Gen-T: Table Reclamation in Data Lakes

Grace Fan
Northeastern University
United States
fan.gr@northeastern.edu

Roee Shraga
Northeastern University
United States
r.shraga@northeastern.edu

Renée J. Miller
Northeastern University
United States
miller@northeastern.edu

ABSTRACT

We introduce the problem of Table Reclamation. Given a Source Table and a large table repository, we aim to find a set of tables that when integrated reproduce the source table as close a possible. Unlike query discovery problems like Query-by-Example or Target, Table Reclamation focuses on reclaiming the Source Table as fully as possible. To do this, we define three measures of table similarity to measure how close our reclaimed table is to the Source Table – two are inspired by the integration literature and a new measure is designed for the reclamation problem. Our search covers not only Select-Project-Join queries, but integration queries with unions, outerjoins, and unary queries with subsumption and complementation operators that have been shown to be important in data integration and fusion. Using reclamation, a data scientist can understand if any tables in a repository can be used to exactly reclaim a tuple in the Source. If not, one can understand if this is due to errors or inconsistencies in null values. We present a solution for Table Reclamation named Gen-T. Gen-T performs table discovery to retrieve a set of candidates tables from the table repository, filters these down to a set of *originating tables*, then integrates these tables to reclaim the Source as close as possible. We show that our solution is efficient and scalable in the size of the table repository with experiments on real data lakes containing up to 15K tables, where the average number of tuples varies from small (web tables) to extremely large (open data tables) up to 1M tuples.

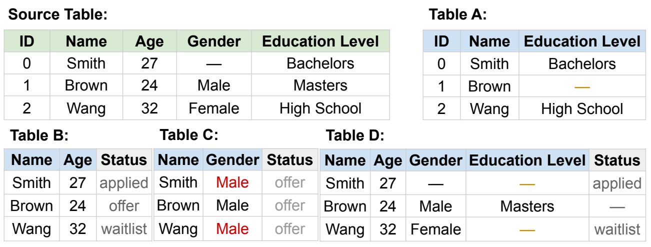
PVLDB Reference Format:
Grace Fan, Roee Shraga, and Renée J. Miller. Gen-T: Table Reclamation in Data Lakes. PVLDB, 14(1): 50 - 60, 2021.
doi:10.14778/3421424.3421431

1 INTRODUCTION

As more tables in data lakes become openly available, users have easier access to more government, academic, and enterprise datasets. In particular, data scientists retrieve these tables to perform integration tasks, run analyses, or input into machine learning models. However, there are concerns around the origins of the tables that are directly retrieved and used for further tasks [1]. The trustworthiness of the data, when used to train downstream machine learning tasks for example, depends heavily on the trustworthiness of the data origins. In addition, common real-world (tabular) datasets are prone to fairness issues (bias and discrimination) and may be used unknowingly (and indirectly) by data scientists [37].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:10.14778/3421424.3421431

This work introduces the problem of **Table Reclamation**: given a *Source Table* and a data lake (a large set of tables), can we find a set of tables (called *originating tables*) that can be combined to reproduce the source table exactly or as close as possible? Using table reclamation, a data scientist can use the trustworthiness (quality) of a set of originating tables to better understand the source table.



Source Table:

ID	Name	Age	Gender	Education Level
0	Smith	27	—	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	High School

Table A:

ID	Name	Education Level
0	Smith	Bachelors
1	Brown	—
2	Wang	High School

Table B:

Name	Age	Status
Smith	27	applied
Brown	24	offer
Wang	32	waitlist

Table C:

Name	Gender	Status
Smith	Male	offer
Brown	Male	offer
Wang	Male	offer

Table D:

Name	Age	Gender	Education Level	Status
Smith	27	—	—	applied
Brown	24	Male	Masters	—
Wang	32	Female	—	waitlist

Figure 1: The Source Table in green contains information about applicants, including their unique ID, Name, Age, Gender, and Education Level. Tables A, B, C, D are possible tables from which the Source Table's instances originated. All common columns with the Source Table are in blue, some of which contain missing (yellow '—') or inconsistent (in red) values to the Source Table's values.

Example 1. *Suppose the user is using the green table in the top left of the Figure 1 as training data. As this table contains instances for applicants, with sensitive values for age, gender, and education level, it is crucial to verify the values and instances. To do so, we trace its values to a subset of tables in the data lake – we find tables A, B, C, and D as possible originating tables. However, we see that Table C contains contradicting non-null values in the "Gender" column compared to the values in the same tuples of the "Gender" column in the Source Table. Thus, to correctly verify every cell value in the Source Table, we will look for other tables in a discovered set of candidate tables that may more faithfully reclaim these tuples.*

Table reclamation is related to the common Query-By-Example (QBE) or Query-By-Target (QBT) that discover a query over input tables that produces an instance-equivalent table to the given example output table [4, 18, 35, 55, 62, 65]. In order to generalize to a data lake setting, we do not assume a complete and correct set of input tables. Rather we use an additional step of finding candidate tables within or across data lakes that may contribute to the Source Table (i.e., that may be originating tables). Also, existing QBE/QBT systems focus primarily on discovering (Select)-Project-Join queries over (largely complete) relational tables [8, 31, 50, 61, 67], with some using both the data values and the schema of the tables. Due to the noise and heterogeneity of data lake tables, these queries may not be sufficient to fully integrate data lake tables to produce

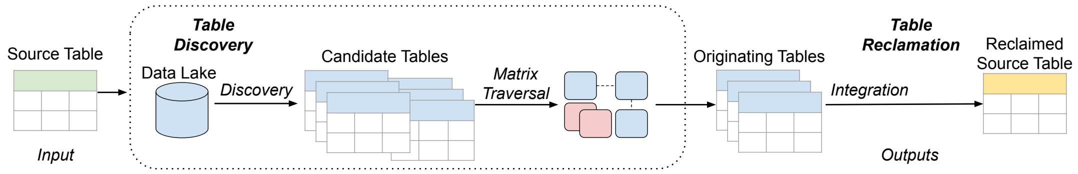
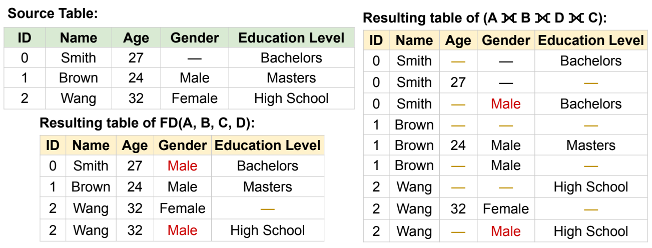


Figure 2: Gen-T Architecture. Given a Source Table, Gen-T finds its originating tables in the Table Discovery phase, produces the reclaimed Source Table from the Table Reclamation phase, and returns the originating tables and the reclaimed Source Table.

the given Source table. So, we aim to recover Select-Project-Join-Union queries using only the data values, since the metadata of data lake tables may be missing or inconsistent [2, 23, 48, 49]. We also consider operations that have proven to be important in data integration and data fusion, namely subsumption and complementation [5]. Both operations have been used to integrate incomplete data. In addition, Rajaraman and Ullman describe the role of subsumption in computing outerjoins over complete data in a way that "preserves all possible connections among facts" [52].

Reclamation focuses on the Source Table and does not assume we know the originating tables. Hence, it can be used to verify tuples. If certain tuples cannot be reclaimed, a data scientist would know these are not derivable from her data lake. Reclamation can also be used to verify values. For example, if the reclaimed table contains nulls where the Source Table has a value, the data scientist can look to see if value imputation might account for the values in her table and can examine whether the value imputation is appropriate for her task. Similarly, if the reclaimed table contains different values from the Source Table, a scientist can investigate whether the source values may be errors or alternatively if they may be valid corrections to errors in the originating tables. Unlike QBE and QBT, the focus is on the data and understanding what data in a Source Table can be verified, not on discovering a reverse-engineered query.

We present a data-driven technique for Table Reclamation in data lakes, named Gen-T, in which we aim to discover a set of originating data lake tables whose integration can reclaim (or reproduce) a Source Table given by the user. Our contributions are the following.
• We define the novel problem of *Table Reclamation* – finding a set of originating tables that can be used to reproduce a Source Table.
• We present a set of properties that a metric for determining how close a reclaimed table is to a Source Table should possess. And based on this we evaluate our methods using tuple-based precision and recall which consider only how many tuples are exactly reclaimed, instance divergence (inspired by instance similarity used in the data exchange literature [3]), and a new metric that uses conditional KL-divergence [17].
• We present a Table Reclamation solution named Gen-T. Gen-T performs table discovery to retrieve a set of candidates tables from the data lake, filters out poor candidates using novel table representations that simulate table integration without performing expensive integrations, and integrates them to produce a table whose values are as close as possible to the Source Table.
• We conduct extensive experiments, showing that Gen-T outperforms all baseline methods when reclaiming Source Tables, reclaiming 5X more Source Tables than the best-performing baseline.
• We show that our solution is efficient and scalable to the size of the data lake with experiments on real data lakes containing up to 15K tables, where the average table size (number of tuples) varies from small (web tables) to extremely large (open data tables) with on average over 1M tuples.


Source Table:

ID	Name	Age	Gender	Education Level
0	Smith	27	—	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	High School

Resulting table of FD(A, B, C, D):

ID	Name	Age	Gender	Education Level
0	Smith	27	Male	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	—
2	Wang	32	Male	High School

Resulting table of (A ⟗ B ⟗ D ⟗ C):

ID	Name	Age	Gender	Education Level
0	Smith	—	—	Bachelors
0	Smith	27	—	—
0	Smith	—	Male	Bachelors
1	Brown	—	—	—
1	Brown	24	Male	Masters
1	Brown	—	Male	—
2	Wang	—	—	High School
2	Wang	32	Female	—
2	Wang	—	Male	High School

Figure 3: Source Table about applicants' data, and possible integrations of tables (with only shared columns with the Source Table) resulting from state-of-the-art integration method using Full Disjunction (FD) and an outer join ⟗.

2 OVERVIEW

A data scientist provides a *Source Table* that she would like to verify by understanding if it can be produced by integrating any combination of tables within a data lake. Specifically, we aim to determine a set of tables from whom the Source Table's values may originate (termed *originating tables*), and use them to reclaim (regenerate) the Source Table. Given our data lake setting where tables can be changed autonomously, we formulate the problem as an approximate search of finding a set of tables that can best be used to reclaim the Source, *as close as possible*.

Unlike prior approaches to *query-by-example* [4, 8, 18, 31, 50, 55, 61, 62, 67] or *by-target* [65] problems, we do not assume that we know the exact set of input tables whose values first formed the Source Table or even **if** the Source Table can be reclaimed. To solve the problem of table reclamation, we thus use a two-step solution. First, we discover tables from the data lake that share values with the Source Table and therefore may have created portions of it , we call these *candidate tables*. Then, we search for ways of combining subsets of these tables to regenerate the Source Table.

Figure 2 shows the general pipeline of Gen-T, given a Source Table and outputting the Reclaimed Source Table and its originating tables. In the Table Discovery phase (Section 5), Gen-T discovers a set of candidate tables whose values may have contributed to the creation of the Source Table. Then, we apply our novel solution of representing tables as matrices in order to simulate table integration via matrix traversal (Section 5.2). The goal of this step is to refine the set of candidate tables to a set of originating tables, and essentially filter out any tables from the set of candidate tables that are not needed in an efficient manner before performing table integration.

Once Matrix Traversal pinpoints a set of originating tables, we integrate these originating tables in the Table Reclamation phase (Section 4) and produce a reclaimed Source Table.

Example 2. *We now return to our running example to illustrate the necessity of pinpointing good originating tables. In Figure 3, the candidates are directly integrated after projecting out columns that do not appear in the Source Table. The bottom left, yellow table is the integration result from the state-of-the-art full disjunction (FD) method [26, 33] and the right table shows the result using one possible outerjoin order that may be learned by Auto-Pipeline (a by-target approach) [65]. The resulting tables contain different values in the Gender column (in red) with respect to the corresponding value in the Source Table. These values originate from Table C. When possible, we need to refine the set of candidate tables to filter out tables like Table C that produce integrated tables with erroneous values that do not make the Source Table.*

The remainder of the paper is outlined as follows: we present related work in Section 3. Going into the solution pipeline of Gen-T, we first assume that we have been given an accurate set of originating tables that we need to integrate, and discuss the Table Reclamation phase in Section 4. Next, Section 5 describes the Table Discovery phase, specifically the Matrix Traversal solution that finds a set of originating table. Finally, the experiments in Section 6 show the effectiveness, scalability, and generalizability of Gen-T.

3 RELATED WORK

We now discuss related work to Table Discovery and Integration. We then discuss work related to finding the origins of a table, referred to as By-Example or By-Target approaches in the literature.

Table Discovery: Table Discovery has a rich literature, specifically keyword search over tables, unionable table search, and joinable table search. Early work such as Octopus [10] along with Google Dataset Search [9], support keyword search over the metadata of tables [2, 43] and smaller scale web-tables [58, 59]. To support data-driven table discovery, systems [24, 49, 54, 70, 71] were then developed to find schema complements, entity complements, joinable tables, and unionable tables.

For joinable table search, early systems use of schema matching or syntactic similarities between tables' metadata, such as Jaccard similarity [40, 64]. LSH Ensemble [71] makes use of approximate set containment between column values and supports set-containment search using LSH indexing. JOSIE [70] uses exact set containment to retrieve joinable tables that can be equi-joined with a column in the user's table. MATE [20] supports multi-attribute join with a user's table. These systems can be used to retrieve a set of candidate tables that have high set similarity with a given user's table.

Table union search was first supported by systems that leverage schema similarity to retrieve unionable table [44, 54]. Using data (rather than metadata), a formal problem statement was first defined by Nargesian et al. [49] who presented a data-driven solution that leverages syntactic, semantic, and natural language measures. This problem was refined by SANTOS [32] to consider relationship semantics in addition to column semantics when retrieving semantically unionable tables. Most recently, Starmie [22] offers a scalable solution to finding unionable tables that leverages the entire table context to encode its semantics. Although our method also retrieves relevant tables to a user's table, we aim to retrieve tables for a specific task – reclaiming the user's table. Finally, other recent work [25] presents a goal-oriented discovery for specific downstream tasks, aiming to augment additional columns. We tailor table discovery towards the goal of reclaiming the Source Table.

Table Integration: Lehmberg et al. [38] stitches unionable tables together, but does not support join augmentation of tables. More recently, ALITE [33] performs full disjunction (FD) [26] to maximally combine tuples from a set of tables (intuitively full disjunction is a commutative and associative form of full outer join). Our goal is to reproduce the given Source Table, which may contain incomplete tuples, so we do not aim to maximally combine tuples if it produces a table that is not identical to the Source Table. Nonetheless, ALITE is a candidate baseline for Gen-T, as it offers a state-of-the-art integration solution.

Preceding the table integration process, there are pre-integration tasks to find alignments between table elements. First, instance-based schema matching determines how the schemas of two tables align to prepare for integration [12, 19, 36, 46, 51, 56]. Our solution aligns schemas by rename the columns in the retrieved tables with the column of the source table that best matches. Entity matching [11, 15, 27, 29, 41, 42, 47, 66, 69] is another common pre-integration task, aiming to align tuples for cleaning or joining tables. In our solution, we assume the Source Table has a key, and thus align tuples by matching using equality on the key.

Finding Origins of Tables: Our problem setting of tracing a Source Table's values back to its origins can be related to Data Provenance [13, 14], which given a query and its output table, explains from where the (values or) tuples originate, why and how they were produced. However, in our problem setting, we need to recover the tables and the integration required to reproduce the the Source Table and thus do not know the query that was originally used to create the Source Table.

Query-By-Example is a popular approach in which systems are given a pair of matching input and output tables, and they need to synthesize a query or transformation from the input to the output. Specifically, systems such as SQL-by-example [62], synthesize a SQL query to produce an output table, given the input table, that contains all equivalent instances of the example output table. Some systems only consider Project and Join operators [28, 31, 50, 67], whereas others also consider the Select operator [8, 61]. For example, Ver [28] aims to find Project-Join views over large tables in which the join path is not known. Some methods output a set of queries rather than one query that could reproduce the example output table, given the input table [18, 55]. AutoPandas [4] performs transformation-by-example by synthesizing Pandas programs rather than SQL queries.

Auto-Pipeline [65] defines a similar problem, Query-By-Target, with the goal of synthesizing the pipeline used to create the target table, given the target table and a set of input tables. Using the synthesized pipeline on the input tables, it then produces a table that "schematically" aligns with the input target table. As the state-of-the-art in this line of work, it is a baseline for our approach. In both By-Example and By-Target paradigms, the set of input tables on which the system synthesizes a query to generate the example or target table is known. And this set of input tables is known to contain the tuples and columns needed such that their integration can reproduce the output table. However, in our problem, we do not assume this is the case. We are only given the Source Table and a data lake as input. From the data lake, we need to search for and filter a set of candidate tables such that by integrating the filtered set of candidate tables (which we call the originating tables), we can reclaim the Source Table as close as possible.

4 TABLE RECLAMATION

We now describe the Table Reclamation step of Gen-T's pipeline. We first assume that we already have the set of tables from which the Source Table originated (originating tables). In this section, we combine them to reclaim the Source Table. First, we define a set of Table Operators in Section 4.1, with which we perform table integration in Section 4.2 to produce a table that reclaims the Source Table. Given the possible reclaimed table, we discuss how to evaluate the quality of the Source Table reclamation in Section 4.3.

4.1 Table Operators

We now discuss the table operators we use for Table Integration. Consider the following table operators performed on one table.

• Projection(π): Project on specified columns of the table.

• Selection(σ): Select tuples that satisfy a specified condition.

• Complementation(κ) [5, 6]: Given tuples t_1, t_2 in the same table, t_1 complements t_2 if they share at least one non-null column value, and t_1 contains some non-null values where t_2 has nulls while t_2 contains some non-null values where t_1 has nulls. The tuples must agree on all non-null values. Applying κ on t_1 and t_2 produces a single tuple that contains all non-null values of either (both) tuples and is null only if both t_1 and t_2 are null. Applying κ on a table produces a table with no complementing tuples and involves repeatedly applying complementation to pairs of tuples.

• Subsumption(β) [5, 26]: Given tuples t_1, t_2 in the same table, t_1 subsumes t_2 if they share some non-null column value(s) and t_1 contains some non-null values where t_2 has nulls. Applying β on a table involves repeatedly applying subsumption and discarding the subsumed tuples (t_2).

Next, we consider the following pairwise-table operators and their known derivatives that may be used in SPJU queries. For these operators, we assume the schemas of the tables have been aligned, so joinable (unionable) columns share the same name and we use the natural version of these operators [53]. These operators are applied to two tables T and S.

• Inner Union($\cup$): Union two tables that share the same schema. This operator is commutative and associative.

• Outer Union($\uplus$) [16]: Union two tables, even if their schemas are not equal. The resulting table contains the union of the columns from both tables. If a column C is missing from one table (T) but appears in the other table (S), then in the result, the tuples of S contain a null ($\perp$) in their C column. This operator is commutative and associative.

• Inner Join($\bowtie$): Two tuples t and s are joinable if they share the same values on all common columns (columns with the same name). The schema of the result of the inner join contains the union of the columns and the table contains all joinable tuples. If T and S contain no common columns, then the inner join is the *Cross Product* (meaning it contains (t, s) for all tuples $t \in T$ and $s \in S$). This operator is commutative and associative.

• Left Join($⟕$): The left join contains the inner join and in addition, for each tuple t in T that does not join with any tuple in S, the left join contains a tuple t' that is equal to t on all columns in T and is null on all columns in $S - T$.

• Outer Join ($⟗$): The outer join contains the left join and in addition, for each tuple s in S that does not join with any tuple in T, the outer join contains a tuple s' that is equal to s on all columns in S and is null on all columns in $T - S$.

Given our set of table operators, we apply them to a set of originating tables to reclaim the Source Table. We first align the columns of the originating tables with the source table. We can apply any schema matching algorithm for this, but given our goal of reclamation, we use the value overlap to do the alignment. Specifically, we rename each column of an originating table with the name of the source column with which it shares the most values (breaking ties arbitrarily) and remove any columns from the originating tables that do not share values with the source table (as these columns are not useful in reclamation).

To make our reclamation search more efficient, we will make use of the following result. Outer Union and the set of unary operators above can be used to represent any SPJU query. Using this result our search will focus on outer union and the unary operators.

THEOREM 3 (REPRESENTATIVE OPERATORS). *Given two tables that contain no duplicate tuples, and no tuples can be subsumed or complemented, for all SPJU queries, there exists an equivalent query consisting of only the Outer Union and the four unary operators (select, project, complementation, and subsumption).*

Proof Sketch. Let T_1, T_2 be two tables that share common columns C. For each pairwise table operator, Inner Union, Inner Join, Left Join, Outer Join, Cross Product, there exists an equivalent query consisting of Outer Union and/or unary operators. For example, Inner Join can be represented using Outer Union ($\uplus$) with selection (σ), subsumption (β), complementation (κ) operators (Equation 8), and Full Outer Join can be represented using Inner Join, Outer Union, and subsumption operators [26] (Equation 10).

$$T_1 \bowtie T_2 = \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2))) \tag{1}$$

$$T_1 ⟗ T_2 = \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2) \tag{2}$$

Due to space constraints, the full proof for each pairwise table operator in SPJU are given in the technical report [21]. ☐

Using our set of operators $\bigoplus = \{\uplus, \sigma, \pi, \kappa, \beta\}$, we will present an efficient way of exploring the integration space to reclaim the Source Table. Instead of traversing through all possible join predicates (columns that two tables join on) in a space consisting of all tables, as in existing by-example and by-target work, we can simply apply Outer Union and the unary operators with conditions based on the Source Table.

4.2 Table Integration

Table Integration takes a set of originating tables ($\mathcal{T}$), a Source Table (S) and using $\bigoplus$ outputs a table (T_{result}) that reclaims S as best as possible. Note that from the previous discovery phase, each table in $\mathcal{T}$ adopts their column names from S for every column that shares values with a column in S. Our table integration method is depicted in Algorithm 1. First, we perform pre-processing by projecting out columns not in S, and selecting tuples whose values are in a primary key or foreign key column in S. Hence, we only keep the columns and tuples that overlap with S (Line 3). Next, we take the Inner Union of tables that share the same schema (Line 4) to reduce the space of tables we need to explore. To prevent the over-combining of tuples that share nulls with tuples in S, for each table $T \in \mathcal{T}$, we find tuples from S that share the same key values and contain nulls. If T also contains nulls in tuples with the same key value in the same columns, then we replace these nulls with a dummy non-null value (Line 5). Before applying our set of integration operators $\bigoplus$, we remove duplicate tuples, subsumed tuples (apply β), and take the resulting tuples of complementation (apply κ) in Line 6. In the integration (Lines 8-13), we explore our set of operators $\bigoplus = \{⊎, \sigma, \pi, \kappa, \beta\}$ As proven in Theorem 3, this is sufficient rather than considering all SPJU operators and conditions. Thus, we first apply Outer Union(⊎). Next, we check if applying Complementation(κ) and Subsumption(β) on the Outer Union result leads to over-combining of tuples (e.g., removing a subsumed tuple that is identical to a tuple in S) and decreases the number of values and tuples shared with S, called similarity with S (we define similarity formally later). After iterating through all tables from the input set, we revert the previous labeling of shared nulls with S (Line 14). If the resulting integration table has fewer columns than S, we pad it with columns from S that are not currently in the resulting table, all containing null values (Line 16). This way, the resulting table shares the same schema with S. Finally, we return the resulting integration as a possible reclaimed table.

Algorithm 1: Table Integration

```
1  Input: $\mathcal{T} = \{T_1, T_2, \ldots T_n\}$: tables to integrate; $S$: the Source Table
2  Output: $T_{\text{result}}$: integration result
3  $\mathcal{T} \leftarrow$ ProjectSelect($\mathcal{T}, S$) //$\sigma, \pi$ ($T \in \mathcal{T}$) on columns, keys in $S$
4  $\mathcal{T}_{\cup} \leftarrow$ InnerUnion($\mathcal{T}$) //Inner Union tables with shared schemas
5  $\mathcal{T}_{\cup} \leftarrow$ LabelSourceNulls($\mathcal{T}_{\cup}$) //Label Nulls shared with Source Table
6  $\mathcal{T}_{\cup} \leftarrow$ TakeMinimalForm($\mathcal{T}_{\cup}$) //Apply $\beta, \kappa$ on each table
7  $T_{⊎} \leftarrow \emptyset$
8  for $T_i \in \mathcal{T}_{\cup}$ do
9      $T_{⊎} \leftarrow T_{⊎} ⊎ T_i$ //Apply outer union ⊎
10     if $S$ similarity does not decrease then
11         $T_{⊎} \leftarrow \kappa(T_{⊎})$                //Apply complementation $\kappa$
12     if $S$ similarity does not decrease then
13         $T_{⊎} \leftarrow \beta(T_{⊎})$                //Apply subsumption $\beta$
14 $T_{\text{result}} \leftarrow$ RemoveLabeledNulls($T_{⊎}$)
15 if $T_{\text{result}}$ has fewer columns than $S$ then
16     add null columns in $T_{\text{result}}$ for each column $\in S \setminus T_{\text{result}}$
17 Output $T_{\text{result}}$
```

4.3 Aligning Tuples

Given a possible reclaimed table, we compare it with the Source Table to see how close they are – this is what we will call *similarity*. Consider a Source Table S with columns C and a possible reclaimed table $\hat{S}$ with the same schema. We say that tables S and $\hat{S}$ *align* if $\hat{S}$ has at least one *aligned tuple*. An aligned tuple $t_{\text{Align}} \in \hat{S}$ is defined with respect to some tuple $t_s \in S$ such that for shared key columns C_k, $t_{\text{Align}}[C_k] = t_s[C_k]$. In other words, an aligned tuple is a tuple in the reclaimed table that shares a key value with some tuple in the Source Table. If there are multiple tuples from the reclaimed table that share a key value with a Source Table's tuple, we select the aligned tuple with the largest number of shared column values with the Source Table's tuple (breaking ties arbitrarily). We now distinguish among three types of aligned tuples, namely a correct tuple, an erroneous tuple, and a nullified tuple. A *correct tuple* $t_{\hat{S}}$ is an aligned tuple that has all of the column values from the Source Table tuple t_s, i.e., $\forall C \in \hat{C} : t_{\hat{S}}[C] = t_s[C]$. An *erroneous tuple* $t_{\hat{S}}$ is an aligned tuple that has at least one non-null *erroneous value* (different column value than the tuple in the Source Table t_s), i.e., $\exists C \in \hat{C} : t_{\hat{S}}[C] \neq t_s[C] \wedge t_{\hat{S}}[C] \neq \bot$. Note that an erroneous value can be a non-null value where the Source Table has a null value. Finally, a *nullified tuple* $t_{\hat{S}}$ is a tuple with a *nullified value* (a null value $\bot$ in a column where the tuple from the Source Table t_s has a non-null value), i.e., $\exists C \in \hat{C} : t_{\hat{S}}[C] \neq t_s[C] \wedge t_{\hat{S}}[C] = \bot$.

Given a possible reclaimed table $\hat{S}$ that shares the same schema with Source Table S, and its aligned tuples to S, we now find how many of the values in S have been reclaimed by the aligned tuples in $\hat{S}$. To do so, we define the Value Similarity Score:

Definition 4 (Value Similarity Score). *Given a table $\hat{S}$ and a Source Table S, the Value Similarity Score (VSS_S) is a similarity score $\in [0, 1]$ with respect to S that captures how similar the values in $\hat{S}$ are to values in S.*

We would like the Value Similarity Score to have the following properties, with respect to Source Table S: *(1) $VSS_S(S) = 1$, (2) $VSS_S(\emptyset) = 0$, (3) Given table $\hat{S}$ with no aligned tuples, $VSS_S(\hat{S}) = 0$, and (4) Given table $\hat{S}_1$ containing m erroneous tuples, each with n erroneous values, and $\hat{S}_2$ with m nullified tuples, each with n nullified values, $VSS_S(\hat{S}_1) < VSS_S(\hat{S}_2)$.*

The best VSS_S of 1 results from equal tables such that all tuples are aligned and share all column values with S, whereas VSS_S results in 0 if there are no aligned tuples between two tables (no key value is shared), or if an empty table is compared to a non-empty table. These properties are similar to those of the Tuple Similarity score from MapMerge [3], which measures the ratio of the number of shared column values between an aligned tuple from $\hat{S}$ and its corresponding tuple in S, vs. the total number of columns in S. Thus, the Tuple Similarity score is 1 if an aligned tuple shares all column values with its corresponding tuple in S, and 0 if no column values are shared.

Consider the fourth property, in which Source Table S is compared to table $\hat{S}_1$ with m erroneous tuples, each with n erroneous values, and also to table $\hat{S}_2$ with the same number of nullified tuples, each also with n nullified values. VSS_S is higher for $(\hat{S}_2, S)$ than for $(\hat{S}_1, S)$. This is desirable due to the fact that a table with erroneous tuples can introduce noise in later downstream tasks, whereas a table with nullified tuples can potentially be integrated with other tables to discover non-null values for the current null values. This is similar to the behavior of Kleene's three-valued logic [7] – 'true' if the degree of truth is 1, 'false' if the degree of truth is 0, and 'unknown' if the degree of truth is between 0 and 1. The maximum truth value of ('unknown', 'false') is 'unknown' since the result can be either true or false. In our case, for table $\hat{S}_2$, which contains nullified (unknown) values, and table $\hat{S}_1$ that contains erroneous (certainly false) values, we also rank the table with nullified tuples higher than the table with erroneous tuples. Consider the following example illustrating how we measure VSS_S:

Source Table:

ID	Name	Age	Gender	Education Level
0	Smith	27	—	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	High School

$\hat{S}_1$

ID	Name	Age	Gender	Education Level
0	Smith	27	Male	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	—

$\hat{S}_2$

ID	Name	Age	Gender	Education Level
0	Smith	—	—	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	—

Figure 4: Aligned tuples between a Source Table (left green table) and two possible reclaimed tables (right yellow tables) from Figure 3, aligned based on key column 'ID'.

Example 5. *Consider Source Table S, and two possible reclaimed table, $\hat{S}_1$ on the bottom left and $\hat{S}_2$ on the right, shown in Figure 3. We align tuples from $\hat{S}_1$ and $\hat{S}_2$ with S on key column 'ID'. If multiple tuples in $\hat{S}_1$ or in $\hat{S}_2$ share an ID value with a tuple in S, we choose the aligned tuple with the largest number of common column values. Thus, we consider the tuples in $\hat{S}_1$ and $\hat{S}_2$ shown in Figure 4 when finding VSS_S. The only difference between the alignments is the tuples with ID=0, where $\hat{S}_1$'s tuple is erroneous ("Male" in red), and $\hat{S}_2$'s tuple is nullified (yellow "—"). Thus, $\text{VSS}_S(\hat{S}_2) > \text{VSS}_S(\hat{S}_1)$.*

Recall that we evaluate Source Table S's similarity (Algorithm 1 Lines 10, 12) before and after applying Complementation(κ) and Subsumption(β) to see if the resulting table contains fewer values and tuples shared with S. With a Value Similarity Score, we can evaluate the resulting table with respect to S by measuring how similar the values in the resulting table are to S's values.

Using a Value Similarity Score as a similarity measure to compare the possible reclaimed table $\hat{S}$ and Source Table S, we aim to solve the following problem:

Definition 6 (Source Table Reclamation). *Given a collection of tables $\mathcal{T}$ and a Source Table S generated from a set of tables $\mathcal{T}^* \subseteq \mathcal{T}$, find a set of originating tables $\hat{\mathcal{T}} \subseteq \mathcal{T}$ such that its integration produces $\hat{S}$ with the maximum Value Similarity Score to S.*

5 TABLE DISCOVERY

Now that we know how to integrate a set of originating tables and evaluate a possible reclamation result, let's describe how we discover a good set of originating tables. This is the Table Discovery phase of Table Reclamation. First, we discover a set of candidate tables from the input data lake in Section 5.1. Then, we discuss the novel methodology that refines this set of tables to the set of originating tables in Sections 5.2 and 5.3.

5.1 Candidate Table Retrieval

Discovering a set of originating tables from a data lake requires discovering tables that share some of the same values as the Source Table in an efficient manner. In the context of data lakes, where metadata is inconsistent or missing, searching using schema names is unreliable [2, 23, 48, 68]. Thus, we use any existing data-driven, approach to table discovery that is scalable in a data lake setting.

With a set of tables returned as relevant to the Source Table, we need to verify the set similarities of their values with the Source Table, especially if they were discovered primarily using table semantics. To do so, we retrieve candidate tables among the previously discovered tables using a set similarity algorithm. This could be done efficiently with a system like JOSIE [70] that computes exact set containment or MATE [20] that supports multi-attribute joins. In addition to finding candidate tables containing columns that have high set similarity with a Source Table, we also *diversify* the set of candidate tables such that each candidate table has minimal overlap with its previous candidate (full algorithm is included in the technical report [21]). This is especially important since data lakes tend to have multiple versions of the same tables [34, 57]. For example, one study (JOSIE [70]) reports that there is a large percentage of duplicate column sets in real data lakes, specifically 98% of column sets in open data and 83% in web tables are duplicates. Duplicates, or near-duplicate tables, are typically ranked adjacent to each other in set similarity results. Thus, if a table is already ranked in the result, we decrease the scores of adjacently ranked similar tables, aiming to rank duplicates and near-duplicates lower. By diversifying the candidates, each candidate for a column in a Source Table S can overlap with different values in S's column. We illustrate this with an example:

Example 7. *Suppose we have the Source Table from Figure 1. In addition to data lake tables A, B, C, D, we also have Table E, an exact duplicate of Table D. If we only rank these tables using set overlap with the Source Table, we would return Tables D and (its duplicate) Table E as top candidates, since all of their columns have high set overlap with those in the Source Table. However, Table E does not add any new information when integrated with Table D. Thus, diversifying the set of candidate tables decreases Table E's score, pushing other tables such as Table A higher in the ranking.*

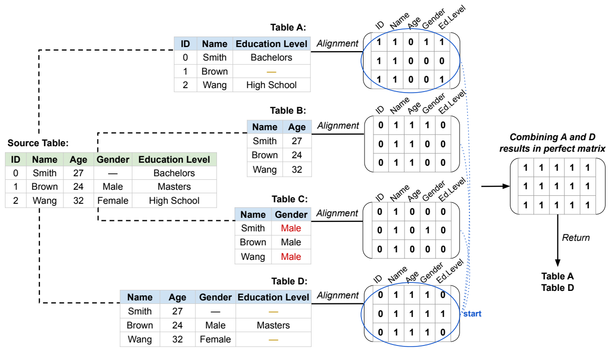
With a diverse set of candidates found for each column in a Source Table S, we ensure that each candidate table still has high set overlap with the Source Table across all related columns. To do so, we find all tuples in a candidate table that contains column values from S. We verify that for each column that has high set overlap with a column in S, it still has high set overlap within these tuples. We then rename each candidate table's column that has high overlap with a column in S to S's column name, thus implicitly performing schema matching between a candidate table and S. Finally, we check if any candidate table can be subsumed by other candidate tables, specifically if their columns and column values are contained in other tables. If so, we remove the subsumed tables from the returned set of candidate tables.

5.2 Matrix Traversal

With the set of candidate tables, we could potentially enter the table integration phase (Section 4.2) using all candidate tables as an originating set of tables and evaluate the Value Similarity Score on the resulting table from integration. However, it may be computationally expensive to use all candidate tables to perform table integration. In order to minimize the integration cost, we need to refine the set of candidate tables to only include a set of originating tables containing the maximum set of aligned tuples with respect to the Source Table. To do so, we emulate the table integration process without performing the expensive computations and see what candidate tables are necessary to reclaim our Source Table. By simulating the alignment of candidate tables' tuples with each other, we can uncover contradicting and erroneous aligned tuples with respect to the Source Table, and discard tables that could decrease the Value Similarity Score.

First, we need to align tuples in candidate tables to tuples in a Source Table. Recall that we assume all tables, including the Source Table, have primary and/or foreign keys. If a candidate table does not have any of the Source Table's keys, we first join it with a copy of another candidate table with which it has a foreign key relationship, and that has a key from the Source Table. This way, all tables after joining via foreign keys have a shared key column with the Source Table and can align its tuples with the Source Table using key values. Note that this pre-processing step is only to emulate table integration and refine a set of candidate tables, and is not needed to perform actual table integration.

To capture tuple alignment, we represent each candidate table and its aligned tuples with the Source Table in the form of a matrix, as illustrated in Figure 5. For each table, we encode its aligned tuples with respect to the tuples in the Source Table, for columns that the table share with the Source Table. To encode aligned tuples, we initialize the matrices to have the same dimensions as the Source Table S, containing the same number of rows and columns as S, such that the matrix indices represent the Source Table's indices. For each key value and its associated column values in Source Table, check if the value appears in the candidate table at the corresponding column and key value. If so, then the matrix has 1 at the same index as the value's index in Source Table, and 0 otherwise.



Source Table:

ID	Name	Age	Gender	Education Level
0	Smith	27	—	Bachelors
1	Brown	24	Male	Masters
2	Wang	32	Female	High School

Table A:

ID	Name	Education Level
0	Smith	Bachelors
1	Brown	—
2	Wang	High School

ID	Name	Age	Gender	Ed.Level
1	1	0	1	1
1	1	0	0	0
1	1	0	0	1

Table B:

Name	Age
Smith	27
Brown	24
Wang	32

ID	Name	Age	Gender	Ed.Level
0	1	1	1	0
0	1	1	0	0
0	1	1	0	0

Table C:

Name	Gender
Smith	Male
Brown	Male
Wang	Male

ID	Name	Age	Gender	Ed.Level
0	1	0	0	0
0	1	0	1	0
0	1	0	0	0

Table D:

Name	Age	Gender	Education Level
Smith	27	—	—
Brown	24	Male	Masters
Wang	32	Female	—

ID	Name	Age	Gender	Ed.Level
0	1	1	1	0
0	1	1	1	1
0	1	1	1	0

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Figure 5: Matrix initialization and traversal with Source Table containing Applicants' data, and candidate tables A, B, C, D.

Next, we simulate table integration by applying the logical Or on the matrices. Taking the logical Or of two matrices takes the maximum value at each position. This is comparable to applying the Outer Union ($\uplus$) of two tables, and applying Subsumption (β) and Complementation (κ) on the resulting table. Taking the Outer Union result of two tables, we apply β, κ such that for two tuples, t_1, t_2 that share the same non-null value at the same column, the resulting tuple t_r is formed such that for every column j, $t_r[j] = \not\perp$ if $t_1[j] \neq \perp$ or $t_2[j] \neq \perp$ and $t_r[j] = \perp$ otherwise.

In matrix representations, we combine two matrices by combining tuples at the same row index, since all tuples are encoded based on the order of the Source Table's keys. Recall that we encode a 1 in the matrix if the corresponding table shares the same value as the Source Table at the same index, and a 0 if it has a null where the Source Table contains a non-null value at the same index. Then, integrating tuples t_1, t_2 can be simulated with respective matrix tuples m_1, m_2. This way, when we combine the values in m_1, m_2 at column j, assuming that $S[i, j] \neq \perp$, the produced tuple m_r contains the following value at position (i, j): $m_r[j] = max(t_1[j], t_2[j])$

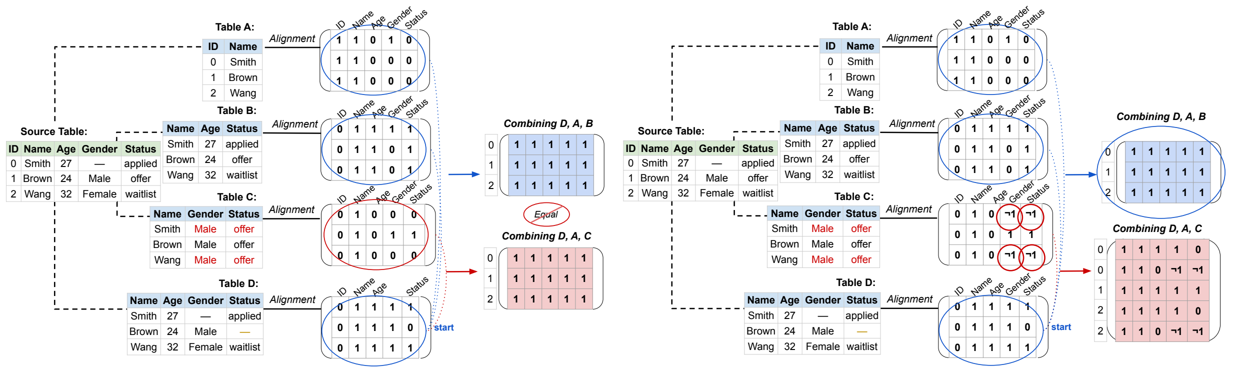
Thus, matrix integration results in a 1 if there is a matching, non-null value to the Source Table's value. Similarly, table integration results in a non-null value if there exists a non-null value in the element-wise integration. Both maximally combine the tuples such that non-null values can replace a null value at the same index.

Example 8. *In Figure 5, the matrix for table D is chosen as the start node in the matrix traversal, as it contains the largest number of values in the Source Table, which is reflected in the percentage of 1's. After exploring other tables to combine with, we find that combining matrices D and A results in a perfect matrix with all 1's, indicating a table integration between tables A and D that would perfectly reclaim the Source Table. We then return tables A and D as the only tables needed in the integration. Thus, the input set of 4 candidate tables is refined to a set containing just Tables A and D.*

Putting it all together, we show the matrix initialization and traversal in Algorithm 2. Given a ranked set of candidate tables $\mathcal{T}$ and Source Table S, we initialize each candidate table's matrix representation of its aligned tuples with respect to S (Line 3). Then, we traverse over the matrices and perform the logical Or operator to combine a pair of matrices in Combine() (Line 9). To evaluate the resulting matrix, we check the fraction of 1's in the matrix, which represent the number of values in the resulting table integration found in the Source Table, thus evaluating its similarity with S or Value Similarity Score with respect to S. At each step of the matrix traversal (Line 13), including the start (Lines 4-5), we thus choose the matrix that results in a matrix containing the most 1's in evaluateSimilarity(). This traversal ends when either all matrices have been traversed (Line 7), or the percentage of 1's in the resulting matrix converges (Lines 17-18). Note that early convergence can be achieved if the lower-ranked tables in the input set do not add new information (can replace 0's with 1's) to the resulting matrix. Finally, we return the set of tables used in the final traversal as the set of originating tables to perform the table integration.

5.3 Three-Valued Matrices

Previously, we use matrices populated with binary values to represent aligned tuples with respect to the Source Table. However, this representation cannot distinguish between nullified and erroneous aligned tuples with respect to the Source Table. Specifically, it does not account for cases in which a tuple in the Source Table and an aligned tuple in a candidate table have different non-null values in the same column, and if a tuple in the Source Table has a null value while the aligned tuple has a non-null value at the same column. Rather, it represents both types of values as 0 in the matrices, as shown in Figure 6(a). In actuality, when we apply Outer Union on two tables with aligned tuples containing different non-null values in the same column, we keep the tuples separate.



(a) Two-Valued Matrix Representation (b) Three-Valued Matrix Representation

Figure 6: A new Source Table and candidate tables A, B, C, D. Figure (a) shows two-valued matrix representations, which does not distinguish between nullified and erroneous values. Figure (b) shows three-valued matrices to make this distinction.

```
Algorithm 2: Matrix Traversal
1  Input: $\mathcal{T} = \{T_1, T_2, \ldots T_n\}$: set of candidate tables; $S$: Source Table
2  Output: $T_{\text{orig}} = \{T_1, T_2, \ldots T_i\}$: refined set of originating tables
3  $\mathcal{M} \leftarrow$ MatrixInitialization($\mathcal{T}$), //Initialize Matrices of $S$ shape
4  $T_{\text{start}} \leftarrow$ GetStartTable($\mathcal{M}$)
5  prevCorrect = mostCorrect $\leftarrow$ evaluateSimilarity($T_{\text{start}}$)
6  $T_{\text{orig}} \leftarrow$ []
7  while $|T_{\text{orig}}| < |\mathcal{T}|$ do
8      if $T_{\text{orig}}$ then
9          $M_c \leftarrow$ Combine($T_{\text{orig}}$) //Iteratively combine each pair of
             consecutive matrices
10     prevCorrect = mostCorrect; nextTable = $\perp$
11     for all tables $T \in \mathcal{T} s.t. T \notin T_{\text{orig}}$ do
12         $M_c \leftarrow$ Combine($M_c$, $T$)
13         percentCorrectVals $\leftarrow$ evaluateSimilarity($M_c$)
14         if percentCorrectVals > mostCorrect then
15             mostCorrect $\leftarrow$ percentCorrectVals
16             nextTable $\leftarrow T$
17     if mostCorrect = prevCorrect then
18         Exit, //Integration did not find more of $S$'s values
19     $T_{\text{orig}} = T_{\text{orig}} \cup$ nextTable
20 return $T_{\text{orig}}$;
```

Example 9. *Suppose a data scientist has the left, green, Source Table in Figure 6(a), that contains applicants' data as instances, and their ID, Age, Gender, and Status of their application. For the candidate tables A, B, C, D, their matrix representations allow the integration of the start matrix D with matrices A and B, and the integration of matrices D, A, and C to result in the same perfect matrix with all 1's. In practice, when we integrate tables D, A, and C, the erroneous values in the Gender and Status columns from table C are passed on to the integration result. However, the current matrix representations do not reflect this behavior.*

Thus, we need to distinguish between nullified and erroneous aligned tuples in the matrix representation (Line 3 in Algorithm 2). To do so, we make use of three-valued matrices, in which we encode a 1 if a candidate table shares the same value with the Source Table at the same index in an aligned tuple, 0 if a candidate table contains a null where the Source Table has a non-null value at the same index, and -1 if they contain contradicting non-null values at the same index (shown in Figure 6(b)). Formally, given Source Table S and candidate table T, we populate position (i, j) for each aligned tuple $t_{\text{Align}} \in T$ in matrix M as:

$$M[i,j] = \begin{cases} 1 & \text{if } S[i,j] = T[i,j] \\ 0 & \text{elif } S[i,j] \neq \perp \wedge T[i,j] = \perp \\ -1 & \text{otherwise} \end{cases} \tag{3}$$

With the amended matrix representations, we now discuss how to combine them during matrix traversal. With three-valued matrices, the logical Or over two matrices takes the maximum of two truth-values at each index. Specifically, if we have two tuples from two matrices that contain a 1 and -1 at the same position, applying logical OR would choose the 1 [7]. However, in practice when applying Outer Union on two tuples with contradicting non-null values, the resulting integration would contain both tuples. Thus, we keep both tuples from the matrices if they contain different non-0 values at the same index. We re-define Combine() (Line 9) between two matrices, given tuples t_1, t_2 at the same row index accordingly.

$$\text{Combine}(t_1, t_2) = \begin{cases} \text{Return } t_1, t_2 & \text{if } t_1[j] \neq t_2[j] \neq 0 \text{ for column } j \\ \text{OR}(t_1, t_2) & \text{otherwise} \end{cases} \tag{4}$$

This way, we keep the two tuples separate if they contain contradicting, non-0 values at the same position, and otherwise apply logical Or and take the maximum of truth values element-wise.

As expected, this new Combine() could result in matrices with more rows than in the Source Table. To account for this, we encode each matrix as a dictionary, with each key value in the Source Table as the key in the dictionary, and the list of aligned tuples in the resulting matrix with respect to a tuple in the Source Table as values. In evaluating the start (Lines 4-5) and resulting matrices (Line 13) in evaluateSimilarity(), we evaluate the Value Similarity Score by taking the aligned tuple with the largest number of aligned values to its corresponding tuple in the Source Table, or the largest number of 1's. Then, we check the number of correct values (1's) vs. the number of erroneous values (-1's), out of the Source Table size. Thus, we treat correct, nullified, and erroneous aligned tuples with respect to the Source Table in different manners, and combine their matrix representations depending on the behavior of applying Outer Union and unary operators.

Example 10. *Given the same tables from Figure 6(a), Figure 6(b) now encodes nullified and erroneous values from the candidate tables differently when forming the matrix representations for the aligned tuples with respect to the Source Table. Now, when we integrate matrices D, A, and C, it produces a matrix that also contain the erroneous encoding, or $\neg 1$. In contrast, integrating matrices D, A, and B still results in a perfect matrix with all 1's. This exactly reflects the behavior of integrating the tables, in which integrating tables D, A, and B perfectly reclaims the Source Table whereas integrating tables D, A, and C contains the erroneous values.*

6 EXPERIMENTS

We now present evaluations on benchmarks with tables containing real instances (Source Tables) from the well-known TPC-H Benchmark [60] and also the T2D Gold [63] Benchmark. We use these benchmarks along with tables from a real data lake. For the data lakes, we use the large SANTOS benchmark [32] and a sample of WDC [39]. Effectiveness experiments in Section 6.2 show that Gen-T is able to perfectly reclaim 11-13 Source Tables, whereas all baselines only perfectly reclaim at most 1 Source Table across benchmarks. Section 6.3 shows scalability experiments in which Gen-T achieves a runtime that is 5X faster than the next-fastest baseline on a large, real data lake. Finally, Section 6.4 shows the generalizability of Gen-T to a different real-world application.

6.1 Experimental Setup

We implement Gen-T in Python on a CentOS server with Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz processor. We evaluate Gen-T using 6 benchmarks whose statistics are outlined in Table 1.

Benchmark	# Tables	# Cols	Avg Rows	Size (MB)
TP-TR Small	32	244	782	3
TP-TR Med	32	244	10.8K	40
SANTOS Large +TP-TR Med	11K	122K	7.7K	11K
TP-TR Large	32	244	1M	3.9K
T2D Gold	515	2,147	74	4
WDC Sample +T2D Gold	15K	75K	14	66

Table 1: Statistics on Data lakes of each benchmark

TP-TR Benchmarks: First, we take the 8 tables from the TPC-H benchmark [60], which contain business information including customers, products, suppliers, nations, etc. Using these tables, we create three versions of a benchmark suite titled TP-Table Reclamation (TP-TR). TP-TR Large has TPC-H tables with original table sizes. TP-TR Med has TPC-H tables that are each 1/100 of its original table's rows, and TP-TR Small has TPC-H tables that are each ~1/1000 of its original table's rows. In addition, to further assess the effectiveness and scalability of our table discovery method, we embed TP-TR Med into a real, large data lake SANTOS Large [32].

To better represent a real-world scenario of a data lake, we populate the TP-TR benchmarks with both incomplete and inconsistent tables. To do so, for the three TP-TR benchmarks, we take each of the 8 tables and create 4 versions of the same table – creating 32 tables in total (detailed in Table 1). For two versions, we randomly nullify different subsets of the values, and for the other two versions, we randomly inject different non-null, or erroneous values in different subsets of values. This way, we evaluate the robustness of our approach when run on benchmarks consisting of tables that contain nullified and erroneous tuples, with respect to the original tables used to create the Source Tables for our benchmarks. Our goal in the table discovery phase is then to filter out the tables with injected non-null noise, so that the resulting reclaimed table would not contain any erroneous value. Instead, we seek to verify that our approach uses the nullified versions rather than the erroneous versions so that combining them can reproduce the Source Table.

We further create queries that produce the Source Tables that we aim to reclaim, from the 8 original tables (without injected nulls and noisy values) of each benchmark. To ensure variations of SPJU queries with no aggregations or string-transformations involved, we create 26 queries, each having a subset of operators $\{\pi, \sigma, \bowtie, ⟕, ⟗, \cup, \uplus\}$. In these 26 queries, the number of operations ranges from 2 (just π, σ), to 9, such that the query with the maximum number of unions contains 4 unioned tables, and the query with the maximum number of joins joins 3 tables. Running the same queries on each TP-TR benchmarks, we create 26 Source Tables for the TP-TR Small benchmark containing an average of 9 columns and 27 rows, and 26 Source Tables for the TP-TR Med and TP-TR Large benchmarks that have an average of 9 columns and 1K rows.

T2D Gold Benchmark: In addition, we explore the real-world application of our method with the T2D Gold Benchmark [63], which takes web tables and matches them to properties from DBpedia. This benchmark was not originally created for the problem of Table Reclamation, so we test the generalizability of Gen-T by seeing if it can reclaim any of this benchmarks' tables. We take 515 raw tables that contain some non-numerical columns and a key column. Since we do not have prior knowledge of whether or not any of these 515 tables can be "reclaimed" as a Source, no Source is known to be able to be perfectly reclaimed from a subset of tables in the benchmark (0 Reclaimable Sources). Thus, we iterate through each of the 515 tables as potential sources. To further assess the effectiveness of Gen-T, we embed T2D Gold tables into a sample of the WDC web table corpus [39], which contains 15K relational web tables.

6.1.1 *Baselines.*

We compare Gen-T against the current state-of-the-art for *by-target synthesis*, Auto-Pipeline [65], and the state-of-the-art for table integration, ALITE [33].

Auto-Pipeline has a similar framework to our problem in discovering the integration that reclaims the Source Table, however Gen-T does not assume to have the perfect set of input tables from which we can synthesize the query that reproduces the Source Table. Auto-Pipeline has both search and deep reinforcement learning approaches, but since we propose an unsupervised approach, we use the search variation as our baseline. Auto-Pipeline's code implementation is not openly available, so we adopted an open reimplementation of their search approach [57], which adapts the framework in Foofah [30] to perform pairwise table operations and determine which of the operators $\{\cup, \bowtie, ⟕, ⟗\}$ to perform at each edge traversal. We call this re-implemented, adapted baseline Auto-Pipeline*. Since Auto-Pipeline's benchmarks contain small tables, and most of their operators are string-transformation operators, we do not consider their benchmarks for our experiments.

To show the need for our Matrix Traversal rather than directly integrating the set of candidate tables returned from Set Similarity

(Section 5.1), we also compare against ALITE and give it the set of candidate tables from Set Similarity as input. Also, since Gen-T first projects and selects on the Source Table's columns and keys before performing integration, we compare with a variation of ALITE, which we call ALITE-PS, that also first performs projection and selection before the table integration. ALITE without projection and selection is much slower as it is creating a much larger integration result, hence ALITE-PS is a fairer comparison.

For each of the three baselines, on the TP-TR benchmarks, we create another variant in which we give each method a specific integrating set of tables as input, rather than the full set of candidate tables returned from Set Similarity. Since we know what subset of tables from the 8 original tables were used to create the 26 Source Tables, we know that a perfect reclamation of each Source Table contains variants of these tables. Thus, for all original tables used to create each Source Table, the integrating set of tables includes all variations (2 tables with nullified values and 2 tables with non-null erroneous values for each original table) of these tables.

6.1.2 Metrics. For effectiveness, we evaluate how much of the values in Source Table have been reclaimed, or how similar the values in the reclaimed table are to those of the Source Table. Thus, the Source Tables are essentially our ground truth in that we see how much of its rows or values we can re-produce.

Precision and Recall: Consider a Source Table S and reclaimed table from a method $\hat{S}$. From the measure Tuple Difference Ratio (TDR) introduced in ALITE [33], we derive two similarity measures, Recall and Precision, that measure the # of tuples in the intersection of S and $\hat{S}$ relative to the # of tuples in each table. Recall $= |S \cap \hat{S}|/|S|$ and Precision $= |S \cap \hat{S}|/|\hat{S}|$. In addition to metrics that measure the similarity between the tuples of a reclaimed table and a Source Table, we also include finer-grain metrics that measure the number of values that do not match within aligned tuples (tuples with the same key value). If there are multiple aligned tuples with respect to one tuple in the Source Table (multiple tuples in the reclaimed table with the same key value), then we consider the tuple that contains the largest number of column values shared with the corresponding tuple in the Source Table. This way, there is at most 1 aligned tuple in the reclaimed table for each tuple in the Source Table. In these measures, which we denote as *divergence measures*, the ideal score is 0 (the reclaimed table is identical to the Source Table). Specifically, we measure how many of the tuples of the Source Table are not found in the reclaimed table (using *Instance Divergence*), and how many of the values found in reclaimed tuples differ from those in the Source Table (using *Conditional KL-Divergence*). This enables us to measure the nullified and erroneous values in the reclaimed table aligned tuples, with respect to the Source Table (see Section 4.3).

Instance Divergence: We measure how many missing values there are in each aligned tuple, with respect to its corresponding tuple in the Source Table. To do so, we define Instance Divergence, which is the inverse of a measure introduced in MapMerge [3] that they refer to as Instance Similarity. Given a Source Table S and a reclaimed table $\hat{S}$, the Instance Divergence (Inst-Div.) is computed as follows.

Consider two aligned tuples s from S and $t_{\text{Align}} = match(s)$ from $\hat{S}$ with the same key value. Let $s \cap t_{\text{Align}}$ be the number of non-key columns on which s and t_{Align} agree. If t_{Align} does not exist (tuple s was not reclaimed), then $s \cap match(s)$ is defined as 0. Also let n be the cardinality of S defined as the number of non-key columns.

$$\text{Inst-Div.} = 1 - \frac{\sum_{s \in S, match(s) \in \hat{S}} \frac{|s \cap match(s)|}{n}}{|S|} \tag{5}$$

For each aligned tuple (see Section 4.3) in the reclaimed table, we find the fraction of overlapping values it has with its aligned tuple in the Source Table. Then, we average over all aligned tuples, with respect to the total number of tuples in the Source Table, and take the inverse to find the Instance Divergence. For a reclaimed table containing all tuples and all values from the Source Table, the Inst-Div. score is 0. Otherwise if no value from the Source Table is found in the reclaimed table, the score is 1.

Conditional KL-divergence: Finally, we want to capture how erroneous the values may be in the aligned tuples from a reclaimed table with respect to tuples in a Source Table. Thus, we also consider the Conditional KL-divergence of a reclaimed table, with respect to a Source Table, conditioned on the probability that the key values from the Source Table are found in the reclaimed table. We adopt the traditional definition of conditional KL-divergence [17, 45], and also add a penalization for erroneous values, such that the score is higher for reclaimed tables containing erroneous values as opposed to nulls in their aligned tuples with the Source Table. Given column C shared between a Source Table and a reclaimed table T, suppose we have probability distributions, $\mathcal{P}$ for C in the Source Table and $\mathcal{Q}$ for C in the reclaimed table. We condition on the key values in primary key column K. The conditional KL-divergence (or conditional relative entropy) between $\mathcal{P}$ and $\mathcal{Q}$ of sample space X of column C conditioned on key K is as follows:

$$D_{KL}(Q||P) = -\sum_{x \in X, k \in K} P(x|k) \log\left(\frac{Q(x|k)(1 - Q(\neg x|k))}{P(x|k)}\right) \tag{6}$$

Given n non-key columns C in a Source Table we take the average D_{KL} for each column divided by the probability of a key value in T matching a key value from the Source Table ($Q(K)$) and the number of non-key columns (n). Then, the conditional KL-divergence of the reclaimed table is as follows:

$$D_{KL}(T) = \frac{D_{KL}(Q_1||P_1) + D_{KL}(Q_2||P_2) + \cdots + D_{KL}(Q_n||P_n)}{Q(K) * n} \tag{7}$$

The conditional KL-divergence of the reclaimed table is a score $\in [0, \infty)$, with 0 being the ideal score. There is no upper limit on this metric since it naturally approaches ∞ when no key value from the Source Table is found in the reclaimed table.

We report the average effectiveness scores over all Source Tables.

Efficiency Measures: For efficiency, we measure the average runtimes for all Source Tables, as well as the average ratio of the output size of the reclaimed table to the size of the Source Table (creating a large reclaimed table can significantly increase runtimes).

6.2 Effectiveness

Table 2 reports the similarity and divergence scores, respectively, for all four TP-TR benchmarks across all methods. For all methods on TP-TR Small, TP-TR Med, and TP-TR Large benchmarks, we input candidate tables discovered from just Set Similarity (Section 5.1). For methods run on SANTOS Large +TP-TR Med, we first discover

Method	TP-TR Small		TP-TR Med		SANTOS Large +TP-TR Med		TP-TR Large	
	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
ALITE	0.704	0.128	0.662	0.202	—	—	—	—
ALITE w/ integrating set	0.745	0.133	0.694	0.202	0.694	0.202	—	—
ALITE-PS	0.805	0.539	0.880	0.556	0.842	0.554	0.775	0.521
ALITE-PS w/ integrating set	0.833	0.552	0.880	0.569	0.880	0.569	0.880	0.569
Auto-Pipeline*	0.674	0.272	—	—	—	—	—	—
Auto-Pipeline* w/ integrating set	0.683	0.289	—	—	—	—	—	—
Gen-T	**0.958**	**0.839**	**0.976**	**0.867**	**0.976**	**0.867**	**0.971**	**0.816**

Method	TP-TR Small		TP-TR Med		SANTOS Large +TP-TR Med		TP-TR Large	
	Inst-Div.	D_{KL}	Inst-Div.	D_{KL}	Inst-Div.	D_{KL}	Inst-Div.	D_{KL}
ALITE	0.095	1.332	0.100	35.831	—	—	—	—
ALITE w/ integrating set	0.086	1.197	0.085	36.348	0.085	36.348	—	—
ALITE-PS	0.040	0.655	0.009	3.524	0.011	4.629	0.049	21.978
ALITE-PS w/ integrating set	0.037	0.688	0.009	3.524	0.009	3.524	0.009	3.524
Auto-Pipeline*	0.158	2.574	—	—	—	—	—	—
Auto-Pipeline* w/ integrating set	0.133	2.109	—	—	—	—	—	—
Gen-T	**0.015**	**0.130**	**0.004**	**1.326**	**0.004**	**1.326**	**0.004**	**1.490**

Table 2: Similarity and Divergence Measures of Gen-T and baselines on all TP-TR benchmarks, given the same set of candidate tables from Set Similarity. If there are no results for some method, then it timed out for most if not all Source Tables.

relevant tables from the large data lake using Starmie [22], a state-of-the-art self-supervised system for scalable table discovery. Hence, it can discover a set of candidate tables for the Source Table from a large data lake. Although the primary use case of Starmie was table union search, it was shown to apply to other search semantics such as table discovery to improve the performance of downstream machine learning tasks via feature discovery (join search) and column clustering. Following Starmie, we run Set Similarity to find syntactically similar tables among the returned tables from Starmie.

As the sizes and/or number of tables increase across the TP-TR benchmarks, some baselines timeout for most Source Tables. Auto-Pipeline* times out for every benchmark except for TP-TR Small, and ALITE times out on TP-TR Large benchmark. We discuss scalability and timeouts in Section 6.3.

Across all benchmarks, Gen-T outperforms the baselines for all metrics, while perfectly reclaiming 15-17 Source Tables across all benchmarks. The baselines ALITE-PS and Auto-Pipeline* only perfectly reclaim 3 Source Tables and 1 Source Table across the benchmarks on which they do not time out, respectively, and ALITE does not perfectly reclaim any. In fairness, ALITE is an integration method and does not take the Source Table into account (it is not "target-driven" like Auto-Pipeline*). In terms of similarity (top table of Table 2), Gen-T outperforms the top performing existing baseline method (ALITE) by 36-47% in Recall and by 71%-329% in Precision across all TP-TR benchmarks. For the divergence measures (bottom table of Table 2), we see that Gen-T produces tables that contain fewer nullified values in its aligned tuples with respect to the Source Table (Inst-Div.), as well as fewer erroneous values in its aligned tuples, which is reflected in the lower D_{KL} scores than the baselines.

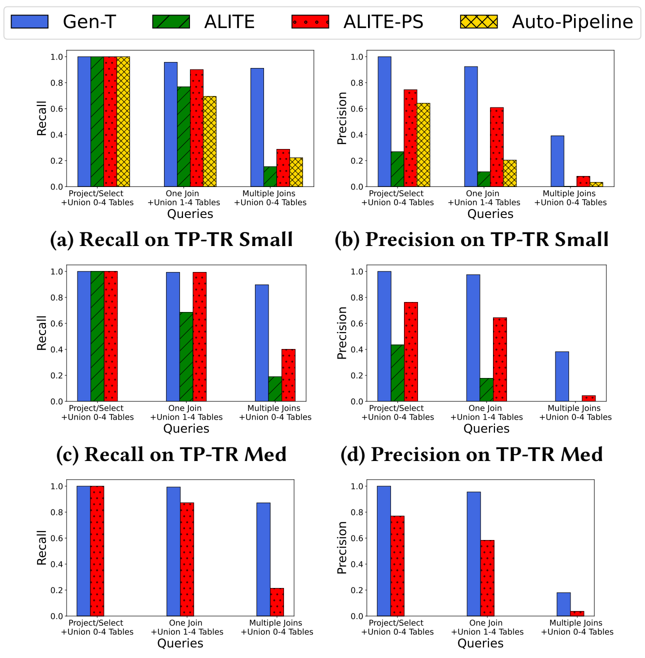
Even compared to each baseline that is given specified integrating sets of tables rather than a large set of candidates ('w/ integrating set'), Gen-T performs much better. Thus, the matrix traversal method (Section 5.3) used in Gen-T to refine the set of originating tables works well in filtering out misleading tables that could be integrated to produce tables containing erroneous values with respect to the Source Table. We provide examples and Benchmark samples corresponding to these examples in our repository.[1]

To better understand the performance of the methods on different types of queries used to initially create the Source Tables, we perform an analysis of the similarity measures for all methods on different types of queries used to form the Source Tables in TP-TR benchmarks, shown in Figure 7. Ranging from simple queries (that just performing Projection, Selection, and Union) to more complex queries (joining up to 4 tables and unioning up to 4 tables), we see that Gen-T outperforms the baselines on queries of all complexities used to initially create the Source Table. Thus, not only does the matrix traversal display effectiveness, but the set of operators used in table integration represents well different types of queries.

6.3 Scalability

Figure 8 shows the scalability of all methods across benchmarks as the number of tables and/or the size of tables grows larger, from the smallest TP-TR benchmark (TP-TR Small), to the tables from TP-TR Med embedded in a large data lake (SANTOS Large). Figure 8(a) reports average runtimes for all methods across all four benchmarks. Since all methods are given the same sets of candidate tables, we report the runtimes starting from ingestion of the candidate tables. For Gen-T, this time includes the time it takes to initialize the matrix representations, traverse over matrices to refine the set of candidate tables, and integrate the originating tables to produce an output table. For the other methods, this time only includes the integration time. We find that Auto-Pipeline* only runs on TP-TR Small without timing out, and ALITE, which performs full disjunction, is exponential in time and times out for the last two benchmarks. We set the timeouts for each method according to the details reported in respective papers and with respect to the data lake size. Specifically, we use a 30min timeout for TP-TR Small

[1] https://github.com/northeastern-datalab/gen-t



(a) Recall on TP-TR Small (b) Precision on TP-TR Small

(c) Recall on TP-TR Med (d) Precision on TP-TR Med

(e) Recall on TP-TR Large (f) Precision on TP-TR Large

Figure 7: Recall and Precision of different types of queries that produce Source Tables over the TP-TR Benchmarks.

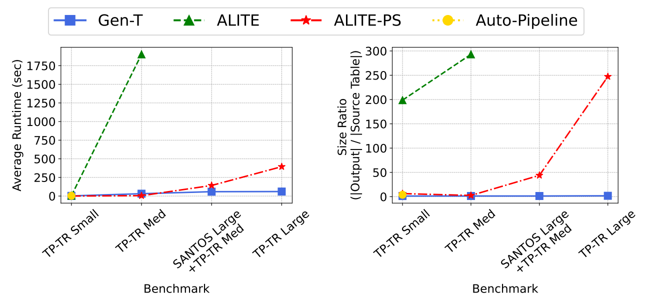
benchmark, 7hrs timeout for TP-TR Med and SANTOS Large +TP-TR Med benchmarks and 24hrs timeout for TP-TR Large benchmark.

Gen-T, although having an overhead of matrix initialization and traversal, has a more consistent runtime across all benchmarks compared to all baselines. Gen-T is 2X faster in runtime compared to Auto-Pipeline* on the TP-TR Small. On TP-TR Med, Gen-T is 60X faster than ALITE and on the TP-TR Large, Gen-T is 7X faster than ALITE-PS. Thus, the matrix representations and refinement of the set of originating tables seems to be beneficial in cutting the cost of integration, a prevalent issue as shown by the baselines.

In Figure 8(b), we report the average output sizes, or number of cell values in the reclaimed tables, with respect to the average Source Table sizes. As the number and sizes of tables grow across benchmarks, the output sizes relative to the sizes of the Source Table (expected output size) can easily grow at a fast rate if the integration is among more or larger tables, especially if it includes noisy tables from the real data lake (SANTOS Large). The output sizes for Gen-T remain consistent across benchmarks, being 1.3-1.8X larger than the average Source Table's size. This trend largely accounts for the higher precision of Gen-T, compared to the baselines, since Gen-T produces tables that mostly consist of tuples from the Source Table. Thus, Gen-T's runtimes and output sizes remain consistent across benchmarks of different sizes, even when the expected tables for integration are immersed in a large, real data lake.

6.4 Generalizability

We also experiment with the T2D Gold benchmark to see how well we can apply Gen-T to a real-world scenario with Web Tables. In this case, we do not know *whether* or *how* the tables were originally generated. Accordingly, we attempt to reclaim each table using a subset of other tables in the benchmark by iterating through each



(a) Average Runtimes in sec. (b) Average Ratio of Output Size (total # values) / Source Table Size

Figure 8: Scalability via Average Runtime (sec) and Ratio of Output Sizes to Source Table sizes over all TP-TR Benchmarks

515 tables as potential Source Tables.Gen-T is able to successfully reclaim three Source Tables from an integration of multiple tables (5-6 tables), such that the output has perfect Recall, Precision, Instance-Div., and near-perfect D_{KL}. Gen-T also finds duplicate tables for 12 Source Tables, or 6 sets of duplicates. This indicates that we can apply Gen-T in a different domain, even if no sources are known to be reclaimed, and retrieve successful reclamations. Baseline methods are able to reclaim 12-13 Source Tables, which are included in the 15 Source Tables reclaimed by Gen-T. Specific results are given in a technical report [21].

Method	Recall	Precision	Inst-Div.	D_{KL}
ALITE	0.956	0.490	0.009	0.627
ALITE-PS	0.956	0.796	0.009	0.627
Auto-Pipeline*	0.881	0.725	0.088	19.261
Gen-T	0.956	1.000	0.009	0.627

Table 3: Sources over T2D Gold immersed in the WDC Sample data lake for which all methods have non-empty outputs.

We then run experiments on a data lake consisting of both T2D Gold and WDC Sample tables. This way, we can evaluate how well the methods perform when the set of candidate tables returned from Set Similarity may contain irrelevant or misleading tables from the WDC Sample benchmark. As shown in Table 3, we report the similarity and divergence scores for all methods on 33 of the common sources from T2D Gold for which all methods have non-empty, reasonably-sized output tables. We can see that Gen-T outperforms the baselines for all measures, even having perfect precision of 1.0. In contrast, the baseline methods that are given the candidate tables from Set Similarity integrate all candidate tables and produce tables that contain a lot of additional tuples.

7 CONCLUSION

Table Reclamation is essential in verifying if a data lake supports the tuples (facts) in a Source table and identifying possible erroneous or nullified values. Our results show that despite the large search space, Gen-T can solve the reclamation problem efficiently for source tables with keys. In future work, we will relax the key assumption and look at combining reclamation with provenance enabling data scientists to ask provenance queries over erroneous or other source values. This will require understanding provenance over queries containing subsumption and complementation operators.

REFERENCES

[1] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79.

[2] Marco D. Adelfio and Hanan Samet. 2013. Schema Extraction for Tabular Data on the Web. *Proc. VLDB Endow.* 6, 6 (2013), 421–432.

[3] Bogdan Alexe, Mauricio A. Hernández, Lucian Popa, and Wang Chiew Tan. 2012. MapMerge: correlating independent schema mappings. *VLDB J.* 21, 2 (2012), 191–211.

[4] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 168:1–168:27.

[5] Jens Bleiholder and Felix Naumann. 2008. Data fusion. *ACM Comput. Surv.* 41, 1 (2008), 1:1–1:41.

[6] Jens Bleiholder, Sascha Szott, Melanie Herschel, and Felix Naumann. 2010. Complement union for data integration. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE Computer Society, 183–186.

[7] Leonard Bolc and Piotr Borowik. 2013. *Many-valued logics 1: Theoretical foundations.* Springer Science & Business Media.

[8] Angela Bonifati, Radu Ciucanu, Aurélien Lemay, and Slawek Staworko. 2014. A Paradigm for Learning Queries on Big Data. In *Proceedings of the First International Workshop on Bringing the Value of "Big Data" to Users, Data4U@VLDB 2014, Hangzhou, China, September 1, 2014*. ACM, 7.

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

[10] Michael J. Cafarella, Alon Y. Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proc. VLDB Endow.* 2, 1 (2009), 1090–1101.

[11] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD*. 1335–1349.

[12] Chen Chen, Behzad Golshan, Alon Y. Halevy, Wang-Chiew Tan, and AnHai Doan. 2018. BigGorilla: An Open-Source Ecosystem for Data Preparation and Integration. *IEEE Data Eng. Bull.* 41, 2 (2018), 10–22.

[13] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (2009), 379–474.

[14] James Cheney and Wang-Chiew Tan. 2018. Provenance in Databases. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_283

[15] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2019. End-to-End Entity Resolution for Big Data: A Survey. *CoRR* abs/1905.06397 (2019).

[16] E. F. Codd. 1979. Extending the Data Base Relational Model to Capture More Meaning (Abstract). In *SIGMOD*. 161.

[17] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of information theory (2. ed.).* Wiley.

[18] Daniel Deutch and Amir Gilad. 2016. QPlain: Query by explanation. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 1358–1361.

[19] Hong Hai Do and Erhard Rahm. 2002. COMA - A System for Flexible Combination of Schema Matching Approaches. In *Proc. VLDB Endow.* 610–621.

[20] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2022. MATE: Multi-Attribute Table Extraction. *Proc. VLDB Endow.* 15, 8 (2022), 1684–1696.

[21] Grace Fan, Roee Shraga, and Renée J. Miller. 2023. Technical Report on Gen-T: Table Reclamation on Data Lakes. https://github.com/northeastern-datalab/gen-t/blob/main/gen-t-technical-report.pdf

[22] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. 2023. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. *Proc. VLDB Endow.* 16, 7 (2023), 1726–1739.

[23] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing Quality to Data Lakes. In *SIGMOD*. 2089–2092.

[24] Raul Castro Fernandez, Essam Mansour, Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery. In *ICDE*. 989–1000.

[25] Sainyam Galhotra, Yue Gong, and Raul Castro Fernandez. 2023. METAM: Goal-Oriented Data Discovery. *CoRR* abs/2304.09068 (2023).

[26] César A. Galindo-Legaria. 1994. Outerjoins as Disjunctions. In *SIGMOD*. 348–358.

[27] Lise Getoor and Ashwin Machanavajjhala. 2012. Entity Resolution: Theory, Practice & Open Challenges. *Proc. VLDB Endow.* 5, 12 (2012), 2018–2019.

[28] Yue Gong, Zhiru Zhu, Sainyam Galhotra, and Raul Castro Fernandez. 2023. Ver: View Discovery in the Wild. *CoRR* abs/2106.01543.

[29] Sairam Gurajada, Lucian Popa, Kun Qian, and Prithviraj Sen. 2019. Learning-Based Methods with Human-in-the-Loop for Entity Resolution. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, Wenwu Zhu, Dacheng Tao, Xueqi Cheng, Peng Cui, Elke A. Rundensteiner, David Carmel, Qi He, and Jeffrey Xu Yu (Eds.). 2969–2970.

[30] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*. 683–698.

[31] Dmitri V. Kalashnikov, Laks V. S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *SIGMOD*. 337–350.

[32] Aamod Khatiwada, Grace Fan, Roee Shraga, Zixuan Chen, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2023. SANTOS: Relationship-based Semantic Table Union Search. In *SIGMOD*.

[33] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proc. VLDB Endow.* 16 (2022), 932–945.

[34] Maximilian Koch, Mahdi Esmailoghli, Sören Auer, and Ziawasch Abedjan. 2023. Duplicate Table Discovery with Xash. *BTW 2023* (2023).

[35] Martin Koehler, Edward Abel, Alex Bogatu, Cristina Civili, Lacramioara Mazilu, Nikolaos Konstantinou, Alvaro A. A. Fernandes, John A. Keane, Leonid Libkin, and Norman W. Paton. 2021. Incorporating Data Context to Cost-Effectively Automate End-to-End Data Wrangling. *IEEE Trans. Big Data* 7, 1 (2021), 169–186.

[36] 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 *ICDE*. 468–479.

[37] Tai Le Quy, Arjun Roy, Vasileios Iosifidis, Wenbin Zhang, and Eirini Ntoutsi. 2022. A survey on datasets for fairness-aware machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 12, 3 (2022), e1452.

[38] Oliver Lehmberg and Christian Bizer. 2017. Stitching Web Tables for Improving Matching Quality. *Proc. VLDB Endow.* 10, 11 (2017), 1502–1513.

[39] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A Large Public Corpus of Web Tables containing Time and Context Metadata. In *WWW (Companion Volume)*. 75–76.

[40] Oliver Lehmberg, Dominique Ritze, Petar Ristoski, Robert Meusel, Heiko Paulheim, and Christian Bizer. 2015. The Mannheim Search Join Engine. *J. Web Semant.* 35 (2015), 159–166.

[41] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proc. VLDB Endow.* 14, 1 (2020), 50–60.

[42] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, Jin Wang, Wataru Hirota, and Wang-Chiew Tan. 2021. Deep Entity Matching: Challenges and Opportunities. *ACM J. Data Inf. Qual.* 13, 1 (2021), 1:1–1:17.

[43] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proc. VLDB Endow.* 3, 1 (2010), 1338–1347.

[44] Xiao Ling, Alon Y. Halevy, Fei Wu, and Cong Yu. 2013. Synthesizing Union Tables from the Web. In *IJCAI*. 2677–2683.

[45] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press.

[46] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. 2002. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *ICDE*, Rakesh Agrawal and Klaus R. Dittrich (Eds.). IEEE Computer Society, 117–128.

[47] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). 19–34.

[48] 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.

[49] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (2018), 813–825.

[50] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco M. Manquinho. 2020. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. *Proc. VLDB Endow.* 13, 12 (2020), 2853–2856.

[51] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB J.* 10, 4 (2001), 334–350.

[52] Anand Rajaraman and Jeffrey D. Ullman. 1996. Integrating Information by Outerjoins and Full Disjunctions. In *PODS*, Richard Hull (Ed.). ACM Press, 238–248. https://doi.org/10.1145/237661.237717

[53] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems (3. ed.).* McGraw-Hill.

[54] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Y. Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding related tables. In *SIGMOD*. 817–828.

[55] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering queries based on example tuples. In *SIGMOD*. 493–504.

[56] Roee Shraga, Avigdor Gal, and Haggai Roitman. 2020. ADnEV: Cross-Domain Schema Matching using Deep Similarity Matrix Adjustment and Evaluation. *Proc. VLDB Endow.* 13, 9 (2020), 1401–1415.

[57] Roee Shraga and Renée J. Miller. 2023. Explaining Dataset Changes for Semantic Data Versioning with Explain-Da-V. *Proc. VLDB Endow.* 16, 6 (2023), 1587–1600.

[58] Roee Shraga, Haggai Roitman, Guy Feigenblat, and Mustafa Canim. 2020. Ad Hoc Table Retrieval using Intrinsic and Extrinsic Similarities. In *WWW*. ACM / IW3C2, 2479–2485.

[59] Roee Shraga, Haggai Roitman, Guy Feigenblat, and Mustafa Canim. 2020. Web Table Retrieval using Multimodal Deep Learning. In *SIGIR*. 1399–1408.

[60] TPC. 2014. http://www.tpc.org/

[61] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *SIGMOD*. 535–548.

[62] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 452–466.

[63] WDC. 2017. http://webdatacommons.org/webtables/goldstandard.html

[64] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. InfoGather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*. 97–108.

[65] Junwen Yang, Yeye He, and Surajit Chaudhuri. 2021. Auto-Pipeline: Synthesize Data Pipelines By-Target Using Reinforcement Learning and Search. *Proc. VLDB Endow.* 14, 11 (2021), 2563–2575.

[66] Dongxiang Zhang, Yuyang Nie, Sai Wu, Yanyan Shen, and Kian-Lee Tan. 2020. Multi-Context Attention for Entity Matching. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 2634–2640.

[67] Meihui Zhang, Hazem Elmeleegy, Cecilia M. Procopiuc, and Divesh Srivastava. 2013. Reverse engineering complex join queries. In *SIGMOD*. 809–820.

[68] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *SIGMOD*. 1951–1966.

[69] Chen Zhao and Yeye He. 2019. Auto-EM: End-to-end Fuzzy Entity-Matching using Pre-trained Deep Models and Transfer Learning. In *WWW*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). 2413–2424.

[70] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864.

[71] 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.

A APPENDIX

A.1 Table Operators

Suppose we have two tables, T_1, T_2, that share common columns C, and are in their minimal forms in which there are no duplicates and no tuples that can be subsumed or complemented. We show that for each pairwise table operator, Inner Union, Inner Join, Left Join, Outer Join, Cross Product, there exists an equivalent query consisting of Outer Union and/or unary operators. (SP of SPJU queries are accounted for by the unary operators).

LEMMA 11 (INNER UNION). *Inner Union($\cup$): it is known that if the schemas of two tables are equal, then Inner Union = Outer Union*

LEMMA 12 (INNER JOIN). *Inner Join ($\bowtie$):*

$$T_1 \bowtie T_2 = \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2))) \tag{8}$$

LEMMA 13 (LEFT JOIN). *Left Join ($⟕$) [26]:*

$$T_1 ⟕ T_2 = \beta((T_1 \bowtie T_2) \uplus T_1) \tag{9}$$

LEMMA 14 (OUTER JOIN). *Full Outer Join ($⟗$) [26]:*

$$T_1 ⟗ T_2 = \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2) \tag{10}$$

LEMMA 15 (CROSS PRODUCT). *Cross Product($\times$): We denote columns in T_1 and T_2 as $T_1.C$ and $T_2.C$, respectively. Consider a constant column c.*

$$T_1 \times T_2 = \kappa(\pi((T_1.C, c), T_1) \uplus \pi((T_2.C, c), T_2)) \tag{11}$$

Thus, $\uplus, \sigma, \pi, \kappa, \beta$ operators form queries that are equivalent to all SPJU queries.

A.1.1 Proof of Lemma 12[Inner Join].

Given two tables T_1, T_2 that join on a set of common columns C, such that T_1, T_2 are in their minimal forms in which they contain no duplicate tuples and no tuples can be subsumed or complemented , $T_1 \bowtie T_2$ can be expressed by an equivalent query containing Outer Union, complementation, and subsumption. Specifically, $T_1 \bowtie T_2$ is equivalent to query $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$.

PROOF. We first prove that all tuples in $T_1 \bowtie T_2$ are contained in $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$. Let tuple $t \in T_1 \bowtie T_2$, such that join columns C's values in t appear in both $T_1.C$ and $T_2.C$, and are non-null: $t.C \in T_1.C \cap T_2.C$ s.t. $t.C \neq \perp$.

When applying $\beta(\kappa(T_1 \uplus T_2))$, only tuples with common non-null values $T_1.C_i = T_2.C_i \neq \perp$ in same column(s) i are complemented and subsumed. This is similar to tuple t, which is formed by joining on $T_1.C_i = T_2.C_i$. Thus, tuple t is derived by selecting on tuples from $\beta(\kappa(T_1 \uplus T_2))$ with non-null C values in both $T_1.C$ and $T_2.C$, so $t \in \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$.

Next, we show that all tuples in $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$ are found in $T_1 \bowtie T_2$. Let tuple $t' \in \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$. Here, all C values in t' are non-null values found in both $T_1.C$ and $T_2.C$ as a result of selection. From $\beta(\kappa(T_1 \uplus T_2))$, t' contains all values from all columns in T_1 and T_2 in a single tuple, formed by complementing and subsuming based on common C values. Thus, $t' \in T_1 \bowtie T_2$.

We have thus shown that all tuples from $T_1 \bowtie T_2$ are found in $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$ and vice versa, and so $T_1 \bowtie T_2$ is an equivalent query to $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$. □

A.1.2 Proof of Lemma 13[Left Join].

Given two tables T_1, T_2 that join on a set of common columns C, such that T_1, T_2 are in their minimal forms in which there are no duplicates and no tuples can be subsumed or complemented , $T_1 ⟕ T_2$ can be expressed by an equivalent query containing Outer Union and subsumption. Specifically, $T_1 ⟕ T_2$ is equivalent to query $\beta((T_1 \bowtie T_2) \uplus T_1)$.

PROOF. We first prove that the resulting table of $T_1 ⟕ T_2$ is contained in the resulting table of $\beta((T_1 \bowtie T_2) \uplus T_1)$:

Let tuple $t \in T_1 ⟕ T_2$. There are two cases for join column C's values in tuple t: $t.C \in T_1.C \cap T_2.C$ (i.e., $t.C$ values are in both $T_1.C$ and in $T_2.C$) and $t.C \in T_1.C \setminus T_2.C$ (i.e., $t.C$ values are only in $T_1.C$ and not in $T_2.C$). Since we are performing left join on T_1 and T_2, $t.C \notin T_2.C \setminus T_1.C$.

(1) $t.C \in T_1.C \cap T_2.C \implies t \in (T_1 \bowtie T_2)$. Since t is in the inner join result and contains more non-Null values than other tuples with C values only in T_1 or T_2, it would not be subsumed when applying $\beta((T_1 \bowtie T_2) \uplus T_1)$.
(2) $t.C \in (T_1.C \setminus T_2.C) \implies t \in \beta((T_1 \bowtie T_2) \uplus T_1)$. Since T_1 is in its minimal form, and t does not share any C values with any tuple in T_2, it is not subsumed when applying β to $(T_1 \bowtie T_2) \uplus T_2$, and thus appear as is in $\beta((T_1 \bowtie T_2) \uplus T_1)$.

Thus, all tuples from $T_1 ⟕ T_2$ are contained in the resulting table of $\beta((T_1 \bowtie T_2) \uplus T_1)$.

Next, we show that the resulting tuples of $\beta((T_1 \bowtie T_2) \uplus T_1)$ are contained in the resulting table of $T_1 ⟕ T_2$.

Let's consider tuple $t' \in \beta((T_1 \bowtie T_2) \uplus T_1)$. There are two cases for C values in tuple t': $t'.C \in T_1.C \cap T_2.C$ and $t'.C \notin T_1.C \cap T_2.C$.

(1) $t'.C \in (T_1.C \cap T_2.C) \implies t' \in (T_1 \bowtie T_2)$. Since $(T_1 \bowtie T_2) \subseteq (T_1 ⟕ T_2)$, $t' \in (T_1 ⟕ T_2)$.
(2) All tuples in $((T_1 \bowtie T_2) \uplus T_1)$ are either subsumed by tuples from $(T_1 \bowtie T_2)$, or are in $T_1 \setminus (T_1 \bowtie T_2)$. Thus, $t'.C \notin T_1.C \cap T_2.C \implies t' \in T_1 \setminus (T_1 \bowtie T_2) \implies t' \in T_1 ⟕ T_2$.

Thus, all tuples from $\beta((T_1 \bowtie T_2) \uplus T_1)$ are contained in the resulting table of $T_1 ⟕ T_2$.

Now that we have shown that tuples from $T_1 ⟕ T_2$ are contained in the resulting table of $\beta((T_1 \bowtie T_2) \uplus T_1)$ and vice versa, we have shown that $\beta((T_1 \bowtie T_2) \uplus T_1)$ is an equivalent query to $T_1 ⟕ T_2$. □

A.1.3 Proof of Lemma 14[Outer Join].

Given two tables T_1, T_2 that join on a set of common columns C, such that T_1, T_2 are in their minimal forms in which there are no duplicates and no tuples can be subsumed or complemented, $T_1 ⟗ T_2$ can be expressed by an equivalent query containing Outer Union and subsumption. Specifically, $T_1 ⟗ T_2$ is equivalent to query $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

PROOF. We first prove that the resulting table of $T_1 ⟗ T_2$ is contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$:

Let tuple $t \in T_1 ⟗ T_2$. There are three cases for join column C's values in tuple t: $t.C \in T_1.C \cap T_2.C$ (i.e., $t.C$ values are in both $T_1.C$ and in $T_2.C$), $t.C \in T_1.C \setminus T_2.C$ (i.e., $t.C$ values are only in $T_1.C$ and not in $T_2.C$), and $t.C \in T_2.C \setminus T_1.C$ (i.e., $t.C$ values are only in $T_2.C$ and not in $T_1.C$).

(1) $t.C \in T_1.C \cap T_2.C \implies t \in T_1 \bowtie T_2$. Tuple t is a result of inner joining two tuples from T_1, T_2 on shared values in common columns C. This is similar to taking $T_1 \uplus T_2$, and applying subsumption and complementation on tuples with shared values in C (Lemma 12) to get t. Since t does not share any values in C with other tuples, it cannot be subsumed. Thus, $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

(2) $t.C \in T_1.C \setminus T_2.C \implies t \in (T_1 ⟕ T_2) \setminus (T_1 \bowtie T_2)$. When we take $(T_1 \bowtie T_1) \uplus T_1$, we append all tuples from T_1 to $T_1 \bowtie T_2$. After applying subsumption, all tuples from T_1 that are used in $T_1 \bowtie T_2$ are subsumed by tuples from $T_1 \bowtie T_2$ on shared values in C. Thus, the only tuples remaining are tuples like t in $(T_1 ⟕ T_2) \setminus (T_1 \bowtie T_2)$. Since t does not share any common values with any tuple in T_2, it is not subsumed when taking $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$, and so $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

(3) $t.C \in T_2.C \setminus T_1.C \implies t \in (T_2 ⟕ T_1) \setminus (T_1 \bowtie T_2)$. Taking the subsumption of $\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2$ removes all tuples from T_2 that are subsumed by tuples in $T_1 \bowtie T_2$. Since the remaining tuples in T_2 cannot be subsumed by any tuple from T_1 not in $T_1 \bowtie T_2$, $t \in (T_2 ⟕ T_1) \setminus (T_1 \bowtie T_2)$. Thus, $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

Thus, all tuples from $T_1 ⟗ T_2$ are contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

Next, we show that all tuples in $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ are contained in the resulting table of $T_1 ⟗ T_2$. Let's consider tuple $t' \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$. There are two cases for C values in tuple t': $t'.C \in T_1.C \cap T_2.C$ and $t'.C \notin T_1.C \cap T_2.C$.

(1) $t'.C \in (T_1.C \cap T_2.C) \implies t' \in (T_1 \bowtie T_2)$. Since $(T_1 \bowtie T_2) \subseteq (T_1 ⟗ T_2)$, $t' \in (T_1 ⟗ T_2)$.

(2) All tuples in $((T_1 \bowtie T_2) \uplus T_1) \uplus T_2$ are either subsumed by tuples from $(T_1 \bowtie T_2)$, are in $T_1 \setminus (T_1 \bowtie T_2)$, or are in $T_2 \setminus (T_1 \bowtie T_2)$. Thus, $t'.C \notin T_1.C \cap T_2.C \implies t' \in (T_1 \uplus T_2) \setminus (T_1 \bowtie T_2) \implies t' \in T_1 ⟗ T_2$.

Thus, all tuples from $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ are contained in the resulting table of $T_1 ⟗ T_2$.

Now that we have shown that tuples from $T_1 ⟗ T_2$ are contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ and vice versa, we have shown that $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ is an equivalent query to $T_1 ⟗ T_2$.

□

A.1.4 Proof of Lemma 15[Cross Product]. Given two tables T_1, T_2, each with columns C_{T_1}, C_{T_2} respectively and do not share any columns, and a constant column c, $T_1 \times T_2$ can be expressed by an equivalent query containing Outer Union, projection, and complementation. Specifically, $T_1 \times T_2$ is equivalent to query $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$.

Proof. Since T_1 and T_2 do not share any columns, the complementation operator cannot be applied to $T_1 \uplus T_2$. Thus, we project on all columns C_{T_1} and constant column c in T_1, and columns C_{T_2}, c in T_2. This way, T_1, T_2 now share all values in c and we can apply complementation on $\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2)$ since T_1, T_2. Thus, we iteratively apply complementation on all tuples from T_1 on all tuples from T_2 to form all tuples in $T_1 \times T_2$. Recall that in every tuple in $T_1 \times T_2$, every value in $t.C_{T_1}$ is from T_1 and every value in $t.C_{T_2}$ is from T_2. Therefore, every tuple in $T_1 \times T_2$ is contained in $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$ and every tuple in $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$ is contained in $T_1 \times T_2$, and so $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$ is an equivalent query to $T_1 \times T_2$. □

A.2 Set Similarity

Algorithm 3: Set Similarity

```
1  Input: $\mathcal{T} = \{T_1, T_2, \dots T_n\}$: set of data lake tables; $S$: Source Table;
     $\tau$: Similarity Threshold
2  Output: $\mathcal{T}' = \{T_1, T_2, \dots T_n\}$: a set of candidate tables with high
     syntactic overlap with $S$
3  $\mathcal{T}'_{\text{scores}} \leftarrow \{\}$ //Store a list of scores for each candidate table
4  for all $S$ columns $c \in C$ do
5      $\mathcal{T}_C$, overlapScores $\leftarrow$ SetOverlap($\mathcal{T}, c, \tau$)
6      $\mathcal{T}_C$, diverseOverlapScores $\leftarrow$ diversifyCandidates($\mathcal{T}_C, c, \tau$)
7      for all tables $T \in \mathcal{T}_C$ do
8          $\mathcal{T}'_{\text{scores}}[T]$ += diverseOverlapScores[$T$]
9  Order $\mathcal{T}'_{\text{scores}}$ by average diverseOverlapScores, in descending order
10 $\mathcal{T}' \leftarrow$ keys($\mathcal{T}'_{\text{scores}}$)
11 for all tables $T \in \mathcal{T}'$ do
12     alignedTuples $\leftarrow$ tuples in $T$ that contain $S$'s column values
13     if set overlap of $T$ values in alignedTuples with $S < \tau$ then
14         Discard $T$;
15     Remove $T$ if its values are contained in another table $T' \in \mathcal{T}'$
16     Rename $T$ columns to aligned $S$ columns
17 return $\mathcal{T}'$;
```

We find candidate tables with values that have high set overlap with those in a Source Table. As shown in Algorithm 3, we perform Set Similarity with an input set of data lake tables $\mathcal{T}$, the Source Table S, and a similarity threshold τ (Line 1), and output a set of candidate tables (Line 2). We first find a set of candidate tables where each table contains a column whose set overlap with a column from S (overlapScore) is above a specified threshold (Lines 4-8). This can be done efficiently with a system like JOSIE [70] that computes exact set containment or MATE [20] that supports multi-attribute joins. In addition, when finding tables with columns that have a high set overlap with columns in S, we call diversifyCandidates() (Line 6) to ensure that each candidate table not only has a high overlap with S, but also has minimal overlap with the previous candidates, shown in Diversify Candidates Algorithm 4.

Formally, given candidate table $T_i \in \mathcal{T}$ s.t. $i > 0$, the previous candidate table, T_{i-1}, and Source Table S, we diversify a set of candidate tables uses the following formula to rank the candidates, in descending order:

$$\text{diverseOverlapScore} = \frac{|T_i \cap S|}{|S|} - \frac{|T_i \cap T_{i-1}|}{|T_i|} \tag{12}$$

When finding diverseOverlapScore, we find the set overlap of T_i with S vs. the set overlap of T_i with the previous candidate T_{i-1}. This way, we arrange the set of candidate tables to ensure diversification of candidates.

After we find candidate tables for each column in the Source Table, we average over all overlap scores such that each is for a Source Table's column with which they share many values, and rank them in descending order of averaged scores (Line 9). With a set of candidate tables, we find tuples in each candidate table

Algorithm 4: Diversify Candidates

```
1  **Input**: $c$: column from Source Table; $\mathcal{T}_C = \{T_1, T_2, \ldots T_n\}$: set of
     candidate tables with columns having high overlap with $c$; $\tau$:
     Similarity Threshold
2  **Output**: $\mathcal{T}'_C = \{T_1, T_2, \ldots T_n\}$: a set of diverse candidate tables
3  $\mathcal{T}_{\text{scores}} \leftarrow \{\}$
4  **for** *all tables* $T \in \mathcal{T}_C$ **do**
5      $C \leftarrow$ column from $T$ with highest set overlap with $c$
6      $\text{Ind}_T \leftarrow$ index of $T$ in $\mathcal{T}_C$
7      **if** $\mathit{Ind}_T = 0$ **then**
8          Continue;
9      $C_{\text{prev}} \leftarrow$ column from $\mathcal{T}_C[\text{Ind}_T - 1]$ with highest set overlap
         with $c$//Get column from previous candidate table with high
         overlap with $c$
10     prevColOverlap $\leftarrow (C \cap C_{\text{prev}})/|C|$ //Set overlap with
         previous column
11     sourceColOverlap $\leftarrow (C \cap c)/|c|$ //Set overlap with column
         from Source table
12     **if** *sourceColOverlap* $< \tau$ **then**
13         Continue;
14     overlapScore $\leftarrow$ sourceColOverlap $-$ prevColOverlap
15     $\mathcal{T}_{\text{scores}}[T] \leftarrow$ overlapScore
16 Order $\mathcal{T}_{\text{scores}}$ by values in descending order
17 $\mathcal{T}'_C \leftarrow \mathcal{T}_{\text{scores}}$.keys
18 **return** $\mathcal{T}'_C$;
```

that contain column values from S. Within these aligned tuples, we check if each aligned column in a candidate table, with respect to a column in S, still has high set overlap (above threshold τ). If not, we remove them (Line 14). Next, we remove any subsumed candidate table, whose columns and column values are all contained in another candidate table (Line 15). We then rename each candidate tables' columns to the names of S's columns with which they align (Line 16), thus implicitly performing schema matching between S's columns and the columns from the candidate tables that have overlapping values with S's columns. Finally, we return the set of candidate tables.