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 a new measure of table similarity, called value similarity, to measure how close our reclaimed table is to a Source Table, a measure grounded in instance similarity used in data exchange. 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. However, there may be concerns around the origins of the tables that may need to be addressed before they are used for further tasks [1]. Considering the rise of generative AI models, such as ChatGPT [54], there is an increasing need to verify the AI outputs, for example, using values from data lake tables [64].

Example 1. *Suppose a user prompts ChatGPT[a] to "Show demographics of employees in Top US tech companies in 2021", for which ChatGPT returns two tables for gender and racial demographics (top blue tables in Figure 1). However, after consulting Microsoft's Diversity Report from 2021 [50], the user finds that ChatGPT's returned statistics for Microsoft (first row of top two tables) that contradict those reported by Microsoft itself (bottom green tables).*

[a]To create this and the next example we used the free research preview of GPT 3.5.

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 [40].

Example 2. *Suppose a user prompts ChatGPT with two applicants' information, and asks ChatGPT to return a list of companies for which they can receive offers and their starting salary at each company (Figure 2). These two applicants, one of whom is "James Smith" from Napa Valley, CA and the other being "Malik Brown" from Jackson, MS, have the same credentials. For "Malik Brown", the starting salaries is $20-35K less than those for "James Smith" for the same companies. Without confirming these starting salaries, the user could unknowingly use these tables with embedded biases and pollute further pipelines with these biases.*

We introduce 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 see from where data in a Source table may hve been derived o better understand the Source Table.

GPT Table 1:Gender Demographics

Company Name	% Male Employees	% Female Employees
Microsoft	61%	39%
Amazon	55%	45%
Google	67%	33%

GPT Table 2: Racial Demographics

Company Name	% White	% Asian	% Black	% Hispanic	% Other
Microsoft	54%	21%	13%	7%	5%
Amazon	54%	21%	12%	9%	4%
Google	51%	24%	7%	12%	6%

MS Table 1:Gender Demographics

Company Name	% Male Employees	% Female Employees
Microsoft	70.3%	29.7%

MS Table 2: Racial Demographics

Company Name	% White	% Asian	% Black	% Hispanic	% Other
Microsoft	48.7%	35.4%	5.7%	7%	3.2%

Figure 1: ChatGPT returns the top blue tables when prompted to "Show demographics of employees in Top US tech companies in 2021". The output for *Microsoft* contradicts data from Microsoft's diversity report (bottom green tables).

Unlike the well-known problem of Data Provenance [14, 15], we do not have prior knowledge of the query or tables that wer originally used to create a Source Table. Instead, we focus on recovering possible origins (tables) that confirm the data values and facts in a Source Table 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, 19, 38, 59, 67, 70]. In order to generalize to a data lake setting, we do not assume we know a complete and correct set of input tables. Rather we use an additional

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

Prompt:
My name is James Smith, and I am from Napa Valley, CA. I have a Bachelors degree in Electric Engineering, and worked in an Amazon warehouse for 4 years. I have learned Python and Java by myself, but never used it in a job. I am applying for an SDE role at tech companies. Given my credentials, please list in tabular form the companies I can get into, and my starting salary at each company given my credentials.

Table 1: GPT Response for James Smith

Company Name	Starting Salary (Approx.)
Microsoft	$110,000 - $130,000
Amazon	$100,000 - $120,000
Google	$120,000 - $140,000

Prompt:
My name is Malik Brown and I am from Jackson, MS. I have a Bachelors degree in Electric Engineering, and worked in an Amazon warehouse for 4 years. I have learned Python and Java by myself, but never used it in a job. I am applying for an SDE role at tech companies. Given my credentials, please list in tabular form the companies I can get into, and my starting salary at each company given my credentials.

Table 2: GPT Response for Malik Brown

Company Name	Starting Salary (Approx.)
Microsoft	$80,000 - $100,000
Amazon	$80,000 - $100,000
Google	$85,000 - $105,000

Figure 2: Two prompts provide ChatGPT with two applicants' information that differ in names and cities, and ask for a list of starting salaries at companies from which they can receive offers. For "Malik Brown" in Table 2, his starting salaries are $20-35K less than those of "James Smith" in Table 1.

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, 34, 55, 66, 72], 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 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, 25, 52, 53]. We also consider operations that have proven to be important in data integration and data fusion of incomplete data, namely subsumption and complementation [5].

Reclamation focuses on a 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 value imputation is appropriate for her task. Similarly, if the reclaimed table contains different values from the source, 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 reclaimed, 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 new Value Similarity score that is a principled extension of instance similarity used in the data exchange literature [3]), and show how it can be computed efficiently in our setting.
- We present an approximate Table Reclamation solution named Gen-T that 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 on real and synthetic data lakes, showing that Gen-T outperforms all baseline methods when reclaiming Source Tables, reclaiming 5X more Source Tables than the best-performing baseline. We perform an ablation study illustrating the sensitivity of our approach to erroneous data (data that cannot be reclaimed) and to incomplete data.
- 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.

2 OVERVIEW

A data scientist provides a *Source Table* that she would like to reclaim 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 which 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, 19, 34, 55, 59, 66, 67, 72] or *by-target* [70] 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. In addition, we assume the Source Table to have a (possibly multi-attribute) key, which can be found using existing mining techniques [9, 32]. This is a restriction, but it is made to make the instance comparison which is done often in the algorithm efficient. Without a key, instance similarity requires homomorphism checks with is NP-hard [3]. However, we do not assume tables in a data lake have keys or any foreign key relationships. In general, we do not assume that metadata is available for any tables (column names are included in examples only for clarity). To solve the problem of table reclamation, we 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 3 shows the pipeline of Gen-T, given a Source Table and outputting a (perhaps partially) 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. Gen-T thus uses an approximate algorithm to search for and integrate a set of originating tables. To efficiently retrieve a set of candidate tables from a data lake, Gen-T uses an existing, data-driven table discovery method that has no guarantees for this problem setting. Gen-T then prunes the candidate tables to a set of originating tables by computing

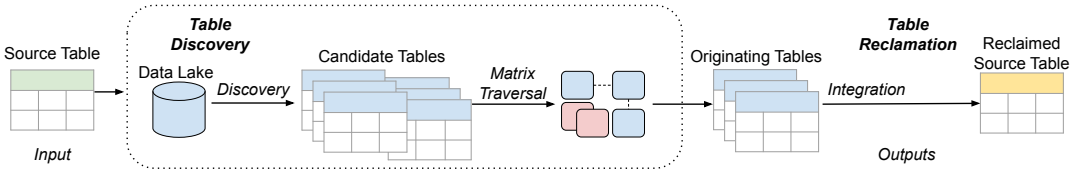


Figure 3: Gen-T Architecture. Given a Source Table, Gen-T finds its originating tables in the Table Discovery phase, produces the reclaimed Source Table from the Table Reclamation phase, and returns the originating tables and the reclaimed Source Table.

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
Smith	27
Brown	24
Wang	32

Table C:

Name	Gender
Smith	Male
Brown	Male
Wang	Male

Table D:

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

Resulting table of FD(A, B, C, D): $\hat{S}_1$

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): $\hat{S}_2$

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 4: Source Table (in green) contains applicants' information, such as ID, Name, Age, Gender, and Education Level. Tables A, B, C, D (in blue) are possible tables from which the Source Table's instances originated. Missing values and inconsistent values w.r.t. Source Table are depicted in yellow ('—') and red, respectively. Tables on the right (in yellow) are possible integrations of tables resulting from integration methods using Full Disjunction (FD) and outer join (⟗).

each candidate table's similarity with a given Source Table and simulating 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 3. *Suppose a user is using the top left, green table in Figure 4 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. We demonstrate the consequence of directly integrating all these tables, including Table C, by first projecting out columns that do not appear in the Source Table. The top right, yellow table is the integration result from the state-of-the-art full disjunction (FD) method [28, 36] and the bottom right table shows the result using one possible outerjoin order that may be learned by Auto-Pipeline (a by-target approach) [70]. 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 [11] along with Google Dataset Search [10], support keyword search over the metadata of tables [2, 46] and smaller scale web-tables [62, 63]. To support data-driven table discovery, systems [26, 53, 58, 75, 76] 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 [43, 69]. LSH Ensemble [76] makes use of approximate set containment between column values and supports set-containment search using LSH indexing. JOSIE [75] uses exact set containment to retrieve joinable tables that can be equi-joined with a column in the user's table. MATE [21] 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 [47, 58]. Using data (rather than metadata), a formal problem statement was first defined by Nargesian et al. [53] who presented a data-driven solution that leverages syntactic, semantic, and natural language measures. This problem was refined by SANTOS [35] to consider relationship semantics in addition to column semantics when retrieving semantically unionable tables. Most recently, Starmie [24] 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 [27] 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. [41] stitches unionable tables together, but does not support join augmentation of tables. More recently, ALITE [36] performs full disjunction (FD) [28] 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 [13, 20, 39, 49, 56, 60]. Our solution aligns schemas by rename the columns in the retrieved tables with the column of the source table that best matches. Entity matching [12, 16, 29, 31, 44, 45, 51, 71, 74] 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 [14, 15], 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 [67], 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 [30, 34, 55, 72], whereas others also consider the Select operator [8, 66]. For example, Ver [30] 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 [19, 59]. AutoPandas [4] performs transformation-by-example by synthesizing Pandas programs rather than SQL queries.

Auto-Pipeline [70] 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

To describe the Table Reclamation step of Gen-T's pipeline, we first, define a set of Table Operators (Section 4.1), with which we perform table integration to produce a reclaimed source table S.ection 4.2). Given a possible reclaimed table, we discuss how to evaluate the quality of the reclamation in Section 4.3.

4.1 Integration Operators

We now present operators that we will show are sufficient for integration, inspired by recent work on data lake integration [36]. First the unary operators.

- Projection(π): Project on specified columns of the table.
- Selection(σ): Select tuples that satisfy a specified condition.
- Subsumption(β) [28]: Given tuples t_1, t_2 in the same table, t_1 subsumes t_2 if for every attribute on which they are both non-null they have the same value, and t_1 contains one or more attributes with a non-null value where t_2 has nulls. Applying subsumption we remove t_2. Applying β on a table involves repeatedly applying subsumption and discarding the subsumed tuples (t_2).
- 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 a less restrictive condition than subsumption, 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 values on which they are both non-null. 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.

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 the Outer Union operator [57].

- Outer Union($\uplus$) [17]: 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. Note that when applied to tables with the same schema, outer union is the same as inner union.

To make reclamation search more efficient, we will make use the fact that 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 4 (Representative Operators). *Given two tables that contain no duplicate tuples, and no tuples that 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).*[1]

Using this set of operators $\bigoplus = \{\uplus, \sigma, \pi, \kappa, \beta\}$, we present an efficient algorithm to explore the integration space to reclaim a Source Table. Instead of traversing through all possible join paths

[1]The proof is included in our technical report [23].

in a space consisting of all tables, as in existing by-example and by-target work, we apply Outer Union and the unary operators with conditions based on a 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. Our table integration method, in which all tables in $\mathcal{T}$ contribute to S's reclamation, is depicted in Algorithm 1. We currently assume that all originating tables given as input to the Table Integration step have been integrated to contain S's columns. Later in Section 5, we describe Table Integration's previous step in the pipeline, Table Discovery. There, when we discuss how a set of candidate tables is refined to a set of originating tables, we will describe how we achieve this goal of ensuring that all originating tables contain S's columns.

Preprocessing: First, we perform pre-processing by projecting out columns not in S (π), and for tables that contain a key column from S, selecting tuples whose values are in the source key column (σ). Hence, we only keep columns and tuples that overlap with S (ProjectSelect() on Line 3). Next, we union all candidate tables that share the same schema (recall outer union on such tables is the same as inner union) (InnerUnion() on Line 4) to reduce the space of tables we need to explore. To prevent over-combining tuples that share nulls with tuples in S, for each table $T \in \mathcal{T}$, we find tuples in S that share the same key values and contain nulls in the same columns, and replace these nulls in T with a unique labeled non-null value (LabelSourceNulls() on Line 5). Before integration, we remove duplicate tuples, subsumed tuples (β), and take the resulting tuples of complementation (κ) (TakeMinimalForm() on Line 6).

Integration: We iterate though the tables formed in preprocessing, all of which contain the source key column. We first select a $T_i \in \mathcal{T}_\cup$ initially setting the reclamation set to empty (T_{result}). For each T_i, we outer union it with T_{result}. Next, we check if applying Complementation(κ) and Subsumption(β) on T_{result} results in a table that is more similar to S. For this we use the function evaluateSimilarity() defined in the next subsection. This lets us check if these operators are over-combining tuples (e.g., removing a subsumed tuple that is identical to a tuple in S) and decreasing the number of values shared with S. We do this iteratively for all tables from the input set. After iterating through all tables from the input set, we revert the previous labeling of shared nulls with S (RemoveLabeledNulls() on Line 14). To ensure that the resulting integrated table (T_{result}) has the same schema as S, we add null columns in T_{result} for every column it does not share with S (Line 16). Finally, we return the resulting integration as a possible reclaimed table.

4.3 Value Similarity Score

Given a possible reclaimed table, we compare it with the Source Table to see how close they are. The problem of comparing database instances is prevalent in many applications such as analyzing how a dataset has evolved over time (e.g., data versioning), evaluating data cleaning solutions (e.g., compare a clean instance produced by a data repair algorithm against a gold standard), or comparing solutions generated by data exchange or transformation systems. A similarity score requires the computation of a mapping between

Algorithm 1: Table Integration

```
1  Input: 𝒯 = {T_1, T_2, ... T_n}: tables to integrate; S: the Source Table
2  Output: T_result: integration result
3  𝒯 ← ProjectSelect(𝒯, S) //σ, π (T ∈ 𝒯) on columns, keys in S
4  𝒯_∪ ← InnerUnion(𝒯) //Inner Union tables with shared schemas
5  𝒯_∪ ← LabelSourceNulls(𝒯_∪) //Label Nulls shared with Source Table
6  𝒯_∪ ← TakeMinimalForm(𝒯_∪) //Apply β, κ on each table
7  T_⊎ ← ∅
8  for T_i ∈ 𝒯_∪ do
9      T_⊎ ← T_⊎ ⊎ T_i //Apply outer union ⊎
10     if evaluateSimilarity(κ(T_⊎), S) ≥ evaluateSimilarity(T_⊎, S)
         then
11         T_⊎ ← κ(T_⊎)                    //Apply complementation κ
12     if evaluateSimilarity(β(T_⊎), S) ≥ evaluateSimilarity(T_⊎, S)
         then
13         T_⊎ ← β(T_⊎)                    //Apply subsumption β
14 T_result ← RemoveLabeledNulls(T_⊎)
15 if T_result has fewer columns than S then
16     add null columns in T_result for each column ∈ S \ T_result
17 Output T_result
```

the tuples across instances, which can be used to explain the result. The most general measures rely on homomorphism checking and are NP-hard [22]. However, since the source table has a key, we will assume that tuples map (i.e., are *aligned tuples*) iff they share the same values on the key attributes.

Our *value similarity score* is a generalization of *instance similarity* define by Alexe et al. [3] which has been widely used in the integration literature. Instance similarity was developed to quantify the preservation of data associations when source data is exchanged into target database [3]. Instance similarity relies on computing homomorphisms (as it is designed for data exchange where keys are not assumed). Considering two relations with the same schema, if two tuples are mapped, they define the *tuple similarity* as the ratio of the number of values that are shared over the size (cardinality) of the tuples. (Note that since this is data exchange, two tuples cannot be mapped if they disagree on any non-null values.) In our setting, we can map tuples that differ on their non-null values. Hence, we define an *error-aware tuple similarity* that penalizes for mismatching (erroneous) values.

Definition 5. *Given two tuples s and t with the same schema and same number of non-key attributes, n, that share the same key value. Let $\alpha(s,t)$ be the number of non-key attributes on which s and t share the same value and $\delta(s,t)$ be the number of non-key attributes on which s and t have different values and t is a non-null value. Then the* **error-aware tuple similarity** *is:*

$$E(s,t) = (\alpha(s,t) - \delta(s,t))/n \tag{1}$$

Note the **tuple similarity** *defined by Alexe et al. [3] is $\alpha(s,t)/n$.*

Since a tuple can map to more than one tuple, Alexe et al. define *instance similarity* using the maximum tuple similarity score and we do the same in defining *value similarity* but we use error-aware tuple similarity rather than tuple similarity. For completeness (and since in our experiments we use both the established measure *instance similarity* and our new *value similarity*), we define both.

Definition 6. *Let S be a source table (with key K) and T a possible reclaimed table with the same schema and same number of non-key*

attributes n. Note that T need not satisfy the key constraint. For a tuple $s \in S$, let $m(s) = \{t \in T | s[K] = t[K]\}$. Then the **instance similarity** *of S and T is:*

$$\text{Instance Similarity} = \frac{\sum_{s \in S} max_{t \in m(s)}(\alpha(s,t)/n)}{|S|} \quad (2)$$

The **value similarity** *of S and T is (for a score in range[0,1]):*

$$\text{Value Similarity} = \frac{\frac{1}{2} * \sum_{s \in S} max_{t \in m(s)}(\alpha(s,t) - \delta(s,t) + n)/n}{|S|} \quad (3)$$

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

Example 7. *Consider Source Table S (with key column "ID") from Figure 4 (top-left green table), and two possible reclaimed tables, $\hat{S}_1$ (top-right yellow table) and $\hat{S}_2$ (bottom-right yellow table). If there were multiple tuples for a single key, we would find the one with the highest (error-aware) tuple similarity. Notice the instance similarity score of S and $\hat{S}_1$ is higher than of S and $\hat{S}_2$.*

$$\hat{S}_1 : t_0 = 3/4, t_1 = 4/4, t_2 = 3/4 \rightarrow (3/4 + 4/4 + 3/4)/3 = 0.833$$

$$\hat{S}_2 : t_0 = 2/4, t_1 = 4/4, t_2 = 3/4 \rightarrow (2/4 + 4/4 + 3/4)/3 = 0.75$$

However, we want to favor $\hat{S}_2$ that contains nullified (unknown) values (one matching the source correctly and two missing values), over $\hat{S}_1$, which has reclaimed a possibly erroneous value for a source null. The Value Similarity score is:

$$\hat{S}_1 : t_0 = (3-1)/4, t_1 = 4/4, t_2 = 3/4$$

$$\rightarrow \frac{\frac{1}{2} * ((3-1+4)/4 + ((4+4)/4) + ((3+4)/4))}{3} = 0.875$$

$$\hat{S}_2 : t_0 = 3/4, t_1 = 4/4, t_2 = 3/4$$

$$\rightarrow \frac{\frac{1}{2} * ((3+4)/4 + ((4+4)/4) + ((3+4)/4))}{3} = 0.917$$

This way, $\hat{S}_2$ has a higher similarity with S than $\hat{S}_1$.

Consider another example in which the Source Table only has tuple t_0; tuple t_0 in $\hat{S}_1$ is [0, –, –, –, –]; t_0 in $\hat{S}_2$ is [0, 'Smith', 27, 'Male', 'Masters']. In this case, $\hat{S}_1$ has a Value Similarity score of $\frac{\frac{1}{2} * ((1-0+4)/4)}{1} = \frac{5}{8} = 0.625$. $\hat{S}_2$ would have a Value Similarity score of $\frac{\frac{1}{2} * ((2-2+4)/4)}{1} = \frac{1}{2} = 0.5$. $\hat{S}_1$, even though it shares fewer values with S, has a higher Value Similarity score than $\hat{S}_2$ that contains erroneous values.

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

Definition 8 (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 an $\hat{S}$ with the maximum Value Similarity Score to S.*

5 TABLE DISCOVERY

We now turn to Table Discovery (first set in Figure 3). Our goal is to find a good set of originating tables from which we can effectively reclaim a source table. First, we discover a set of candidate tables in Section 5.1. Then, we discuss a novel methodology that refines this set into a set of originating tables in Sections 5.2 and 5.3.

5.1 Candidate Table Retrieval

Discovering a set of originating tables 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, 25, 52, 73]. However, we can any use 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. 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 [75] that computes exact set containment or MATE [21] 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 other candidates (full algorithm is included in the technical report [23]). This is especially important since data lakes tend to have multiple versions of the same tables [37, 61]. For example, one study (JOSIE [75]) 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. Note that for private data lakes, although there are still duplicate columns due to lakes containing multiple versions of the same table [61], the percentage of duplicate columns may not be as high. By diversifying candidates, each candidate for a column in Source Table S may overlap with different values in S. We illustrate this with an example:

Example 9. *Suppose we have the Source Table from Figure 4. 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 related columns. To do so, we find all tuples in a candidate table that share column values with 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 a set of diversified candidate tables, we could potentially enter the table integration phase (Section 4.2) using all candidate tables as an originating set of tables. 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 a maximum set of aligned tuples with respect to the Source Table. To do so, we emulate the table integration process without performing the expensive integration 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 (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. To align tuples based on shared key values with the Source Table, we need to ensure that each candidate table contains a key column of the Source Table. If it does not, we greedily find a best way to join it with candidates that include the source key. We use standard join cardinality estimation to find a path that covers the most source key values and is as close to functional as possible (this procedure is denoted as Expand()). [2] This way, all tables can align its tuples with the Source Table using key values.

To capture tuple alignment, we represent each candidate table and its aligned tuples with the Source Table in the form of a matrix. 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.

Next, we simulate table integration by applying the logical Or on the matrices, which takes the maximum value at each position. When we combine values in matrices 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): $m_r[j] = max(t_1[j], t_2[j])$. 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. Thus, both table and matrix integrations maximally combine the tuples such that non-null values can replace a null value at the same index.

Putting it all together, we show the matrix initialization and traversal in Algorithm 2. Given a set of candidate tables $\mathcal{T}$ and Source Table S, we first ensure that each candidate table contains a key column from S by joining candidate tables that do not share a key column with S with those that do (Expand() on line 3). This way, we align tuples in each candidate table with respect to S and initialize each candidate table's matrix representation (Line 4). Then, we traverse over the matrices and perform the logical Or operator to combine a pair of matrices in Combine() (Line 10). 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 14), including the start (Lines 5-6), 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 8), or the percentage of 1's in the resulting matrix converges (Lines 18-19). 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 table integration. If a candidate table that was joined with other candidates to contain a key column from S (from Expand() from line 3) becomes an originating table, we include its expanded form in the returned set.

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. 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.

Thus, we need to distinguish between nullified and erroneous aligned tuples in the matrix representation (Line 4 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

[2] The Expand algorithm is included in the technical report [23].

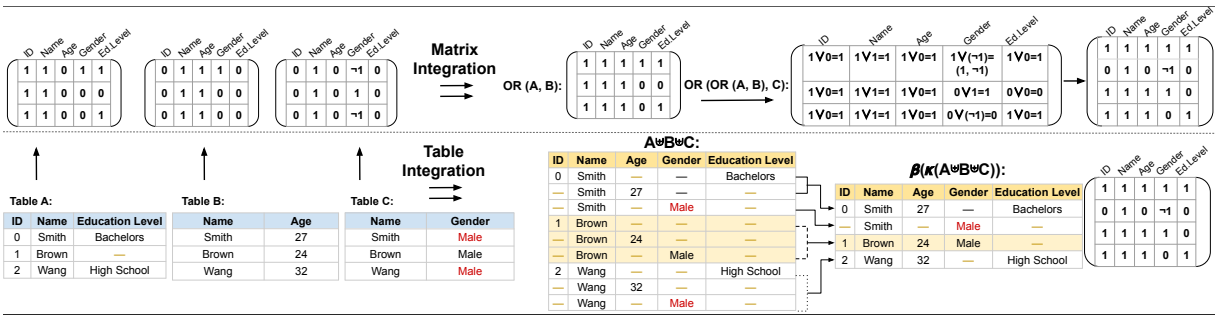


Figure 6: Matrix initialization and integration of tables A, B, C given the Source Table from Figure 4 simulate their table integration. The result of matrix integration is equivalent to the matrix representation of the table integration result.

Algorithm 2: Matrix Traversal

1 **Input**: $\mathcal{T} = \{T_1, T_2, \dots T_n\}$: set of candidate tables; S: Source Table
2 **Output**: $T_{\text{orig}} = \{T_1, T_2, \dots T_i\}$: refined set of originating tables
3 $\mathcal{T} \leftarrow$ Expand($\mathcal{T}, S$) //join tables without source key
4 $\mathcal{M} \leftarrow$ MatrixInitialization($\mathcal{T}$), //Initialize Matrices of S shape
5 $T_{\text{start}} \leftarrow$ GetStartTable($\mathcal{M}$)
6 prevCorrect = mostCorrect $\leftarrow$ evaluateSimilarity(T_{start})
7 $T_{\text{orig}} \leftarrow [\,]$
8 **while** $|T_{\text{orig}}| < |\mathcal{T}|$ **do**
9 **if** T_{orig} **then**
10 $M_c \leftarrow$ Combine(T_{orig}) //Iteratively combine each pair of consecutive matrices
11 prevCorrect = mostCorrect; nextTable = $\perp$
12 **for** *all tables* $T \in \mathcal{T}s.t.T \notin T_{\text{orig}}$ **do**
13 $M_c \leftarrow$ Combine(M_c, T)
14 percentCorrectVals $\leftarrow$ evaluateSimilarity(M_c)
15 **if** *percentCorrectVals* > *mostCorrect* **then**
16 mostCorrect $\leftarrow$ percentCorrectVals
17 nextTable $\leftarrow$ T
18 **if** *mostCorrect* = *prevCorrect* **then**
19 Exit, //Integration did not find more of S's values
20 $T_{\text{orig}} = T_{\text{orig}} \cup$ nextTable
21 **return** T_{orig};

index, and -1 if they contain contradicting non-null values at the same index (shown in Figure 6). 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:

$$M[i,j] = \begin{cases} 1 & \text{if } S[i,j] = T[i,j] \\ 0 & \text{elif } S[i,j] \neq \perp \wedge T[i,j] = \perp \\ -1 & \text{otherwise} \end{cases} \quad (4)$$

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 at the same index, the resulting integration would contain both tuples. Thus, we need to keep both tuples from the matrices if they contain different non-0 values at the same index. We re-define Combine() (Line 10) between two matrices, given tuples t_1, t_2 at the same row index accordingly.

$$\text{Combine}(t_1, t_2) = \begin{cases} \text{Return } t_1, t_2 & \text{if } t_1[j] \neq t_2[j] \neq 0 \text{ for column } j \\ \text{OR}(t_1, t_2) & \text{otherwise} \end{cases} \quad (5)$$

This way, we keep 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. The amended Combine() function is illustrated in Example 10.

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 dictionary key, and the list of aligned tuples in the resulting matrix with respect to a tuple in the Source Table as values.

Example 10. *Given the Source Table from Figure 4, Figure 6 shows the result of integrating tables A, B, and C and their matrix representations. We start with matrix A with the largest number of correct values. Integrating matrices A and B produces more correct values after taking logical OR's of 0's and 1's (from the Combine() function). When combining its resulting matrix with matrix C, we find a (1) and (¬1) in the first tuple for column "Gender". In this case, we keep both tuples from OR(A, B) and C. For all other value-pairs, we take the logical OR. When we integrate tables A, B, C, we also find tuples for "Name"=Smith to be separate, since we check if we over-combine tuples and replace correct nulls (Algorithm 1). As a result, the matrix representation of this table integration result is exactly the output of the matrix integration, and so table integration is simulated by integration of matrix representations.*

For evaluateSimilarity() of the start (Lines 5-6) and resulting matrices (Line 14), 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. To find the Value Similarity score (Equation 3) between a tuple t in a resulting matrix that shares a key value with a Source tuple s, we set $\alpha(s, t)$ to be the number of non-key attributes for which tuple t has (1)'s, representing shared values between t and s. For $\delta(s, t)$, we take the number of non-key attributes for which t has (-1)'s, representing different, non-null values. 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.

6 EXPERIMENTS

We now present evaluations on benchmarks with tables containing real instances (Source Tables) from the well-known TPC-H Benchmark [65] and also the T2D Gold [68] Benchmark. We use these benchmarks along with tables from a real data lake. For the data lakes, we use the large SANTOS benchmark [35] and a sample of WDC [42]. There are no existing solutions for reclamation. Hence, our baselines are versions of related techniques that we modify to solve the reclamation problem. 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 that Gen-T generalizes 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
TP-TR Large	32	244	1M	3.9K
SANTOS Large +TP-TR Med	11K	122K	7.7K	11K
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: We use the 8 tables from the TPC-H benchmark [65], 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 each, 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. For the majority of the experiments, the amount of nulls (respectively, erroneous values) is 50% meaning that we randomly take 50% of each table's values and replace them with nulls (respectively, replace them with different new strings). In an ablation study (Section 6.2 last two paragraphs), we vary (indepedently) the amount of nulls/erroneous values from 10 to 90%. 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.

The Source Tables for the TP-TR benchmarks are created using all 8 of the original (unmodified) TPC tables over which we randomly generated 26 queries each having a subset of operators $\{\pi, \sigma, \bowtie, \ltimes, \overline{\ltimes}, \cup, \uplus\}$. 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. We ran the same queries on each TP-TR benchmark to 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.

SANTOS Large +TP-TR Med Benchmark: 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 [35]. In doing so, we evaluate how well Gen-T prunes a potentially large set of candidate tables retrieved from a large data lake to a smaller set of originating tables that can more accurately reclaim a Source Table when integrated. We use the same Source Tables as for TP-TR Med.

T2D Gold Benchmark: In addition, we explore the real-world application of our method with the T2D Gold Benchmark [68], 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. We do not have prior knowledge of whether or not any of these 515 tables can be "reclaimed" as a Source,. Thus, we iterate through each of the 515 tables as potential sources.

WDC Sample +T2D Gold: To further assess the effectiveness of Gen-T, we embed T2D Gold tables into a sample of the WDC web table corpus [42], which contains 15K relational web tables. This way, we can examine how well Gen-T prunes a large set of candidate tables found from a large table corpus to a small set of originating tables that can be integrated to reclaim a Source Table.

6.1.1 Baselines. We compare Gen-T against the current state-of-the-art for *by-target synthesis*, Auto-Pipeline [70], and the state-of-the-art for table integration, ALITE [36] where we have modified both to be solutions for the reclamation problem.

Auto-Pipeline has a similar framework to our problem in discovering the integration (query or pipeline) that reclaims a 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 query-search and deep reinforcement learning approaches, but since we propose an unsupervised approach, we use the query-search variation as our baseline. Auto-Pipeline's code implementation is not openly available, so we adopted an open reimplementation of their search approach [61], which adapts the framework in Foofah [33], and, for a fair comparison, revised their set of table operators to only contain table operators that Gen-T considers ($\{\sigma, \pi, \cup, \bowtie, \ltimes, \overline{\ltimes}\}$). 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 to match the Source Table before the table integration. Hence, ALITE-PS, also includes unary operators to make its performance more comparable with Gen-T. ALITE without projectand selectis much slower as it is creating a larger integration result, hence ALITE-PS is a fairer comparison in terms of performance.

For each of baseline (Auto-Pipeline*, ALITE, and ALITE-PS), 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. We know what subset of tables from the 8 original tables were used to create the 26 Source Tables, so, we know that a perfect reclamation 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 a 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 many of its values we can re-produce. In an aligned tuple in an Output table with respect to a Source Table, it contains erroneous values if there is a non-null, different value at a given column compared to a value in the same column in the Source Table. Similarly, it contains nullified values if it contains a null value in a column where the Source tuple contains a non-null value (see Section 4.3).

Precision and Recall: Consider a Source Table S and reclaimed table from a method $\hat{S}$. From the measure Tuple Difference Ratio (TDR) introduced by Khatiwada et al. [36], 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 (Inst-Div.), which is the inverse of Instance Similarity, introduced by Alexe et al. [3] (see Equation 2 in Section 4.3):

$$\text{Inst-Div.} = 1 - \text{Instance Similarity} \tag{6}$$

Conditional KL-divergence: Finally, we want to capture how many erroneous values there are in 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 [18, 48], and also add a penalization for erroneous values, such that the score is higher (diverges more) for reclaimed tables containing erroneous values as opposed to nulls in their aligned tuples with the Source Table (for more information, please visit the technical report [23]).

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 relevant tables from the large data lake using Starmie [24], 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.

Some baselines timeout for most Source Tables as the sizes and/or number of tables increase across the benchmarks. 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-PS) by 11-25% in Recall and by 48-56% 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.

Method	TP-TR Small Recall	TP-TR Small Precision	TP-TR Med Recall	TP-TR Med Precision	SANTOS Large +TP-TR Med Recall	SANTOS Large +TP-TR Med Precision	TP-TR Large Recall	TP-TR Large 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	**0.954**	**0.799**	**0.976**	**0.867**	**0.976**	**0.867**	**0.971**	**0.807**

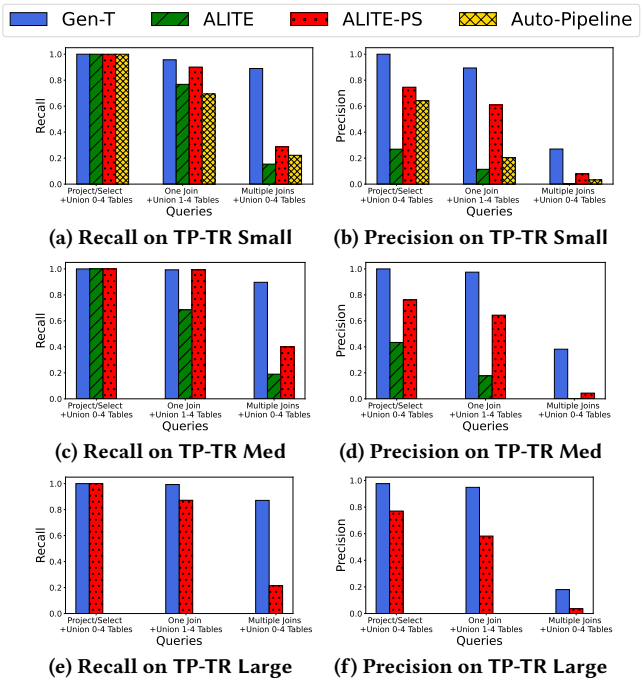
Method	TP-TR Small Inst-Div.	TP-TR Small D_{KL}	TP-TR Med Inst-Div.	TP-TR Med D_{KL}	SANTOS Large +TP-TR Med Inst-Div.	SANTOS Large +TP-TR Med D_{KL}	TP-TR Large Inst-Div.	TP-TR Large D_{KL}
ALITE	0.095	1.332	0.100	35.831	—	—	—	—
ALITE w/ integrating set	0.086	1.197	0.085	36.348	0.085	36.348	—	—
ALITE-PS	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**	**0.165**	**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.

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.[3]

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 perform 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.

Effectiveness of Pruning in Gen-T: For a more detailed analysis, we analyze Recall, Precision, and F1-Scores of Gen-T and baseline, ALITE-PS, on TP-TR Med for each of its 26 Source Tables. [4] Note that ALITE-PS directly integrates a set of candidate tables, whereas Gen-T first prunes the set of candidate tables to a set of originating tables before performing table integration. Gen-T outperforms ALITE-PS in Precision for all Source Tables, and outperforms ALITE-PS in Recall for 24 of 26 total Source Tables. This shows that ALITE-PS, which directly integrates candidate tables without pruning, reclaims more Source Tuples than Gen-T, which does include a pruning step, for only a few Source Tables. Also, Gen-T outperforms ALITE-PS in F1-Score for all Source Tables, showing that even if ALITE-PS outperforms Gen-T in Recall, it does not impact the F1-Score.



(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.

Tuning % Erroneous vs. Nullified Values: Lastly, we analyze Gen-T's performance when run on data lake tables with different number of erroneous and nullified values in TP-TR Med tables (Figure 8). So far, TP-TR Med tables have 50% erroneous values in erroneous versions and 50% nulls in nullified versions (intersection point on the graph where Gen-T has 0.867 Precision). Now, we tune the percentage of values replaced with non-null, random strings (blue line in Figure 8) in erroneous versions, while the nullified versions always contain 50% nulls. Similarly, we tune the percentage

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

[4] All accompanying graphs can be found in the technical report [23].

of values replaced with nulls (red line in Figure 8) while holding the erroneous versions constant. For Gen-T to produce a perfect reclamation of a Source Table, it should only have tables with injected nulls in its set of originating tables so that these nulls can be replaced with correct values during table integration.

As data lake tables have more values replaced with erroneous values (blue line), Gen-T is more likely to contain tables with nullified tuples in its set of originating tables, which results in an integrated table with higher precision. On the other hand, as we tune the percentage of values replaced with nulls (red line), precision decreases. As more nulls are injected, these tables also have fewer correct values. Gen-T is thus more inclined to have originating tables with 50% erroneous values, or 50% correct values, leading to a final integration with lower precision.

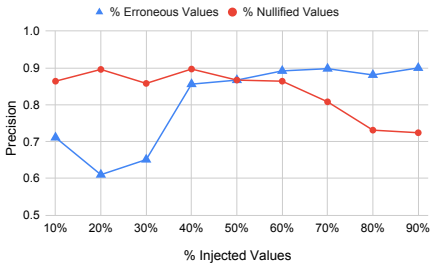


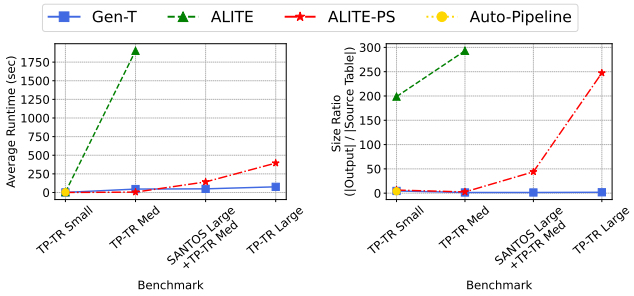
Figure 8: Gen-T Precision as TP-TR Med has different % of Erroneous (blue triangles) and Nullified Values (red circles).

6.3 Scalability

Figure 9 shows the scalability of all methods across benchmarks as the number and/or size of tables grows. Figure 9(a) reports average runtimes for all methods across all four benchmarks, starting from ingestion of the candidate tables. For Gen-T, this time includes the time it takes to prune the set of candidate tables to a set of originating tables, that is integrated to produce an output table. For other methods, this time only includes integration. 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 as 30min for TP-TR Small, 7hrs for TP-TR Med and SANTOS Large +TP-TR Med and 24hrs for TP-TR Large.

Gen-T has a more consistent runtime across all benchmarks compared to all baselines. Gen-T is 3X faster compared to Auto-Pipeline* on TP-TR Small. On TP-TR Med, Gen-T is 40X faster than ALITE and on TP-TR Large, Gen-T is 5X faster than ALITE-PS. Thus, pruning candidate tables to originating tables seems to cut the cost of integration, a prevalent issue as shown by the baselines.

In Figure 9(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). Output sizes for Gen-T remain consistent across all benchmarks (1.4-4.5X larger than the average Source Table size). This trend largely accounts for the higher precision of Gen-T, since its output tables mostly consist of Source tuples. In contrast, output sizes for ALITE, ALITE-PS, and Auto-Pipeline* are 200-300X, 2.5-250X, and 4X larger than Source Tables' sizes, respectively. Thus, Gen-T's runtimes and output sizes remain consistent across benchmarks of different sizes.



(a) Average Runtimes in sec. **(b) Average Ratio of Output Size**

Figure 9: Scalability via Average Runtime (sec) and Ratio of Output Sizes to Source Table sizes over all TP-TR Benchmarks

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 515 tables as potential Source Tables. Gen-T is able to successfully reclaim 3 Source Tables from an integration of multiple tables (5-6 tables), such that the outputs have 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.

We then run experiments on a data lake consisting of both T2D Gold and WDC Sample tables. This way, we evaluate how well the methods perform when a set of candidate tables returned from Set Similarity may contain irrelevant or misleading tables from the WDC Sample benchmark. For 33 of the common sources for which all methods have non-empty, reasonably-sized output tables, Gen-T outperforms the baselines, even having perfect precision of 1.0. In contrast, baselines that integrate all candidate tables produce tables that contain many additional tuples, leading to much lower precision scores (best-performing baseline has precision 0.796).

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 with regard to Source Tables. To allow users to add preferences over columns they wish to reclaim, we will also add weights in our Value Similarity scoring function. We will also 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] Leon Bornemann, Tobias Bleifuß, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2020. Natural Key Discovery in Wikipedia Tables. In *WWW*. ACM / IW3C2, 2789–2795.

[10] 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.

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

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

[13] 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.

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

[15] 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

[16] 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).

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

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

[19] 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.

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

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

[22] Ronald Fagin, Phokion G. Kolaitis, Alan Nash, and Lucian Popa. 2008. Towards a theory of schema-mapping optimization. In *PODS*, Maurizio Lenzerini and Domenico Lembo (Eds.). ACM, 33–42. https://doi.org/10.1145/1376916.1376922

[23] 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

[24] 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.

[25] 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.

[26] 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.

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

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

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

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

[31] 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.

[32] Lan Jiang and Felix Naumann. 2020. Holistic primary key and foreign key detection. *J. Intell. Inf. Syst.* 54, 3 (2020), 439–461.

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

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

[35] 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*.

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

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

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] Microsoft. 2021. https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RWNrak

[51] 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.

[52] 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.

[53] 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.

[54] OpenAI. 2023. https://chat.openai.com/ Free Research Preview.

[55] 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.

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

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

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

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

[64] Nan Tang, Chenyu Yang, Ju Fan, Lei Cao, and Alon Halevy. 2023. VerifAI: Verified Generative AI. *CoRR* abs/2307.02796 (2023).

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

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

[67] 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.

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

[69] 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.

[70] 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.

[71] 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.

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

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

[74] 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.

[75] 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.

[76] 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 TABLE RECLAMATION

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 ($\bowtie$):*

$$T_1 \bowtie T_2 = \sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2))) \tag{7}$$

Lemma 13 (Left Join). *Left Join ($\rtimes$) [28]:*

$$T_1 \rtimes T_2 = \beta((T_1 \bowtie T_2) \uplus T_1) \tag{8}$$

Lemma 14 (Outer Join). *Full Outer Join ($\rtimes\ltimes$) [28]:*

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

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.*

$$T_1 \times T_2 = \kappa(\pi((T_1.C, c), T_1) \uplus \pi((T_2.C, c), T_2)) \tag{10}$$

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 $\sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$.

Proof. We first prove that all tuples in $T_1 \bowtie T_2$ are contained in $\sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$. 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. $t.C \neq \bot$.

When applying $\beta(\kappa(T_1 \uplus T_2))$, only tuples with common non-null values $T_1.C_i = T_2.C_i \neq \bot$ 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 $t \in \sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$.

Next, we show that all tuples in $\sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$ are found in $T_1 \bowtie T_2$. Let tuple $t' \in \sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$. 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 $\sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$ and vice versa, and so $T_1 \bowtie T_2$ is an equivalent query to $\sigma(T_1.C = T_2.C \neq \bot, \beta(\kappa(T_1 \uplus T_2)))$. □

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 , $T_1 \rtimes T_2$ can be expressed by an equivalent query containing Outer Union and subsumption. Specifically, $T_1 \rtimes T_2$ is equivalent to query $\beta((T_1 \bowtie T_2) \uplus T_1)$.

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

Let tuple $t \in T_1 \rtimes T_2$. 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 $T_1 \rtimes T_2$ 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 $T_1 \rtimes T_2$.

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 $(T_1 \bowtie T_2) \subseteq (T_1 \rtimes T_2)$, $t' \in (T_1 \rtimes T_2)$.
(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, $t'.C \notin T_1.C \cap T_2.C \implies t' \in T_1 \setminus (T_1 \bowtie T_2) \implies t' \in T_1 \rtimes T_2$.

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

Now that we have shown that tuples from $T_1 \rtimes T_2$ 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 $T_1 \rtimes T_2$. □

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, $T_1 \rtimes\ltimes T_2$ can be expressed by an equivalent query containing Outer Union and subsumption. Specifically, $T_1 \rtimes\ltimes T_2$ 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 $T_1 \rtimes\ltimes T_2$ is contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$:

Let tuple $t \in T_1 \rtimes\ltimes T_2$. 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) $t.C \in T_1.C \setminus T_2.C \implies t \in (T_1 \rtimes T_2) \setminus (T_1 \bowtie T_2)$. 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 $(T_1 \rtimes T_2) \setminus (T_1 \bowtie T_2)$. 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) $t.C \in T_2.C \setminus T_1.C \implies t \in (T_2 \rtimes T_1) \setminus (T_1 \bowtie T_2)$. 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 $T_1 \bowtie T_2$, $t \in (T_2 \rtimes T_1) \setminus (T_1 \bowtie T_2)$. Thus, $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

Thus, all tuples from $T_1 \rtimes T_2$ 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 $T_1 \rtimes T_2$. 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 $(T_1 \bowtie T_2) \subseteq (T_1 \rtimes T_2)$, $t' \in (T_1 \rtimes T_2)$.

(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, $t'.C \notin T_1.C \cap T_2.C \implies t' \in (T_1 \uplus T_2) \setminus (T_1 \bowtie T_2) \implies t' \in T_1 \rtimes T_2$.

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

Now that we have shown that tuples from $T_1 \rtimes T_2$ 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 $T_1 \rtimes T_2$.

□

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: 𝒯 = {T_1, T_2, ... T_n}: set of data lake tables; S: Source Table;
     τ: Similarity Threshold
2  Output: 𝒯' = {T_1, T_2, ... T_n}: a set of candidate tables with high
     syntactic overlap with S
3  𝒯'_scores ← {} //Store a list of scores for each candidate table
4  for all S columns c ∈ C do
5      𝒯_C, overlapScores ← SetOverlap(𝒯, c, τ)
6      𝒯_C, diverseOverlapScores ← diversifyCandidates(𝒯_C, c, τ)
7      for all tables T ∈ 𝒯_C do
8          𝒯'_scores[T] += diverseOverlapScores[T]
9  Order 𝒯'_scores by average diverseOverlapScores, in descending order
10 𝒯' ← keys(𝒯'_scores)
11 for all tables T ∈ 𝒯' do
12     alignedTuples ← tuples in T that contain S's column values
13     if set overlap of T values in alignedTuples with S < τ then
14         Discard T;
15     Remove T if its values are contained in another table T' ∈ 𝒯'
16     Rename T columns to aligned S columns
17 return 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 [75] that computes exact set containment or MATE [21] 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:

$$\text{diverseOverlapScore} = \frac{|T_i \cap S|}{|S|} - \frac{|T_i \cap T_{i-1}|}{|T_i|} \tag{11}$$

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 5: Expand

Input: $\mathcal{T} = \{T_1, T_2, \ldots T_n\}$: set of candidate tables; k: Source Table's key column(s)
Output: $\mathcal{T}_k = \{T_1, T_2, \ldots T_n\}$: set of candidate tables that all now contain Source Table's key column
//Initialize Graph
nodes = candidate tables
edges = tables that have joinable columns
edge Weights = value overlap of joinable columns
start_nodes = {candidate tables that do not contain k}
end_nodes = {candidate tables that contain k}
for *each start in start_nodes* **do**
    //Initialize sets and dictionaries
    visited ← set() //visited nodes
    node_weights ← {} //maximum weights before each node
    descendant ← {} //best child for each node
    max_weight ← 0 //Maximum weight found so far
    end_node ← None //end node for a given start node
    //Initialize *stack* for DFS
    stack ← *stack* + *start*
    visited ← *visited* + *start*
    while *stack is not empty* **do**
        node ← *stack.pop*()
        unvisited_children ← children of *node* not in *visited*
        //Current child's weight is the weight of the path so far, including the edge weight between node and current child
        for *each child in unvisited_children* **do**
            child_weight ← *node_weights*[*node*] + *weight*(*node*, *child*)
            //update descendant if it contains the maximum sum of weights so far
            if *child_weight* > *node_weights*[*child*] **then**
                node_weights[*child*] ← *child_weight*
                descendant[*child*] ← *node*
                if *child is in end_nodes* **then**
                    if *child_weight* > *max_weight* **then**
                        //child has k and the maximum weighted path so far
                        max_weight ← *child_weight*
                        end_node ← *child*
                stack ← *stack* + *child*
            visited ← *visited* + *child*
    if *end_node is not null* **then**
        //reconstruct path with maximum sum of weights by reversing path, starting with found end node
        path ← []
        current_node ← *end_node*
        while *current_node is in descendant* **do**
            path ← *path* + *current_node*
            current_node ← *descendant*[*current_node*]
        path.reverse()
        table ← first node in *path*
        for *each join_table in path*[1 :] **do**
            table ← *join*(*table*, *join_table*)
        $\mathcal{T}_k \leftarrow \mathcal{T}_k$ + *table*

Algorithm 4: Diversify Candidates

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; τ: Similarity Threshold
Output: $\mathcal{T}'_C = \{T_1, T_2, \ldots T_n\}$: a set of diverse candidate tables
$\mathcal{T}_{\text{scores}} \leftarrow \{\}$
for *all tables* $T \in \mathcal{T}_C$ **do**
    $C \leftarrow$ column from T with highest set overlap with c
    $\text{Ind}_T \leftarrow$ index of T in $\mathcal{T}_C$
    if $Ind_T = 0$ **then**
        Continue;
    $C_{\text{prev}} \leftarrow$ 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
    prevColOverlap ← $(C \cap C_{\text{prev}})/|C|$ //Set overlap with previous column
    sourceColOverlap ← $(C \cap c)/|c|$ //Set overlap with column from Source table
    if *sourceColOverlap* < τ **then**
        Continue;
    overlapScore ← sourceColOverlap − prevColOverlap
    $\mathcal{T}_{\text{scores}}[T] \leftarrow$ overlapScore
Order $\mathcal{T}_{\text{scores}}$ by values in descending order
$\mathcal{T}'_C \leftarrow \mathcal{T}_{\text{scores}}$.keys
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.

B MATRIX REPRESENTATIONS

B.1 Expanding Candidate Tables

In order to represent candidate tables as matrices, their tuples need to align with those in the Source Table. However, not all candidate tables may share a key column with the Source Table. Thus, we need to join a given candidate table that does not share a key column with the Source Table with those that do. This way, tuples from all candidate tables can be aligned with tuples from a Source Table using key values.

As illustrated in Expand Algorithm 5, we traverse a graph that consists of candidate tables as nodes and we find a join path between candidate tables that do not have a Source Table's key (start nodes), and candidate tables that do (potential end nodes). If two candidate tables can join on common columns, their nodes are connected with an edge. For each edge, we use standard join-size cardinality estimation to find edge weights [57].

After we find a path from a start node to an end node, we iteratively join all tables in the path, resulting in a table that shares a key column with the Source Table. This way, all candidate tables share a key column with the Source Table.

B.2 Two-Valued vs. Three-Valued Matrix Representations

After aligning tuples in candidate tables to Source Table's tuples that share the same key values, we can represent candidate tables as matrices to show how similar the values in candidate tables are to those in the Source Table. These matrices have the same dimensions and indices as the Source Table. First, we consider matrices that consist of binary values, where a 0 in tuple i, column j represents a value at index (i, j) in the candidate table that is different from the value in the same position in the Source Table, and a 1 represent common values in the same indices.

However, populating alignment matrices with binary values does not fully encode how much values in candidate tables *differ* from those in the Source Table. Specifically, this representation does not distinguish between nullified values in candidate tables (null values in candidate tables at index (i, j) where there is a non-null value in the Source Table at the same position), and erroneous values (different non-null values in candidate tables at index (i, j) from those in Source Tables at the same position).

Example 16. *Suppose a data scientist has the left, green, Source Table in Figure 10(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.*

Instead, we encode matrix representations using three values, where at a given index in an aligned tuple, there is a 1 for a value shared between a candidate table and the Source Table, 0 for a null value in the candidate table where there is a non-null value in the Source Table, and -1 for a non-null value in the candidate table that differs from the value in the Source Table.

Example 17. *Given the same tables from Figure 10(a), Figure 10(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 such that, when we take the best-matching aligned tuples for every key value (tuple with highest number of 1's), we get a matrix that has two 0's (first, third, fourth tuples). 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.*

C METRICS

Conditional KL-divergence: 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 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:

$$D_{KL}(Q||P) = -\sum_{x \in X, k \in K} P(x|k) \log\left(\frac{Q(x|k)(1 - Q(\neg x|k)}{P(x|k)}\right) \quad (12)$$

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:

$$D_{KL}(T) = \frac{D_{KL}(Q_1||P_1) + D_{KL}(Q_2||P_2) + \cdots + D_{KL}(Q_n||P_n)}{Q(K) * n} \quad (13)$$

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.

D ABLATION STUDY

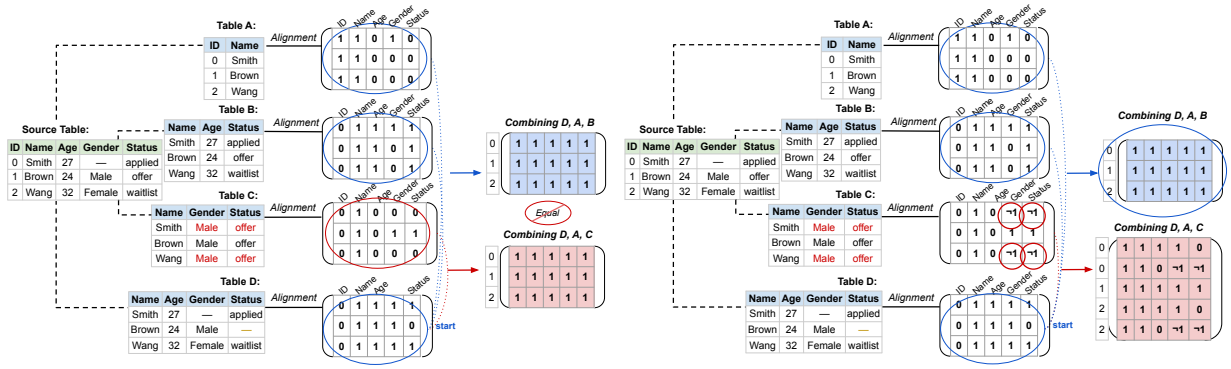
We compare Gen-T, that prunes the set of candidate tables to a set of originating tables before table integration, with baseline ALITE-PS, that directly integrates a set of candidate tables (Figure 11) in Recall, Precision, and F1 Score on TP-TR Med benchmark. ALITE-PS only outperforms Gen-T for 2 out of 26 Source Tables in Recall (Figure 11(a)). Even so, Gen-T still outperforms ALITE-PS on all Source Tables in F1 Score (Figure 11(c)), showing that ALITE-PS's higher Recall scores on 2 of the Source Tables are marginal, leaving no impact on the F1 Scores.

E GENERALIZABILITY

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.

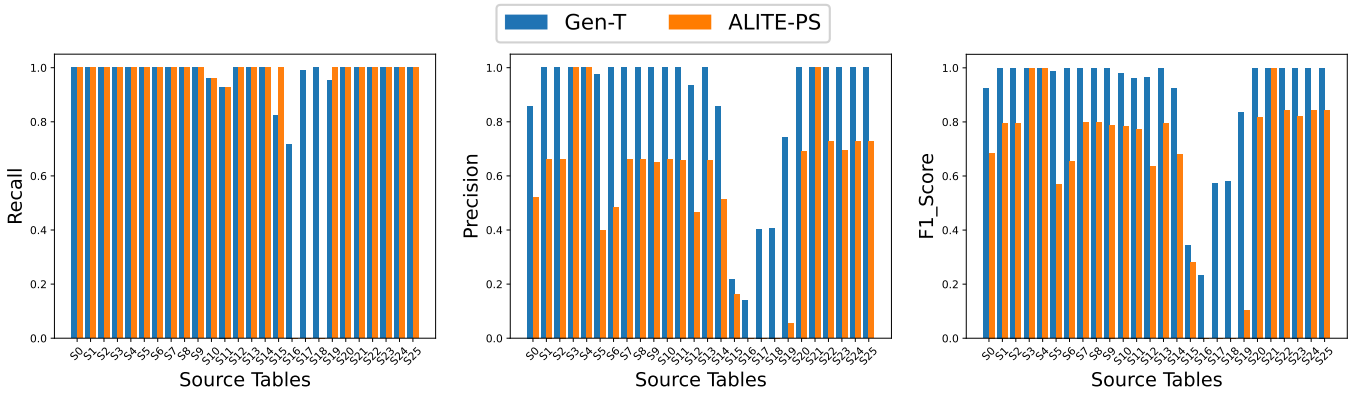
As shown in Table 3, we report the similarity and divergence scores for all methods on 33 of the common sources from T2D Gold for which all methods have non-empty, reasonably-sized output tables. We can see that Gen-T outperforms the baselines for all measures, even having perfect precision of 1.0. In 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.



(a) Two-Valued Matrix Representation

(b) Three-Valued Matrix Representation

Figure 10: 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.



(a) Recall

(b) Precision

(c) F1 Score

Figure 11: Recall, Precision, and F1 Scores of Gen-T and ALITE-PS for each Source Table in TP-TR Med benchmark.