Finding Related Tables in the Data Lake

Yi Zhang
University of Pennsylvania
Philadelphia, PA 19104
yizhang5@seas.upenn.edu

Zachary G. Ives
University of Pennsylvania
Philadelphia, PA 19104
zives@cis.upenn.edu

ABSTRACT
An increasing number of data science applications build on data lakes, which are schema-agnostic repositories of data files and data products. However data lakes typically offer limited capabilities for organizing and managing data. Data scientists and analysts need richer capabilities for querying for related tables and datasets in the lake, in order to find additional training data in similar tables, to find related and linkable tables, and to find other tables produced by the same workflows and thus similar in role. We focus on data science use cases within computational notebook software, such as Jupyter Notebook. However, our methods generalize to settings where computational tasks involve execution of programs or scripts. We explore how to find related tables, efficiently track a multitude of intermediate results and data products, and exploit similarity when storing related tables.

PROVLD Reference Format:

1. INTRODUCTION
The data lake has emerged as a flexible, schema-agnostic repository for data sources and analysis results (“data products”), providing a single point of access for all data products independent of form. Typically, the data lake is an abstraction over a distributed file system, an object store, or even a centralized repository of computational notebooks (as in Mathematica, RStudio, Jupyter Notebook/Jupyter-Lab, Apache Zeppelin, etc.). Data lakes offer significant data access benefits, but, as with file systems and key-value stores, they do little to help a user find the most relevant data, understand relationships among data products, or integrate heterogeneous data sources or products.

The challenges of data lake management arise quickly in collaborative settings, as well as those in which data processing methodologies are changing. When data products are updated across versions, processed across (often similar) computational stages in a workflow, used to train machine learning classifiers, or analyzed by users, a simple folder/file hierarchy no longer suffices. Data lakes were developed to promote reuse of data (and in many cases the associated workflows) — but if users are unaware of what is available, or unable to trust what they find, they end up reinventing their own schemas, import processes, and cleaning processes. This not only leads to inefficiencies and redundant work, but also inconsistency in data processing, irregularity in data representation, and challenges in maintainability. Much as good software engineering promotes independent, modular, maintainable, and reusable software components, we need ways of promoting reusable units of data. As a building block, we need to help users find semantically related and similar datasets. Prior work on data lake management has focused on finding mappings to common schemas [14] and on profiling data to find joinable tables [7, 10, 7], but we believe a broader set of capabilities is what is needed.

We focus in this paper on tabular data that can be imported into an RDBMS, as with CSVs, certain structured files, and R and Pandas dataframes. We argue that during data science computations, users often want to search the data lake, not only by keyword, but using a table, to find other related (e.g., similar or linkable) tables. We develop a broad notion of query-by-table that captures several different classes of problems that data scientists always meet with: (need to be changed) find tables with similar schema and data (“similar” tables), find data produced through similar computational processes (“role-similar” tables), and find data linkable through join steps, possibly also involving record linking (“linkable” tables). Targeting collaborative data science settings, we formalize these problems and develop similarity measures and solutions.

We assume our search problem occurs over the inputs and outputs of data science workflows comprised of modules; the workflows may be captured in shell scripts, computational workflows [30, 13, 26], or, as is commonplace today, computational notebooks in tools such as Jupyter Notebook, Apache Zeppelin, or RStudio. We further assume that the output of each module is stored in the data lake, with a record of its provenance. (Provenance capture can be accomplished through operating system event logging [28], provenance APIs [39], or provenance-generating tools [1, 21, 34].)

As the user is working with data in the lake, e.g., cleaning data, linking data, conducting analyses, or training machine learning classifiers, our SYSTEM J1 platform helps them find

1 Name changed to preserve anonymity
additional resources. We briefly summarize several use cases for finding related tables.

**Example 1.1 (Augmenting training/validation data).** Often, data is captured in multiple sessions (perhaps by multiple users) using the same sensor device or tool for the same/different purposes or problems. Given a table from one such session, the user may wish to augment his or her data, to form a bigger training or validation set for a machine learning or data analytic algorithm.

**Example 1.2 (Linking data via ontologies).** Particularly in the life sciences, records in one database may have identifiers (e.g., “accession numbers”) linking to entries in another database or ontology. Such entries may transitively reference other entries. Each such entry brings in additional fields that may be useful to the user or to a machine learning algorithm. It can be helpful for users to know about such links that are implicit in the data.

**Example 1.3 (Augmenting Features).** Another common task especially in machine learning is to find additional/alternative features for the given data instances. In the collaborative setting, one data scientist may perform a specific feature engineering on a data set, while another may do it in a different way. It can be helpful for data scientists to be recommended with other feature engineering possibilities.

**Example 1.4 (Data Cleaning).** Given a widely used table, a data scientist may want to see examples of how the table is loaded or cleans, what analysis have been performed on it, and so on. In contrast to looking for alternative feature engineering, the tables derived by other data cleaning processes are generally identical in most entries, while different in a few columns/rows/entries.

**Example 1.5 (Data for Domain Adaption).** Usually, machine learning and data science workflows leverage a fixed dataset for training and testing. While a typical risk here is overfitting the dataset, to have a better understanding of the generality of the workflow (even cross-domain), the user may want to get a diverse set of compatible related datasets to validate their workflows.

In contrast to query-by-example [42, 41] or keyword-based table search [33, 4], in the above scenarios we are not necessarily looking for tables with specific keywords or rows — but rather based on a stronger notion of relatedness that may consider schema similarity, record-level overlap, description of data and workflows and similarity in the workflows that create or use specific tables. All of these tasks can be arbitrarily complex and challenging, so in this paper we choose a tractable subset of the space. We make the following contributions:

- Measures of table relatedness that consider both row and column overlap, and accommodate approximate matching.
- Top-k algorithms for finding related tables, and pruning heuristics to make this tractable at scale.
- Algorithms for efficient table storage, which also allow us to speed up subsequent searches.
- Experimental validation of our methods’ accuracy and scalability.

Section 2 defines the table search problem and proposes measures for table similarity and relatedness. Section 3 then develops methods for querying for similar tables. Building upon this, Section ?? shows how we can efficiently store many overlapping and derived tables in a way that both speeds up search and reduces overall space consumption. In Section 4 we describe SYSTEM J, which implements the query and storage schemes proposed in this paper. We then experimentally evaluate the SYSTEM J implementation in Section 5, before describing related work in Section 6 and concluding in Section 7.

## 2. RELATED TABLES

As data scientists are conducting their tasks — combining ad hoc and repeated workflows made up of multiple steps or modules — if they could easily find related tables rather than re-creating new ones, this would help foster many of the benefits we associate with good software engineering. Dataset reuse will ultimately make the data analysis process more regular, but it should also provide natural ways of leveraging common work on cleaning, curation, and standardization.

In this section, we formalize the problem, first by providing more detail on the types of workflows and environments we target, then by outlining the objectives of table search, and finally by defining a model of relatedness or utility $\text{sim}(T', T)$ between a given table $T'$ and our current table $T$.

### 2.1 Workflows and the Data Lake

While many of our capabilities will work simply by comparing static tables, we in fact target a data lake environment in which we can observe tasks, provenance, and updates across time. Our work with scientific collaborators (primarily in biomedical fields like genetics, radiology, and neuroscience) involves three different settings, which we describe next. Our work primarily targets the first of these settings.

We formalize a workflow graph $WF_i$ as a directed bipartite graph, in which certain data objects nodes $D_i = \{D_1, \ldots, D_m\}$ are input to computational modules $M_i = \{M_1, \ldots, M_n\}$, via labeled directed edges. The labels represent parameter names, input indices, etc. Finally each module in $M_i$ produces zero or more output additional data objects that belong to set $D_i$; in turn these can be input to other computational stages. In this paper, our focus is on data objects that can be represented in (possibly non-1NF) tables.

We next summarize where the workflows may originate.

**Computational Notebooks.** Our primary target environment for SYSTEM J is computational notebook software, such as Jupyter Notebook, JupyterLab, Apache Zeppelin, and RStudio. Computational notebook software captures scripts, workflows, and exploratory work by enabling interleaving of source code cells (effectively, code modules in a workflow) with their output (in the form of rendered visualizations or tabular output). Additionally, cells in the computational notebook may produce side effects in terms of state (variables whose values are set, and readable by the next cell to execute) or files (which may be read by a future cell execution that may even occur in another notebook).
Such environments typically store data in the local filesystem or on the cloud. Moreover, they do not fully preserve either the history of cell versions and outputs, nor the order in which cells were executed — which may result in non-reproducible notebooks. However, like several recent projects introducing reproducible notebooks [6, 24, 32], we have replaced the notebook software’s storage layer with the SYSTEM J data lake storage subsystem. Our storage layer provides a notebook storage and retrieval API to the computational notebook software, but internally it tracks versioning of the code cells, dependencies between cells that occur due to state sharing, and interactions between notebooks and the shell or terminal via files. Please see Figure 1 for an example.

SYSTEM J converts every notebook $N_i$ into a workflow graph $WF_i$, in which code cells become the computational modules in $M_i$. Input files read by the notebook, formatted text (MarkDown) cells, and outputs (files, rendered text, rendered images) become the data objects in $D_i$. Edges between data objects and computational modules are labeled with the names (and, optionally, subscripts) of the program variables associated with those objects.

Scientific Workflow Systems. In some more regularized settings, scientists make use of workflow systems [30, 26, 13] to define and execute their workflows, while also capturing provenance. Here, we typically differentiate between the workflow specification or definition and the workflow run or execution. Similarly to how we handle notebooks, for a workflow run $W_i$, we define a workflow graph $WF_i$ in which the inputs (files and parameters) and outputs become elements in $D_i$ and the code modules (in this case, often binary code instead of source code) become computational modules in $C_i$. Edges in the workflow graph are labeled with the names or indices defined in the workflow specification.

Scripts and Shell Commands. Finally, many users, particularly in genomics, define workflows using shell or Python scripts, which in turn run compiled open-source modules for different workflow stages. For such settings, we can use operating system event and I/O logging tools [29, 36, 29, 15] to track the programs being executed, their inputs, and their command lines — forming a “workflow run” where the “workflow definition” occurs in a script.

2.2 Searching Dimensionality

To enable improved reuse of data in data science, our goal is to provide search capabilities integrated into the computational notebook or data science environment. The user can highlight any input or intermediate table of output table, and ask to search for related tables — possibly restricting the search process with additional filter criteria such as author, attribute name or content, or the name of a computational process that was involved in the provenance of the search result.

In the following section, we introduce several building blocks which can support the required search capabilities.

2.2.1 Table Overlap

Our starting point is to try to define a measure of similarity or overlap between tables that are similar but not identical. Intuitively, we want to measure the overlap, in terms of rows, which mainly captures the overlapped instances; and in terms of columns, which mainly captures the overlapped schema. Both of them should be tolerant of small changes to both schema and values.

Building block for table overlap: Relation mappings. When computing similarity between a pair of tables $S, T$, a key aspect will be the notion of a relation mapping $\mu$ consisting of a key mapping and a schema mapping. Intuitively, the key mapping attributes $k_S, k_T$ capture the pairs of attributes whose values must be equal to establish a relationship between the rows, which are part of a candidate key; the schema mapping attribute sets $m_S, m_T$ capture the broader pairs of attributes that directly correspond (but are not necessarily part of a key, i.e., they are in a superkey). We more formally define the relation mapping below.

2.2.1.1 Overlap with Exact Matches.

For the problem of table overlap, we start with a strong assumption of exact matching, which we will ultimately relax. Given two tables $S, T$, we seek a mapping $\mu$ that relates tuples $s \in S, t \in T$.

**Definition 1 (Key Mapping).** If $S$ and $T$ overlap, we expect that there is at least one candidate key $k_S$ for relation $S$ and $k_T$ for relation $T$, each containing $n$ attributes,
such that if $\theta_{k_S,k_T} = (k_{S_1} = k_T_j) \land \ldots \land (k_{S_n} = k_T_i)$, then $k_S \rightarrow S \forall k_{S, k_T} T$ and $k_T \rightarrow S \forall k_{S, k_T} T$.

We say $\theta_{k_S,k_T}$ establishes a bijective key mapping $K$ between pairs $k_S$, and $k_T$, for $1 \leq i \leq n$.

Often, attributes that are not part of a candidate key are in fact mappable between relations $S$ and $T$, so the mapping key attributes do not fully define the relationship between pairs of tuples.

Definition 2 (Schema Mapping). We define a schema mapping $\Gamma_{m_S,m_T}$, where $|m_S \subseteq S| = |m_T \subseteq T| = k$, as a bijective mapping between pairs of attributes $s_j \in m_S$, $t_j \in m_T$, $1 \leq j \leq k$. Initially we assume that the domains of mapped attributes $s_j, t_j$ are the same, which we term direct schema mappings (we will revisit this in Section 2.2.3 to allow conversions between mapped values).

Now we can define the relation mapping, which will be a parameter to all of our similarity measures.

Definition 3 (Relation Mapping). A relation mapping between relations $S$ and $T$, $\mu(S,T)$, is a four-tuple $(m_S, m_T, k_S, k_T)$ such that $|m_S \subseteq S| = |m_T \subseteq T|$, $|k_S \subseteq m_S| = |k_T \subseteq m_T|$. $\Gamma_{m_S,m_T}$ is a schema mapping between $S, T$, and $k_S, k_T$ form a one-to-one key mapping $K$.

Definition 4 (Overlap with Relation Mapping). Given two tables $S, T$ and a relation mapping $\mu = (m_S, m_T, k_S, k_T)$ between the tables, we define two components: row similarity, $\text{sim}_{\mu}^{\text{row}}(S, T)$, and column similarity, $\text{sim}_{\mu}^{\text{col}}(S, T)$. We use Jaccard similarity for each component. First, we consider row overlap; given that $k_S \rightarrow m_S$ and $k_T \rightarrow m_T$:

$$\text{sim}_{\mu}^{\text{row}}(S, T) = \frac{|\pi_{k_S}(S) \cap \pi_{k_T}(T)|}{|\pi_{k_S}(S) \cup \pi_{k_T}(T)|}$$  \quad (1)

For column similarity, we consider the overlap between the schemas of $S$ and $T$, denoted by vectors $\bar{S}, \bar{T}$.

$$\text{sim}_{\mu}^{\text{col}}(S, T) = \frac{|m_S|}{|S| + |T| - |m_S|}$$  \quad (2)

2.2.2 Overlap with Approximate Matches.

In real world datasets that are not controlled by a DBMS, key constraints are occasionally violated due to errors, data consistency issues, and joint datasets that have not been fully de-duplicated. Thus, we relax our constraints above to incorporate approximate key constraints, and extend the similarity metric, in case only approximate key is available. Approximate functional dependencies, a building block towards approximate keys, have been heavily studied in the database literature, with a focus on practical algorithms for their discovery [17, 19, 9, 38, 5]. For our purposes of defining a similarity metric, the specific algorithm is an orthogonal question.

If $k_S \rightarrow S$ holds exactly, then all tuples with the same value in $k_S$ should have the same value in relation $S$. However, in the approximate setting some tuples may not satisfy $k_S \rightarrow S$. We can collect this portion of $S$ into a subset $S_n$, such that for each value of $s \in S_n[k_S]$, there exist multiple tuples in $S_n$.

Definition 5 (Approximate Key Constraints). Given a candidate approximate key, $k_S \subseteq S$, we define a factor $\gamma_{k_S}(S)$ to measure how well $k_S$ serves as a key of $S$. Adapting a metric proposed in Wang et al. [38], we measure the expected number of tuples in the table associated with each value of the approximate key, $\pi_{k_S}(S)$. Note that the factor is equal to 1 if an exact functional dependency holds. Formally, $\gamma_{k_S}(S)$ is defined as follows:

$$\sum_{v \in \pi_{k_S}(S)} \frac{|\sigma_{k_S=v}(S)|}{|\pi_{k_S}(S)|} = \frac{|S|}{|\pi_{k_S}(S)|}$$  \quad (3)

Thus, if $k_S \subseteq S$, and $\gamma_{k_S}(S) \approx 1$, then we say $k_S$ is an approximate key of $S$.

Definition 6 (Approximate Relation Mapping). We define an approximate relation mapping to be a relation mapping $\mu(S,T) = (k_S, k_T, m_S, m_T)$, with an approximation factor, $\gamma(S,T)$, based on how closely the mapped portions of the relations (approximately) satisfy $k_S \rightarrow m_S$ and $k_T \rightarrow m_T$. Formally, $\gamma(S,T)$ is:

$$\frac{|\pi_{m_S}(S)| + |\pi_{m_T}(T)|}{|\pi_{k_S}(S) \cup \pi_{k_T}(T)|}$$  \quad (4)

Now we must adapt the definition of $\text{sim}_{\mu}^{\text{row}}(S, T)$ and $\text{sim}_{\mu}^{\text{col}}$ in Definition 4 to compute the similarity while accounting for the approximate relation mapping. Let $\mu(S,T) = (k_S, k_T, m_S, m_T)$ and $\gamma(S,T)$ be the approximation factor as specified above. For the column similarity we would like to penalize the cases where mapped attributes in $m_S \cup m_T$ do not fully satisfy the key constraint:

$$\text{sim}_{\mu}^{\gamma}(S, T) = \frac{|k_S| + \frac{|m_S \setminus k_S|}{\gamma(S,T)}}{|S| + |T| - |m_S|}$$  \quad (5)

2.2.2 Table Containment.

Besides the table overlap, it is also useful to tell the user how many columns or rows of the table is in the overlap with the query table. We leverage the relation mapping defined in Definition 3 to propose our containment metrics.

Definition 7 (Containment with relation mapping). Given two tables $S, T$ and a relation mapping $\mu = (m_S, m_T, k_S, k_T)$ between the tables, we define two containment measures, row containment $\text{cnt}_{\mu}^{\text{row}}(S, T)$:

$$\text{cnt}_{\mu}^{\text{row}}(S, T) = \frac{|\pi_{k_S}(S) \cap \pi_{k_T}(T)|}{|\pi_{k_T}(T)|}$$  \quad (6)

and column containment $\text{cnt}_{\mu}^{\text{col}}(S, T)$:

$$\text{cnt}_{\mu}^{\text{col}}(S, T) = \frac{|m_T|}{|T|}$$  \quad (7)

Note that both $\text{cnt}_{\mu}^{\text{row}}(S, T)$ and $\text{cnt}_{\mu}^{\text{col}}(S, T)$ are asymmetric.

2.2.3 Table Linkability.

Rather than looking for tables that are overlapped, a data scientist often needs to find tables that augment or link to the current result. The data scientist’s task might be to find and join with a correspondence table that was produced by a
record linking tool [7, 23], subsequently allowing us to compute table similarity with tables using different domains for their keys (using the techniques of Section 2.2.1); to expand composite features into sub-features; to replace current feature representation with an alternative one; or to add other related data or metadata. Our notion of table similarity in Section 2.2.1 assumed that there was (approximately) a one-to-one mapping between rows, and that it was preferable the tables differ by only a few columns. In this section, we consider settings in which we expect one-to-many (or in rare cases many-to-many) links between tuples, where adding non-matching columns is in fact desirable.

As in the previous section, we assume the presence of a relation mapping \( \mu(S, T) = (m_S, m_T, k_S, k_T) \). Here, however, we expect that either \( k_S \to S \) or \( k_T \to T \); i.e., we can think of \( k_S \) and \( k_T \) as members of a key/foreign-key join.

In the first case, there will be a high row overlap between \( T \) and \( S \mid k_S \to k_T \), which means \( \text{sim}_{\text{row}}(T, S \mid k_S \to k_T) \) will be high. In the second case, the high overlap will exist between \( S \) and \( S \mid k_S \to k_T \), which means \( \text{sim}_{\text{row}}(S, S \mid k_S \to k_T) \) is high.

**Definition 8 (Table Linkability).** Given tables \( S, T \) and a relation mapping \( \mu(S, T) = (m_S, m_T, k_S, k_T) \), the linkability of \( S \) and \( T \) \( \text{lin}_\mu(S, T) \) is:

\[
\max(\text{sim}_{\text{row}}^\mu(S, S \mid k_S \to k_T), \text{sim}_{\text{row}}^\mu(T, S \mid k_S \to k_T))
\]

(8)

### 2.2.4 Description Similarity

The information of why and how a table is derived is also important when considering table similarity especially when there is minimal row overlap. In this paper, we study two types of information, one is the description (metadata) of the source data and the generating workflow, and we will introduce the other one, table provenance in the next subsection.

In this work, we assume the description of the source data and the generating workflow is stored in a key-value style.

Typically, the description can be the problem type, whose value can be classification, regression, clustering, the domain of the source data, such as health care, finance, insurance, a text introduction of the workflow and etc.

We denote the description space of table \( S \) as \( \Theta(S) \), and \( \Theta(S) = \{ (\theta_i, v_i, f_i) | 1 \leq i \leq N \} \), where \( \theta_i \) represents a specific type of description, \( v_i \) is the corresponding value of \( \theta_i \). Given another table \( T \) with its configuration space \( \Theta(T) \), \( f_i \) is a similarity function that is used to measure the similarity between \( v_i \) and \( v'_i \), where \( (\theta_i, v_i, f_i) \in \Theta(S) \) and \( (\theta_i, v'_i, f_i) \in \Theta(T) \).

Therefore, given table \( S, T \) and a description space, the description similarity \( \text{sim}_{\text{desc}}(S, T) \) is defined as follows:

\[
\sum_{i=1}^{N} w_i \cdot f_i(v_i, v'_i)
\]

(9)

where \( w_i \) is the weight of the similarity of a specific feature, such that \( \sum_{i=1}^{N} w_i = 1 \), and \( (\theta_i, v_i, f_i) \in \Theta(S) \) and \( (\theta_i, v'_i, f_i) \in \Theta(T) \).

### 2.2.5 Provenance Similarity

The previous similarity measures focus on similarity of table content or meta information of the generating workflow and the source data. However, another useful type of similarity involves similarity of a table’s purpose or role: outputs of the same or very similar workflow. Here we introduce a notion of provenance similarity based on similar workflows. To formally define the similarity, we will start with variable dependency graph which captures the dependencies among variables in the computational notebook, then extract a subgraph for a specific variable as its provenance graph, and define the similarity based on the edit distance between provenance graphs.

**Definition 9 (Variable Dependency Graph).** A variable dependency graph of a notebook is a directed acyclic graph with labels on edges denoted as \( G = (V, E, F) \). \( V \) represents the vertices consisting of all variables detected in the notebook. \( E(G) \) represents the labeled directed edges, and for any triple \((u, v, l) \in E(G)\), where \( u, v \in V(G) \), and \( l \in F(G) \), it means that variable \( u \) depends on variable \( v \) via operator \( l \). Note that, table dependency graph is generated by extracting the assignment relationships, variables and functions from the source code of the computational notebook. To extract all of the information, we parse the source code to an abstract syntax tree (ast).

Listing 1: Example 2.1

```python
import pandas as pd
import numpy as np

inFp='train.csv'
df_train=pd.read_csv(inFp)
var='SalePrice'
data=pd.concat([df_train['GrLivArea'], df_train[var]])
```

![Figure 2: Caption](image)

**Figure 2: Caption**

**Figure 3: Caption**

**Example 2.1.** We show an example of Listing 1 by Figure 2. The nodes represent the variables detected in the source code, including \( \text{inFp} \), \( \text{df_train} \), \( \text{var} \), and \( \text{data} \). The operators used here are functions such as \( \text{pd.read.csv} \) and pd.concat. The edges consist of all assignments in the source code. For example, data is outputted by running pd.concat on \( \text{df_train} \) and \( \text{var} \). Therefore, it is connected by two edges with the same label from \( \text{var} \), \( \text{df_train} \) respectively.

Therefore, a variable dependency graph depicts how a variable depends on and affects other variables. Then, we will introduce variable provenance graph for each variable.

**Definition 10 (Variable Provenance Graph).** Given a variable dependency graph \( G = (V, E, F) \), a variable provenance graph \( PG(v) \) where \( v \in V \) is a subgraph of \( G \), which describes all variables that affect \( v \) and their relationships. For example, Figure 2 is \( PG(\text{‘data’}) \), and Figure 3 is \( PG(\text{‘df_train’}). \)
**Definition 11 (Variable Provenance Similarity).** We define the variable provenance similarity between two variables via graph isomorphism. Given $G = (V, E, F)$ and $G' = (V', E', F')$, a graph isomorphism from $G$ to $G'$ is a bijective function $f: E \rightarrow E'$, s.t. $\forall (u, v, l) \in E, \exists (u', v', l') \in E'$ such that $u' = f(u), v' = f(v)$ and $l = l'$. Meanwhile, $\forall (u', v', l') \in E', \exists (u, v, l) \in E$ such that $u = f^{-1}(u'), v = f^{-1}(v')$ and $l = l$.

Then, we define provenance similarity between two variables $v_a$ and $v_b$, where $v_a \in V$ and $v_b \in V'$ based on edit distance between $PG(v_a)$ and $PG(v_b)$ denoted as $eds(PG(v_a), PG(v_b))$, which is the number of edit operations in the optimal alignments that make $PG(v_a)$ reach $PG(v_b)$. The edit operation on a graph $G$ is an insertion or deletion of a vertex/edge or relabeling of an edge. The costs of different edit operations are assumed to be equal in this paper.

Given $S, T$, we denote their provenance similarity as $sim_p(S, T)$ as:

$$sim_p(S, T) = edt(PG(S), PG(T))$$

2.2.6 Null Value Decrements

Null value is an important signal when looking for alternative data cleaning output. We define the measure as follows:

**Definition 12 (Null Value Decrement).** Given two tables $S, T$, the null value decrement $\Delta_0(S, T)$ is:

$$\Delta_0(S, T) = |Null(S)| - |Null(T)|$$

where $Null(S)$ and $Null(T)$ represent the number of null value entries in $S$ and $T$ respectively.

2.3 Searching the Lake

As alluded in the introduction, we seek to support several different classes of search for tables — each of which is useful for different data science tasks and user information needs. Based on the search building blocks, we summarize the data science tasks with its corresponding searching dimensionalities in Table 1.

**Table Relatedness.** At times, finding tables with similar schema and/or content and/or description and/or provenance can both help one data scientist learn from others, and help the data scientist accumulate additional and linkable data. In all of those cases listed in Table 1, the user can benefit greatly from seeing or using other examples.

**Definition 13 (Table Relatedness).** Given tables $S, T$ and a relation mapping $\mu(S, T) = (m_S, m_T, k_S, k_T)$, the table relatedness of $S$ and $T$, $Rel^\mu_{ij}(S, T)$ is:

$$\sum_i \omega_i \phi_i(S, T, \mu)$$

where $\Omega = \{w_i\}$, $\phi_i(S, T, \mu)$ represents a searching feature between $S$ and $T$.

Specifically, when searching for augmenting training/validation data, usually we will require the table returned with exactly the same schema, and a high description similarity, so that the table can fit the machine learning algorithm as well as relate to the task to be solved. To maximize the benefit, a table with a lower row containment will bring more new rows, which is more useful than returning overlapped instances (high row similarity) which is not helpful for further training or validating.

**Linking ontologies and augmenting features** require tables that are joinable via a key with the base table, but they may otherwise share few columns — i.e., low column containment or with different provenance —i.e., low provenance similarity.

**Alternative data cleaning** is looking for overlapped tables while different in a few columns and/or rows or entries, i.e., a high table overlap while generated by a different workflow (a low provenance similarity). Here, we also consider the change of the null value, which can reveal in some level if it is a process of data cleaning.

In terms of searching for data that current workflow can be transferred to, we will require a high provenance similarity, so that the table can be guaranteed to be tested in the current workflow, while it probably used to serve as a different purpose. It is important because in this way we can test the generality of the workflow. Similar to looking for additional training data, the row overlap is not useful, while we can tune the column and description similarity to control the relatedness between problems.

3. QUERYING FOR RELATED TABLES

Given the similarity metrics from the previous section, we now develop algorithms for finding related tables. To compute a table’s relatedness, we must identify the best relation mapping between the source table $S$ and the candidate table $T$ (Section 3.1.1) and compute necessary building blocks for table relatedness (Section 3.1.3). Once we have a set of similarity scores we must compute the top-$k$ results using effective approximation and pruning strategies to limit our search space (Section 3.2).

3.1 Computing Table Relatedness

In the previous section, we defined our measures for table relatedness under the assumption we were given a relation mapping $\mu(S, T)$. In this section, we describe how we find this mapping and then use it to compute our mapping related similarities. Since relation mapping $\mu(S, T) = (k_S, k_T, m_S, m_T)$, we first detect the schema mapping $\Gamma_{m_S,m_T}$, and then seek $\theta_{k_S,k_T}$ from them.

Besides relation mapping related similarities, computing description similarity and null value decrements is NP-hard [40]. Here, we propose an estimation of provenance similarity and leverage it into our searching framework.

3.1.1 Detecting Schema Mappings

To find the schema mapping $\Gamma_{m_S,m_T}$ between $S$ and $T$, we model it as an integer linear programming problem [22], much as in prior work [12]. Formally, we denote $x_{ij}$ as the binary variable indicating if $s_i \in S$ is matched to $t_j \in T$, according to some attribute similarity function that may take schema or data into account [35, 23]. That is:

$$x_{ij} = \begin{cases} 1, & \text{if } \Gamma_{m_S,m_T}(s_i) = t_j \\ 0, & \text{otherwise} \end{cases}$$
Here, we assume that each attribute in \( \bar{S} \) can be matched to at most one attribute in \( \bar{T} \) and vice versa. Thus, the objective is to find a mapping \( \Gamma_{m_S,m_T} \) satisfying the constraints that can maximize the matching score as follows:

\[
\arg\max_{\Gamma} \sum_{i,j} x_{ij} \text{MatchingScore}(i,j)
\]

s.t. \( x_{ij} \in \{0, 1\}, \forall i, \sum_j x_{ij} \leq 1, \forall j, \sum_i x_{ij} \leq 1 \) \hspace{1cm} (13)

By default, we use Jaccard similarity between pairs of columns as their matching score:

\[
\text{MatchingScore}(i,j) = \frac{\pi_s(i) \cap \pi_t(j)}{\pi_s(i) \cup \pi_t(j)} \hspace{1cm} (14)
\]

Finally, we use a greedy algorithm proposed by Papadimitriou [31] to get the best schema mapping.

### 3.1.2 Detecting Key Mappings

However, to measure table overlap, table containment or table linkability, we additionally need to determine how rows map — meaning we need to find a key mapping as a subset of the schema mapping.

For a given schema mapping, the choice of key mappings determines which rows are mapped together, and thus it affects the corresponding similarity scores, i.e., \( sim_{row}^{\mu}(S, T) \), \( sim_{col}^{\mu}(S, T) \), \( cnt_{row}^{\mu}(S, T) \), \( cnt_{col}^{\mu}(S, T) \), \( lin_{\mu}(S, T) \). Our goal is to find a key mapping \( K = (k_S, k_T) \) with equipijoin predicate \( \theta_{k_S, k_T} \) which maximizes the table overlap or the table linkability. Note that we use the same key detected for table overlap to compute table containment.

Since table overlap includes both row and column overlap, we denote the table overlap as:

\[
sim_{\mu}(S, T) = \beta sim_{row}^{\mu}(S, T) + (1 - \beta) sim_{col}^{\mu}(S, T) \hspace{1cm} (15)
\]

where parameter \( \beta \) allows us to adjust the weight on row and column terms.

Therefore, we are to maximize \( sim_{\mu}(S, T) \) (for some given parameter \( \beta \) and schema mapping \( \Gamma_{m_S,m_T} \)) to find the key mapping \( K \) between \( S \) and \( T \). Formally,

\[
sim_{\mu}(S, T) = \arg\max_{k_S,k_T} sim_{\mu}(S, T) \hspace{1cm} (16)
\]

In terms of looking for one-to-many links between rows to compute table linkability, similarly, we are to find \( K \) that can maximize \( lin_{\mu}(S, T) \).

Unfortunately, choosing the subset of a schema mapping as a key mapping to maximize the similarity is a variation of the classic, NP-hard subset-selection problem. Fortunately, the key mapping for two tables typically has a very small size, especially for linkable tables (key-foreign key joins seldom match on more than 2-3 keys). Therefore, we do not actually need to explore all subset combinations of attribute pairs from \( S \) and \( T \), but limit the size of \( \theta_{k_S, k_T} \) to be a small integer denoted as \( k_S \). Thus, the key mapping we are looking for is:

\[
K : \arg\max_{k_S,k_T} \sum_{k_S \mid |k_T| \leq k_S} sim_{\mu}(S, T) \hspace{1cm} (17)
\]

or

\[
K : \arg\max_{k_S,k_T} \sum_{k_S \mid |k_T| \leq k_S} lin_{\mu}(S, T) \hspace{1cm} (18)
\]

and these are optimization problems which are tractable. We combine the chosen \( \Gamma_{m_S,m_T} \) and \( K \) to form our relation mapping \( \mu \).

### 3.1.3 Estimating Provenance Similarity

In this section, we present a method that can efficiently estimate the provenance similarity between two tables based on their variable provenance graphs.

The basic idea of our estimation is to transform a graph structure to a multiset of star structures proposed in [40].

#### 3.1.3.1 Star Edit Distance

To estimate the provenance similarity between \( S \) and \( T \), we first derive the corresponding variable provenance graphs, which are \( PG(S) \) and \( PG(T) \) respectively. The key idea of estimating the graph edit distance between \( PG(S) \) and \( PG(T) \) is representing each graph to be a set of star structures whose graph edit distance can be easily obtained. Based on the edit distances between star structures of \( PG(S) \) and \( PG(T) \), we can estimate the lower and upper bound of their graph edit distance, a.k.a their provenance similarity. In the following part, we first introduce the star structure and then elaborate how to compute the edit distance between two star structures.

**Definition 14 (Star Structure).** A star structure \( s \) is a labeled single-level, rooted tree which can be represented by a 3-tuple \( s_r = (r, L, f) \), where \( r \) is the root vertex, \( L \) is the set of leaves. Note that edges only exist between \( r \) and any vertex in \( L \), and no edge exists among vertices in \( L \). \( f \) is the labeling function on edges.

Given \( PG(T) = \langle V, E, F \rangle \), we can represent it with a set of star structures, denoted as \( PG_s(T) = \{ s_r \mid r \in V \} \), where \( s_r = (r, L, f) \) and \( L \subset E, f \subset F \).
structures their star edit distance as $sdt(s_1, s_2)$. Formally,

$$sdt(s_1, s_2) = ||f_1| - |f_2|| + M(f_1, f_2)$$  \hspace{1cm} (19)$$

where $M(f_1, f_2) = \max\{|\Psi_{f_1}\|, |\Psi_{f_2}|\} - |\Psi_{f_1} \cap \Psi_{f_2}|$, and $\Psi_f$ represents the multiset of $f$.

### 3.1.3.3 Lower Bound of Edit Distance

Based on the star representation of the variable provenance graph, we introduce a mapping distance between two star representations, and we will leverage it to provide a lower bound of the graph edit distance between two variable provenance graphs.

**Definition 16 (Mapping Distance).** Given two star representations $PG_s(S)$ and $PG_s(T)$, assume that $\Upsilon$ is a bijective mapping between $s_i \in PG_s(S)$ and $s_j \in PG_s(T)$. The distance $\zeta$ between $PG_s(S)$ and $PG_s(T)$ is

$$\zeta(PG_s(S), PG_s(T)) = \min_{s_i \in PG_s(S)} \sum_{s_i \in PG_s(S)} sdt(s_i, \Upsilon(s_i))$$  \hspace{1cm} (20)$$

Detecting $\Upsilon$ that can minimize the mapping distance is also a combinatorial optimization problem, which is the same as detecting the schema mapping described in 3.1.1. Therefore, we use the same greedy solution to find $\Upsilon$, so that we can compute the mapping distance efficiently.

Given the mapping distance between $PG_s(S)$ and $PG_s(T)$, the provenance similarity between $S$ and $T$ satisfies:

$$\zeta(PG_s(S), PG_s(T)) \leq 4 \cdot edt(PG_s(S), PG_s(T))$$  \hspace{1cm} (21)$$

Proof. Let $P = (p_1, p_2, ..., p_l)$ be an alignment transformation from $PG_s(S)$ to $PG_s(T)$, such that $PG_s(S) = h_0 \rightarrow h_1 \rightarrow ... \rightarrow h_l = PG(T)$, where $h_{i-1} \rightarrow h_i$ indicates that $h_i$ is derived from $h_{i-1}$ by performing $p_i$. According to our provenance similarity defined in Definition 10, $p_i$ can be one of vertex/edge insertion, deletion and relabeling of an edge. Therefore, we can assume that $P$ consists of $z_1$ edge insertion/deletion, $z_2$ vertex insertion/deletion, and $z_3$ edge relabeling, and the graph edit distance between $PG(S)$ and $PG(T)$ is $z = z_1 + z_2 + z_3$. In terms of edge insertion/deletion, every time it happens, only two vertices are affected, and for each vertex, it will at most increase a cost of 2 on mapping distance between $PG_s(S)$ and $PG_s(S')$. Therefore, if there are $z_1$ edge insertion/deletion in the transformation from $PG(S)$ to $PG(T)$, the mapping distance increases at most 2$z_1$. In terms of vertex insertion/deletion, since there are only labels on edges, no additional cost will be generated to mapping distance when inserting or deleting a vertex. In terms of edge relabeling, it will also affect two vertices, therefore assuming $z_3$ edge relabeling, the increasing cost on mapping distance is at most $2z_3$.

Above all, we will get the following conclusion:

$$\zeta(PG_s(S), PG_s(T)) \leq 4 \cdot z_1 + 2 \cdot z_2 + 3 \cdot z_3$$

$$\leq 4z$$

$$\leq 4 \cdot \text{edt}(PG(S), PG(T))$$  \hspace{1cm} (22)$$

In the following computation, we will use the lower bound of $\text{edt}(PG(S), PG(T))$ to estimate the provenance similarity between two given tables $S$ and $T$.

### 3.2 Querying for Tables

Given a source table $S$ and the algorithms above for computing specific table similarity components, our objective is to find the top-$k$ tables to maximize the relatedness score requested by the user based on their specific need. Without loss of generality, we assume that our goal is to find the $k$ tables with highest table relatedness, i.e., $\text{Rel}_{\Omega}$. Formally, given table $S$ and the data lake $\Sigma = \{T_1, ..., T_n\}$, we seek top-$k$ tables $\Omega = \{T_i | T_i \in \Sigma, \forall T_j \notin \Omega, \text{Rel}_{\Omega}(S, T_i) > \text{Rel}_{\Omega}(S, T_j) \}$, $|\Omega| = k$.

To return the top-$k$ tables, a strawman solution is a form of exhaustive search: we take each table $T \in \Sigma$, find a relation mapping, then compute row, column similarity and the other components, respectively, and ultimately rank our matches to return the top-$k$ tables. Among all of the computations, the most expensive parts are 1) schema mapping and 2) key detection from the mapped schema. 3) estimating the provenance similarity.

Of course, the strawman solution is not efficient, because 1) it repeatedly computes schema mappings without reusing any mappings already detected; 2) it separates the computation between schema mapping and key mapping detection, and actually we do not need to compute schema mapping for two tables without a promising key mapping and vice versa. 3) based on the component similarity scores, we may be able to quickly prune some tables from consideration. Top-$k$ query processing is a well-studied problem [25, 18, 20], with Fagin’s Threshold Algorithm [8] forming a strong foundation. Building upon the insights of reusing schema mapping and the top-$k$ query processing literature, we use several key ideas to prune the search space.

### 3.2.1 Reusing and Indexing Schema Mapping

To return the top-$k$ related tables, we need to do schema mapping with each table $T \in \Sigma$ that is explored, which is a big cost if the number of tables in the repository is continuously increasing. Therefore, to improve efficiency, we need to reduce the time of doing schema mapping, which we do heuristically by a two-staged mapping detection strategy. Specially, we assume that in the same or related computational workflows, the user rarely either renames the same column of a table in different workflow steps or shares the name across different columns in different tables. Therefore, a detected schema mapping can usually be reused across different tables with common columns in the same or related computational workflows. Concretely, our two-staged mapping detection strategy consists of two steps, (1) offline indexing table schema via schema-based schema mapping, (2) online mapping detection via instance-based schema mapping.

**Indexing Tables via Workflow Graph.** We first introduce workflow graph constructed by the tables in the repository, then elaborate how we can use it to index attributes of tables.

**Definition 17 (Workflow Graph).** Given a multitude of computational notebooks, denoted as $NB = \{nb_1, ..., nb_l\}$, for each notebook $nb_i \in NB$, we denote its computational modules as $M(nb_i) = \{M_1, ..., M_{n_i}\}$. For any $M_i$, we denote its source code as $c(M_i)$. We define a workflow graph $WG = (V, E)$, $V \subseteq \cup_{i=1}^{n} M(nb_i)$ such that $\forall v_i, v_j \in V, c(v_i) \neq c(v_j)$, and $\exists v_k \in \cup_{i=1}^{n} M(nb_i), \forall v_k \in V, c(v_k) \neq
Given a repository of computational workflows, we first construct a workflow graph $WG$, then detect the connected components of $WG$ denoted as $C(WG) = \{C_M, \ldots, C_M\}$, and for each $C_M \in C(WG)$, it belongs to one of $C(WG)$.

Intuitively, the connected component usually reflects the collaborated workflows or a workflow with different versions, so that some computational modules are shared across different workflows. Given $C(WG)$, and for each source or intermediate table derived by the workflow steps in each $C_M \in C(WG)$, we do schema mapping via attribute names. Specially, we replace the matching score defined in Equation 3.1.1 as:

$$
MatchingScore(i, j) = \begin{cases} 
1, & \text{if } s_i = t_j \\
0, & \text{otherwise} 
\end{cases}
$$

which can be computed very efficiently. Then, for each $C_M$, we have a schema mapping that can map each attribute of each table derived by $CM$ to a unique id. Note that for each attribute, we uniformly sample a set of unique values from its corresponding columns with a fixed sample size, which will be used when doing online schema mapping. We denote this table as $V(CM)$ for each $CM \in C(WG)$.

**Mapping Detection.** Given table $S$, we use algorithm described in Section 3.1.1 to detect the schema mapping between $S$ and $V(CM)$. Then, based on the built index, we can construct the schema mapping between $S$ and each table $T$ by the mapping between $S$ and $V(CM)$ and the indexed mapping between $T$ and $V(CM)$.

### 3.2.2 Pruning

Through reusing detected schema mappings by indexing, we only need to do instance-level schema mapping $|C(WG)|$ times, when we are to query a new table. While it is still a huge cost as the number of tables is increasing. Furthermore, another huge cost when computing table relatedness is detecting the key mapping among mapped attributes.

To further improve the efficiency, we consider the idea that for each table $T \in \Sigma$, if we can have some clues telling us that $Rel^p_{\mu,\gamma}(S, T)$ will be very low even though we have not computed the relation mapping related similarity components, we then do not need to either perform schema mapping between $S$ and $T$ or detect the key mapping.

Therefore, we have two strategies. One is that we can use the similarity components that can be computed faster to prune the amount of the tables that we need to compute relation mapping related similarity components to avoid detecting schema or key mapping.

The other one is that we can use an estimation of the goodness of the key mapping to prune the tables that need to do schema mapping.

In the following parts, we will elaborate the two strategies one by one.

**Pruning Relation Mapping.** Intuitively, our idea is based on considering schema mapping and key detection together. Concretely, among our search components, both computing $\text{sim}_{\mu,\gamma}^{\nu,\rho}$ and computing $\text{lin}_{\mu}(S, T)$ requires schema mapping and approximate one-to-one or one-to-many links between rows. Let’s first consider computing $\text{sim}_{\mu,\gamma}^{\nu,\rho}$.

Our basic idea is that computing $\text{sim}_{\mu,\gamma}^{\nu,\rho}$ only requires a schema mapping between keys rather than the full schema mapping between two tables. Therefore, our strategy is we can detect schema mapping only between pairs of columns that are very likely to be the one-to-one mapping key, and leave the computation of the full schema mapping until we have to compute the full similarity score.

Formally, given the query table $S$, and for each table $T \in \Delta$, we detect the top-$q$ candidate keys from $S$ and $T$ respectively, which means we compute the $q$ attributes denoted as $Q(S) = \{s_i\}$, such that $s_i \in S$, $|Q(S)| = q$, and $\forall s_i \in Q(S), s_i \in S \setminus Q(S), \gamma(s, \gamma(s)) \geq \gamma(s, \gamma(s))$, and similarly, we define $Q(T)$ for $T$.

Then, we limit the schema mapping between the attributes from $Q(S)$ and $Q(T)$, which can accelerate the time of doing schema mapping. Since a good relation mapping has to be constructed by good keys coming from two tables simultaneously, such pruning will not affect the quality of the results.

We denote the detected mapping as $\Gamma^{m^q}_{m^q,m^q}$.

**Similarity Components.** To find an efficient solution to the top-$k$ processing problem, we observe that simply applying the threshold algorithm is insufficient, as computing table relatedness is itself expensive. Based on the mapping reusing strategy and the pruning of relation mapping, we have some partial results of some of the search components. Now we will put all those pieces together and propose an efficient algorithm that can compute the top-$k$ related tables efficiently. The basic idea is that we seek a way of pruning the most expensive parts of the computation, in particular the problem of detecting the full schema mappings and the key mappings.

Given tables $S, T \in \Delta$ and the schema mapping $\Gamma^{m^q}_{m^q,m^q}$, between $Q(S)$ and $Q(T)$, we can get a set of candidate relation mapping $Q(\mu(S,T)) = \{(m^q, m^q, k_S, k_T)m^q \subseteq Q(S), m^q \subseteq Q(T), k_S \in m^q, k_T \in m^q\}$.

Therefore, we can get following upper bounds:

$$
\text{sim}_{\mu,\nu}^{\rho,\sigma}(S, T) \leq \max_{\mu \in Q(\mu(S,T))} \text{sim}_{\mu,\nu}^{\rho,\sigma}(S, T) \tag{24}
$$

Note that we can obtain this upper bound, when we compute the schema mapping $\Gamma^{m^q}_{m^q,m^q}$.

Considering the one-to-many mapping, we denote $Q(\mu(S,T)) = \{(m^q, m^q, k_S, k_T)m^q \subseteq S, m^q \subseteq T, k_S \in m^q, k_T \in m^q\}$, we can get a similar upper bound for table linkability.

$$
\text{lin}_{\mu}(S, T) \leq \max_{\mu \in Q(\mu(S,T))} \text{lin}_{\mu}(S, T) \tag{25}
$$

As for column similarity, we can get:

$$
\text{sim}_{\mu,\nu}^{\rho,\sigma}(S, T) \leq \frac{|k_S|}{\max(|S|, |T|)} + \frac{1}{\gamma(\mu)} \left(\min(|S|, |T|) - |k_S|\right)
$$

$$
\leq \frac{\min(|S|, |T|)}{\max(|S|, |T|)} \tag{26}
$$
Similarly, we can also get the upper bounds for containment based similarity components as follows:

\[
\text{cnt}_{\text{col}}^\alpha(S, T) \leq \max_{\mu \in Q} \text{cnt}_{\text{col}}^\alpha(S, T) \tag{27}
\]

\[
\text{cnt}_{\text{col}}^\alpha(S, T) \leq \frac{\min(|S|, |T|)}{|T|} \tag{28}
\]

Besides the upper bounds above, we can efficiently compute \(\Delta_i\) and denote its upper bound as \(\omega_i\), and compute efficiently as \(\tilde{\omega}_i\), and denote its upper bound as \(\chi_i\), and compute efficiently as \(\tilde{\chi}_i\), and denote its upper bound as \(U_i(S, T, \mu)\).

\[
\Omega_i = \sum_i \omega_i \phi_i(S, T, \mu)
\]

\[
= \sum_i \omega_i \phi_i(S, T) + \sum_i \omega_j \phi_j(S, T, \mu) \tag{29}
\]

\[
= \sum_i \omega_i \phi_i(S, T) + \sum_j \omega_j U(\phi_j(S, T, \mu))
\]

\[
= U_i(S, T)
\]

Therefore, the basic idea of our pruning technique is to avoid computing a full schema mapping and key mapping for each table \(T\) in \(\Sigma\), unless it is clearly needed. Specifically, we rank each table in \(\Sigma\) by computing terms (a) \(\omega_i \phi_i(S, T, \mu)\) and computing an upper bound on the corresponding terms (b) \(\omega_j \phi_j(S, T, \mu)\). For many tables, terms (a) are low, thus we do not need to compute terms (b), that require full schema mapping and key mapping detection.

Now our top-k search problem becomes one with objective function \(U_i(S, T)\), with the attributes, i.e., \(\{\omega_i \phi_i(S, T)\}\) and \(\{\omega_j U(\phi_j(S, T, \mu))\}\). We observe that all of them can be computed directly without the full schema mapping and key detection, which is very fast and thus possible to pre-compute and sort.

In contrast, computing \(\phi_i(S, T, \mu)\) is slow. Therefore, we only compute those numbers on demand. From the perspective of the Threshold Algorithm [8], we therefore want to do sequential access over the values of \(\{\omega_i \phi_i(S, T)\}\) and \(\{\omega_j U(\phi_j(S, T, \mu))\}\) and random access (on-demand computation) of \(\{\omega_i \phi_i(S, T, \mu)\}\). We term our approach the staged threshold algorithm.

Let \(\Sigma^*\) denote the ranked list of tables in \(\Sigma\) according to \(\{\omega_i \phi_i(S, T, \mu)\}\). In sorted order, for each table \(T\) in \(\Sigma^*\), we detect the key \(\theta_{k, k, \Gamma}\) from the schema mapping \(\Gamma_{m, n}\), as described in Section 3.1, then compute other mapping related search components. If \(T\) is one of the top-k tables so far, we remember it as a candidate for the top-k.

Let us use \(T_i^\alpha\) to denote the \(i^{th}\) table in component \(i\) which we do sorted access. We then define a threshold value \(\tau_p = \sum_i \omega_i \phi_i(S, T_i^\alpha, \mu) + \sum_j \omega_j U(\phi_j(S, T_i^\alpha, \mu))\). When at least \(k\) tables have been seen whose score is at least \(\tau_p\), we may terminate the search, otherwise we set the threshold value to \(\tau_{p+1}\) and compute the similarity for the next table in \(\Sigma^*\). Pseudocode for the algorithm is shown in Algorithm 1.

**Approximation.** If most of \(\omega_i \phi_i(S, T, \mu)\) have similar values, our staged threshold algorithm will be limited in its ability to prune. Here, we can trade off efficiency and precision, by leveraging an approximate variant of our algorithm based on the approximate version of Fagin’s Threshold Algorithm [8]. Here, we use an early stop strategy by introducing a new parameter \(\alpha\), with some value \(\alpha > 1\) specifying the required level of approximation. The algorithm will then stop when the \(k^{th}\) biggest similarity of the tables remembered is higher than \(\frac{1}{\alpha} \cdot \tau_p\), such that a table \(T'\) not in the top-\(k\) set satisfies the condition that \(Rel_{\phi_i}(S, T') \leq \alpha \cdot Rel_{\phi_i}(S, T)\) for every other table \(T\) inside the top-\(k\) set. This approximation relaxes the threshold test of our algorithm, making it possible to return approximate answers, which is much faster than our vanilla staged threshold algorithm.

**Algorithm 1 Top-k Search for Related Tables**

**Input:** Query Table \(S\), Table Set \(\Sigma\), \(k\)

**Output:** Top-k Tables

1: for all \(T \in \Sigma\) do
2: \hspace{1em} for all \(i\) do
3: \hspace{2em} compute \(\omega_i \phi_i(S, T)\)
4: \hspace{1em} end for
5: \hspace{1em} for all \(j\) do
6: \hspace{2em} compute \(\omega_j U(\phi_j(S, T, \mu))\)
7: \hspace{1em} end for
8: end for
9: \(\Sigma^* \leftarrow \) sorted access in parallel for each \(\omega_i \phi_i(S, T)\)
10: \(\text{result} = [\], \text{R} = [\]
11: for \(p = 1\) to \(|T|\) do
12: \hspace{1em} \(\text{threshold} = \tau_p\)
13: \hspace{1em} for all \(T \in T_p[\Sigma^*]\) do
14: \hspace{2em} \(\theta_{k, k, \Gamma} \leftarrow\) detect the key from \(\Gamma_{m, n}\).
15: \hspace{2em} \(\Gamma_{m, n} \leftarrow\) do schema mapping between \(S\) and \(T\).
16: \hspace{2em} \(\text{R.append}((T, Rel_{\phi_i}(S, T)))\)
17: \hspace{1em} end for
18: \hspace{1em} sort \(R\) by \(Rel_{\phi_i}(S, T)\)
19: \hspace{2em} \(lv \leftarrow\) the least or the \(k^{th}\) value in \(R\)
20: \hspace{2em} if \(lv \geq \text{threshold}\) then
21: \hspace{3em} break
22: \hspace{1em} end if
23: end for
24: \(\text{result} \leftarrow\) top-\(k\) tables in \(R\)
25: return \(\text{result}\)

**4. SYSTEM IMPLEMENTATION**

In this section, we outline how the components described in Section 2, 3 and 7 fit together within System J, shown in Figure 4. We implement the system with Python 2 and Postgres SQL database with around 5K line of code.

**System J** links into Jupyter Notebook’s ContentsManager interface to store data and interfaces with PostgresSQL as its storage backend. As shown in the figure, it provides two main services. The first one is table creation, which happens transparently whenever a Jupyter cell is executed. As data products are generated by the cell and exposed as dataframes, our table storage optimizer will be invoked. As described in Section 7, the optimizer makes a cost-based decision about how to store the output dataframe(s) in PostgresSQL using a cost model.

The second stage of System J, which hooks into Jupyter’s client-side user interface, is searching for related tables. The user selects a cell output table as a query, then specifies what type of table to look for; and our table search component retrieves the top-\(k\) related tables, according to the
similarity, linkability, and role-similarity measures described in Section 2. As described in Section 3, we use the staged threshold algorithm to do top-k search, and take the advantage of the table registry and caching to speed up the search process.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate System J on real data science workflows with their source datasets, and compare it with several alternatives. We consider the following questions: (1) What is the execution-time speedup provided by our staged threshold-based algorithm? (2) How do the exact and approximate versions of the top-k algorithm perform, in terms of running time? (3) How is the quality of the tables returned by our system, compared with those returned by alternatives such as keyword-based table search? (4) What storage gains can we achieve when we are using our storage backend?

5.1 Experimental Setting

We provide an overview of our experimental setup, including how we obtained our data and workload.

5.1.1 Data Sets and Workloads

Workflows (Computational Notebooks). In our experiments, we evaluate our system on real data science workflows we found from Kaggle 3. Specifically, we downloaded a set of Jupyter Notebooks with their source data, as created by users addressing 5 different public tasks. We divide these into three different categories: Machine Learning (ML), typically including workflow steps such as feature selection, feature transformation (construction) and cross-validation; Exploratory Data Analysis (EDA), typically including data cleaning, univariable study, multivariable study, hypothesis testing, etc.; Combined which includes a pipeline doing both exploratory data analysis and testing some machine learning models. We describe our 5 tasks in Table 2, and note our workflows cover many different styles of data analysis.

To run our experiments, we downloaded 3 highly rated notebooks for each task to our workflow repository. For each source dataset, we separated it into two disjoint subsets. We then modified each of the source notebooks to use one or the other subset — thus obtaining a total of 30 different notebooks, conforming to 15 unique workflows.

Table 2: Workflows used in our experiment.

<table>
<thead>
<tr>
<th>Task</th>
<th>Category</th>
<th># of Input Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicting Sales Price 3</td>
<td>EDA</td>
<td>1</td>
</tr>
<tr>
<td>Classifying Patients 4</td>
<td>ML</td>
<td>2</td>
</tr>
<tr>
<td>Insurance Forecasting 5</td>
<td>Combined</td>
<td>1</td>
</tr>
<tr>
<td>Predicting Survival on Titanic 6</td>
<td>Combined</td>
<td>2</td>
</tr>
<tr>
<td>Studying Mental Health on Tech Survey 7</td>
<td>Combined</td>
<td>1</td>
</tr>
</tbody>
</table>

Tables. We ran all of the notebooks in our repository, and stored all variables output by each cell, if their type belonged to one of Pandas DataFrame, NumPy Array or List, as tables within our System J PostgreSQL-based storage backend. Overall, there are 1226 tables captured and stored.

Queries. We generated a search tables workload for finding similar and linkable tables as follows. For each input dataset from our 5 tasks (Table 2), we sample 6 different tables with 50 rows and 5 columns as query tables. Overall, we have 42 different queries for similar and linkable tables.

To search for role-similar tables, we randomly sample 42 outputs from the workflow runs.

5.1.2 Comparisons and Metrics

Searching for Related Tables. We evaluate both efficiency and quality of System J, when searching for top-k related tables.

1. Efficiency. To evaluate the efficiency of our top-k searching algorithms, we compare System J with brute-force search (BF), our staged threshold based algorithm (STA), and three α− approximation of STA where α = 1.1 (STA-1.1), α = 1.3 (STA-1.3) and α = 1.5 (STA-1.5). Each algorithm is evaluated twice, with and without leveraging our caching system as a storage backend speedup. We will add -S as a suffix after the notation of the algorithm if using our storage backend speedup. We evaluate efficiency by its running time.

2. Top-k approximation. Leveraging α approximation in our staged threshold algorithm or leveraging our storage backend speedup are approximate, and heuristic, respectively, algorithms for top-k search. Thus, we compare the results returned by the approximate and heuristic algorithms, versus those returned by the standard top-k search algorithm, which computes optimal results according to the similarity score. Here we compute the mean MAP (Mean Average Precision) of the top-20 results of each query as the metric.

3. Quality. We also evaluate the quality of the tables, by comparing the results returned by System J as well as the tables returned by a keyword based table search algorithm, denoted as DS, following the matching strategies of [33, 4]. DS is an algorithm implemented by transforming the table to a vector, and searching related tables by computing similarity between vectors. Here, we transform the table to

---

3https://www.kaggle.com
a vector by computing tf-idf score for each text entry in the table, and compute the similarity by cosine distance.

To evaluate the quality, when searching for similar tables, we will compute (a) precision of the tables returned, by evaluating if the instances of the table are all from the input data, (b) recall, by counting how many instances of the input data have been covered so far, when $k$ tables are returned.

Since each query is generated synthetically from the input data, we can compute the precision and recall by adding an additional index, which is invisible to our search and storage layers. For linkable tables, since it is not well-defined how many tables can be joined, we only report the precision by looking at whether there is a plausible primary key-foreign key join between the tables. For role similar tables, since it is an exact search problem, we only report on efficiency.

Storing tables. We evaluate our storage backend for System J by comparing storage without any compression (N-S), our storage mechanism leveraging storing union and join (UJ-S), and our storage mechanism leveraging union, join and intensional storage (LUJ-S). We compute the storage gain and average query cost for each table to show the benefit of our system.

5.2 Performance of Searching Tables

5.2.1 Analysis of Searching Efficiency.

In Figure 5, we show the running time comparison of different algorithms when searching for top-$k$ related tables. In all of our three tasks, we can observe from the figures that both our storage backend speedup and the approximate staged threshold algorithm significantly reduce the running time. This is due to a reduction in the times spent on schema mapping, as well as in shortening the top-$k$ search process.

Specifically, the staged threshold algorithm can help reduce the time of searching for top-$k$ related tables, which can achieve $1.1\times 1.2\times$ speedup for similar tables, $1.3\times 1.4$ for linkable tables and $1.6$ for role similar tables. We can further achieve gains by leveraging $\alpha$ approximation. For example, a 1.1 approximation can lead to a $8.9\times$ speedup for top-1 similar tables, and $5.5\times$ speedup for top-1 linkable tables, and the greatest speedups are more than $30\times$, when we are searching for top-1 similar and linkable tables with 1.5 approximation. Note that there is only a marginal improvement when using these approximations and searching for role-similar tables. This is because when computing role-similar tables, we are not looking at the row similarity but only the column similarity. We still need to compute the similarity of the workflow many times to adjust the top-$k$ threshold.

Besides the search process, the time used in detecting schema mappings is consistent across the different algorithms and tasks. Our storage backed speedup, which uses caching and limited exploration, can achieve about $10\times$ speedup by reusing the schema mapping detected when the tables were stored. In this way, we can also quickly prune the tables that are not similar in schema, which can further speed up the search process. Putting those pieces together, from the figures we observe that we only need 1.5 – 2 seconds to get the top-1 related tables using both 1.5 approximation and our storage backend speedup; around 2 seconds to return the top-20 role-similar tables; and around 10 seconds to return the top-20 similar and linkable tables. This is much faster than the brute force search for top-$k$ related tables.

5.2.2 Analysis of Approximation Quality

Neither $\alpha$ approximation nor the storage backend speedup are guaranteed to produce the exact top-$k$ results. Therefore, we need to analyze how different are the results returned by the standard staged threshold algorithm, and those returned by approximate algorithms.

To measure the difference, for each query, we compute the mean average precision (MAP) of the top-20 results returned by all of our approximate algorithms, by treating the results returned by the vanilla staged threshold algorithm as the ground truth. To have a better understanding of what are the results returned by the approximate algorithms, we vary the size $k$ of the results returned by the vanilla algorithm to observe the change of the mean MAP of the approximate algorithms.

The results are shown in Figure 6. We observe that as $k$ is larger, the mean MAPs of all of our approximate algorithms become higher, and for searching similar tables, most of the approximate algorithms’ mean MAPs are very close to 1 when $k = 30$. This means that the top-20 results returned by our approximate algorithms mostly overlap with the actual top-30 results of the optimal algorithms. For linkable tables and role similar tables, we get similar results for 1.1 and 1.3 approximation of STA; here, leveraging the storage backend speedup adds more variability compared with other approximate algorithms, as we have a bigger speedup.

<table>
<thead>
<tr>
<th>$@k$</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.6667</td>
<td>0.7053</td>
</tr>
<tr>
<td>10</td>
<td>0.6381</td>
<td>0.6263</td>
</tr>
<tr>
<td>15</td>
<td>0.5460</td>
<td>0.5614</td>
</tr>
<tr>
<td>20</td>
<td>0.5095</td>
<td>0.5789</td>
</tr>
</tbody>
</table>

5.2.3 Analysis of Search Result Quality

Table 3 reports the quality of similar tables returned by our system and alternatives, given the gold standard established by synthetically creating the search tables. Here, as in [33, 4], we use keyword indexing on the tables in the data lake, and we define a keyword query based on the textual contents of the search table. The first three columns report the precision and the last three report the recall of top-$k$ similar tables. We observe that the results returned by our system, whether we use storage backend speedup or not, are consistently better than keyword-based table search, on both precision and recall. We get the same conclusion from the task of searching for linkable tables in Figure 4.
reason is clear: some text entries, for example ‘M/F’ in the column of gender, ‘yes/no’ in the column of a binary feature, ‘south/north/east/west’ in the column of region are usually not useful in matching either similar or linkable tables due to their popularity, and there is no way to measure dependencies among columns in this way. Therefore, the results demonstrate that keyword based table search is not enough, when we are to search related tables with high utility from the data lake, especially in data analytic settings.

6. RELATED WORK

Data management for computational notebooks is an emerging area, with Aurum 8 focused on indexing and searching tables via keywords, and multiple efforts [6, 24, 32] addressing issues of versioning and provenance tracking. System J builds over similar infrastructure but focuses on effective search and efficient storage over it. A different body of work focuses specifically on storing versioned (and branched) tables, including DataHub, Orpheus, and Decibel [3, 16, 27]. Some underlying techniques (such as separating the differences from the shared parts of the data) are shared with System J, but our work focuses on storing unrelated tables efficiently.

The problems of managing, linking, and querying structured and unstructured data in situ have been of interest to the database community for some time. The overall vision was articulated in the work on dataspaces [11], although even today, many aspects of that vision remain open challenges for the community. More recent tools such as the Q System [37] and the system built by Belhajjame et al [2] attempt to combine automatic link discovery and user feedback to integrate data in something resembling a data lake.

Data Civilizer [7] maintains profiles on database tables to attempt to discover links, and Constance [14] seeks to further exploit semantic mappings to mediated schemas in order to enable query reformulation. Our efforts are complementary to that work, focusing on exploiting our knowledge of tables’ provenance, defining different query classes and notions of table relevance, and supporting top-k query answering. Finally, keyword search over tables [33, 4].

7. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the problems of searching for related tables: tables that are similar, that can be linked, and that serve similar roles because they were computed using a similar workflow. We developed similarity measures, novel strategies for top-k and approximate top-k search, and more efficient storage. We evaluated using real Jupyter notebooks as workflows, and showed good search result quality and efficient performance.

As future work, we hope to extend our work to handle nested data, and other data types that are not strictly tabular.

8http://mitdbg.github.io/aurum-datadiscovery/
8. REFERENCES


