper bound by replacing $rel(T_j^q, T^*)$ with $rel(T_j^q, T^\circ), index(\Sigma'_j, T^\circ) = d$. Meanwhile, we maintain a priority queue of k highest lower bounds, and if the k highest lower bound exceeds the upper bound of all other candidate mappings, we then get the top-k mappings for two tables.

If we can not detect the top$-k$ mappings after accessing all tables that have been read, we then read the next d tables from Σ'_i and Σ'_j, and update the corresponding lower bounds and upper bounds for related mappings. If we still can not detect the top$-k$ mappings, we then do sorted access to the newly read tables from both lists to continue the detection. We will keep reading d tables from the input streams in each stage until we detect the top$-k$ mappings. The pseudo code is as shown in Algorithm 2.

5.2.2 Top$-k$ Mappings for multiple stream inputs. We then want to generalize our top$-k$ mapping detection to multiple input streams. Concretely, in the first stage, we read d inputs from all sorted lists. Then we choose a sequence of base tables to detect the mappings. For example, if the join predicates exist between T_1^q and T_2^q, and T_1^q and T_3^q, we can first detect the top$-d$ sub-mappings for $\langle T_1^q, T_2^q \rangle$ and then use them to detect the top$-k$ mappings for $\left\langle \langle T_1^q, T_2^q \rangle, T_3^q \right\rangle$. If there is any step where we fail to detect the top-d sub-mappings, which means we should read more tables from the 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-mappings is derived so that we can use it to continue the detection of the top$-k$ complete mappings.

Algorithm 2 Mapping Detection for Two Stream Inputs

Require: $V_q, T_1^q, T_2^q, \Sigma, d, k$
  STAGE = 1
  $\Sigma'_1 \leftarrow$ sort the tables in Σ based on $rel(T_1^q, T), \forall T \in \Sigma$
  $\Sigma'_2 \leftarrow$ sort the tables in Σ based on $rel(T_2^q, T), \forall T \in \Sigma$
  $\Gamma_1 \leftarrow \Sigma'_1[0:d], \Gamma_2 \leftarrow \Sigma'_2[0:d]$
  while True **do**
    for $(i = (\text{STAGE} - 1) * d, i < \text{STAGE} * d, i++)$ **do**
      $TL \leftarrow$ all tables that can be joined to $\Gamma_1[i]$.
      $TR \leftarrow$ all tables that can be joined to $\Gamma_2[i]$.
      $\Gamma_L = \{\sigma | \sigma(T_1^q) = \Gamma_1[i], \sigma(T_2^q) = T_y, T_y \in TR\}$
      $\Gamma_R = \{\sigma | \sigma(T_2^q) = \Gamma_2[i], \sigma(T_1^q) = T_x, T_x \in TL\}$
      $\Gamma_{[1,2]} \leftarrow \Gamma_L \cup \Gamma_R$
      for all $\sigma \in \Gamma_{[1,2]}$ **do**
        $LB_\sigma \leftarrow LB\big(rel_\sigma(V_q)\big), UB_\sigma \leftarrow UB\big(rel_\sigma(V_q)\big)$
      end for
      maintain the top-k list of mappings based on LB_σ.
      if the kth value is higher than the highest upper bound of the other mappings **then**
        return top-k mappings
      end if
    end for
    $\Gamma_1 \leftarrow \Sigma'_1[0 : \text{STAGE} * d], \Gamma_2 \leftarrow \Sigma'_2[0 : \text{STAGE} * d]$
    update lower bounds and upper bounds
    if updated kth lower bound is higher than the updated highest upper bound of the other mappings **then**
      return top-k mappings
    end if
    STAGE + = 1.
  end while

6 SYSTEM IMPLEMENTATION

The algorithms described in the previous sections have all been implemented in System J. Our implementation closely mirrors Figure 1 and consists of:

(1) A set of *web services* that can be invoked from the data science IDE to store, index, and query for tables.
(2) A middleware layer, constructed in Python 3.8, for managing both loading and top-k querying of data.
(3) An underlying relational repository, implemented in PostgreSQL 12, for capturing the data lake as well as auxiliary structures including sketches.

Data science IDE.. Currently we use Jupyter Notebook as the IDE, and use Jupyter's plugin architecture to (1) interact with the Python kernel to discover and load Pandas dataframes, dictionaries, and lists after each cell is executed; (2) provide GUI tools to select a dataframe and search for additional data.

Middleware layer. The System J middleware layer is invoked by web services from the IDE. For loading data, it receives a serialized copy of the dataframe or other data structure. For querying, it receives a serialized copy of the query table of interest. We are currently investigating the possibility of using Apache Arrow or similar mechanisms to share the same in-memory representation of the data across sub-systems. Our current architecture provides slightly greater flexibility at the cost of some performance. As our ranking functions, we build upon the library of scoring functions used in the Juneau system [29], which we forked from the open-source release (https://github.com/juneau-project/juneau).

Storage layer. We make heavy use of PostgreSQL's support for user-defined functions to integrate sketches. We re-implemented LSHE (which was done in-memory in GO) and other sketches using a mix of C user-defined functions and PL/pgSQL. Naturally, this substantially reduced the 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.

7 EXPERIMENTAL ANALYSIS

In this section, we evaluate our system on a collection of real data science workflows with hierarchical source datasets, and compare the results with alternatives. To evaluate the performance, we consider following questions: (1) What execution-time speedup can be provided by our correlated top−k algorithmic framework and data profiling, varying queries with different numbers of joins. (2) How is the quality of the sets of the tables returned by our system, especially whether the system can return complementary data sources, compared with the alternatives?

7.1 Experimental Settings

In this part, we will introduce how we set up the experiments, including how we obtained the data, workloads, how we generated the queries and the tables we will search for.

7.1.1 Workloads and Datasets. Our experiments use real data science workflows from kaggle.com. It includes (1) 102 Jupyter Notebooks with their source data used in [29]; (2) 227 new Jupyter Notebooks with their source data, which is either in JSON format or includes multiple CSV tables that can be written into a JSON object. The new workloads included in this experiments cover a variety of tasks and data domains. We describe some of them in Table 1. Note that since we focus on searching for complementary data sources, we identify some specific fields in the data (listed in Table 1), so that we can check whether our algorithm and alternatives return identical or diverse sets of related tables.

We ran all of the notebooks in our repository. Whenever we identify hierarchical data objects (i.e., nested list, dictionary) in a running cell, we parse it, obtain its base tables, and re-write them as a view on these base tables. We then stored and indexed all these base tables and other regular tables (Pandas dataframe) in PostgreSQL.

7.1.2 Storage and Indexing. Overall, our corpus stored and indexed over 12000 tables, with an overall size of approximately around 20G. For each table stored, we kept its sketches, and in total the size of the sketches of all tables in the corpus is around 165MB. Data profiles and indices were created periodically. In the end, we identified 165 individual profiles and 196 composite data profiles, whose size including their sketches for matching is around 2.3 MB. Examples of data profiles include neighbourhood, longitude, latitude, country code, airline, etc.

7.1.3 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 System J.

7.1.4 Queries and Performance Metrics.

Query Groups. We developed queries to study the efficiency and quality of our system. The queries consist of JSON datasets used in the notebooks, which in turn can be specified as join-union-nest queries over "shredded" relations as we have described previously in this paper.

To study the efficiency, we divide the queries into groups based on the number of joins required, i.e., queries with 2, 3, and more than 4 joins. Statistics are reported in Table 2. Since we have merged the tables with the same schema as described in Section 3.2, there are few views deeper than 4 joins.

To evaluate the query answering quality, we leveraged the workloads reported in Table 1, and used the views derived from those workloads as queries, so that we can clearly evaluate the recall of the query results.

Query Answering Efficiency. We report the average running time of a query to evaluate the efficiency. For each group of queries, we randomly sample 10 views from it, and compare the average running time with alternatives.

Query Answering Quality. To evaluate the quality of the results returned, we report (1) the ratio of the boost of the relevance score varying position k compared with the baseline; (2) the mean recall of the complementary domains of the query in our corpus, for example, as shown in Table 1, if our query view is about publications whose publisher is *ACM*, we hope the tables returned can include the publications published by *IEEE*.

7.2 Performance of Searching Tables

7.2.1 Query Efficiency. In this part, we want to evaluate the effectiveness of our correlated top−k algorithm and the data profiling as indices.

As a baseline, we use two heuristics that allow us to reuse existing single-table data lake search tools such as [29]: in one case, we break a hierarchical dataset into its constituent relations, separately search for top matches to each, and then rank the top-scoring join expressions over these. As a second option, we consider a nested-loops-style implementation where we find the best matches to the root table in the hierarchical data; then find the best-scoring joinable tables to these, and finally filter to join expressions that only match the structure of our hierarchical data.

More specifically, we compare the running time of the search with (1) our full system (SJ) including the data profiling serving as indices and the multi-way co-related top−k algorithmic framework; (2) the multi-way co-related top−k algorithm without using data

Table 1: Samples from experimental workflows

Task	Data Name	Example Data Complementary Fields
Citation network analysis	DBLP citation network[5]	Papers published by *ACM* and *IEEE*
Peek into the Airbnb activity	Airbnb Seattle [6] and Boston[7] Open Data	Airbnb activities in *Seattle* and *Boston*
Explore key education statistics	World Bank: Education Data[8] & GHNP Data[9]	Topics including *Education* and *Global Health, Nutrition, and Population*
Predict flight delays	2015 Flight Delays and Cancellations[10]	Flights depart from *LAX*, *LAS* and *JFK*
Simulate a specific market strategy	Daily stock market prices[11]	Stocks of listed companies in *NASDAQ*, *S&P500*, *NYSE*, and *Forbes 2000*

Table 2: Statistics: Number of Joins v.s. Number of Views

# of Joins	# of Views
2	336
3	75
4+	7

profiles as indices (NPS); (3) a straw-man top−k algorithm that fetch 2k tables from each input stream, and compute the best top−k sets of tables by conducting the product of the top-2k tables from each stream (BL). Note that, all of the implementations mentioned above leveraging sketches mentioned in 4.2 to map columns and tables, without which is not feasible to compute the results in a reasonable time. Furthermore, to fairly compare with (1) and (3), (3) also uses data profiles as indices.

Table 3 reports the running time of all these methods when issuing different groups of queries. As shown in the table,our full system is better than alternatives in most of the cases, especially when k is larger and the query is more complex. Our system can get the results 50× faster than BL when $k = 20$ and the queries have 3 or more than 3 joins, which shows that our algorithmic framework is more efficient due to a limit number of explorations of combinations compared with the baseline. Furthermore, we can also observe that leveraging data profiling as indices can always bring us a speed-up, due to the reason that it reduced the times of computation of mappings among the tables.

7.2.2 Query Answering Quality. To evaluate the quality of the top-k search, we first compared the relatedness score of the sets of the tables returned by our full system compared with the results returned by the baseline. We conduct this comparison because our multi-way co-related top-k algorithm is to find the optimal, while the baseline is a heuristic method without any guarantee. Therefore, we report the ratio of the boost of the relatedness score result from SJ in Table 2. We can observe from the figure that the tables returned by our SJ is always much more related than the tables returned by BL, which is due to the fact that BL only explore the combinations of the top-2k tables from each input stream.

To have a better understanding of the query results, we further study if our system can find complementary data for the queries. To verify that, for each query listed in Table 1, since we have an understanding of what domains they are related to, we checked the top−5 sets of tables returned by SJ and BL to understand whether they return the complementary data. We report the mean recall of related domains we observed for SJ and BL in Table 3. As shown in the figure, BL are less biased to explore new combinations since it only focuses on the top ranked tables in each stream. Our system SJ instead has more chance to explore new combinations due to its capability to find combinations with higher join score.

Table 3: Average running time (sec) of returning top−k sets of tables.

D_2	5	10	15	20
BL	0.0181	0.0192	0.0205	0.0226
NPS	0.0158	0.0161	0.0202	0.0212
SJ	0.0088	0.0129	0.0162	0.0166
D_3	5	10	15	20
BL	0.0185	0.3195	1.8002	5.6885
NPS	0.0168	0.0558	0.1027	0.1376
SJ	0.0095	0.0248	0.0494	0.1036
D_{4+}	5	10	15	20
BL	0.0258	0.6017	3.1853	10.078
NPS	0.0332	0.1113	0.1603	0.2135
SJ	0.0322	0.0833	0.1490	0.2017

8 CONCLUSIONS AND FUTURE WORK

This paper developed core mechanisms for System J, which goes beyond traditional data lake search tools to consider results that require *combinations* of indexed tables. System J indexes JSON, XML, or Pandas dataframe results, by converting them to a relational form; and conversely, it assembles relation to returns appropriately structured matches to search datasets that are hierarchical.

We developed several key contributions in our work:

- Storage strategies for hierarchical data, by storing the data in relations and capturing associated foreign key and structuring information.
- Novel automated index generation techniques, extending ideas from data profiles, and scaling sketch techniques from prior work to out-of-memory settings.

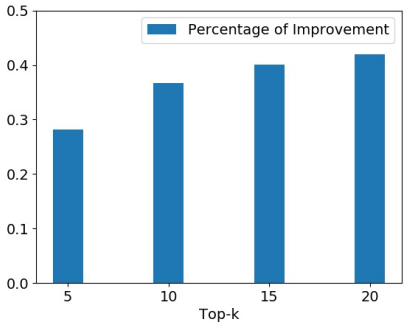

Figure 2: Percentage of Improvement of relatedness score of SJ compared with BL.

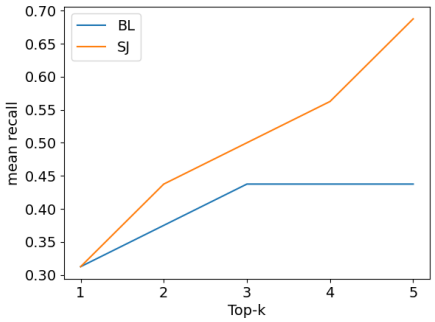

Figure 3: Mean Recall of Related Domains for the Query Data at K

- Novel techniques and scoring functions for efficiently doing *correlated top-k matching* across multiple tables that can be combined to form a hierarchical, join query result.

Our experimental results demonstrate significant benefits both in quality and efficiency, when compared to direct extensions of the prior state of the art. Compared to a baseline strategy that directly leverages top-k search over individual tables, then combines them — we see significant improvements in the quality of our answers (as measured by the ranking algorithm). We also see up to 50x speedups in efficiency, even with the 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. 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.

As 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 best match results judged to be of most relevance to data scientists performing real tasks.

REFERENCES

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. Citeseer, 487–499.

[2] 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.

[3] 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.

[4] 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.

[5] 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*.

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

[7] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2021. COCOA: COrrelation COefficient-Aware Data Augmentation.. In *EDBT*. 331–336.

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

[9] 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.

[10] 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.

[11] 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.

[12] Rihan Hai, Sandra Geisler, and Christoph Quix. [n.d.]. Constance: An Intelligent Data Lake System. In *Proc. of ACM SIGMOD Conf. on Management of Data*.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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

[18] 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.

[19] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A Hearst, et al. 2021. Lux: Always-on Visualization Recommendations for Exploratory Data Science. *arXiv preprint arXiv:2105.00121* (2021).

[20] 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

[21] 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

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

[23] Mark A Roth, Herry F Korth, and Abraham Silberschatz. 1988. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)* 13, 4 (1988), 389–417.

[24] 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.

[25] 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.

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

[27] 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).

[28] 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

[29] 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

[30] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 847–864. https://doi.org/10.1145/3299869.3300065

[31] 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