Technical Report on Integrating Data Lake Tables

Aamod Khatiwada
Northeastern University
Boston, Massachusetts, USA
khatiwada.a@northeastern.edu

Roee Shraga
Northeastern University
Boston, Massachusetts, USA
r.shraga@northeastern.edu

Wolfgang Gatterbauer
Northeastern University
Boston, Massachusetts, USA
w.gatterbauer@northeastern.edu

Renée J. Miller
Northeastern University
Boston, Massachusetts, USA
miller@northeastern.edu

ABSTRACT

We have made tremendous strides in providing tools for data scientists to discover new tables useful for their analyses. 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 progress in using it 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 empirically show that ALITE can outperform previous algorithms for computing the Full Disjunction. ALITE relaxes previous assumptions that tables share common attribute names (which completely determine the join columns), are complete (without null values), and have acyclic join patterns. To evaluate ALITE, we develop and share three new benchmarks for integration that use real data lake tables.

PVLDB Reference Format:
Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. Technical Report on Integrating Data Lake Tables. PVLDB, 16(4): 932 - 945, 2022.
doi:10.14778/3574245.3574274

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

1 INTRODUCTION

The number of public datasets has grown immensely in open data platforms [57, 58, 80]. Also, 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 datasets such as keyword search [11, 12, 59, 77] and table search (using the data within their table as a query) [10, 29, 49, 51, 58, 82]. Such a process usually outputs a collection of data lake tables that may enrich their analysis [57]. Existing techniques usually discover unionable [12, 42, 49, 58], joinable [27, 56, 77, 79, 80], and related tables [10, 20, 79].

Example 1. *Consider the data lake tables about football stadiums shown in Fig. 1. We have added a* `TID` *(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 [58] and the top-2 joinable tables [80] 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. Join search finds tables that join on an indicated column (in this case* `Location`*), but does not 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.*

After discovery, data scientists would often integrate the discovered tables before analyzing and applying statistical tools. Such integration not only extends their data but also allows them to answer queries that go beyond a single table. Consider Tables T_2, T_3 and T_4 in Fig. 1 and assume a football team has data scientists assisting in finding a new coach. Specifically, the team looks for an experienced coach who has handled teams playing in front of large crowds in new stadiums. So, they may use queries such as *"coaches who coach teams having stadiums established after 2000, that accommodate at least 50 thousand spectators"*. The information required here goes beyond a single table. In our example, one needs (at least) to integrate T_2, T_3 and T_4 to obtain such facts, for example, `Dan Campbell` who coaches the `Detroit Lions` that uses `Ford Field` Stadium established in `2002` and having a capacity of `65k` (f_7 in Fig. 2). Prior search methods do not address this "post-discovery" phase and do not answer the important question of how to integrate tables (relations) obtained by 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) [58]. 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 just leave* `Location`*. For the joinable tables, a join on only* `Location` *of T_1 with T_4 leads to tuples like t_{11} being omitted and the result has two* `Stadium` *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.*

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574274

T_1

TID	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

TID	Stadium	Location	Opened
t_5	Soldier Field	Chicago	1924
t_6	Ford Field	Michigan	2002

T_3

TID	Team	Location	Coach
t_7	Houston Texans	Texas	Lovie Smith
t_8	Green Bay Packers	Wisconsin	Matt LaFleur
t_9	Detroit Lions	Michigan	Dan Campbell

T_4

TID	Stadium	Location	Capacity
t_{10}	NRG Stadium	Texas	±
t_{11}	Ford Field	Michigan	65k

T_5

TID	Stadium	Location	Team
t_{12}	Lambeau Field	Wisconsin	Green Bay Packers
t_{13}	±	Ohio	Cleveland
t_{14}	Sofi Stadium	California	±

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

Within the data integration literature, *Full Disjunction (FD)* [33] has been understood as a natural way of assembling partial pieces of information (facts) such that it maximizes the connections among these facts [67]. Indeed, Rajamaran and Ullman describe FD as a relation with nulls (denoted by ⊥) such that every set of join-consistent tuples appears within an FD tuple, with a concrete value or ⊥ in each attribute not found within the set of tuples [67]. Here, join-consistent is defined as common attributes (attributes with the same name), so this is effectively a *natural FD*. The widely known outer-join [47, 68] is not associative (hence, the result may depend on the order in which tables are integrated) and does not aim to maximize the connections among the integrated tuples [33, 53].

Example 3. *Outer-join and outer-union keep all tuples and columns and pad non-matching tuples (respectively, columns) with nulls [17, 47]. The outer-union of the tables from Fig. 1 is depicted in Fig. 2(a). It does not maximally connect the facts in the original tables. Here, ± indicates a missing value (missing null) in the original tables and ⊥ represents a null introduced by the outer-union operator (produced null). In particular, outer union includes partial facts like t_{10} that are made redundant by more complete facts like t_1. Similar observations can be made of the outer-join results. So, Galindo-Legario defined the Full Disjunction (FD) [33]. Informally, it removes redundant facts and produces, in this case, the first 8 tuples (mustard colored) of Fig. 2(b). FD can be viewed as an associative version of outer join [18] and has been used to integrate information across relational tables [18, 60] and web tables [60]. Notice in this example, FD uses information from all five tables so it is important to be able to compute FD over (possibly large) sets of tables.*

But using FD in data lakes poses several important challenges: 1) We cannot rely on common attributes having the same name as in our example. Instead, we must discover what the common attributes are [57]. We will use schema matching for this purpose [65]. Notice that we are not given just two schemas that need to be matched, rather we have a set of tables all of which could potentially share attributes with some or many of the other tables. Hence, we will use *holistic schema matching* [66]. 2) We cannot assume that integrated datasets are complete (that is, they may actually contain null values or partial facts). 3) Prior work on Full Disjunction has been done on relatively small relations (with only 1000 or so tuples per relation [18]) or assumes the common attributes form graphs with specific acyclic structures [67]. To the best of our knowledge, the only work on using FD on larger datasets requires that all joins be done on attributes having key-to-foreign-key relationships [60], 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.

We assume data scientists use table discovery algorithms to identify a set of tables that they wish to integrate. Regardless of the search technique, we wish to find the best way to integrate the tables. Specifically, we propose a table integration technique ALITE (Align and Integrate) that first applies schema matching to identify common columns in a set of tables to be integrated. Matched columns are given the same integration ID. We then apply a natural FD over the tables using integration IDs as attribute names.

Contributions. (1) To the best of our knowledge, we introduce the problem of integrating data lake tables obtained using table discovery algorithms. (2) We propose a new holistic schema matching algorithm for sets of tables that outperforms the state-of-the-art matchers on the real data lake tables in our integration benchmarks. (3) We compare the FD used as the integration semantics in ALITE to several other semantics and show the difference and superiority of FD. Empirically, we show FD's superiority for a downstreaming task of entity resolution [45]. (4) We propose a novel algorithm to compute the FD by using complementation and subsumption operators in a novel way. We show that the use of these operators permits optimizations that make the computation faster than the state-of-the-art techniques, in practice. Specifically, ALITE scales better than the state-of-the-art FD algorithms on data lake tables, which are typically large and may have complex join graphs. (5) We introduce and share several open data integration benchmarks.

2 PRELIMINARIES

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

Table 1: Symbols used in this paper and their definitions

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

Notation. Table 1 summarizes the used notation. For any operator, let $\mathcal{S}$ (and its size s) and $\mathcal{F}$ (and its size f) denote the collective set of all input and output tuples, respectively. We use two types of nulls: ± denotes a *missing null* i.e., missing value from an incomplete input relation to be integrated and ⊥ denotes a *produced null*, a null value that is introduced by an operator during integration.

TID	Stadium	Location	Team	Opened	Coach	Capacity
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	Ford Field	Michigan	⊥	2002	⊥	⊥
t_7	⊥	Texas	Houston Texans	⊥	Lovie Smith	⊥
t_8	⊥	Wisconsin	Green Bay Packers	⊥	Matt LaFleur	⊥
t_9	⊥	Michigan	Detroit Lions	⊥	Dan Campbell	⊥
t_{10}	NRG Stadium	Texas	⊥	⊥	⊥	±
t_{11}	Ford Field	Michigan	⊥	⊥	⊥	65k
t_{12}	Lambeau Field	Wisconsin	Green Bay Packers	⊥	⊥	⊥
t_{13}	±	Ohio	Cleveland	⊥	⊥	⊥
t_{14}	Sofi Stadium	California	±	⊥	⊥	⊥

(a) T1 ⊎ T2 ⊎ T3 ⊎ T4 ⊎ T5

OID	TIDs	Stadium	Location	Team	Opened	Coach	Capacity
f_1	$\{t_1, t_7, t_{10}\}$	NRG Stadium	Texas	Houston Texans	⊥	Lovie Smith	±
f_2	$\{t_2\}$	AT&T Stadium	Texas	Dallas Cowboys	⊥	⊥	⊥
f_3	$\{t_3\}$	Paul Brown	Ohio	±	⊥	⊥	⊥
f_4	$\{t_{13}\}$	±	Ohio	Cleveland	⊥	⊥	⊥
f_5	$\{t_4\}$	Sofi Stadium	California	Angeles Chargers	⊥	⊥	⊥
f_6	$\{t_5\}$	Soldier Field	Chicago	⊥	1924	⊥	⊥
f_7	$\{t_6, t_9, t_{11}\}$	Ford Field	Michigan	Detroit Lions	2002	Dan Campbell	65k
f_8	$\{t_8, t_{12}\}$	Lambeau Field	Wisconsin	Green Bay Packers	⊥	Matt LaFleur	⊥
f_9	$\{t_3, t_{13}\}$	Paul Brown	Ohio	Cleveland	⊥	⊥	⊥
f_{10}	t_{14}	Sofi Stadium	California	±	⊥	⊥	⊥

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

(b) Output tuples generated using different operators

Figure 2: Result of integrating the tables in Fig. 1 using different techniques. The table in (a) is the result of outer unioning the five tables. The table in (b) is the union of tuples obtained using FD (first eight tuples in mustard), a variant called tuple-set FD (which is the FD plus f_{10}), and Complement Union (⊞). A unique Output ID (OID) is provided for each output tuple for clarity.

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 running example is unrealistic as 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 is not the case in most realistic examples. Hence, we begin by using schema matching to assign *integration ids* to columns such that two matched columns will have the same id and two columns that are not matched 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 (Section 4).

2.2 Integration Operators

We assume that the reader is familiar with the elementary relational algebra operators like union (∪), join (⋈) and outer join(⟗) [68] based on which, we now introduce some (less well known) operators that we use as components of an integration solution.

Outer Union (⊎) is an extension to the union operator. It unions tables even if they do not have the same schema [17]. 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 union the padded relations.

Example 5. *The outer union of the tables in Fig. 1 is shown in Fig. 2(a). Here, the input size ($|\mathcal{S}|$ = 14) is the same as the output size ($|\mathcal{F}|$ = 14) 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), denoted by $t_1 \sqsupset t_2$, 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 values (either missing or produced) than t_2 [7, 33]. We denote the subsumption operation using β where $\beta(r)$ contains all tuples of r that are not subsumed by another tuple in r. Applying subsumption to the outer union result is called the *minimum union* (⊕) [33]. An example of minimal union (⊕) of tables in Fig. 1 is the set $\{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{11}, t_{12}, t_{13}\}$ in Fig. 2. This is because the tuple t_{10} is subsumed by t_1 and t_{14} is subsumed by t_4. Here, the size of the input ($|\mathcal{S}|$ = 14) is larger than the output ($|\mathcal{F}|$ = 12).

Complementation (κ). Two different tuples t_1 and t_2 having the same schema *complement* each other if: 1) there is at least one column A on which they have equal and non-null values; 2) for every column A where both tuples are non-null, the tuples must have the same value on A; 3) there is at least one column A on which t_1 is non-null and t_2 is null (missing or produced); and 4) there is at least one column A on which t_2 is non-null and t_1 is null (missing or produced) [7, 9]. The complementation of t_1 and t_2 is a tuple t_3 where 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 (hence, equal). 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* (⊞).

Example 6. *In Table(a) of Fig. 2, tuples t_3 and t_{13} complement each other. Their complementation is denoted as f_9 in Fig. 2(b), i.e., $\kappa(t_3, t_{13}) = f_9$. 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_1 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 $\{f_3, f_4\}$. Note that complementation union may not remove all subsumable tuples (e.g. f_{10}, which can be subsumed by f_5, is not complemented by f_5 since the fourth condition of complementation is not met).*

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) [33]. His proposal is essentially

a commutative, associative 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, as 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 an integrated tuple. Note that integrating those tuples that have missing nulls on their *common columns* may produce semantically incorrect tuples. Consider tuples t_3 from T_1, and t_{13} from T_5, while they share the value `Ohio` on `Location`, the value of `Stadium` is known in t_3 (`Paul Brown`), it is unknown in t_{13}. Therefore, we will not integrate these tuples. Notably, FD was later proposed as the right semantics for integrating data [67].

Example 7. *The FD of the five tables from Fig. 1 is the set of tuples $\{f_1, ..., f_8\}$ depicted in mustard in Fig. 2(b). Unlike complementation union (Example 6), FD does not overcombine tuples t_3 and t_{13} since* `Team` *in t_3 is unknown. Hence, it contains f_3 and f_4 after integration and does not produce f_9. Also, f_{10} is subsumed by f_5.*

FD has been shown to produce what has been called *maximally integrated tuples* [41].

Definition 8 (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 tuple(s) of r [41].*

We follow Kanza and Sagiv [41] by defining FD based on maximally integrated tuples.

Definition 9 (Full Disjunction (FD)). *The Full Disjunction of the tables $T_1, T_2, \ldots T_n$, with input tuples $\mathcal{S}$, is the set of all maximally integrated tuples that can be generated from $\mathcal{S}$.*

In Section 5, we will introduce an algorithm that computes FD based on Definition 9. The FD definition we use [41] is based on tuples [33], rather than tuple-sets ($\text{FD}_{\text{tuple-set}}$) [18, 19]. $\text{FD}_{\text{tuple-set}}$ applies subsumption based on the set of tuples from which an integrated tuple is produced (call the tuple-set) [19]. Subsumption is only applied between two tuples if the tuple-set of one is a superset of the tuple-set of the other. Note that $\text{FD}_{\text{tuple-set}}$ yields a set of maximally integrated tuples, but might contain integrated tuples that subsume each other, as we discuss in the next example.

Example 10. *Fig. 2(b) illustrates the difference between FD and $\text{FD}_{\text{tuple-set}}$. To understand the subsumption in $\text{FD}_{\text{tuple-set}}$, first consider integrated tuples f_3 and f_9 whose tuple-sets are $\{t_3\}$ and $\{t_3, t_{13}\}$, respectively (depicted in the* `TIDs` *column in Fig. 2). The tuple-set of f_9 contains all tuples in the tuple-set of f_3. Therefore, under $\text{FD}_{\text{tuple-set}}$, f_9 subsumes f_3. However, if we consider f_5 and f_{10} having tuple-sets $\{t_4\}$ and $\{t_{14}\}$, respectively, neither is a superset of the other. Therefore, $\text{FD}_{\text{tuple-set}}$ does not perform subsumption on these two tuples and returns both. In contrast, the tuple f_{10}, is not produced by FD as it gets subsumed by f_5.*

2.4 Solution Overview

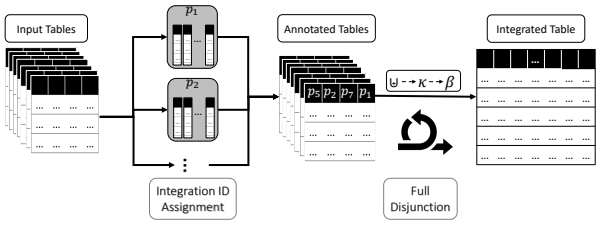

Figure 3: An overview of ALITE.

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. We assume that we are given a set of tables. The first step (Fig. 3 left part) is to assign each column with a column header which we call an *Integration ID* (Section 4). After assigning such IDs, the tables are annotated (Figure 3 middle part). We can then apply 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 [65], a long-standing problem of identifying correspondences among database attributes. Numerous algorithmic attempts have been suggested over the years for handling the matching problem, e.g., COMA [26], Similarity Flooding [54], BigGorilla [15], and ADnEV [71]. A common assumption for most of this work is the existence of consistent and complete metadata, an unrealistic assumption in data lake tables [57]. Recently, Koutras et al. explored the use of traditional schema matching methods in the scope of data lakes [46]. 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 [36, 63, 72], mainly revolving around web tables and assuming metadata is reliable and complete. Some work [3, 63] uses a clustering-based approach. However, contrary to the clustering-based approach we will suggest (Section 4), they use schema information rather than data values. Recall that data lake tables generally lack reliable metadata [30, 57].

Other related work includes unionable [12, 42, 49, 58], joinable [27, 56, 77, 79–81], and related [10, 20, 79] table search, for which the designed methods are usually based on column relationships. For example, in order to find unionable tables, TUS [58] first aims at finding unionable columns. Similarly, Bogatu et al. [10] assesses table relatedness by assessing their attribute relatedness. Our work uses a similar methodology to TUS [58] based on column embeddings. But here we make use of an embedding that was designed for tables, namely, TURL [24]. We are also, to the best of our knowledge, the first to make use of TURL [24] for holistic matching of data lake tables.

Full Disjunction. Full disjunction (FD) was initially defined by Galindo-Legaria as an associative alternative for the outer join operator [33]. 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) [33]. 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 iff 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 [67]. Kanza and Sagiv showed that FD can be computed for any arbitrary set of tables in $O(n^5 s^2 f^2)$ time [41] where, recall n is the number of input tables, s is the total 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 [76]. Other work computes the $\text{FD}_{\text{tuple-set}}$, rather than FD [18, 19]. Cohen and Sagiv introduced an algorithm that computes k $\text{FD}_{\text{tuple-set}}$ tuples in a given ranking order [19] and improved the worst-case time complexity over Kanza and Sagiv [41] to compute the full results. Cohen et al. also proposed an algorithm called *BICOMNLOJ* that computes each $\text{FD}_{\text{tuple-set}}$ tuple with polynomial delay [18]. 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 [19]. Note that both *INCREMENTALFD* [19] and *BICOMNLOJ* [18] perform subsumption in terms of tuple-sets ($\text{FD}_{\text{tuple-set}}$) rather than actual tuples. Hence, they may produce subsumable tuples in their $\text{FD}_{\text{tuple-set}}$ result (specifically, they may produce a proper superset of the FD). Note that, when there are no missing values ($\pm$) and subsumable tuples in the input relations, these algorithms compute the FD [33]. As data lake tables may contain many missing values and subsumable tuples, we use FD. Our experiments (Section 6) show that in real data lakes the difference between the FD and the $\text{FD}_{\text{tuple-set}}$ [18, 19] can be substantial and that the original definition of FD, which maximally integrates tuples, is preferred.

Recently, Paganelli et al. [60] revised Cohen and Sagiv's *INCREMENTALFD* [19] and Cohen et al.'s *BICOMNLOJ* [18] to compute the FD in a distributed environment. They also introduced a new algorithm called *ParaFD* that outperforms *INCREMENTALFD* and *BICOMNLOJ* while computing FD using multiple machines. ParaFD first finds all the spanning trees in the scheme graph of the input table schemas. Then, it applies outer join based on Hash-star join to integrate tables following the order on each spanning trees by using Primary Key-Foreign Key relationships. Finally, it applies subsumption to obtain the FD. *ParaFD* allows *missing nulls* but it can be used only for sets of relational tables on which all joins are key to foreign-key joins. We consider the general case of arbitrary joins. To modify ParaFD for arbitrary joins, one needs to use full outer join without Hash-star join over each spanning tree. But for real data lake tables forming complex scheme graphs, the number of spanning trees can be very large. For instance, for a complete scheme graph (i.e. each table is connected to each of the other tables) having n tables, the number of spanning trees may be on the order of n^{n-2} [1]. One needs to apply outer join over each spanning tree making ParaFD inefficient and similar to the baseline suggested by Galindo-Legaria [33].

Other research considers integrating data from relational and web tables and handling conflicts between the data values [6–9]. Bleiholder et al. introduced complement union operator that integrates tuples under uncertainty (*a conflict between a null value and a non-null value*) [9]. In the absence of *missing nulls* ($\pm$), the complement union operator is the same as FD. Yet, in the common case of tables with *missing nulls*, complement union may over-combine the tuples having null values on the join columns (see Example 6).

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 [40]. So, 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 11 (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 [72]. 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 [58], for example, uses embeddings from fastText [39], 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 [24]. 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. Hierarchical clustering works iteratively. First, each data point (in our case attribute) is assigned with a cluster. Then, at each iteration the two closest clusters (by some metric) are merged to generate a new cluster. The algorithm terminates when all the data points are assigned to the same cluster. The hierarchical clustering result is usually described

using a dendrogram, which is a type of tree that illustrates the different clusters that can be generated at each iteration [48]. Using the dendrogram, we select a specific cluster as we discuss next.

Selecting the Number of Integration IDs. An important parameter in any clustering algorithm is the number of clusters [35], 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 [43]. Several clustering quality evaluation methods exist in the literature based on inter-cluster and intra-cluster distances [13, 23, 69]. 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 [5] to determine the number of clusters that maximizes some (unsupervised) clustering quality measure. In the experiments we use the well-known Silhouette Coefficient [69].[1]

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 $m_1, m_2 \dots m_n$ are the number of columns in the input tables $T_1, T_2 \dots T_n$, the minimum number of clusters is given by $\max(m_1, m_2 \dots m_n)$. 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. We show a figure that zooms in the left part of Fig. 3 and summarizes the Column Integration ID assignment in Appendix A.1.

Example 12. *Consider Fig. 1, we need to assign integration IDs to the columns of tables. Here, we expect to have six clusters, each for {Stadium, Location, Team, Opened, Coach, Capacity} 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 (3), to the maximum (15). Recall* `TID` *is not a real column and was added so we can clearly refer to tuples. The Silhouette coefficient over the TURL embeddings is computed for all values from 3 to 15 and plotted in Appendix A.1. It starts from 3 and has a maximum at 6 . Then it decreases monotonically from 7 to 15. Hence, we would pick 6 as the optimal number of clusters and the clustering created in this simple example does reflect the ground truth – e.g., the four* `Stadium` *attributes are assigned the same integration ID, and the single* `Opened` *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 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. The two main properties the algorithm uses are that the output is composed of

[1]Additional details are available in Appendix A.1.

Algorithm 1: ALITE Full Disjunction

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

all maximally integrated tuples over the input tuples (Definition 9) and should not contain subsumable 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.

1. 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 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 13. *We use our running example (Fig. 1) throughout the description of the algorithm for clarity. Since we have four missing nulls in the tables (one each on T_1 and T_4 and two in T_5), we replace them with four distinct labeled nulls. After replacement, they are treated similar to other non-null values.*

2. Outer union. Now we outer union all the input tables and store the resulting tuples in a set U_{ou} (Line 4). The outer union of 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 uses complementation to return all the maximally integrated tuples along with (possibly) subsumable tuples.

3. Complementation Step (Algorithm 2). The objective of this step is to generate all the maximally integrated tuples. First, we prepare two sets to perform the complementation: U_{temp} and U_{comp}, and initialize both to U_{ou}. Later on, U_{comp} holds the complementation result. We start complementing the tuples in U_{temp} with outer unioned tuples U_{ou}. (Line 4). For each tuple in U_{temp}, we look for 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 9-12). However, if a tuple in U_{temp} does not have any complementing tuples, we add the tuple itself to U_{comp} (Line 13-14). This ensures that the tuples having no join partners are also included

Algorithm 2: Complement

```
1  Input: A set of outer unioned tuples U_ou
2  Output: A set of tuples after complementation U_comp
3  U_comp ← U_ou; U_temp ← ∅
4  while U_temp ≠ U_comp do
5      U_temp ← U_comp; U_comp ← ∅
6      for t_1 ∈ U_temp do
7          complement_count = 0
8          for t_2 ∈ U_ou do
9              R, complement_status ← κ(t_1, t_2)
10             if complement_status then
11                 U_comp ← U_comp ∪ R
12                 complement_count ← complement_count + 1
13         if complement_count = 0 then
14             U_comp ← U_comp ∪ {t_1}
15 Output U_comp
```

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. If they are not equal, there may be tuples that can be complemented. So, we go for another round of complementation. Note that the outer loop (Line 4-14) never takes more than $n-1$ rounds (even less in practice). The reason is simple: complementation can only combine tuples from different tables. So, there are at most $n-1$ such steps for any tuple. Also from monotonicity of the process, if a tuple does not get complemented in one step, it can not be complemented in a future step.

Example 14. *Consider Table (a) and (b) of Fig. 2. 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. In the complementation first round (Line 4-14), both U_{ou} and U_{temp} are the same and they contain the tuples t_1, t_7 and t_{10}. All these three tuples integrate with each other. Assume that the labeled null in t_{10} was replaced by a distinct non-null value $\pm_1$. So, the complementation operator integrates them pairwise (Line 9) to generate intermediate tuples $\kappa(t_1, t_7)$ (equal to f_1 except Capacity = $\perp$), $\kappa(t_1, t_{10})$ (equal to f_1 except Coach = $\perp$ and Capacity = $\pm_1$), and $\kappa(t_7, t_{10})$ (equal to f_1 except Capacity = $\pm_1$) (see Section 2.2). All these tuples are added to U_{comp}. Similarly, tuples t_8 and t_{12} complement each other producing f_8, which is added to U_{comp} (Line 11). On the other hand, t_5 does not have any integrating partners and, hence, is added to U_{comp} itself (Line 14). After the first complementation round, $U_{comp} = U_{ou} \setminus \{t_1, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\} \cup \{\kappa(t_1, t_7), \kappa(t_1, t_{10}), \kappa(t_7, t_{10}), \kappa(t_6, t_9), \kappa(t_6, t_{11}), \kappa(t_9, t_{11}), f_8\}$ 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 (Line 4-14). As mentioned earlier, U_{ou} is always the same. So, tuples $\kappa(t_1, t_7)$ and $\kappa(t_1, t_{10})$ in U_{temp} complement with tuples t_{10} and t_7 respectively. They both produce the same tuple that is equal to $\kappa(t_7, t_{10})$ which is already in U_{temp} from the first iteration. As U_{temp} is the set, the newly generated duplicates are discarded. After this round, we again move U_{comp} to U_{temp} and empty U_{comp}. In the next round, no tuples in U_{temp} have complementing partners in U_{ou}. So, the complementation terminates and $U_{comp} = U_{temp} = \{\kappa(t_7, t_{10}), f_2, f_3, f_4, f_5, f_6, f_7, f_8, t_{14}\}$.*

4+5. Remove labeled nulls and subsumption. Once the complementation is done, we remove the subsumable tuples to get the FD. Notice however, we have replaced the missing nulls ($\pm$) 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 ($\pm$) (Line 6 of Algorithm 1) and then use subsumption (Line 7) to remove the non-maximally integrated tuples.

Example 15. *We now replace the unique labeled nulls in each tuple with a missing null ($\pm$). This step converts $\kappa(t_7, t_{10})$ to f_1. Finally, we apply subsumption to U_{comp} and get rid of tuple t_{14} (Algorithm 1, Line 7). This ensures that the final result is the set of FD tuples i.e., $\{f_i\}, i \in [1, 8]$ where, i is an integer.*

For subsumption, we use the null-value based partitioning algorithm introduced by Bleiholder et al. that computes subsumption in $O(s \log s)$ time where, s is the number of input tuples [8]. 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.

5.2 Efficient Complementation

For subsumption, we used an existing fast algorithm. For complementation, we describe a novel optimization based on partitioning of the tuples. Recall that two tuples having different non-null values on a common column cannot complement each other. So, we avoid the comparison between such tuples by assigning them to different partitions. Then we apply complementation within each partition using Algorithm 2, reducing its computation time.

Example 16. *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 (see 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 a common column* `Stadium`*. Hence, we safely avoid any comparison between t_1 and t_2. Also, as $t_{10}[Stadium]$ = NRG Stadium, it has a possibility of complementing t_1. So, we compare t_1 and t_{10}. Notice however, tuple t_7 complements t_{10} even though t_7 has a produced null on* `Stadium`*. Therefore, a tuple having a produced null on a common column should still be compared with other tuples.*

We intend to make each partition fairly small, i.e., keep the number of tuples in each partition less than a positive integer θ where, $\theta \ll s$. Bleiholder et al. suggested to partition tuples using the values of selected partitioning column(s) [9]. 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 produced null values in the partitioning column(s), they are added to all other partitions. Now, the complementation can be applied on the tuples within each partition. But partitioning with a single or even

a group of columns may still produce large partitions. So, 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. The tuples in the produced null partition should be added to each of the other partitions. Hence, in order to reduce the number of tuples in the produced null partition, we first sort the columns in ascending order of the number of produced nulls they contain. Then, we partition the tuples by value of each column one by one. Also, the tuples from produced null partitions, when added in other partitions, may create duplicate partitions i.e., the partitions having exactly the same tuples. To discard such duplicate partitions, we index each partition based on its tuples.

Example 17. *Consider the table in Fig. 2(a), which is the outer union of the tables in Fig. 1. Each missing null is replaced by a distinct value. Let the threshold for partitioning be $\theta = 4$. The partitioning order of the columns based on the number of produced nulls is {Location, Stadium, Team, Coach, Opened, Capacity}. In the first round, we partition by* `Location` *which gives six partitions $P_1 = \{t_1, t_2, t_7, t_{10}\}$, $P_2 = \{t_3, t_{13}\}$, $P_3 = \{t_4, t_{14}\}$, $P_4 = \{t_5\}$, $P_5 = \{t_6, t_9, t_{11}\}$, $P_6 = \{t_8, t_{12}\}$. As P_1 does not have less than 4 tuples at the end of the first round, we again partition P_1 into smaller partitions using* `Stadium` *column. This gives two more partitions $P_{11} = \{t_1, t_7, t_{10}\}$ and $P_{12} = \{t_2, t_7\}$. Note that t_7 has a produced null in the partitioning column. So, we add t_7 to both P_{11} and P_{12}. At the end of second round, all the partitions have size less than 4. Hence, we do not further partition using other columns and the input to Algorithm 2 by Algorithm 1 are partitions $P_{11}, P_{12}, P_2, P_3, P_4, P_5, P_6$.*

To optimize, we slightly modify Algorithm 1 (Line 5). Specifically, we apply partitioning over the outer unioned tuples (Algorithm 1 Line 4) and apply complementation over each partition one-by-one. The complementation over each partition is then unioned before replacing distinct labeled nulls with the missing nulls.

5.3 Full Disjunction Algorithm Analysis

Correctness. We now present a theorem on the correctness of Algorithm 1. [2]

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 $T_1, T_2, \ldots T_n$.*

Time Analysis. Recall that the objective of ALITE is to integrate data lake tables discovered using table search techniques. Generally, such tables form schema graphs that may have complex cycles. Our choice of using a complementation operator enables us to optimize the production of maximally integrated tuples. Furthermore, we also optimize the subsumption operator separately, which makes ALITE faster in practice than the baselines for computing FD over data lake tables. We will show the superiority of ALITE over baselines in different conditions experimentally in Section 6. Further details on time complexity are available in Appendix A.2.

6 EXPERIMENTS

We now evaluate the two steps involved in ALITE.

[2] Proof in Appendix A.2

6.1 Experimental Setup

We implement ALITE and all the baselines using Python 3.7 and run experiments using a CentOS server having Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz processor. The main objective of our experiments is to answer: (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) Is it 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.

Embedding Generation. Recall that we use pre-trained embeddings to represent the columns for clustering (and integration ID assignment). Before using TURL [24], our method of choice to generate embeddings, we pre-process the tables using their implementation [74]. This phase includes, for example, generating a Wikipedia entity dictionary to map values in the tables. TURL was designed for web tables and, hence, has a limited capacity in terms of the number of rows and columns it can use to create embeddings (mean of ~20 rows and ~2 columns [24]). Since typical data lake tables are much larger (see Fig. 4), to cope with such a limitation, we designed an iterative embedding generation approach for each column. First, we randomly sample 50 rows and generate the corresponding column embedding by averaging the representations of each row. Then, we iteratively 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 less 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 [62]. Based on our objective of obtaining dense, but well-separated clusters, we use the Silhouette Coefficient as a clustering quality measure [69]. We select the number of clusters (column predicates) that maximizes the Silhouette Coefficient (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., F_1-score are used as the evaluation measures for schema matching [14, 32, 71]. So, we 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 match during the evaluation. Specifically, the total number of matches is the sum of the number of column pairs belonging to the same cluster and the number of singleton clusters. Formally, let $\mathcal{T}_{\mathcal{M}}$ be true column pair matches according to the ground truth

and $\hat{\mathcal{T}}_M$ be the matches according to a method. We define Precision (P), Recall (R) and F_1-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 F_1-score for each set of tables to be integrated 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 is 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 [67]. 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 and f' be output size of a competing method (e.g. outer join). The TDR is given by $\frac{f \cap f'}{f}$. If the competing method produces all FD tuples, TDR is equal to 1 and it is equal to 0 if it produces none of them.

6.3 Baselines

Column Integration ID Assignment. Recall that we use a clustering approach and pre-trained embeddings created for the tables' columns [24] to find the column integration IDs. Other existing natural language embeddings were successfully adopted for similar tasks such as table search [10, 58] and column annotation [73]. Here, we compare the performance of such embeddings also for our task. Like in table search [10, 58], we use **fastText** [39, 55] embeddings of columns as done for column annotation [73], and we use **BERT** [25] embeddings. We use a publicly available Fasttext model [31] using Gensim python package [34]. We generate BERT embeddings [4] using the commonly used hugging face package [28].

We also compare our Column Integration ID Assignment with existing schema matching methods. There are numerous matching approaches in the literature [26, 46, 52, 71]. However, most work relies on metadata, which we aim to avoid in our setting. Recently, in Valentine, Koutras et al. performed a detailed analysis of existing schema matching methods in a data lake setting [46]. Based on their analysis, we select the Distribution Based method (**DB**), proposed by Zhang et al., as a baseline [78]. DB 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 [70]. A threshold is applied over this score to decide the column similarity. We use a threshold of 0.15 suggested by Zhang et al. [78]. Also, we reproduce *DB* using the open source code in Valentine [75]. For completeness, we compare ALITE against other schema-based matching methods available in Valentine over a benchmark having real schemas. Specifically, we compare **CUPID** [52], **COMA** [26] and Similarity Flooding **(SF)** [54]. We also report a Jaccard Similarity and Levenshtein Distance method (**JLM**) used as a baseline in Valentine [46]. We use default parameters from the respective papers. Note that the holistic schema matching works with a set of tables whereas the pairwise schema matching methods work only between a pair of tables (or schemas). So, for fair evaluation, we make all the pairwise methods holistic. We apply pairwise schema matching between every pair of tables in the set of tables to be integrated. Then, the method returns all the column pair matches, which we use to compute P, R and F_1 (Section 6.2).

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 [60]. In a data lake, we are often not joining on keys and foreign keys, so we mainly compare ALITE against ParaFD in a benchmark having such relationships. However, to understand how accurate ParaFD can be in the real tables that may not necessarily have PK-FK relations, we report its TDR on a benchmark having real data lake tables.

We also use **BICOMNLOJ**, which computes the FD with a polynomial delay between tuples [18]. 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 computes $\text{FD}_{\text{tuple-set}}$, if the input contains nulls its output may contain some subsumable tuples (see Example 10). So, 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 [8]. Since an open-source implementation is not available for either ParaFD or BICOMNLOJ, we reproduce them using the information provided in the paper. We implement ParaFD to run on a single machine for fair comparison. The reproduced implementations are publicly available in our github repository [2]. Also, we run **outer join** to integrate the tables and use its output size to report TDR. As outer join is not associative, the order of integration makes a significant difference [33]. Applying outer join in a connected-prefix ordering of the input tables can yield FD for γ-acylic case [18]. Therefore, we find the connected-prefix ordering by performing DFS transversal over the input scheme graph and use it to compute the outer join [18].

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 4: Benchmarks used in the experiments.

Figure 4 summarizes all benchmarks used in different experiments along with their statistics. Each benchmark contains multiple tables with different schemas and each schema may be used by multiple tables. All the benchmarks are publicly available [2].

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*. For example, T_1-T_5 (Fig. 1) is

an integration set containing 5 tables to be integrated. We run the column integration ID Assignment 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 [58]. First, we select 65 real data lake tables from US Open Data [37], Canada Open Data [21], and UK Open Data [22] and consider them as seed tables. Each seed has a different schema and is used to generate an integration set. We partition 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 schema 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 ground truth. 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 $f > s$. [3] 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 open data lakes. Therefore, to experiment with 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 a 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. [58] 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 $f < 0.5s$. Furthermore, five integration sets have $f \approx s$ and six integration sets have $f > s$.

IMDB. As ParaFD can only be used accurately for the tables having PK-FK relationships, we also use an IMDB dataset, having such relationships for our experiments [38]. 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 [18]. 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.

[3] We consider $f \approx s$ when $\frac{|f-s|}{s} \leq 0.05$.

6.5 Column Integration ID Assignment Results

Benchmark		Method							
		Baseline					ALITE		
		CUPID	COMA	SF	JLM	**DB**	fastText	BERT	**TURL**
Align	**P**	-	-	-	-	*0.95*	**0.96**	0.92	0.93
	R	-	-	-	-	0.89	0.92	*0.97*	**0.97**
	F_1	-	-	-	-	0.91	0.94	*0.94*	**0.95**
Real	**P**	0.45	**0.82**	0.26	0.49	0.72	0.69	0.73	*0.78*
	R	0.70	0.63	**0.91**	*0.91*	0.80	0.81	0.77	0.76
	F_1	0.47	0.69	0.30	0.56	0.72	*0.72*	0.71	**0.76**

Best score
Second Best score

Figure 5: Average precision, recall and F_1 over the *Align* and *Real* benchmarks for column integration ID assignment.

We now report the effectiveness of column integration ID assignment, followed by an empirical analysis of its efficiency. Fig. 5 shows the evaluation results for the *Align* and *Real* benchmarks. Recall that ALITE uses a clustering-based approach to find the column integration IDs that uses pre-trained embeddings created using TURL [24]. So, first we compare ALITE's precision, recall and F_1-Score using TURL-based embeddings against fastText and BERT. We use the same experimental setups for all three methods (Section 6.1). TURL gives comparable or even better precision and recall against the baselines (Fig. 5). In terms of F_1-score (the best overall metric that combines both precision and recall), TURL performs better than the baselines. This validates our choice of using table-based embedding (TURL) instead of natural language embeddings (fastText and BERT) for data lake tables. We will explore other ways to embed 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 ALITE on the *Align* benchmark. However, ALITE outperforms *DB* by more than 8% in terms of Recall. In *Real* benchmark, DB has the lower precision and higher recall than ALITE. Still, in terms of F_1 (the combined metric), ALITE outperforms DB by more than 4% on both benchmarks. The main reason for the lower performance of *DB* is that it relies on value overlap and ignores the semantics (e.g., synonyms). Specifically, *DB*'s precision is impacted by the presence of homographs [50]– the same values having different meanings– in the non-matching columns, and its recall is impacted by the presence of synonyms in the matching columns that can not be captured by value overlap. Also, *DB* uses information only within a pair of columns to make the matching decision whereas, ALITE considers all the columns together in a holistic way, which enhances its performance.

Moreover, we analyze the performance of schema-based methods for column ID assignment in Real Benchmark. Recall that Align Benchmark's tables are generated using seed tables such that the

aligning columns have the same column headers (Section 6.4). So, we do not evaluate schema-based methods (CUPID, COMA, SF and JLM) in Align Benchmark. In Real Benchmark, we observe that COMA has better P and F_1 than other schema-based methods but it has lower R and F_1 than DB (baseline) and ALITE. Specifically, ALITE outperforms COMA, the best schema-based method, by ~10 % in F_1-score. This is because the tables contain unreliable schemas and applying similarity measures over them leads to incorrect aligning. CUPID also shows weaker performance due to the same reason. We observe that SF and JLM have the top-2 recalls among all the methods. However, they have lower precision and F_1-score. This is because they align most columns within the same cluster which increases their recall but penalizes precision. Hence, the schema-based methods are not effective in the data lake setting.

The column integration ID assignment is considered as an offline task. Yet, ALITE's clustering is much faster than the pair-wise comparison done by the baseline (*DB*). Specifically, ALITE takes ~10 minutes for *Align* and ~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 (~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 ~80 minutes while TURL takes ~7 minutes. For the *Real* benchmark, they take approximately the same time (~15 minutes).

6.6 Full Disjunction Results

Now we compare ALITE's FD algorithm efficiency (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).

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. [18]. 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 [18], *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 (see repository [2]). We again fix the number of tuples on each input table to 1k for each of the 8 tables, i.e., s = 8000. 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. Fig. 6(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

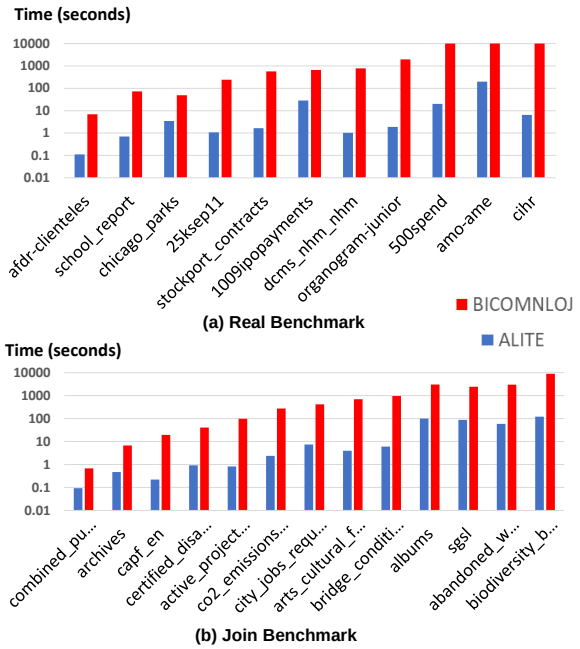

Figure 6: Integration time (Y-axis, log scale) in (a) *Real* Benchmark and (b) Join Benchmark. The integration sets in X-axis are arranged in ascending order of input size, some of the names are truncated for conciseness. A 10k second cut-off was used in both benchmarks. Due to space considerations, integration sets that did not meet the cut-off time in Join benchmark are provided in the technical report [2].

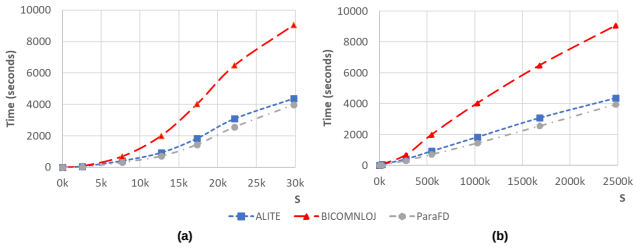

Figure 7: Integration time in the IMDB benchmark for (a) different input size and (b) different output size.

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. 6 (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. 6(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.

Next, we apply ParaFD over Real Benchmark to see if it can yield FD results in data lake tables. ParaFD completes the integration within the cut-off time for only 3 out of 11 integration sets and only 2 of them are equal to FD result. ParaFD is slow in *Real* because it computes all the spanning trees over the schema graph and computes outer join over each of them (see Section 3). Accordingly, we also implement an approximate version of ParaFD where we do not apply the cut-off time but compute output tuples using at most 100 spanning trees. The approximate version yields FD result for only 5 out of 11 integration sets. For other 6 sets, the average TDR is 0.82, i.e., ParaFD misses around 18 % of tuples. Also, it takes an average of 9268 seconds per integration set, which is slower by magnitudes than ALITE (Fig. 6(a)). The integration time and TDR on each integration set is provided in the github repository [2].

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. 7 (a), we vary 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. 7 (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. Due to space constraints, we provide other details like the number of tables on each integration set, the number of columns, input size, output size, and missing nulls size with the supplementary materials [2].

FD against outer join. We now show the importance of using FD against outer join empirically in real data lake tables (*Real* Benchmark). We provide a bar graph in Appendix A.4 that shows each integration set of this benchmark in X-axis and TDR in Y-axis. 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* 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, the outer join can sometimes produce the full FD. For two integration sets (chicago_parks and 1009ipopayments), the outer join is able to generate more than half of the FD tuples. But for other sets, *TDR* is very low, which shows that the outer join produces incomplete tuples and hence, magnifies the importance of FD to best integrate real data lake tables.

Integration Method	Integrated Table Size	$\lvert T \rvert$	$\lvert T \cap T^* \rvert$	P	R	F_1
Full Disjunction	121	98	78	**0.795**	**0.838**	**0.816**
Outer join	114	109	37	0.339	0.397	0.366

Figure 8: Results of applying ER over FD and outer join output. `Integrated Table Size` is the number of input tuples to ER, which is the output size of integration methods.

Entity Resolution (ER). Lastly, we analyze the use of FD (rather than outer join) for the downstream application of entity resolution (ER). To create ground truth, we inject duplicate tuples into a real table. We then partition the table into four tables and integrate them back using outer join and FD. Over these tables, we apply ER and verify if the tuples in the original table are reproduced. Specifically, we use Magellan's *py_entitymatching* [64] to find and remove matching tuples. Given a table T (resulting table after applying ER and removing duplicates from outer joined or FD table) and a ground truth table T^* (i.e. clean table), we compute precision (P), recall (R) and F1-score (F_1) as follows:

$$P = \frac{|T \cap T^*|}{|T|}, R = \frac{|T \cap T^*|}{|T^*|}, F_1 = \frac{2 * P * R}{P + R}$$

In other words, precision and recall measure the portion of clean tuples in T and the portion of clean tuples that are covered by T respectively. Additional details on the experimental setup are provided in Appendix A.4.

We report P, R and F_1 of applying ER over FD and outer join output in Fig. 8. The results indicate that applying ER over FD table is better than outer join table in terms of both P and R and by ~123% in terms of F_1. Since outer join is not able to integrate the maximal information, its result contains incomplete tuples having null values. This reduces the information available for the entity resolution algorithm and impacts de-duplication accuracy.

7 CONCLUSION

We introduce a novel problem of integrating data lake tables after discovery and present ALITE that aims 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 show that ALITE's new FD algorithm is more efficient than existing baselines, in practice. We also show the effectiveness of using FD to best integrate the real data lake tables.

ACKNOWLEDGMENTS

This work was supported in part by NSF under award numbers IIS-1762268, IIS-1956096 and IIS-2107248.

REFERENCES

[1] Martin Aigner and Günter M Ziegler. 1999. Proofs from the Book. *Berlin. Germany* 1 (1999).
[2] ALITE. 2022. https://github.com/northeastern-datalab/alite
[3] 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.
[4] Hugging Face BERT base model (uncased). 2022. https://huggingface.co/bert-base-uncased

[5] 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).

[6] Jens Bleiholder, Melanie Herschel, and Felix Naumann. 2011. Eliminating NULLs with Subsumption and Complementation. *IEEE Data Eng. Bull.* 34, 3 (2011), 18–25. http://sites.computer.org/debull/A11sept/DataFusion1.pdf

[7] 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

[8] Jens Bleiholder, Sascha Szott, Melanie Herschel, Frank Kaufer, and Felix Naumann. 2010. Subsumption and complementation as data fusion operators. In *EDBT 2010, 13th International Conference on Extending Database Technology, Proceedings (ACM International Conference Proceeding Series)*, Vol. 426. ACM, 513–524. https://doi.org/10.1145/1739041.1739103

[9] Jens Bleiholder, Sascha Szott, Melanie Herschel, and Felix Naumann. 2010. Complement union for data integration. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010*. IEEE Computer Society, 183–186. https://doi.org/10.1109/ICDEW.2010.5452760

[10] 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

[11] 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 (WWW '19)*. ACM, 1365–1375. https://doi.org/10.1145/3308558.3313685

[12] Michael J. Cafarella, Alon Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proc. VLDB Endow.* 2, 1 (2009), 1090–1101. https://doi.org/10.14778/1687627.1687750

[13] Tadeusz Caliński and Jerzy Harabasz. 1974. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods* 3, 1 (1974), 1–27.

[14] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD Conference 2020*. ACM, 1335–1349. https://doi.org/10.1145/3318464.3389742

[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] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An Overview of End-to-End Entity Resolution for Big Data. *ACM Comput. Surv.* 53, 6, Article 127 (dec 2020), 42 pages. https://doi.org/10.1145/3418896

[17] 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

[18] 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 (VLDB '06)*. VLDB Endowment, 739–750.

[19] Sara Cohen and Yehoshua Sagiv. 2007. An incremental algorithm for computing ranked full disjunctions. *J. Comput. Syst. Sci.* 73, 4 (2007), 648–668. https://doi.org/10.1016/j.jcss.2006.10.015

[20] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding Related Tables. In *SIGMOD Conference 2012*. ACM, 817–828. https://doi.org/10.1145/2213836.2213962

[21] Canada Open Data. 2020. https://open.canada.ca/en/open-data

[22] UK Open Data. 2020. https://data.gov.uk/

[23] 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

[24] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proc. VLDB Endow.* 14, 3 (2020), 307–319. https://doi.org/10.5555/3430915.3442430

[25] 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).

[26] 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*. 610–621. https://doi.org/10.1016/B978-155860869-6/50060-3

[27] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. In *37th IEEE International Conference on Data Engineering, ICDE 2021*. IEEE, 456–467. https://doi.org/10.1109/ICDE51399.2021.00046

[28] Hugging Face. 2022. https://huggingface.co

[29] Grace Fan, Jin Wang, Yuliang Li, Dan Zhang, and Renée J. Miller. 2022. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. *CoRR* abs/2210.01922 (2022). https://doi.org/10.48550/arXiv.2210.01922

[30] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing Quality to Data Lakes. In *SIGMOD Conference 2016*. ACM, 2089–2092. https://doi.org/10.1145/2882903.2899391

[31] fastText. 2022. https://fasttext.cc/docs/en/english-vectors.html

[32] Avigdor Gal, Haggai Roitman, and Roee Shraga. 2021. Learning to Rerank Schema Matches. *IEEE Trans. Knowl. Data Eng.* 33, 8 (2021), 3104–3116. https://doi.org/10.1109/TKDE.2019.2962124

[33] César A. Galindo-Legaria. 1994. Outerjoins as Disjunctions. In *SIGMOD Conference 1994*. ACM, 348–358. https://doi.org/10.1145/191839.191908

[34] Gensim. 2022. https://radimrehurek.com/gensim

[35] Johannes Grabmeier and Andreas Rudolph. 2002. Techniques of Cluster Algorithms in Data Mining. *Data Min. Knowl. Discov.* 6, 4 (2002), 303–360. https://doi.org/10.1023/A:1016308404627

[36] 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 and Data Mining*. ACM, 429–438. https://doi.org/10.1145/1081870.1081920

[37] The home of the U.S. Government's open data. 2020. https://data.gov/

[38] IMDB. 2022. https://datasets.imdbws.com/

[39] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[40] Jaewoo Kang and Jeffrey F. Naughton. 2003. On Schema Matching with Opaque Column Names and Data Values. In *SIGMOD Conference 2003*. ACM, 205–216. https://doi.org/10.1145/872757.872783

[41] 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 (PODS '03)*. ACM, 78–89. https://doi.org/10.1145/773153.773162

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

[43] 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.

[44] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeffrey F. Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, and Vijay Raghavendra. 2016. Magellan: Toward Building Entity Matching Management Systems. *Proc. VLDB Endow.* 9, 12 (2016), 1197–1208. https://doi.org/10.14778/2994509.2994535

[45] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of Entity Resolution Approaches on Real-World Match Problems. *Proc. VLDB Endow.* 3, 1–2 (2010), 484–493. https://doi.org/10.14778/1920841.1920904

[46] 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 *37th IEEE International Conference on Data Engineering, ICDE 2021*. IEEE, 468–479. https://doi.org/10.1109/ICDE51399.2021.00047

[47] Michel Lacroix and Alain Pirotte. 1976. Generalized joins. *ACM Sigmod Record* 8, 3 (1976), 14–15.

[48] Peter Langfelder, Bin Zhang, and Steve Horvath. 2008. Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut package for R. *Bioinform.* 24, 5 (2008), 719–720. https://doi.org/10.1093/bioinformatics/btm563

[49] Oliver Lehmberg and Christian Bizer. 2017. Stitching Web Tables for Improving Matching Quality. *Proc. VLDB Endow.* 10, 11 (2017), 1502–1513. https://doi.org/10.14778/3137628.3137657

[50] Aristotelis Leventidis, Laura Di Rocco, Wolfgang Gatterbauer, Renée J. Miller, and Mirek Riedewald. 2021. DomainNet: Homograph Detection for Data Lake Disambiguation. In *EDBT 2021*. OpenProceedings.org, 13–24. https://doi.org/10.5441/002/edbt.2021.03

[51] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proc. VLDB Endow.* 3, 1–2 (2010), 1338–1347. https://doi.org/10.14778/1920841.1921005

[52] Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. 2001. Generic schema matching with cupid. In *vldb*, Vol. 1. Citeseer, 49–58.

[53] David Maier. 1983. *The theory of relational databases*. Vol. 11. Computer science press Rockville.

[54] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. 2002. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proceedings of the 18th International Conference on Data Engineering, 2002*. IEEE Computer Society, 117–128. https://doi.org/10.1109/ICDE.2002.994702

[55] Tomás Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in Pre-Training Distributed Word Representations. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018*. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2018/summaries/721.html

[56] Renée J. Miller. 2018. Open Data Integration. *Proc. VLDB Endow.* 11, 12 (2018), 2130–2139. https://doi.org/10.14778/3229863.3240491

[57] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989. https://doi.org/10.14778/3352063.3352116

[58] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (2018), 813–825. https://doi.org/10.14778/3192965.3192973

[59] 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 (2021), 2863–2866. https://doi.org/10.14778/3476311.3476364

[60] 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

[61] George Papadakis, Ekaterini Ioannou, and Themis Palpanas. 2020. Entity Resolution: Past, Present and Yet-to-Come. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020*. OpenProceedings.org, 647–650. https://doi.org/10.5441/002/edbt.2020.85

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. https://doi.org/10.5555/1953048.2078195

[63] Jin Pei, Jun Hong, and David A. Bell. 2006. A Novel Clustering-Based Approach to Schema Matching. In *Advances in Information Systems, 4th International Conference, ADVIS 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 4243. Springer, 60–69. https://doi.org/10.1007/11890393_7

[64] py_entitymatching. 2016. https://github.com/anhaidgroup/py_entitymatching

[65] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB J.* 10, 4 (2001), 334–350. https://doi.org/10.1007/s007780100057

[66] Erhard Rahm and Eric Peukert. 2019. Holistic Schema Matching. In *Encyclopedia of Big Data Technologies*. Springer. https://doi.org/10.1007/978-3-319-63962-8_12-1

[67] 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 (PODS '96)*. ACM, 238–248. https://doi.org/10.1145/237661.237717

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

[69] 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

[70] 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

[71] Roee Shraga, Avigdor Gal, and Haggai Roitman. 2020. ADnEV: Cross-Domain Schema Matching using Deep Similarity Matrix Adjustment and Evaluation. *Proc. VLDB Endow.* 13, 9 (2020), 1401–1415. https://doi.org/10.14778/3397230.3397237

[72] Weifeng Su, Jiying Wang, and Frederick H. Lochovsky. 2006. Holistic Schema Matching for Web Query Interfaces. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Proceedings (Lecture Notes in Computer Science)*, Vol. 3896. Springer, 77–94. https://doi.org/10.1007/11687238_8

[73] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çagatay Demiralp, Chen Chen, and Wang-Chiew Tan. 2022. Annotating Columns with Pre-trained Language Models. In *SIGMOD Conference 2022*. ACM, 1493–1503. https://doi.org/10.1145/3514221.3517906

[74] TURL. 2020. https://github.com/sunlab-osu/TURL

[75] Valentine. 2021. https://github.com/delftdata/valentine

[76] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, 1981*. IEEE Computer Society, 82–94.

[77] Jiang Zhan and Shan Wang. 2007. ITREKS: Keyword Search over Relational Database by Indexing Tuple Relationship. In *Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007 (Lecture Notes in Computer Science)*, Vol. 4443. Springer, 67–78. https://doi.org/10.1007/978-3-540-71703-4_8

[78] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. 2011. Automatic discovery of attributes in relational databases. In *SIGMOD Conference 2011*. ACM, 109–120. https://doi.org/10.1145/1989323.1989336

[79] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *SIGMOD Conference 2020*. ACM, 1951–1966. https://doi.org/10.1145/3318464.3389726

[80] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD Conference 2019*. ACM, 847–864. https://doi.org/10.1145/3299869.3300065

[81] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-Join: Joining Tables by Leveraging Transformations. *Proc. VLDB Endow.* 10, 10 (2017), 1034–1045. https://doi.org/10.14778/3115404.3115409

[82] Erkang Zhu, Ken Q. Pu, Fatemeh Nargesian, and Renée J. Miller. 2017. Interactive Navigation of Open Data Linkages. *Proc. VLDB Endow.* 10, 12 (2017), 1837–1840. https://doi.org/10.14778/3137765.3137788

A APPENDIX

A.1 Assigning Column Integration IDs

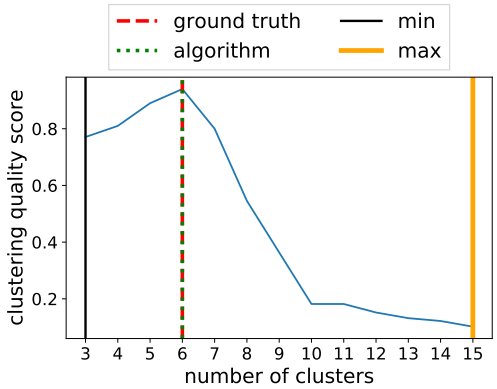

Figure 9: An example scenario of determining the number of clusters for the input tables in Fig. 1. Here, X-axis shows the number of clusters and Y-axis shows the clustering quality score. The green and red lines show the number of clusters according to the algorithm and ground truth respectively.

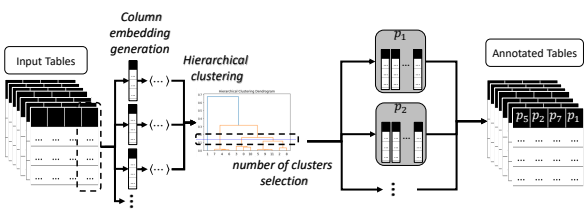

Figure 10: Assigning Column Integration IDs sketch

Fig. 10 zooms in the left part of Fig. 3, described in Section 4. We get a set of tables as input, generate column embeddings, which are used in hierarchical clustering of columns to generate integration ids that are used to annotate the input tables.

Silhouettes Coefficient. Let $\{p_1, p_2, ..., p_K\}$ be K clusters, each containing a set of attributes. The silhouette coefficient $\psi_s(A)$ for an attribute A in a cluster is defined as:

$$in(A) = \frac{\sum_{A' \in p \setminus \{A\}} d(A, A')}{|p|} \qquad out(A) = \min_{1 \le i \le K, i \ne j} \frac{\sum_{A' \in p_i} d(A, A')}{|p_i|}$$

$$\psi_s(A) = \frac{out(A) - in(A)}{\max(in(A), out(A))}$$

where $d(\cdot, \cdot)$ is a distance measure . The silhouette of a set of clusters is calculated by averaging over all attributes of all clusters.

A.2 Integrating Tables

Proof of Theorem 18 (FD Correctness).

Proof. Let, $\mathcal{F}$ be the set of tuples in the natural full disjunction of $T_1, T_2, \ldots T_n$. 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 $T_1, T_2, \ldots T_k$ where, k $\le$ 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' $\sqsupset$ 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 $t_i[A] \ne \pm$ and $t_j[A] \ne \pm$. 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. 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 13- 14 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. □

A.3 FD vs Outer join in Practice

Recall Section 1 that integrating tables is beneficial for answering queries over multiple tables and also for other downstream applications. However, the accuracy of such answers depends on the accuracy of integration process. A straightforward example is aggregation queries. For instance, recalling Fig. 2, if a user is interested in counting the number of teams from *Texas*, a simple aggregation over Fig. 2(a) (outer union) would (incorrectly) return 4 while aggregating Fig. 2(b) (FD) returns 2.

Furthermore, we now discuss the benefit of applying FD (instead of outer join/union) also for another downsteam application, the one of Entity Resolution (ER). ER aims to find and remove duplicate tuples in a table [16, 45, 61]. Using a real example from our experiments , we now show that by applying FD instead of outer join, we better prepare the ground for applying ER. The same concept can be used to understand the difference of using union operator.

Example 19. *Consider tables T_6, T_7 and T_8 shown in Fig. 11 (a) that describe home stadiums of football players who play for different*

T_6

TID	Player	Team
t_{15}	Dez Bryant	Dallas Cowboys
t_{16}	T. Romo	±

T_7

TID	Team	Stadium
t_{17}	Dallas Cowboys	AT and T Stadium
t_{18}	±	AT & T

T_8

TID	Player	Stadium
t_{19}	Tony Romo	AT and T stadium
t_{20}	T. Romo	AT & T

(a) Input Tables

$T_6 ⋈ T_7 ⋈ T_8$

OID	TID	Player	Team	Stadium
f_{11}	$\{t_{15}, t_{17}\}$	Dez Bryant	Dallas Cowboys	AT and T Stadium
f_{12}	$\{t_{16}\}$	T. Romo	±	⊥
f_{13}	$\{t_{18}\}$	⊥	±	AT & T
f_{14}	$\{t_{19}\}$	Tony Romo	⊥	AT and T stadium
f_{15}	$\{t_{20}\}$	T. Romo	⊥	AT & T

(b) Output Table generated using outer join operator

Player	Team	Stadium
Dez Bryant	Dallas Cowboys	AT and T Stadium
T. Romo	±	⊥
⊥	±	AT & T
Tony Romo	⊥	AT and T stadium

(d) Entity Resolution over outer join result

FD(T_6, T_7, T_8)

OID	TID	Player	Team	Stadium
f_{11}	$\{t_{15}, t_{17}\}$	Dez Bryant	Dallas Cowboys	AT and T Stadium
f_{16}	$\{t_{20}\}$	T. Romo	⊥	AT & T
f_{17}	$\{t_{17}, t_{19}\}$	Tony Romo	Dallas Cowboys	AT and T stadium

(c) Output Table generated using FD operator

Player	Team	Stadium
Dez Bryant	Dallas Cowboys	AT and T Stadium
Tony Romo	Dallas Cowboys	AT and T stadium

(e) Entity Resolution over FD result

Figure 11: Entity resolution after integrating the tables.

teams. The results of applying outer join and FD over these tables are shown in Fig. 11 (b) and Fig. 11 (c) respectively. The remaining tuples after applying a distance-based ER algorithm [44] on outer join and FD results are shown in Fig. 11(d) and Fig. 11(e) respectively. Outer join produces more output tuples than FD; yet, it does not produce any tuple about Tony Romo's Team. FD, on the other hand, produces output tuple f_{17} that provides this information. Furthermore, since outer join produces incomplete tuples, ER could not resolve f_{13} and f_{14}. Hence, it is better to use FD instead of outer join to integrate the tables. We further provide experimental evidence in Section 6.

A.4 Experiments

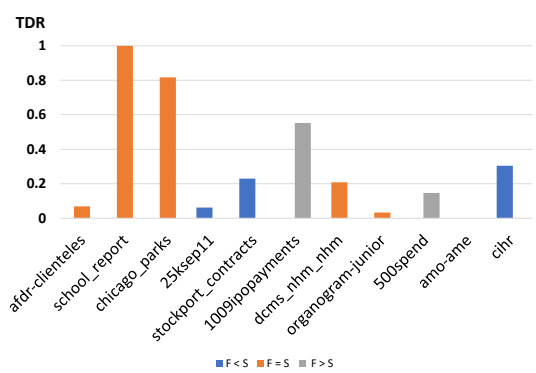

Figure 12: Tuple Difference Ratio (TDR) of the integration sets in the *Real* Benchmark. We show each integration set of this benchmark on the X-axis and TDR in Y-axis (see Section 6.2). We show the schemas based on three categories: $S < F$, $S \approx F$ and $S > F$.

ER experiment setup: We generate this scenario in a similar fashion to Appendix A.3. We start with a real table containing information about players, teams, facilities, etc. and consider it as ground truth i.e., a clean table. Next, we randomly select 10 % of its tuples and add their duplicates/noise (at most two for each) in the clean table, which we call dirty table. For example, for tuple having (player name, team, facility) = (Mark Andrews, Ravens, M & T Bank Stadium), we add a duplicate (Andrews, Ravens, MT Stadium). Then we partition the dirty table into four tables having different schema and randomly add the rows on each of them such that they retrieve information in the dirty table.

To evaluate, we integrate four generated tables using outer join and FD. Over these tables, we apply entity resolution and verify if the tuples in the clean table are reproduced. Specifically, we use Magellan's *py_entitymatching* package to find matching tuples in the integrated tables [64]. Based on the matching results, we remove the duplicates from the tables. Given a table T (resulting table after applying ER and removing duplicates from outer joined or FD table) and a ground truth table T^* (i.e. clean table), we compute precision (P), recall (R) and F1-score (F_1) as follows:

$$P = \frac{|T \cap T^*|}{|T|}, R = \frac{|T \cap T^*|}{|T^*|}, F_1 = \frac{2 * P * R}{P + R}$$

In other words, precision and recall measure the portion of clean tuples in T and the portion of clean tuples that are covered by T respectively.