Technical Report on Integrating Data Lake Tables

Aamod Khatiwada, Roee Shraga, Renée J. Miller
Northeastern University
{khatiwada.a,r.shraga,miller}@northeastern.edu

ABSTRACT

Over the last decade, we have made tremendous strides in providing tools for data scientists to discover new tables useful for their tasks. But despite these advances, the proper integration of discovered tables has been under-explored. An interesting semantics for integration, called Full Disjunction, was proposed in the 1980's, but there has been little advancement in using Full Disjunction for data science to integrate tables culled from data lakes. We provide ALITE, the first proposal for scalable integration of tables that may have been discovered using join, union or related table search. We show that ALITE can outperform (both theoretically and empirically) previous algorithms for computing the Full Disjunction. ALITE relaxes previous assumptions that tables share a common attribute names (which completely determine the join columns), are complete (without null values), and have acyclic join patterns. To evaluate our work, we develop and share three new benchmarks for integration that use real data lake tables.

PVLDB Reference Format:
Aamod Khatiwada, Roee Shraga, Renée J. Miller. Technical Report on Integrating Data Lake Tables. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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

1 INTRODUCTION

In recent years, the number of publicly available datasets has grown immensely in open data platforms [46, 47, 64]. In addition, individual corporations have a wealth of data stored in their own data lakes. Analyzing and integrating such datasets can help governments and enterprises in making decisions and plans. Data scientists, as the main users of data, use different techniques to discover and explore the available datasets using methods such as keyword search [9, 10, 48, 61] and table search (using the data within their table as a query) [8, 40, 41, 47]. The output of such a discovery process usually includes a collection of data lake tables that may be used to enrich their analysis [14]. Existing discovery techniques usually aim to discover unionable [10, 40, 47], joinable [26, 45, 61, 63–65], and related tables [8, 19, 63].

Example 1. *Consider the data lake tables about football stadiums shown in Fig. 1. We have added a* `Tuple ID` *column in each table to permit us to refer to tuples. Assume that a data scientist uses table* T_1 *as a* query table *to search for the top-2 unionable tables [47] and the top-2 joinable tables [66] from a data lake. Let* T_2 *and* T_3 *be the union search results and* T_4 *and* T_5 *be the join search results because they join with* T_1 *(approximately) on the* `Location` *columns. Join search finds tables that join on an indicated column (in this case* `Location`*), but does discover if there are other common (integratable) columns. For simplicity, assume that the common columns on these tables are already detected and have identical column headers. Note that in practice this will not be the case.*

In a real-world scenario, after discovery, data scientists wish to integrate the discovered tables before analyzing and applying statistical tools. Unfortunately, the previously mentioned methods do not address this "post-discovery" phase and the important question of how to integrate the tables (relations) obtained by the table search technique(s).

Example 2. *The standard relational union operator needs all tables to have exactly the same schema. However, this is not the case even for union search results (where tables that union on a subset of attributes can be retrieved) [47]. So to integrate the tables in Fig. 1, one can project out non-common columns and union on only the common columns. For* T_1*,* T_2*, and* T_3*, this would be just leave* `Location`*. For the joinable tables, a join on* `Location` *of* T_1 *with* T_4 *and* T_5 *leads to tuples like* t_{11} *being omitted and other tuples having two* `Stadium` *and* `Team` *attributes. Worse, the natural join operator i.e.,* $T_1 \bowtie T_4 \bowtie T_5$ *returns an empty set because* T_4 *and* T_5 *do not have joining tuples. The problem gets more complicated if we try to integrate all five tables using these operators.*

Within the data integration literature, *Full Disjunction (FD)* [30] has been understood as a good way of assembling partial pieces of information (facts) in such a way that maximizes the connections among these facts [53]. Indeed, Rajamaran and Ullman describe the Full Disjunction as a relation with nulls (represented by $\perp$) such that every set of join-consistent tuples appears within a tuple of the Full Disjunction, with concrete value or $\perp$ in each attribute not found within the set of tuples. Here, join-consistent is defined as common attributes (attributes with the same name), so this is effectively a *natural Full Disjunction*. The widely known outer-join [54] is not associative (hence the result depends on the order in which tables are integrated) and does not aim to maximize the connections among the integrated tuples [30].

Example 3. *Outer-join and outer-union keep all tuples and columns, but pad non-matching tuples (respectively, columns) with nulls [30, 54]. For example, the outer-union of the five tables is depicted in Figure 2(a). As can be seen in this example, the outer-union does not maximally connect the facts in the original tables. Here,* $\pm$ *indicates a missing value in the original tables and* $\perp$ *represents a*

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

T_1

Tuple ID	Stadium	Location	Team
t_1	NRG Stadium	Texas	Houston Texans
t_2	AT&T Stadium	Texas	Dallas Cowboys
t_3	Paul Brown	Ohio	±
t_4	Sofi Stadium	California	Angeles Chargers

T_2

Tuple ID	Stadium	Location	Opened
t_5	Soldier Field	Chicago	1924
t_6	State Farm	Arizona	2006

T_3

Tuple ID	Team	Location
t_7	Houston Texans	Texas
t_8	Green Bay Packers	Wisconsin

T_4

Tuple ID	Stadium	Location
t_9	Paul Brown	Ohio
t_{10}	AT&T Stadium	Texas

T_5

Tuple ID	Stadium	Location	Team
t_{11}	Lambeau Field	Wisconsin	Green Bay Packers
t_{12}	±	Ohio	Cleveland
t_{13}	Sofi Stadium	California	±

Figure 1: Tables about football stadiums, their locations and home teams. The objective is to integrate the five tables. `Tuple ID` is not a real data column, used for illustration. Also, metadata like column headers may not be available in real data lake tables and are used for illustration purpose. ± represents null values present in the input tables (*missing nulls*).

null introduced by the outer-union operator. In particular, it includes partial facts like t_{10} that are made redundant by more complete facts like t_2. Similar observations can be made of the outer-join results. Hence, Galindo-Legario defined the Full Disjunction (FD) [30] Informally, it gets rid of redundant facts and produces, in this example, the first 9 tuples (mustard colored) of Fig. 2(b). The FD can be viewed as an associative version of the outer join [17] and has been used to integrate information across relational tables [17, 49] and web tables [49].

However, using FD in data lakes poses several important challenges. First, we cannot rely on common attributes having the same name as in our example, rather we must discover what are the common attributes [46]. In real data lakes, Codd's unique name assumption does not hold so an attribute `Location` in one table may integrate with (be joinable with) and attribute with a different name like `Place` in another table. Similarly, two attributes names `Location` may have different meanings. Second, we cannot assume that integrated datasets are complete (that is, they contain no null values or partial (redundant) facts Finally, previous work on Full Disjunction has been done on relatively small relations (with only 1000 or so tuples per relation [17]) or assumes the common attributes form graphs with specific acyclic structures [53]. To the best of our knowledge, the only work on using FD on larger datasets requires that all joins be done on attributes having a key-to-foreign-key relationships [49], a strict requirement that makes the technique only applicable within well-designed enterprise scenarios, not the possibly messy tables retrieved from data lakes commonly used in data science.

In this work, we will assume data scientists use one or more table discovery algorithms to identify a set of tables they wish to integrate. Regardless of the search technique, we wish to find the best way to integrate the tables. Some search techniques use the semantics of columns, embeddings of columns, and other information about columns and tables to improve the retrieved results [8, 12, 13, 19, 47]. However, such information is rarely outputted with the search results. Even if we decide to modify search techniques to provide additional information, different techniques may vary in terms of semantics and format. Therefore, integrating the tables on the basis of metadata that are present in the tables or obtained from the search techniques is inadequate and may lead to bad integrations.

Based on these observations, we propose a table integration technique termed ALITE (Align and Integrate) that first assigns integration IDs (to be introduced in Section 2) to each column of the input data lake tables and then applies a natural FD over the integration IDs to obtain an integrated table.

1.1 Contributions

We now summarize the main contributions of this paper.

- To the best of our knowledge, we introduce here, for the first time, the problem of integrating data lake tables obtained using table discovery algorithms.
- As a first step, prior to the integration, we propose a technique to assign integration IDs that can be used to align columns during the integration process.
- We study multiple semantics that can be a possible fit for integrating the tables, among which, Full Disjunction (FD) lays the groundwork for our solution.
- We propose a novel scalable algorithm to compute FD by using complementation and subsumption operators. We show that the use of these operators permits optimizations that make the computation faster than the state-of-the-art techniques, both theoretically and experimentally. Specifically, the proposed algorithm scales better than the state-of-the-art technique for data lake tables, which are typically large and may have complex join graphs.
- We introduce several publicly available open data integration benchmarks to the research community.

2 PRELIMINARIES

We now provide the building blocks for integrating tables in a data lake setting, namely, specifying the notation we will use and the basic integration operators, after which we formally define the problem we address.

Notation. Table 1 summarizes the notation we use in the paper. For any operator, $\mathcal{S}$ (and its size S) and $\mathcal{F}$ (and its size F), denote the collective set of all input and output tuples, respectively. Also, we use two types of nulls, ± denotes a *missing value* from any incomplete input relation to be integrated and $\perp$ denotes a *produced null*, a null value that is introduced by an operator during the integration process.

Tuple ID	Stadium	Location	Team	Opened
t_1	NRG Stadium	Texas	Houston Texans	⊥
t_2	AT&T Stadium	Texas	Dallas Cowboys	⊥
t_3	Paul Brown	Ohio	±	⊥
t_4	Sofi Stadium	California	Angeles Chargers	⊥
t_5	Soldier Field	Chicago	⊥	1924
t_6	State Farm	Arizona	⊥	2006
t_7	⊥	Texas	Houston Texans	⊥
t_8	⊥	Wisconsin	Green Bay Packers	⊥
t_9	Paul Brown	Ohio	⊥	⊥
t_{10}	AT&T Stadium	Texas	⊥	⊥
t_{11}	Lambeau Field	Wisconsin	Green Bay Packers	⊥
t_{12}	±	Ohio	Cleveland	⊥
t_{13}	Sofi Stadium	California	±	⊥

(a) T1 ⊎ T2 ⊎ T3 ⊎ T4 ⊎ T5

Output ID	Tuple ID	Stadium	Location	Team	Opened
f_1	$\{t_1, t_7\}$	NRG Stadium	Texas	Houston Texans	⊥
f_2	$\{t_2, t_{10}\}$	AT&T Stadium	Texas	Dallas Cowboys	⊥
f_3	$\{t_7, t_{10}\}$	AT&T Stadium	Texas	Houston Texans	⊥
f_4	$\{t_3\}$	Paul Brown	Ohio	±	⊥
f_5	$\{t_{12}\}$	⊥	Ohio	Cleveland	⊥
f_6	$\{t_4\}$	Sofi Stadium	California	Angeles Chargers	⊥
f_7	$\{t_5\}$	Soldier Field	Chicago	⊥	1924
f_8	$\{t_6\}$	State Farm	Arizona	⊥	2006
f_9	$\{t_8, t_{11}\}$	Lambeau Field	Wisconsin	Green Bay Packers	⊥
f_{10}	$\{t_3, t_{12}\}$	Paul Brown	Ohio	Cleveland	⊥
f_{11}	t_{13}	Sofi Stadium	California	±	⊥

- $FD(T_1, T_2, T_3, T_4, T_5) = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9\}$
- $FD_{tuple\text{-}set}(T_1, T_2, T_3, T_4, T_5) = FD(T_1, T_2, T_3, T_4, T_5) \cup \{f_{11}\}$
- $T_1 \boxplus T_2 \boxplus T_3 \boxplus T_4 \boxplus T_5 = FD(T_1, T_2, T_3, T_4, T_5) - \{f_4, f_5\} \cup \{f_{10}, f_{11}\}$

(b) Output tuples generated using different operators

Figure 2: Result of integrating the tables in Fig. 1 using different techniques. The table in (a) shows the result of outer unioning the five tables. The table in (b) shows the union of tuples obtained using FD (first nine tuples in mustard), a variant called tuple-set FD (which is the FD plus tuple f_{11}) and Complement Union (⊞). A unique Output ID is provided for each output tuple for clarity. The output tuples of each algorithm are shown below the table w.r.t. the FD's output.

Table 1: Symbols used in this paper and their definitions

Symbol	Definition
$\mathcal{T}$ $(n = \lvert\mathcal{T}\rvert)$	Set of Tables
T (T_i)	A table (the i_{th} table in $\mathcal{T}$)
A $(T.A)$	A column (Column A in Table T)
m_i	Arity of Table T_i
$\mathcal{A}(T)$	Schema, i.e. set of columns, of T
t	A tuple
r	A set of tuples
$t[A]$	The value of t on the column A
$\mathcal{S}$ $(S = \lvert\mathcal{S}\rvert)$	Set of all input tuples
$\mathcal{F}$ $(F = \lvert\mathcal{F}\rvert)$	Set of output tuples
±	Null denoting a missing value
⊥	Null produced by an operator

Example 4. *Consider Table T_1 of Fig. 1. The schema of T_1 is $\mathcal{A}(T_1) = \{Stadium, Location, Team\}$. Tuple t_3 = {Paul Brown, Ohio, ±} has attribute value $t_3[Stadium]$ = Paul Brown and a missing null on the Team column, i.e., $t_3[Team] = \pm$.*

2.1 Finding Common Columns

Our introduction example is unrealistic because common (or in relational terms join-consistent) columns from different tables have the same name and columns that are not common have different names. This will not be the case in most realistic examples. Hence, we will begin by assigning *integration ids* to columns such that two common columns will have the same integration id and two columns that are not common will have different ids. We will ensure no two columns in the same table share an integration id. Accordingly, we will set $\mathcal{A}(T)$ to be the set of integration ids of T's columns.

2.2 Integration Operators

We assume that the reader is familiar with the elementary relational algebra operators like union, join and outer join [54] based on which, we now introduce some (less well known) operators that we use in later sections as components of an integration solution. Also, we assume two columns in different tables are *common columns* iff they have the same integration id.

Outer Union (⊎). Outer union is an extension to the union operator that unions tables even if they do not have the same schema [16]. We use ⊎ to denote outer union, i.e., the outer union between T_1 and T_2 is denoted by $T_1 \uplus T_2$. For each $A \in \mathcal{A}(T_1) - \mathcal{A}(T_2)$, we pad T_2 with a new column A containing nulls (specifically ⊥). Similarly, for each $A \in \mathcal{A}(T_2) - \mathcal{A}(T_1)$, we pad T_1 with a new column A containing nulls. We then compute the union of the padded relations.

Example 5. *An example of outer unioning tables in Fig. 1 is shown in Fig. 2(a). In this example, the size of the input ($\lvert\mathcal{S}\rvert$ = 13) is the same as the size of the output ($\lvert\mathcal{F}\rvert$ = 13) but the output may be smaller if there are duplicates.*

Subsumption (β). Given two different tuples t_1 and t_2 having the same schema, the tuple t_1 (subsuming tuple) *subsumes* t_2 (subsumed tuple) if all the non-null values of t_2 are equal to that of t_1 on the respective columns and t_1 has fewer null (either missing or produced) values than t_2 [5, 30]. Using subsumption, we can remove the subsumed tuple which is less informative than the subsuming tuple. We denote the subsumption operation using β, e.g., applying subsumption to a set of tuples r is denoted by $\beta(r)$ where, $\beta(r)$ contains all tuples of r that are not subsumed by another tuple in r, and "t_1 subsumes t_2" is denoted by $t_1 \sqsupset t_2$. Applying subsumption to the outer union result is called the *minimum union* ($\bigoplus$) [30].

Example 6. *An example of minimal union ($\bigoplus$) of tables in Fig. 1 is the set $\{t_1, t_2, t_3, t_4, t_5, t_6, t_9, t_{11}, t_{12}\}$. This is because the tuple t_7*

is subsumed by t_1, t_8 is subsumed by t_{11}, t_{10} is subsumed by t_2, t_8 is subsumed by t_{11}, and t_{13} is subsumed by t_4. In this example, the size of the input ($|\mathcal{S}| = 13$) is larger than the output ($|\mathcal{F}| = 9$). Note that t_9 and t_3 do not subsume each other.

Complementation (κ). Two different tuples t_1 and t_2 having the same schema *complement* each other to give a new tuple t_3 if they have equal and non-null values in at least one column, all non-null values at same column positions are equal to each other, and t_1 has at least one column with a null (missing or produced) and t_2 has a non-null value and vice-versa [5, 7]. For any column A, $t_3[A] = t_1[A]$ if either $t_1[A]$ is non-null or both $t_1[A]$ and $t_2[A]$ are non-null (and equal) or $t_1[A] = \pm$. Otherwise,if $t_2[A]$ is non-null $t_3[A] = t_2[A]$. For the case where both values are null, if $t_1[A] = t_2[A] = \perp$ then $t_3[A] = \perp$ otherwise (at least one of the nulls is missing) $t_3[A] = \pm$. The complementation operator (κ) replaces all complemented pairs of tuples with their complementation. Note that a tuple that results from complementation could be complemented by other tuples so the complementation operator is the iterative result of applying complementation to a relation until it contains no further complementing tuples. Applying complementation over a set of outer unioned tuples, is known as *complement union* and denoted by $\boxplus$.

Example 7. *In Table(a) of Fig. 2, tuples t_3 and t_{12} complement each other. Their complementation is denoted as f_{10} in Fig. 2(b), i.e., $\kappa(t_3, t_{12}) = f_{10}$. So complementation can* overcombine *tuples that do not agree on all their common attributes. In this example, T_5 asserts that* `Cleveland` *is a team in* `Ohio` *with an unknown stadium while T_5 asserts that* `Paul Brown` *is a stadium in* `Ohio`*. But we do not definitively know that* `Paul Brown` *is the stadium of* `Cleveland`*. The complement union of the tables in Fig. 1, i.e., $T_1 \boxplus T_2 \boxplus \cdots \boxplus T_5$ is the set of tuples in Table (b) of Fig. 2 excluding tuples $\{f_4, f_5\}$. Note that complementation union may contain subsumable tuples like f_{11} (which can be subsumed by f_6).*

2.3 Full Disjunction

The operators of the previous section offer possible semantics for integrating tables. In 1994, Galindo-Legario proposed a different semantics called Full Disjunction (FD). His proposal is essentially a commutative, associate form of outer-join. We will now define the terms that we need to define FD. We say that $t_1 \in T_1$ and $t_2 \in T_2$ are *connected tuples* if their schemas overlap, i.e., $\mathcal{A}(T_1) \cap \mathcal{A}(T_2) \neq \emptyset$. As in outer join, two connected tuples $t_1 \in T_1$ and $t_2 \in T_2$ can be integrated (or joined) if and only if $t_1[A] = t_2[A]$, $t_1[A] \neq \pm$ and $t_2[A] \neq \pm$, $\forall A \in \mathcal{A}(T_1) \cap \mathcal{A}(T_2)$. The tuples generated after an integration are referred to as *integrated tuples*. When more than two tables are involved, the integration can be viewed as an iterative process in which an integrated tuple can be further integrated with another connected tuple, following the same conditions as before. Finally, like in outer join, if an input tuple t can not be integrated with other tuples, it will be padded by produced nulls ($\perp$) and considered as integrated tuple. Note that integrating those tuples that have missing nulls on their *common columns* may produce semantically incorrect tuples. Consider tuples t_9 from T_4, and t_{12} from T_5, while they share the value `Ohio` on `Location`, the value of `Stadium` is known in t_9 (`Paul Brown`), it is unknown in t_{12}. Therefore, we will not integrate these tuples. Notably, FD was later proposed as the right semantics for integrating web data [53].

Example 8. *The FD of the five tables from Fig. 1 is the set of tuples $\{t_1, ..., t_9\}$ depicted in mustard in Table (b) of Fig. 2.*

FD has been shown to produce what has been called *maximally integrated tuples* [37]. We follow Kanza and Sagiv [37] by defining FD based on maximally integrated tuples.

Definition 9 (Maximally Integrated Tuple). *Given a set of integrated tuples r. Any tuple $t \in r$ is said to be a maximally integrated tuple if it is not subsumed by any other tuples of r [37].*

We now define FD based on maximally integrated tuples [37].

Definition 10 (Full Disjunction (FD)). *The Full Disjunction of the tables $T_1, T_2, \ldots T_n$ is the set of all maximally integrated tuples that can be generated from S input tuples of $T_1, T_2, \ldots T_n$ [37].*

In Section 5, we will introduce an algorithm that computes FD based on Definition 10. The FD definition we use [37] is based on tuples [30], rather than tuple-sets ($FD_{tuple-set}$) [17, 18]. $FD_{tuple-set}$ applies subsumption based on sets of tuples rather than the actual tuples [18] and considers a tuple set r_1 as the subsumption of tuple set r_2 if r_1 is a superset of r_2. Note that $FD_{tuple-set}$ yields a set of maximally integrated *tuple sets*, but might contain tuples that subsume each other, as we discuss in the next example.

Example 11. *To illustrate the difference between FD and $FD_{tuple-set}$, consider Fig. 2(b). To understand the subsumption in $FD_{tuple-set}$, first consider f_4 and f_{10} as tuple sets having tuples $\{t_3\}$ and $\{t_3, t_{12}\}$ respectively. As f_{10} contains all tuples of f_4 (t_3 in this case), f_4 is contained in f_{10} and hence, f_{10} is the superset of f_4. Therefore, according to [17, 18], f_{10} subsumes f_4. However, if we consider tuple sets f_6 and f_{11} having tuples $\{t_4\}$ and $\{t_{13}\}$, respectively, f_6 is not a superset of f_{11} so, accordingly, f_6 does not subsume f_{11}. Therefore, $FD_{tuple-set}$ returns the tuple f_{11}, which is not produced by FD as it gets subsumed by f_6.*

2.4 Solution Overview

In the previous sections, we introduced and formally defined our problem of integrating data lake tables. Fig. 3 illustrates the entire ALITE pipeline that we propose as a solution.

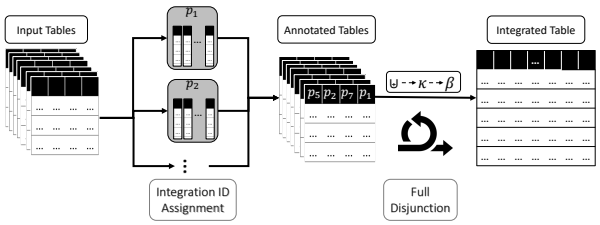

Figure 3: An overview of the integration process. The Integration ID assignment block assigns new integration ids to each column in the input tables. If two columns have the same integration id, they are equated in the natural Full Disjunction. Then, the natural Full Disjunction is applied to get the integrated table.

We assume that we are given a set of tables. The first step (left part of Fig. 3) is to assign each column with a column header which we call *Integration ID* (Section 4). After assigning such IDs, the tables are annotated (middle part of Figure 3) and can be used to be used in applying FD to integrate the tables (Section 5).

3 RELATED WORK

We now discuss related work mainly revolving around assigning the column integration IDs and applying FD.

Assigning Column Integration IDs. The problem of assigning column integration IDs aims at providing correspondences between columns that can be integrated. In a traditional database setting, this problem is usually referred to as schema matching [52], a longstanding problem of identifying correspondences among database attributes. Numerous algorithmic attempts have been suggested over the years for handling the matching problem, e.g., COMA [25], Similarity Flooding [43], BigGorilla [15], and ADnEV [57]. A common assumption for most of this work is the existence of consistent and complete metadata, an unrealistic assumption in data lake tables. Recently, Koutras et al., explored the use of traditional schema matching methods in the scope of data lakes [39]. However, the work covered is limited to finding pairs of matching columns whereas, our objective is to assign integration IDs to a set of tables to be integrated in a holistic manner. Holistic schema matching, i.e., matching a set of schemas at the same time, has received some attention in the literature [33, 51, 58], mainly revolving around web tables and assuming metadata is reliable and complete. Specifically, some work [1, 51] presents a clustering based approach similar to what we will present (Section 4). However, their clustering method is applied over schema names rather than data values as we suggest here. Recall that data lake tables generally lack reliable and consistent metadata [46].

Other related work includes searching for unionable [10, 40, 47], joinable [26, 45, 61, 63–65], and related [8, 19, 63] tables, for which the designed methods are usually based on column relationships. For example, in order to find unionable tables, TUS [47] first aims at finding unionable columns. Similarly, T3L [8] assesses attribute relatedness to assess how related tables are. Our work uses a similar methodology to TUS [47] and T3L [8] based on embeddings. However, here we make use of an embedding that was designed for tables, namely, TURL [23]. We are also, to the best of our knowledge, the first to make use of TURL [23] for data lake tables.

Full Disjunction. Full disjunction (FD) was initially defined by Galindo-Legaria as an associative alternative for the outer join operator [30]. Galindo-Legaria used algebraic relationships to express the outer join in terms of inner join and minimum union, which is known as join-disjunctive form or Full Disjunction (FD) [30]. The inner joins between each table pair, triple, etc., are computed. The resulting tuples are then outer-unioned and the subsumable tuples are removed to get the FD. Rajaraman and Ullman showed that a fixed ordering of outer joins can give the FD if and only if the input tables form a γ-acylic hypergraph. Hence, for the γ-acylic case, the FD can be computed in linear time in the output size [53]. Kanza and Sagiv showed that FD can be computed for any arbitrary set of tables in $O(n^5S^2F^2)$ time [37] where, recall n is the number of input tables, S is the total of number of input tuples, and F is the number of FD output tuples. This is the first work to show that FD can be computed for any set of tables in polynomial time in input-output complexity [60]. Other work considers FD in terms of tuple sets rather than actual tuples [17, 18]. Cohen and Sagiv introduced an algorithm that computes k FD tuples in a given ranking order [18]. Cohen and Sagiv [18] improved the worst-case time complexity over Kanza and Sagiv [37] to compute the full results. Cohen et al. also proposed an algorithm called *BICOMNLOJ* that computes each FD tuple with polynomial delay [17]. As we want to integrate all input tables, we compute the full FD result instead of a partial result. The worst-case time complexity of *BICOMNLOJ* to compute full FD is linear in the output size which is an improvement over the prior work [18]. Note that both *INCREMENTALFD* [18] and *BICOMNLOJ* [17] perform subsumption in terms of tuple sets rather than actual tuples. Hence, they may produce subsumable tuples in their FD result (specifically, they may produce a proper superset of the FD). Note that, when there are no missing values (±) and subsumable tuples in the input relations, these algorithms compute the FD [30]. The difference between the FD and tuple-set based FD is illustrated in Example 11. As data lake tables may contain many missing nulls and subsumable tuples, we use FD. We show in our experiments (Section 6) that in real data lakes the difference between the FD and the tuple-set FD [17, 18] can be substantial and that the original definition of FD, which maximally integrates tuples is preferred.

Recently, Paganelli et al. [49] revised Cohen and Sagiv's *INCREMENTALFD* [18] and Cohen et al.'s *BICOMNLOJ* [17] to compute the FD in a distributed environment. Paganelli et al. introduced a new algorithm called *PARAFD* that outperforms *INCREMENTALFD* and *BICOMNLOJ* while computing FD using multiple machines. However, *PARAFD* can be used only for sets of relational tables on which all joins are key to foreign-key joins. In our work, we consider the general case of arbitrary joins.

Integrating Relational and Web Tables. Other research considers integrating data from relational and web tables and handling conflicts between the data values [4–7]. Data conficts have been divided into two categories: (i) *contradiction*, which is a conflict between two non-null values when there is a key or an equality-generating dependency asserting the values should be equal and (ii) *uncertainty*, which is a conflict between a null value and a non-null value [5]. Bleiholder et al. introduced complement union operator that integrates tuples under uncertainty [7]. In the absence of *missing nulls* (±), the complement union operator is the same as FD. Yet, in the common case of tables with *missing nulls* complement union may combine the tuples having null values on the join columns. As we argued in Example 7, complement union can over-combine tuples even when they do not agree on all common attributes.

4 ASSIGNING COLUMN INTEGRATION IDS

We now explain the first stage of ALITE, namely assigning integration IDs to the columns of the input tables. We assume that the schemas $\mathcal{A}(T) = A_1, A_2, \ldots, A_m$ of all the tables $T \in \mathcal{T}$ are opaque [36]. Accordingly, our goal, in this stage, is to annotate the columns with *integration IDs*. An integration ID $p(A) \in \mathcal{P}$ is associated with each column. The same integration ID can be associated with a set of columns – these column match (and will

be integrated). We now formally define the column integration ID assignment problem.

Definition 12 (Column Integration ID Assignment Problem). *Given a set of input tables* $\mathcal{T} = \{T_1, T_2, \ldots T_n\}$, *the* column integration ID assignment problem *is to assign each column an integration ID in* $\mathcal{P}$ *such that columns in the same table get distinct integration IDs.*

$$\forall T \in \mathcal{T} \nexists A \in \mathcal{A}(T), A' \in \mathcal{A}(T) \wedge A \neq A' \; p(A) = p(A')$$

As discussed in Section 3, this problem can also be seen as a variation of holistic schema matching [58]. Specifically, it can be seen as a $1:1$ matching constraint, in which an attribute can match at most one attribute from each of the other tables and cannot be matched with an attribute from the same table.

Finding Column Integration IDs with ALITE. We now aim to find column integration IDs by positioning the problem as clustering over the columns. In order to apply clustering over columns, we use their values (assuming the metadata is missing or unreliable) to create embeddings over which a clustering algorithm can be applied. Formally, a column A is embedded into a numeric vector $vec(A)$, allowing the creation of a similarity matrix. Obtaining an embedding for data lake columns is far from trivial. TUS [47], for example, uses embeddings from fastText [35], a word embedding method based on natural text representations, to assess column unionability of string columns. Recently, Deng et al., have proposed TURL that creates embeddings based on a representation of each table [23]. In this work, we explore the use of TURL to represent columns of data lake tables. Once the embeddings for the columns are set, we need to define a similarity/distance measure to be used in the clustering algorithm (in our experiments we use euclidean distance). Having defined the embeddings and the distance measure, we follow a hierarchical clustering methodology to create the clusters out of which the column integration IDs are obtained. We ensure that the clustering algorithm does not allow columns from the same table to be assigned to the same cluster.

Selecting the Number of Integration IDs. An important parameter in any clustering algorithm is the number of clusters [32], which, in our case corresponds to the number of column integration IDs. While traditional approaches assume that the number of clusters is a given parameter, an alternative is to tie this number to clustering quality [38]. Several clustering quality evaluation methods exist in the literature based on inter-cluster and intra-cluster distances [11, 22, 55]. The objective, which we also share in this paper, is to cluster similar columns together (i.e., reduce intra-cluster distance) and avoid similar columns in different clusters (i.e., increase inter-cluster distance).

We follow an approach similar to the elbow method to determine the number of clusters [3]. Specifically, we look for a number of clusters that maximizes some (unsupervised) clustering quality measure.[1] In what follows, we also need to define the scope of this search, i.e., what are the possible values for the number of clusters. Recall that the columns from the same input table cannot be assigned to the same cluster. Therefore, if $m_1, m_2 \ldots m_n$ are the number of columns in the input tables $T_1, T_2 \ldots T_n$, the minimum number of clusters is given by $\max(m_1, m_2 \ldots m_n)$. Also, the maximum number of clusters is given by $\sum_{i=1}^{n} m_i$. The latter represents the case when each input column forms a separate cluster and the bound can be even tighter if we know that the scheme graph of the input tables is connected.

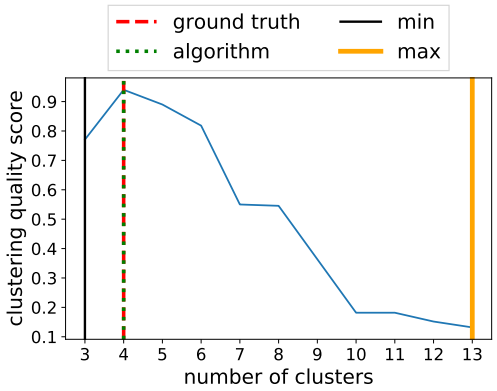

Figure 4: An example scenario of determining the number of clusters for the input tables in Fig. 1. Here, X-axis shows the number of clusters and Y-axis shows the clustering quality score. The green and red lines show the number of clusters according to the algorithm and ground truth respectively.

Example 13. *In our running exampe (Fig. 1), we need to assign integration IDs to the columns of tables. Here, we expect to have four clusters, each for {Stadium, Location, Team and Opened} as our column labels show the ground truth. We use a clustering algorithm that computes the clustering quality score starting from the minimum number of clusters (three), to the maximum (12). Recall* `Tuple ID` *is not a column in the data lake and was just added so we can clearly refer to tuples. The Silhouette coefficient over the TURL embeddings is computed for all values from 3 to 12 and plotted in Fig. 4. It has a maximum at 4, with the next highest value at 5 and monotonically decreasing from 6 to 12. The value at 3 is just under the value at 6. Hence, we would pick 4 as the optimal number of clusters and the clustering created in this simple example does reflect the ground truth. The four* `Stadium` *columns are assigned the same integration ID, and the single* `Opening` *attribute is assigned a different integration ID not shared by other columns.*

5 INTEGRATING TABLES

Once we find the column integration IDs, ALITE uses them to integrate the tables using a novel algorithm for computing Full Disjunction. We show that our algorithm is correct and that both asymptotically and in practice is faster than existing algorithms.

5.1 ALITE FD Algorithm

The input of Algorithm 1 is a set of tables $\mathcal{T}$ to be integrated with each column labeled with its integration ID. A table $T \in \mathcal{T}$ is a set of tuples. The two main properties the algorithm uses are that the output is composed of all maximally integrated tuples over the input tuples (Definition 10) and should not contain subsumable

[1] In the experiments we use the well-known Silhouette Coefficient [55].

Algorithm 1: ALITE Full Disjunction

1 **Input**: A set of tables with integration IDs as column names $\mathcal{T} = \{T_1, T_2, \ldots T_n\}$
2 **Output**: The Natural Full Disjunction of $\mathcal{T}$ $FD(\mathcal{T})$
3 $\mathcal{T}$:= GenerateLabeledNulls($\mathcal{T}$)
4 $U_{\text{ou}} \leftarrow T_1 \uplus T_2 \uplus \cdots \uplus T_n$ //Apply outer union
5 U_{comp} := Complement(U_{ou}, n)
6 U_{comp} := RemoveLabeledNulls(U_{comp})
7 $T' \leftarrow \beta(U_{comp})$ //Apply subsumption
8 Output T'

Algorithm 2: Complement

1 **Input**: A set of outer unioned tuples U_{ou} and the no. of tables n
2 **Output**: A set of tuples after complementation U_{comp}
3 //Prepare temporary set and complementation set to hold tuples
4 $U_{\text{temp}} := U_{\text{ou}}$
5 $U_{\text{comp}} \leftarrow \emptyset$
6 //Start complementation
7 **for** *i in range(1,n)* **do**
8 **for** $t_1 \in U_{temp}$ **do**
9 complement_count = 0
10 **for** $t_2 \in U_{ou}$ **do**
11 R, complement_status = $\kappa(t_1, t_2)$
12 **if** *complement_status* **then**
13 $U_{comp} \leftarrow U_{comp} \cup R$
14 complement_count$\leftarrow$ complement_count + 1
15 **if** *complement_count = 0* **then**
16 $U_{comp} \leftarrow U_{comp} \cup \{t_1\}$
17 **if** $U_{temp} = U_{comp}$ **then**
18 break
19 **else**
20 $U_{temp} \leftarrow U_{comp}$
21 $U_{comp} \leftarrow \emptyset$
22 Output U_{comp}

tuples. ALITE's pseudo code is provided in Algorithm 1. We make use of the following property, complementation (Line 5) over the outer union (Line 4) generates all maximally integrated tuples if the input relations contain no null values. Of course, our data lake tables will contain null values (±) so we begin by replacing these with distinct *labeled nulls* (Line 3). We then apply complementation treating the labeled nulls as distinct so they cannot be equated. We can then replace all distinct labeled nulls with the same missing value (±) (Line 6) and apply subsumption (Line 7) as a final step to compute the FD. Next, we will explain each step in detail.

Generating Labeled Nulls. Complementation produces all maximally integrated tuples only if the input tables have no null values (±). Hence, to prevent over-jealous combining of tuples, we replace nulls (±), with distinct labeled nulls which are not equal to each other, to ±, ⊥, or any constant (non-null) in any table. This avoids undesirable complementation (and accordingly generates only integrated tuples). Specifically, the first step of Algorithm 1 (Line 3) is to replace missing nulls in the input tables with the *distinct labeled nulls* and store them in a set N. This step ensures that the complementation will not integrate tuples having null values on join columns.

Example 14. *We use our running example (Fig. 1) throughout the description of the algorithm for clarity. Since we have three missing nulls in the tables (one on T_1 and two in T_5), we replace them with three distinct labeled nulls. After replacement, they are treated similar to other non-null values.*

Now we outer union all the input tables and store the resulting tuples in a set U_{ou} (Line 4). The result of outer unioning the tables in Fig. 1 is shown in Fig. 2(a). Next, Line 5 passes the set of outer unioned tuples (U_{ou}) and the total number of tables (n) to Algorithm 2 which returns all the maximally integrated tuples along with (possibly) subsumable tuples.

Complementation Step (Algorithm 2). As we mentioned before, the objective of this step is to generate all the maximally integrated tuples. First, we prepare two sets to perform the complementation, namely, U_{temp}, initially a duplicate of U_{ou}, and U_{comp} which will hold the complementation result Line 4-5. We start complementing the tuples in U_{temp} with outer unioned tuples (Line 7). For each tuple in U_{temp}, we find a complementing partner in U_{ou} and if at least one complementing partner is found, we add the result of complementation to U_{comp} (Line 11-14). However, if a tuple in U_{temp} does not have any complementing tuples, we add the tuple to U_{comp} (Line 15-16). This ensures that the tuples having no join partners are also included in the FD results. After we go through all the tuples in U_{temp}, we check if U_{temp} and U_{comp} have the same tuples. If this is true, it means that there are no more complementing tuples left and hence, we stop the complementation (Line 17-19). If they are not equal, there may be tuples that can be complemented. Therefore, we go for another round of complementation.

Example 15. *Consider Table (a) and (b) of Fig. 2 as an example. Table (a) is the result of outer unioning the tables in Fig. 1 and Table (b) holds the resulting tuples given by different integration techniques. Consider Algorithm 2 which has as input a set of tuples to be complemented along with the total number of tables. In the first round of complementation (i.e., Line 7- 21 for i = 1), both U_{ou} and U_{temp} contain the tuples in Table (a). Therefore, the tuple t_7 can be found in U_{temp} and tuple t_{10} can be found in U_{ou}. Clearly, these two tuples can be integrated. The complementation operator integrates them and produces the tuple f_3 shown in Table (b) (Line 11). Similarly, tuples t_8 and t_{11} complement each other producing f_9. Both f_3 and f_9 are added to U_{comp} (Line 13). The tuple t_5 on the other hand, does not have any integrating partners. Hence, t_5 is added to U_{comp} (Line 16). After the first round of complementation, $U_{comp} = U_{ou} \setminus \{t_7, t_8, t_{10}, t_{11}\} \cup \{f_3, f_9\}$ which is different from U_{ou}. Hence, we move U_{comp} to U_{temp}, and empty U_{comp}. Then the algorithm starts a second round of complementation, i.e., Line 7- 21 for i = 2. As mentioned earlier, U_{ou} is the same for this round, but $U_{temp} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_9, t_{12}, t_{13}, f_3, f_9\}$. Clearly, no tuples in U_{temp} have complementing partners in U_{ou}. Therefore, the complementation terminates after this round and $U_{comp} = U_{temp} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_9, t_{12}, t_{13}, f_3, f_9\}$.*

Subsumption. Once the complementation is done, we remove the subsumable tuples to get FD. Notice however, we have replaced the missing nulls (±) with the distinct labeled nulls before complementation. This is to prevent the complementation on the missing nulls. However, to get the maximally integrated tuples, we ensure that there are no subsumable tuples–both on missing nulls and

produced nulls. Therefore, we revert each labeled null to its original missing value (±) (Line 6 of Algorithm 1) and then use subsumption (Line 7) to remove the non-maximally integrated tuples.

Example 16. *Continuing our example, we replace the unique labeled nulls in each tuple with the missing null (±). This step makes tuple t_3 a duplicate of t_9 and hence t_3 gets removed because we are using set semantics. The remaining tuples are returned by Algorithm 1. Finally, we apply subsumption to U_{comp} and get rid of tuples t_8 and t_9 (Algorithm 1, Line 7). This ensures that the final result is the set of FD tuples i.e., $\{f_i\}, i \in [1, 9]$.*

5.2 Full Disjunction Algorithm Analysis

We now analyze the time complexity and correctness of Algorithm 1.

Time Complexity. The worst case time complexity of performing subsumption is quadratic in the number of input tuples [6]. However, we apply subsumption using the *Null-value based partitioning Algorithm* [6] that takes $O[SlogS]$ time where, S is the number of input tuples. The idea is to first partition the input tuples according to their null value pattern. This helps to reduce the number of tuple comparisons for the subsumption check and hence, we can apply subsumption only on tuples within a partition. Note that the number of columns in the integrated table is constant for a given set of tables.

Theorem 17. *The worst-case time complexity of ALITE on input tables $T_1, T_2, \ldots T_n$ is $O[n \cdot max(F, S) \cdot S + FlogF]$ where, S is the total number of tuples in all input tables and F is the output size.*

Proof. In Algorithm 1, the asymptotic time complexity of replacing missing nulls with unique placeholders (Line 3) and outer unioning the input tuples (Line 4) is linear to the input size i.e. $O[S]$. The runtime of Algorithm 1 mainly depends on complementation (Line 5) and subsumption (Line 7). To show the time complexity of Algorithm 1, first we need to determine the maximum number of tuples that can be produced by Algorithm 2, Line 7-21. Note that, the number of tuples in U_{ou} is always constant i.e., S and the number of tuples changes only for U_{temp} which we update with the integrated tuples in each iteration. In some round of iteration, the total tuples in U_{temp} reaches the number of output tuples along with some subsumable tuples b i.e., $(F + b)$. We can consider b as some factor of F. Therefore, the number of tuples in U_{temp} is bounded by $O[F]$ if $F > S$ else, it is bounded by $O[S]$. On what follows, the upper bound of the number of tuples in U_{temp} is $O[max(F, S)]$ and hence, the asymptotic time complexity of Algorithm 2 is $O[n \cdot max(F, S) \cdot S]$. After that we feed the complementation results i.e., $O[F]$ tuples to Line 7 in Algorithm 1, which performs subsumption in $O[FlogF]$ time. Hence, the worst-case time complexity is $O[n \cdot max(F, S) \cdot S + FlogF]$. □

Note that FD is exponential in terms of input complexity due to which its time complexity is expressed in terms of input-output complexity [60]. Also, we follow the FD definition (Definition 10) of Galindo-Legaria [30, 37] rather than tuple-set version [17, 18]. Therefore, the algorithm presented by Kanza and Sagiv [37] ($O[n^5 s^2 F^2]$) is the best algorithm to compare against our time complexity. The best theoretical guarantee for the tuple-set based FD is given by *BICOMNLOJ* [17] whose worst-case time complexity to compute tuple-set FD is $O[F(n^2S + S^2 + nSlogS)]$. However, one needs to apply subsumption to make the output of tuple-set FD equivalent to the FD. By applying the same subsumption algorithm we use in our technique, the worst-case time complexity of *BICOMNLOJ* [17] to compute full FD result becomes $O[F(n^2S+S^2+nSlogS)+FlogF]$. This shows that our algorithm gives the best theoretical guarantee to compute FD. Also, when there are no subsumable tuples in the input relations, both *BICOMNLOJ* and our algorithm give the same result. In that case, our algorithm does not need to perform subsumption and hence, the worst case time complexity is dependent only on the complementation step i.e., $O[n \cdot max(F, S) \cdot S]$ which is also an improvement over Cohen et al. [17]. Note that *BICOMNLOJ* does some extra work to compute the tuples with polynomial delay. This is also a reason for it being slower than us. We quantify the importance of this difference in the time taken by both algorithms to integrate the tables having different input and output sizes in Section 6 experimentally.

Correctness. Next we show that Algorithm 1 computes the Full Disjunction [30].

Theorem 18. *The relation computed by ALITE over a set of input tables $T_1, T_2, \ldots T_n$ is exactly the natural full disjunction of $T_1, T_2, \ldots T_n$.*

Proof. Let, $\mathcal{F}$ be the set of tuples in the natural full disjunction of T_1, T_2,... T_n. Let, t be any tuple such that t $\in \mathcal{F}$. Now, t $\in \mathcal{F}$ iff t is a maximally integrated tuple i.e., t can be in the FD result iff:

- either, t is a maximally integrated tuple because on integrating T_1, T_2,... T_k where, k $\leq$ n, t is produced and not subsumed by any other tuple.
- or, t is a maximally integrated tuple because t $\in \{T_1, T_2, \ldots T_n\}$ and t' $\notin \mathcal{F}$ such that t' $\supset$ t.

Now we show that ALITE does not miss any maximally integrated tuple that fulfills the first condition. For this, we show that the complementation operator produces the maximally integrated tuple that fulfills the first condition. Two tuples $t_i \in T_i$ and $t_j \in T_j$ can be integrated (or joined) iff for each join column $A \in \mathcal{A}(T_i) \cap \mathcal{A}(T_j)$, $t_i[A] = t_j[A]$ and $t_i[A] \neq \pm$ and $t_j[A] \neq \pm$. Also, for joinable tuples $t_i \in T_i$, $t_j \in T_j$ having same value in join columns and having column $A_i \in T_i$ and $A_j \in T_j$, $t_i[A_i]$ and $t_j[A_j]$ are always non-null because even if there are missing nulls, we replace them with distinct labeled nulls. Due to the way outer union operator works, $t_j[A_i] = t_i[A_j] = \perp$ i.e., they fulfill the complementation condition. This ensures that the complementation operator combines the joining tuples. Now, for each tuple in outer unioned partition, ALITE applies up to n iterations of complementation operator. As the setup ensures that complementation operator joins a pair of tuples from two tables in each iteration, it generates all possible maximally integrated tuples after at most n rounds even if the input tuples has join partner in all n input tables. Hence, we do not miss any maximally integrated tuples after this step. i.e., t $\in \mathcal{F}$.

For the second type of maximally integrated tuples, let t be a tuple from any table $T \in \mathcal{T}$ having no join partners. Such a tuple will not complement with any other tuples and remains on the complementation output in its original form padded with produced

nulls at non-join columns (Line 15- 16 of Algorithm 2). Hence, t $\in \mathcal{F}$.

As we apply subsumption after complementation, any non-maximally integrated tuples get subsumed. Hence, ALITE neither misses a maximally integrated tuple, nor produces a subsumable tuples i.e., the relation computed by ALITE over the input tables is exactly equal to the natural full disjunction. This completes the proof. □

5.3 Efficient Complementation

We showed how Algorithm 1 computes FD in $O[n.max(F,S).S + FlogF]$ time. We also discussed the efficient implementation of subsumption. Next, we will describe how we implement complementation more efficiently. Note that Algorithm 2 receives S tuples from Algorithm 1 for complementation, and the time taken by Line 7- 21 in Algorithm 2 is $O[n \cdot max(F,S) \cdot S]$. However, we know that for two tuples to complement each other, they must have the same non-null values on the common column. We illustrate this with an example.

Example 19. *Consider column* `Stadium` *and tuples* t_1, t_2, t_7 *and* t_{10} *of table (a) in Fig. 2. Also recall the necessary conditions for two tuples to complement each other described in Section 2. Since* $t_1[Stadium]$ *= NRG Stadium and* $t_2[Stadium]$ *= AT&T Stadium, they cannot complement each other as they have different non-null values on the common column* `Stadium`*. Hence, we can safely escape the comparison between* t_1 *and* t_2 *while applying complementation. Also, as* $t_{10}[Stadium]$ *= AT&T Stadium, it has a possibility of complementing* t_2*. So, we need to compare* t_2 *and* t_{10}*. Notice however, tuple* t_7 *complements* t_{10} *even though it has a missing null on* `Stadium`*–different from* $t_{10}[Stadium]$ *= AT&T Stadium. Therefore, a tuple having a missing null on the common column should still be compared with all other tuples.*

In general, we can partition the tuples having different non-null values on common columns and apply complementation within each partition using Algorithm 2. This helps to reduce the computation time taken by Line 7- 21 in Algorithm 2. Our objective is to make each partition fairly small, i.e., keep the number of tuples in each partition less than a positive integer θ where, $\theta < S$. Bleiholder et al. suggested to partition tuples using the values of selected partitioning column(s) [7]. The selection of the partitioning column(s) is based on a heuristic that considers the number of non-null and unique values on each column. At first, the tuples having the same non-null values in the partitioning column are kept in separate partitions. If there are tuples having null values in the partitioning column(s), they are added to all the partition. Now, the complementation can be applied on the tuples within each partition. However, partitioning with a single or even a group of columns may still produce large partitions.Therefore, instead of stopping after the first partitioning, we continue the process using other columns one after another until the number of tuples in each partition is less than θ. Recall that the tuples in the null partitions should be added to each of other partitions. Hence, in order to reduce the number of tuples in the null partitions, we first sort the columns in ascending order of the number of nulls they contain. Then, we partition the tuples by value of each column one by one.

6 EXPERIMENTS

We now evaluate the two steps involved in ALITE.

6.1 Experimental Setup

We implemented ALITE and all the baselines using Python 3.7 and performed experiments using a CentOS server having Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz processor. The main objective of our experiments are to answer the following questions.

(1) How accurate is our Column Integration ID Assignment method in comparison to the existing attribute matching techniques?
(2) How well does our FD algorithm scale in comparison to the state-of-the-art FD algorithms?
(3) We also consider whether it is worthwhile to use FD instead of the faster (and widely available) outer-join operator. Specifically, we study how many FD tuples are missed by outer-join when integrating real data lake tables.

We now provide additional implemetation details.

Embedding Generation. Recall that we use pre-trained embeddings to represent the columns for clustering (and integration ID assignment). Before using TURL [23], our method of choice to generate embeddings, we pre-process the tables using their implementation.[2] This phase includes, for example, generating a Wikipedia entity dictionary to map values in the tables. TURL was designed for web tables. Hence, it has a limited capacity in terms of the number of rows and columns it can use in creating embeddings (mean of ~20 rows and ~2 columns, see [23]). Since typical data lake tables are much larger (see Fig. 5), to cope with such a limitation, we designed an iterative embedding generation approach for each column. At the first iteration, we randomly sample 50 rows and generate the corresponding embedding by averaging the representations of each row. Then, at each iteration we sample 50 additional rows and combine them with the current embedding until convergence. Convergence is achieved if the euclidean distance between two consecutive embeddings is smaller than some value (0.05 in our setup).

Hierarchical Clustering. The generated embeddings are used to represent columns for clustering (see Section 4). We implement the clustering algorithm using Agglomerative Clustering module available in scikit learn library [50]. Based on our objective of obtaining dense, but well-separated clusters, we use the Silhouette Coefficient as a clustering quality measure [55]. We selected the number of clusters (column predicates) that maximizes the Silhouette Coefficient following the method of Section 4. We use euclidean distance as a distance metric throughout the experiments.

6.2 Evaluation Measures

To the best of our knowledge, no prior work considers the integration of data lake tables after discovery. So, we compare the different components of our pipeline to some approximate baselines.

Column Integration ID Assignment: The column integration ID assignment can be addressed using schema matching. Generally, precision, recall and their harmonic mean, i.e., F1-score are used as the evaluation measures for schema matching [12, 29, 57]. So, we

[2] https://github.com/sunlab-osu/TURL

use the same three metrics to compare our column integration ID assignment against existing schema matching methods. To assess the quality of a clustering-based solution using binary measures, we consider a pair of columns belonging to the same cluster as a match. Note that a column having no matches forms a singleton cluster, i.e., a cluster having one column. We count each such cluster as a true classification during the evaluation. Formally, let $\mathcal{T}_M$ be the sum of the number of column pairs belonging to the same cluster and the number of singleton clusters according to the ground truth and $\hat{\mathcal{T}}_M$ be the number of column pairs belonging to the same cluster and the number of singleton clusters according to a method. We define Precision (P), Recall (R) and F1-score (F_1) as follows:

$$P = \frac{\mathcal{T}_M \cap \hat{\mathcal{T}}_M}{\hat{\mathcal{T}}_M}, R = \frac{\mathcal{T}_M \cap \hat{\mathcal{T}}_M}{\mathcal{T}_M}, F_1 = \frac{2 \cdot P \cdot R}{P + R} \quad (1)$$

We compute precision, recall and F1-score for each set of tables to be integrated (integration set) and report the average. In addition, we also report the time taken by each method to determine the column predicates.

Full Disjunction: Our objective is to show that our proposed FD algorithm is faster in integrating data lake tables in comparison to the state-of-the-art methods for computing the FD. Therefore, we will report the time taken to compute Full Disjunction by each method. A cut-off of 10k seconds was used when applying FD. Furthermore, it is interesting to see how many tuples generated by FD can also be generated by the relatively faster outer-join over real data lake tables. Recall that outer-join is not an associative operator and there may exist outer-join orderings that yield the semantics of Full Disjunction when the scheme graph of the input tables does not contain a γ-cycle [53]. But the data lake tables to be integrated may contain gamma cycles in which case an outer join may not compute the FD. We quantify this using the Tuple Difference Ratio (TDR) as a success metric. Let F be the FD output size. Let, F' denote an outer join output size. The Tuple Difference Ratio (TDR) is given by:

$$TDR = \frac{F \cap F'}{F} \quad (2)$$

If an outer join produces all FD tuples, TDR is equal to 1 and it is equal to 0 if it produces none of them.

6.3 Baselines

We separate our baselines into column integration ID detection baselines and FD baselines.

Column Integration ID Assignment. Recall that we use a clustering-based approach and pre-trained embeddings created for the tables's columns [23] to find the column integration IDs. Other existing natural language embeddings were successfully adopted for similar tasks such as table search [8, 47] and column type annotation [59]. In what follows, we compare the performance of such embeddings also for our task. Specifically, as done for table search [8, 47], we use **Fasttext** [35, 44] embeddings of columns and as done for column annotation [59], we use **BERT** [24] embeddings. We use a publicly available Fasttext model [28] using the Gensim python package [31]. We generate BERT-base embeddings [2] using the commonly used hugging face package [27].

We also compare our Column Integration ID Assignment technique with existing schema matching methods. There are numerous matching approaches in the literature [25, 39, 42, 57]. However, most work relies on metadata, which we aim to avoid in our setting. Recently, Koutras et al. performed a detailed analysis of the existing schema matching methods in a data lake setting [39]. Based on the result of their analysis, we select a method proposed by Zhang et al. as a baseline [62] (**DB**) that is closely related to our problem. This method discovers clusters of similar attributes in tables using information that includes attribute data types, overlap of the attribute values, and their distribution. Earth Mover's Distance is used to measure the similarity between the column pairs [56]. A threshold is applied over this score to decide the column similarity. We use a threshold of 0.15 suggested by Zhang et al. [62]. Also, we reproduce *DB* using the open source code provided by Koutras et al. [39]. [3]

Full Disjunction. Paganelli et al. recently suggested **ParaFD** to compute the FD of relational tables where all joins are between keys and foreign keys using multiple machines [49]. In a data lake, we are often not joining on keys and foreign keys, so we compare ALITE against ParaFD only in a benchmark having such relationships.

We also use **BICOMNLOJ**, which computes the FD with a polynomial delay between them [17]. As our focus is to compute full FD, we report the performance of **BICOMNLOJ** for computing the full FD. Also, BICOMNLOJ is based on the tuple sets and hence, if the input contains nulls its output may contain some subsumable tuples (see Example 11). Therefore, to ensure that the output produced by this algorithm is the same as other algorithms, we apply subsumption to its final result. For fair comparison, we apply the same subsumption algorithm that we use for our approach [6]. sSince an open-source implementation is not available for either ParaFD or BICOMNLOJ, we reproduce them using the information provided in the paper.[4] The reproduced implementations are publicly available in our github repository.[5]

Moreover, we also ran outer join to integrate the tables (**Outer**) and use its output size to report TDR. As outer join is not associative, the order of integration makes a significant difference [30]. Applying outer join in a connected-prefix ordering of the input tables can yield FD for γ-acylic case [17]. Therefore, we use the connected-prefix ordering for each schema to compute the outer join. The connected-prefix ordering is obtained by performing DFS transversal over the input scheme graph [17].

6.4 Benchmarks

Benchmark	Tables	Columns	Tuples	Integration sets	Experiments
Align	606	4,584	2.2M	65	Integration ID
Real	102	1, 195	219k	11	Integration ID, FD
Join	302	2, 309	1.1M	28	FD
IMDB	6	33	3k - 30k	1	FD

Figure 5: Benchmarks used in the experiments.

Figure 5 summarizes all benchmarks used in different experiments along with their statistics. Each benchmark contains multiple

[3] https://github.com/delftdata/valentine
[4] Note that we implement ParaFD to run on a single machine for fair comparison.
[5] https://github.com/northeastern-datalab/alite

tables with different schemas and each schema may be used by multiple tables. All the benchmarks were made publicly available.[5]

Align. To the best of our knowledge, there are no available data lake benchmarks that could be adapted to evaluate the column integration ID assignment task. So we create a new benchmark called *Align* containing 606 tables divided into 65 non-overlapping sets of tables, which we call *integration sets*. We run the column integration ID Assignment methods over the columns of tables of each integration set and report the average performance. To create this benchmark, we follow a similar technique used to create a table union benchmark [47]. First, we select 65 real data lake tables from US Open Data [34], Canada Open Data [20], and UK Open Data [21] and consider them as seed tables. Each seed table has a different schema and is used to generate an integration set. We partitioned the seed tables by projecting columns and selecting rows (without replacement) to get 606 smaller tables such that all the columns of the small tables that originated from the same seed column have the same integration ID. Accordingly, we have labeled ground truth for the column integration ID assignment. Based on the number of columns and rows in the seed tables, each integration set contains 2 to 30 tables. Note that we do not add or remove missing nulls in the seed tables before partitioning. Therefore, if there is a missing null in the seed table row, it gets copied to the small tables. On average, since these are real data lake tables, we have null values in 50% of the rows. This ensures that our benchmark well-represents the real data lake scenario where such nulls are prevalent.

Real. To understand the performance of different methods in a real data lake environment, we also created the *Real* Benchmark that contains 102 real data lake tables divided into 11 disjoint integration sets. We ensure that the scheme graph of the tables in each integration set is connected. Furthermore, two real tables can have different column headers for the join columns. Therefore, we manually marked the join columns and labeled the groundtruth. We use this benchmark to evaluate the effectiveness of column integration ID assignment and efficiency of FD. It is interesting to evaluate FD computation for different input sizes (S) and output sizes (F). Therefore, we also ensure that the benchmark covers $F < S$, $F \approx S$ and $F > S$ cases. Precisely, in this benchmark, there are three integration sets where $F < S$, five integration sets where $F \approx S$, and three integration sets where $F > S$.[6] The number of tables (n) on each integration set ranges from 5 to 14. Also, S and F ranges from 588 to 76k and 580 to 60k respectively.

Join. Except renaming column headers, we do not modify the Real Benchmark and it contains raw tables searched from the data lakes. Therefore, to experiment our algorithm in broader contexts, like for variation in the input size, output size and the number of tables in each integration set, we create Join Benchmark that contains 28 integration sets generated using 27 seed tables–at most two integration sets from each seed. Each integration set contains 2 to 20 tables. We follow a similar methodology as used in Nargesian et al. [47] as explained in the Align Benchmark, but this time we also consider broader variation in the number of input and output tuples and also their ratio. The input tuple size (S) varies from 266 to 100k and range of output size is from 234 to 12M. There are 17 integration sets with $F < S$ among which six have $F < 0.5S$. Also, five integration sets have $F \approx S$ and six integration sets have $F > S$.

IMDB. As ParaFD can only be used for the tables having PK-FK relationships, we also use an IMDB dataset, having such relationships for our experiments.[7] This is a dataset about movies and their details including ratings, crews, etc. The full dataset contains about 106.8 M tuples in 6 tables. We use this benchmark to study the effect of different input size on the run time. Previous work uses 1k tuples in each table to evaluate the run time [17]. Therefore, to study the trend on similar setting , we sample tuples randomly and vary the input size between 500 to 5000 for each table– around 3k to 30k input tuples in total for our experiments. We preserve PK-FK relationships during sampling.

6.5 Column Integration ID Assignment Results

We now report the effectiveness of column integration ID assignment, followed by an empirical analysis of its efficiency.

Method		Benchmark					
		Align			Real		
		P	R	F_1	P	R	F_1
Baseline	DB	0.953	0.892	0.911	**0.807**	0.708	0.71
ALITE	fastText	**0.956**	0.923	0.94	0.689	**0.806**	0.722
	BERT	0.922	0.965	0.939	0.706	0.756	0.743
	TURL	0.934	**0.968**	**0.947**	0.776	0.762	**0.755**

Figure 6: Precision, recall and F1 over the *Align* and *Real* benchmarks for column integration ID assignment.

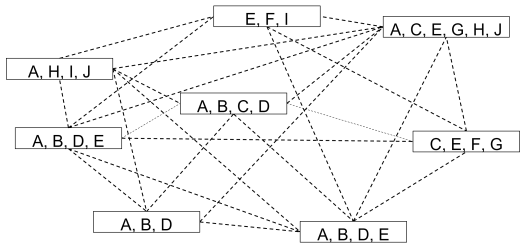

Figure 7: Scheme graph of a synthesized integration set used in the experiments.

Recall that ALITE uses a clustering-based approach to find the column integration IDs that uses pre-trained embeddings created using TURL [23]. So, first we compare ALITE's precision, recall and F1-Score using TURL based embeddings against fastText and BERT embeddings. We use the same experimental setups for all three methods for fair comparison (see Section 6.1). Fig. 6 shows the evaluation results for the *Align* and *Real* benchmarks. We first compare the different embeddings, namely, fastText, BERT and TURL, for ALITE. It is seen that using TURL gives comparable or even better precision and recall against the baselines. In terms of F_1-score, TURL performs better than the baselines. This suggests that a table-based embedding (TURL) performs better that natural

[6] We consider $F \approx S$ when $\frac{|F-S|}{S} < 0.05$.

[7] https://datasets.imdbws.com/

language embedding (fastText and BERT) for data lake tables. We will explore alternative ways to represent data lake tables in future research.

Next, we compare the effectiveness of ALITE's embedding-based technique against *DB* that uses attribute data types, values and distribution to find the similar columns. The *DB* approach has a slightly better precision than our method on the *Align* benchmark. However, ALITE outperforms *DB* by more than 8% in terms of recall and 3% in terms of F_1-score. Also on the *Real* benchmark, *DB* is better than our method in terms of precision by around 3%, but ALITE is better in recall and F_1-score. A possible reason for the preciseness of *DB* is the use of a threshold in pairwise comparisons. Specifically, the default threshold is set to 0.15 and so the precision is fairly high. However, note that accordingly the recall also drops. An additional reason for the lower recall is that *DB* relies on value overlap and semantics (e.g., synonyms) are ignored.

The column integration ID assignment is considered as an offline task. Yet, we note that applying ALITE's clustering is much faster than the pair-wise comparison done by the *DB* baseline. Specifically, ALITE takes about 10 minutes for *Align* and about 15 minutes for *Real* while *DB* takes about 45 minutes (×4.5) for *Align* and about 2 hours (×8) for *Real*. Comparing the embedding generation, fastText is the fastest (total of 28 seconds for *Align* and 3 seconds for *Real*) as the embeddings are pre-defined. TURL and BERT, for which a pre-trained model is used, show somewhat different trends. For the *Align* benchmark BERT takes a total of 80 minutes while TURL takes about 7 minutes. For the *Real* benchmark they take approximately the same time (about 15 minutes).

6.6 Full Disjunction Results

Now we compare the efficiency of ALITE's proposed FD algorithm (Algorithm 1) against the baselines (see Section 6.3). We also analyze the run time of our algorithm by varying the input and output sizes. Finally, we compare the FD output with the outer join output in terms of *TDR* (the relative size of the outputs, see Section 6.1).

6.6.1 ALITE against baselines. Before experimenting with our data lake benchmarks, we conducted a preliminary analysis over three synthetic integration sets ($\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3$) introduced in Cohen et al. [17]. We reproduced these by randomly generating 1000 input tuples in each of the 10 tables in each integration set. Unsurprisingly, since these schema contain biconnected components [17], *BICOMNLOJ* splits the tables into smaller integration sets, computes FD for each of them separately and combine them. Therefore, *BICOMNLOJ* is much faster than ALITE. As a second step, we created a new, more complex, integration set having eight tables that better represents data lake tables. The scheme graph of this integration set is shown in Figure 7. Each rectangular box shows a table and its attributes. Each table pair sharing a common attribute have an edge between them. We again fix the number of tuples on each input table to 1k, i.e., S = 8000 as the integration set has eight tables. We added tuples to the tables in such a way as to create three cases: $F < S$ (F = 3868), $F \approx S$ (F = 7445) and $F > S$ (F = 14204). For all three cases, ALITE outperforms *BICOMNLOJ* by at least one order of magnitude. *BICOMNLOJ* could not optimize the computation because there is only one biconnected component. Note that this is a common case in data lakes due to the presence of complex cycles in the scheme graphs.

We also compare the time taken by ALITE's FD algorithm against the baseline *BICOMNLOJ* in *Real* Benchmark.[8] Figure 8(a) summarizes this experiment. Each pair of bars on the X-axis represents a schema and the Y-axis shows the time taken to integrate the tables by ALITE (blue) and *BICOMNLOJ* (red). The tables in an integration set are ordered by input size such that the smallest is shown on the left and the largest in the right. ALITE's FD algorithm (blue bars) is significantly faster than *BICOMNLOJ* (red bars) over all 11 integration sets. Specifically, the cases where the cut-off was not applied (all but the last three), ALITE boosts the performance of *BICOMNLOJ* by around two orders of magnitude. The reason for this gain comes from the fact that our algorithm partitions tuples according to their complementation patterns and iterates over the tuples only within the partitions. This leads to an interesting insight, showing the impact of the complementation operator in optimizing the FD computation for data lake tables. Another reason is that data lake tables have complex join connections that limit the chances of dividing the tables of integration sets into biconnected components, which is used in *BICOMNLOJ*. We see the same trend on *Join* Benchmark (shown in Fig. 8 (b)) where, ALITE outperforms *BICOMNLOJ* on all integration sets by around one and half orders of magnitude. As in *Real*, we are much faster for the integration sets having different output to input ratio. Also, it is notable that out of 28 integration sets, *BICOMNLOJ* computes the full FD result within the cutoff time for only 13 integration sets that are shown in Fig. 8(b). Generally, *BICOMNLOJ* is able to compute FD within the cutoff time for input sizes less than 45k. For the remaining 15 integration sets, the average integration time by ALITE ranges from 20 seconds to 3827 seconds with an average of 598 seconds–well below the cut-off time (10k seconds) that we used in the experiments. This shows that ALITE is more applicable than the baseline for the data lake tables with large input size. we also observed that tuple-set FD produces over 300 subsumable tuples per integration set in the Real Benchmark which supports the subsumption step in ALITE.

Moreover, we compare ALITE's FD algorithm against both *BICOMNLOJ* and *ParaFD* in *IMDB*– a benchmark having six tables and large number of join connections. As shown in Fig. 9 (a), we vary the number of input tuples (S) from 0 to 30k and observe the runtime. Note that, when we increase the number of input tuples, the output size also increases in this benchmark. Therefore, we also show the integration time with respect to the output size (Fig. 9 (b)). It is seen that ALITE gives comparable performance against *ParaFD* and is more than two times faster than *BICOMNLOJ*. Recall that *ParaFD* needs all joins to be key to foreign-key joins to compute FD. It uses this property to optimize the computations and hence, performs relatively better than other techniques on *IMDB*. However, *ParaFD* cannot be used for the tables without PK-FK relationship. ALITE does not assume the presence of such relationships and hence, can be used to integrate tables forming any scheme graphs. Due to space constraints, we provide other details like number of tables on each integration set, number of

[8] Recall that the second baseline, *ParaFD*, is not applicable for this benchmark, see Section 6.3.

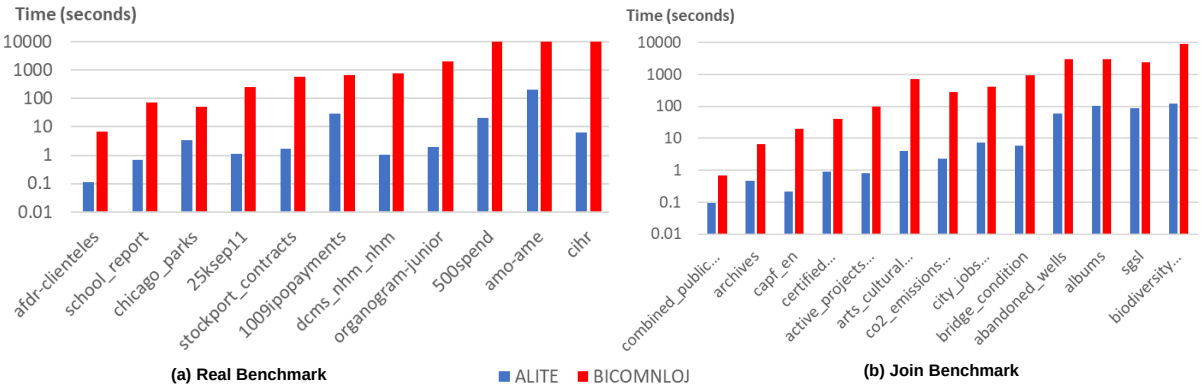

Figure 8: Time taken to integrate tables in (a) *Real* Benchmark and (b) Join Benchmark. The integration sets in X-axis are arranged in ascending order of input size. Y-axis (log scale) shows the integration time. A 10K second cut-off was used in both benchmarks. Some integration set names are truncated for concise representation. Due to space considerations, we only show the integration sets on which the baseline integrates the tables before the cut-off time in Join Benchmark. The details of other integration sets are provided with the supplementary materials.

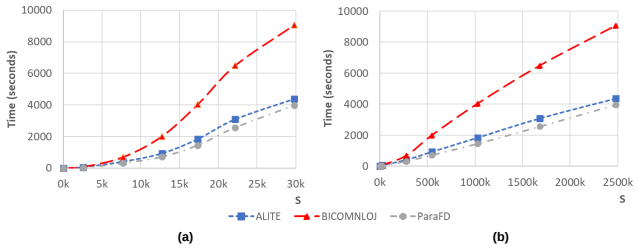

Figure 9: Integration time in the IMDB benchmark for (a) different input size and (b) different output size.

columns, input size, output size, number of missing nulls in each benchmark with the supplementary materials.[5]

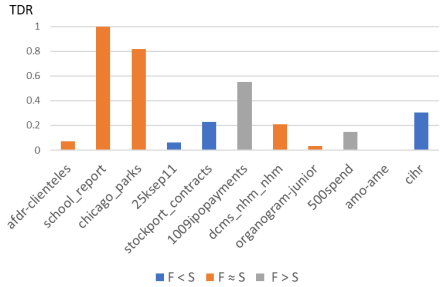

Figure 10: Tuple Difference Ratio (TDR) of the integration sets in the *Real* Benchmark.

6.6.2 Comparison against outer join. Lastly, we show the importance of using FD against outer join empirically in real data lake tables using the *Real* Benchmark. The bargraph in Fig. 10 shows each integration set of this benchmark on the X-axis and TDR in Y-axis (see Section 6.2). We show the schemas based on three categories: $S < F$, $S \approx F$ and $S > F$. Recall that all these schemas contain complex cycles. Out of 11 integration sets, only once is *TDR* is equal to one (school_report), i.e., all FD tuples are generated by outer join. It is interesting that even in the presence of complex cycles, outer join can sometimes produce the full FD. For two integration sets (chicago_parks and 1009ipopayments), outer join is able to generate more than half of the FD tuples. But for other sets, *TDR* is very low, which shows the importance of FD to best integrate real tables.

7 CONCLUSION

We introduce a novel problem of integrating data lake tables after discovery. We also introduce ALITE, a technique to solve this problem in two steps. ALITE first assigns an integration ID to each column and then applies natural full disjunction to integrate the tables. We present and share three open data integration benchmarks that may be of interest to the research community. We show that ALITE's new FD algorithm is more efficient than existing baselines and ALITE does not require tables to have common schemas.

REFERENCES

[1] Basel Alshaikhdeeb and Kamsuriah Ahmad. 2015. Integrating correlation clustering and agglomerative hierarchical clustering for holistic schema matching. *Journal of Computer Science* 11, 3 (2015), 484.

[2] Hugging Face BERT base model (uncased). 2022. https://huggingface.co/bert-base-uncased

[3] Purnima Bholowalia and Arvind Kumar. 2014. EBK-means: A clustering technique based on elbow method and k-means in WSN. *International Journal of Computer Applications* 105, 9 (2014).

[4] Jens Bleiholder, Melanie Herschel, and Felix Naumann. 2011. Eliminating NULLs with Subsumption and Complementation. *IEEE Data Eng. Bull.* 34, 3 (2011), 18–25.

[5] Jens Bleiholder and Felix Naumann. 2009. Data Fusion. *ACM Comput. Surv.* 41, 1, Article 1 (Jan. 2009), 41 pages. https://doi.org/10.1145/1456650.1456651

[6] Jens Bleiholder, Sascha Szott, Melanie Herschel, Frank Kaufer, and Felix Naumann. 2010. Subsumption and complementation as data fusion operators. In *Proceedings of the 13th International Conference on Extending Database Technology*. 513–524.

[7] Jens Bleiholder, Sascha Szott, Melanie Herschel, and Felix Naumann. 2010. Complement union for data integration. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 183–186.

[8] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 709–720. https://doi.org/10.1109/ICDE48307.2020.00067

[9] Dan Brickley, Matthew Burgess, and Natasha Noy. 2019. Google Dataset Search: Building a Search Engine for Datasets in an Open Web Ecosystem. In *The World*

Wide Web Conference (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 1365–1375. https://doi.org/10.1145/3308558.3313685
[10] Michael J. Cafarella, Alon Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proc. VLDB Endow.* 2, 1 (aug 2009), 1090–1101. https://doi.org/10.14778/1687627.1687750
[11] Tadeusz Caliński and Jerzy Harabasz. 1974. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods* 3, 1 (1974), 1–27.
[12] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1335–1349. https://doi.org/10.1145/3318464.3389742
[13] Raul Castro Fernandez, Essam Mansour, Abdulhakim A. Qahtan, Ahmed Elmagarmid, Ihab Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 989–1000. https://doi.org/10.1109/ICDE.2018.00093
[14] Adriane Chapman, Elena Simperl, Laura Koesten, George Konstantinidis, Luis-Daniel Ibáñez, Emilia Kacprzak, and Paul Groth. 2020. Dataset search: a survey. *VLDB J.* 29, 1 (2020), 251–272.
[15] 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.
[16] E. F. Codd. 1979. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.* 4, 4 (dec 1979), 397–434. https://doi.org/10.1145/320107.320109
[17] Sara Cohen, Itzhak Fadida, Yaron Kanza, Benny Kimelfeld, and Yehoshua Sagiv. 2006. Full Disjunctions: Polynomial-Delay Iterators in Action. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 739–750.
[18] S. Cohen and Y. Sagiv. 2007. An incremental algorithm for computing ranked full disjunctions. *J. Comput. Syst. Sci.* 73 (2007), 648–668.
[19] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding Related Tables. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 817–828. https://doi.org/10.1145/2213836.2213962
[20] Canada Open Data. 2020. https://open.canada.ca/en/open-data
[21] UK Open Data. 2020. https://data.gov.uk/
[22] David L. Davies and Donald W. Bouldin. 1979. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1, 2 (1979), 224–227. https://doi.org/10.1109/TPAMI.1979.4766909
[23] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: table understanding through representation learning. *Proceedings of the VLDB Endowment* 14, 3 (2020), 307–319.
[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *ArXiv* abs/1810.04805 (2019).
[25] Hong-Hai Do and Erhard Rahm. 2002. COMA—a system for flexible combination of schema matching approaches. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 610–621.
[26] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 456–467.
[27] Hugging Face. 2022. https://huggingface.co
[28] fastText. 2022. https://fasttext.cc/docs/en/english-vectors.html
[29] Avigdor Gal, Haggai Roitman, and Roee Shraga. 2019. Learning to rerank schema matches. *IEEE Transactions on Knowledge and Data Engineering* (2019).
[30] César A. Galindo-Legaria. 1994. Outerjoins as Disjunctions. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 348–358. https://doi.org/10.1145/191839.191908
[31] Gensim. 2022. https://radimrehurek.com/gensim
[32] Johannes Grabmeier and Andreas Rudolph. 2002. Techniques of cluster algorithms in data mining. *Data Mining and knowledge discovery* 6, 4 (2002), 303–360.
[33] Bin He and Kevin Chen-Chuan Chang. 2005. Making holistic schema matching robust: an ensemble approach. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 429–438.
[34] The home of the U.S. Government's open data. 2020. https://data.gov/
[35] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).
[36] Jaewoo Kang and Jeffrey F Naughton. 2003. On schema matching with opaque column names and data values. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 205–216.
[37] Yaron Kanza and Yehoshua Sagiv. 2003. Computing Full Disjunctions. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (San Diego, California) *(PODS '03)*. Association for Computing Machinery, New York, NY, USA, 78–89. https://doi.org/10.1145/773153.773162
[38] Trupti M Kodinariya and Prashant R Makwana. 2013. Review on determining number of Cluster in K-Means Clustering. *International Journal* 1, 6 (2013), 90–95.
[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 *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 468–479.
[40] Oliver Lehmberg and Christian Bizer. 2017. Stitching Web Tables for Improving Matching Quality. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1502–1513. https://doi.org/10.14778/3137628.3137657
[41] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 1338–1347. https://doi.org/10.14778/1920841.1921005
[42] Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. 2001. Generic schema matching with cupid. In *vldb*, Vol. 1. Citeseer, 49–58.
[43] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. 2002. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th international conference on data engineering*. IEEE, 117–128.
[44] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in Pre-Training Distributed Word Representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.
[45] Renée J Miller. 2018. Open data integration. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2130–2139.
[46] 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 (aug 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116
[47] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (mar 2018), 813–825. https://doi.org/10.14778/3192965.3192973
[48] Paul Ouellette, Aidan Sciortino, Fatemeh Nargesian, Bahar Ghadiri Bashardoost, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2021. RONIN: Data Lake Exploration. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2863–2866. https://doi.org/10.14778/3476311.3476364
[49] Matteo Paganelli, Domenico Beneventano, Francesco Guerra, and Paolo Sottovia. 2019. Parallelizing Computations of Full Disjunctions. *Big Data Research* 17 (2019), 18–31. https://doi.org/10.1016/j.bdr.2019.07.002
[50] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[51] Jin Pei, Jun Hong, and David Bell. 2006. A novel clustering-based approach to schema matching. In *International Conference on Advances in Information Systems*. Springer, 60–69.
[52] Erhard Rahm and Philip A Bernstein. 2001. A survey of approaches to automatic schema matching. *the VLDB Journal* 10, 4 (2001), 334–350.
[53] Anand Rajaraman and Jeffrey D. Ullman. 1996. Integrating Information by Outerjoins and Full Disjunctions (Extended Abstract). In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Montreal, Quebec, Canada) *(PODS '96)*. Association for Computing Machinery, New York, NY, USA, 238–248. https://doi.org/10.1145/237661.237717
[54] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems (3. ed.)*. McGraw-Hill.
[55] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65. https://doi.org/10.1016/0377-0427(87)90125-7
[56] Y. Rubner, C. Tomasi, and L.J. Guibas. 1998. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. 59–66. https://doi.org/10.1109/ICCV.1998.710701
[57] Roee Shraga, Avigdor Gal, and Haggai Roitman. 2020. Adnev: Cross-domain schema matching using deep similarity matrix adjustment and evaluation. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1401–1415.
[58] Weifeng Su, Jiying Wang, and Frederick Lochovsky. 2006. Holistic schema matching for web query interfaces. In *International Conference on Extending Database Technology*. Springer, 77–94.
[59] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çağatay Demiralp, Chen Chen, and Wang-Chiew Tan. 2021. Annotating Columns with Pre-trained Language Models. *arXiv preprint arXiv:2104.01785* (2021).
[60] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*.
[61] Jiang Zhan and Shan Wang. 2007. ITREKS: Keyword search over relational database by indexing tuple relationship. In *International Conference on Database Systems for Advanced Applications*. Springer, 67–78.

[62] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. 2011. Automatic Discovery of Attributes in Relational Databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/1989323.1989336

[63] Yi Zhang and Zachary G Ives. 2020. Finding related tables in data lakes for interactive data science. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1951–1966.

[64] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*. 847–864.

[65] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-join: Joining tables by leveraging transformations. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1034–1045.

[66] 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.