Similarity Measures For Incomplete Database Instances

Boris Glavic, Giansalvatore Mecca, Renée J Miller, Paolo Papotti, Donatello Santoro, Enzo Veltri

Similarity Measures For Incomplete Database Instances

Boris Glavic
bglavic@uic.edu
Illinois Inst. of Technology
USA

Giansalvatore Mecca
giansalvatore.mecca@unibas.it
University of Basilicata
Italy

Renée J. Miller
miller@northeastern.edu
Northeastern University
USA

Paolo Papotti
paolo.papotti@eurecom.fr
EURECOM
France

Donatello Santoro
donatello.santoro@unibas.it
University of Basilicata
Italy

Enzo Veltri
enzo.veltri@unibas.it
University of Basilicata
Italy

ABSTRACT

The problem of comparing database instances with incompleteness is prevalent in applications such as analyzing how a dataset has evolved over time (e.g., data versioning), evaluating data cleaning solutions (e.g., compare an instance produced by a data repair algorithm against a gold standard), or comparing solutions generated by data exchange systems (e.g., universal vs core solutions). In this work, we propose a framework for computing similarity of instances with labeled nulls, even of those without primary keys. As a side-effect, the similarity score computation returns a mapping between the instances' tuples, which explains the score. We demonstrate that computing the similarity of two incomplete instances is NP-hard in the instance size in general. To be able to compare instances of realistic size, we present an approximate PTIME algorithm for instance comparison. Experimental results demonstrate that the approximate algorithm is up to three orders of magnitude faster than an exact algorithm for the computation of the similarity score, while the difference between approximate and exact scores is always smaller than 1%.

1 INTRODUCTION

Organizations adopt "data lakes" for collecting their data. Rather than organizing data in carefully structured warehouses that are managed by administrators, data is now commonly stored in schema-on-read storage systems [36]. The reliance on data lakes is driving new techniques for organizing datasets [33, 51]. In this environment, a crucial task is to compare datasets. Being able to compare instances has multiple uses. First, finding datasets that are similar to an already discovered dataset or user-provided data example (e.g., find more census data or medical records [40, 41]), even if they do not share the same key values. Second, recover dataset version history in a data lake where new versions of datasets may be added to the lake without identifying them as such. Finally, different data exchange and constraint-based data repair algorithms produce different instances that need to be evaluated. Measuring how close the result of an algorithm matches a gold standard solution requires a similarity metric for incomplete databases, i.e., databases with labeled nulls. However, two challenges make the comparison of incomplete datasets difficult.

First, it is not possible in general to rely on metadata – such as keys – to reliably determine a correspondence between the tuples of two incomplete instances, i.e., key values may be missing.

Conference **I**

	Name	Year	Place	Org
t_1	VLDB	1975	Framingham	VLDB End.
t_2	VLDB	1976	*Null*	*Null*
t_3	SIGMOD	1975	San Jose	ACM

Conference $\mathbf{I_1}$

	Name	Year	Place	Org
t_7	SIGMOD	1975	San Jose	ACM
t_8	VLDB	*Null*	Framingham	VLDB End.
t_9	*Null*	1976	Brussels	IEEE
t_{10}	VLDB	*Null*	*Null*	VLDB End.

Conference $\mathbf{I_2}$

	Name	Year	Place	Org
t_{15}	*Null*	1975	*Null*	*Null*
t_{16}	CC&P	1980	Montreal	*Null*
t_{17}	VLDB	1976	Brussels	VLDB End.
t_{18}	VLDB	1975	Framingham	VLDB End.

Figure 1: Three versions of instance I.

Second, many datasets are inherently incomplete, either because the dataset creator has encoded unknown values as nulls or because the dataset is the result of a data curation step. For instance, idiosyncratic encodings of incompleteness may have been replaced with SQL-style nulls [47], a constraint-repair algorithm may have replaced conflicting values with labeled nulls [20], or outliers may have been replaced with nulls.

Data Versioning. Data versioning systems provide similar functionality for datasets that version control systems, like GIT or SVN, provide for files or software. Interest in data versioning is growing with systems like DataHub [12] and Dolt [3]. Such systems provide version management features (e.g., checkout, commit, and merge) for datasets. However, they do not support comparing versions of incomplete datasets to understand what has changed between two versions.

Example 1.1. Consider the relational schema T describing database conferences: Conference(Name, Year, Place, Org). Fig. 1 shows an initial instance (I). This instance contains missing values (denoted by *Null*). In data versioning, nulls are common. As data evolves, not every value of a tuple may be available. Fig. 1 shows two additional versions I_1 and I_2 of I.

A natural question in data versioning is which instance is closer to an original dataset I and how different are two versions. *Similarity* of instances can be used to show users how instances evolve over time by determining the order in which versions were created. Moreover, users may be interested in obtaining a list of differences across two instances, e.g., both updated versions of I contain new tuples (t_9 and t_{16}), two *Null* values in I (t_2) has

Conference $\mathbf{I_3}$

	Name	Year	Place	Org
t_{21}	N_1	1975	Framingham	N_3
t_{22}	N_1	1976	N_4	N_3
t_{23}	N_2	1975	San Jose	N_5

Figure 2: Version of I obtained with data exchange.

been updated to "VLDB End." (t_{17}), etc. The presence of nulls leads to uncertainty about which tuples are updated versions of which other tuple. For example, tuple t_{15} can be mapped to t_1 or t_3; both t_9 and t_{10} can be mapped to t_2. The need for instance similarity metrics for dataset versioning has been recognized in related work [11]. However, unlike our work, [11] does not handle incompleteness and assumes keys.

Empirical Evaluation of Data Cleaning and Integration. Empirical evaluation is important in data integration and data cleaning [9]. To provide a few examples, ST-Benchmark [5], IQ-Meter [44] and iBench [7] are examples of frameworks for data-exchange evaluation, while BART [8] is an error-generation tool for data repair. In data cleaning and integration it is common that systems differ not just in their runtime efficiency but also in terms of the quality of the results they produce. Thus, empirical evaluation of such systems requires testing how similar a system-generated solution is to a known expected solution.

Both data integration and cleaning make use of *labeled nulls*. In data exchange, labeled nulls are used to encode incompleteness in a target instance, e.g., when there are attributes in the target schema that do not have any correspondence to attributes from the source schema [23]. In constraint-based data repair, labeled nulls are used by systems to mark conflicts among values that require user intervention [10, 19, 20, 26, 29, 39].

Labeled nulls encode incompleteness [35] and turn the instances we need to compare into *representation systems* of incomplete databases. For example, in instance I_3 in Fig. 2, labeled nulls N_1 and N_3 encode the fact that the values for Name and Org are unknown for tuple t_{21}, but the values must be the same for attribute Name (Org) across tuples t_{21} and t_{22}. When we compare instances involving these nulls, satisfaction or violation of these constraints must be taken into consideration.

Challenges. The two tasks above are representative examples of applications that require an effective algorithm for comparing instances that (i) are incomplete and (ii) have no *shared key*, i.e., the instances do not have keys or the keys are not consistent across the two instances. Other applications include discovery tasks such as dataset search, given a data example, or detecting data theft and plagiarism [2, 27]. The problem of data lake deduplication aims to find duplicate or near duplicate tables [38] from real data lakes containing incomplete table. Once found, instance comparison would be valuable in understanding how to resolve the (near) duplication. We investigate a problem that is present in all these tasks, which is the one of *comparing incomplete instances without keys*, or *instance-comparison problem* for short.

This problem is challenging for two reasons. First, finding mappings between instances with nulls is related to known computationally hard problems such as checking the existence of homomorphisms between instances [17]. Indeed, we demonstrate that the instance comparison problem is NP-hard. Second, since similarity measurements must be repeated over time in dataset versioning, often with high frequency, and scalability of the tools is often an evaluation parameter, a crucial requirement is that the comparison algorithm is fast and scales to large databases.

Conference

	Id	Name	Year	Place	Org
t_1	1	VLDB	1975	Framingham	VLDB End.
t_2	2	VLDB	1976	Brussels	VLDB End.
t_3	3	SIGMOD	1975	San Jose	ACM

Paper

	Authors	Title	ConfId
t_4	Zloof	Query-By-Example...	1
t_5	Chen	The Entity-Relationship...	1
t_6	Rappaport	File Structure Design...	3

Figure 3: A Ground Instance I_g.

Recent work for comparing instances considers an easier setting with shared keys and without null values and, instead, focuses on solving other related problems such as exploring and summarizing the differences between instances by identifying transformations that map one instance into the other [15, 50]. To the best of our knowledge, there are currently no fast algorithms for comparing database instances with labeled nulls.

Contributions. Our main contributions are:

(1) We formalize the problem of comparing incomplete instances when no keys are available (Sec. 3).
(2) We formalize matches between incomplete instances (Sec. 4), enabling a scoring mechanism for instance comparison (Sec. 5).
(3) We introduce an exact and an approximate algorithm for the instance comparison problem (Sec. 6).
(4) We show experimentally that our approximate algorithm is accurate and scales to large datasets (Sec. 7).

We then discuss related work in Sec. 8 and conclude in Sec. 9.

2 INSTANCES WITH LABELED NULLS

Let us first formalize the notion of a relational instance with labeled nulls (or nulls for short). A *relational schema* R as a finite set $\{R_1, \ldots, R_k\}$ of relation symbols, with each R_i having a fixed arity $n_i \geq 0$. Consider countably infinite domains of constants (Consts) and labeled nulls (Vars). We will use $c_0, c_1, \ldots$ to denote constants and $N_0, N_1, \ldots$ to denote nulls. An *instance* $I = (I_1, \ldots, I_k)$ of R consists of finite relations $I_i \subset (\text{Consts} \cup \text{Vars})^{n_i}$, for $i \in [1, k]$. We denote by $\text{Consts}(I)$ and $\text{Vars}(I)$ the set of constants and nulls in I, respectively. The union of the two sets, $\text{adom}(I) = \text{Consts}(I) \cup \text{Vars}(I)$, is the *active domain* of I. An instance I without nulls ($\text{Vars}(I) = \emptyset$) is called a *ground instance*. We assume the presence of *unique tuple identifiers* in an instance; by t_{id} we denote the tuple with identifier "id" in I. Note that these are not assumed to be semantic keys of the instances, but just provide us with a way to reference tuples in an instance. A *cell* is a location in I specified by a tuple id/attribute pair $t_{id}.A_i$. We denote by $\text{ids}(I)$ the set of tuple ids of instance I. When comparing two instances I and I', we will assume that $\text{ids}(I) \cap \text{ids}(I') = \emptyset$. A mapping $h : \text{adom}(I) \rightarrow \text{adom}(I')$ such that $\forall c \in \text{Consts} : h(c) = c$ is called a *homomorphism* if, $\forall t \in I : h(t) \in I'$. Two instances are *isomorphic*, i.e., they represent the same information, if there exists a bijective homomorphism between I and I'.

Example 2.1. Consider the relational schema T describing database conferences and papers: Conference(Id, Name, Year, Place, Org), Paper(Authors, Title, ConfId). Fig. 3 shows a ground instance I_g of the schema, in which all values come from Consts.

Fig. 4 shows an instance I_n that, in addition to constants from Consts, contains nulls from Vars (N_1, N_2, N_3). This instance might be the result of mapping a source database into the target schema T. Some of the mappings leave unspecified the value of

Conference

	Id	Name	Year	Place	Org
t_7	N_1	VLDB	1975	N_3	VLDB End.
t_8	N_2	VLDB	1976	Brussels	VLDB End.
t_9	3	SIGMOD	1975	San Jose	ACM

Paper

	Authors	Title	ConfId
t_{10}	Zloof	Query-By-Example...	N_1
t_{11}	Chen	The Entity-Relationship...	N_1
t_{12}	Rappaport	File Structure Design...	3

Figure 4: Instance with Labeled Nulls I_n (Data Exchange).

the conference location – thus null value N_3 is present in the conference column – and perform a vertical partition of the source by creating surrogate keys for conferences (N_1 and N_2).

Finally, Fig. 5 shows another instance I_v with null N_1 (we do not report grounded table *Paper* for the sake of space). This might be the result of repairing an instance of the database that is dirty wrt. the functional dependency (FD): Conference : Name → Org. Assume the FD identifies two tuples with conflicting values for the Org attribute – say, "VLDB" and "VLDB End.". In this case, the repair algorithm uses a labeled null to mark the conflict so that a human expert solves it using domain knowledge [31].

Conference

	Id	Name	Year	Place	Org
t_{13}	1	VLDB	1975	Framingham	N_1
t_{14}	2	VLDB	1976	Brussels	N_1
t_{15}	3	SIGMOD	1975	San Jose	ACM

Figure 5: Instance with Labeled Nulls I_v (Data Repair).

3 THE INSTANCE COMPARISON PROBLEM

In this section we state natural requirements for an instance-similarity measure, motivate the concept of instance matches as a natural generalization of the symmetric difference of two ground instances, and define our instance similarity measure as the optimization problem of finding an instance match with a maximal similarity score. We then formally define instance matches in Sec. 4 and explain how to score them in Sec. 5.

A common way to measure the similarity of ground instances is the *symmetric difference* Δ, normalized to a value in $[0, 1]$:

$$\Delta(I, I') = 1 - \frac{|(I - I') \cup (I' - I)|}{|I| + |I'|}$$

In the following, we will use $\text{similarity}(I, I')$ to denote the similarity score of instances I and I' . Obviously, an instance I is maximally similar to itself. Thus, we require:

$$\text{similarity}(I, I) = 1 \tag{1}$$

However, we are comparing incomplete instances represented as instances with labeled nulls. Such an instance represents a set of ground instances, each of which is generated by substituting the nulls in the instance with constants. That is, the identity of a null does not affect the semantics of an instance: renaming a null does not change the incomplete instance represented by an instance with nulls. Thus, isomorphic instances should also be maximally similar as they encode the same set of ground instances:

$$I \text{ is isomorphic to } I' \Rightarrow \text{similarity}(I, I') = 1 \tag{2}$$

Also, two instances that are not isomorphic should receive a score strictly less than 1, e.g., consider $I = \{(N_1), (N_2)\}$, $I' = \{(N_3), (N_4)\}$ and $I'' = \{(N_5), (N_5)\}$. Intuitively, I is more similar to I' than I is to I'', because I and I' are isomorphic and, thus, represent the same set of ground instances. Instances I and I'' share some ground instances, e.g., $I_1 = \{(1), (1)\}$, but not all of them (e.g., $I_2 = \{(1), (2)\}$ is only a ground instance for I, but not I''). Thus, we require:

$$I \text{ is not isomorphic to } I' \Rightarrow \text{similarity}(I, I') < 1 \tag{3}$$

If we compare two ground instances I and I' that do not share any tuples, then I and I' should be minimally similar:

$$I \cap I' = \emptyset \wedge \text{Vars}(I) = \text{Vars}(I') = \emptyset \Rightarrow \text{similarity}(I, I') = 0 \tag{4}$$

Finally, we expect our similarity measure to be symmetric:

$$\text{similarity}(I, I') = \text{similarity}(I', I) \tag{5}$$

Notice that the symmetric difference fulfills Eq. (1) and (3) to (5), but not Eq. (2) as it does not take into account the renaming of nulls. To motivate our notion of instance similarity, let us restate the symmetric difference as follows: find a maximal matching between tuples from I and I' (called a *tuple matching*) such that only tuples that are equal are matched. The symmetric difference score is then twice the size of this matching relative to the sum of the cardinality of the two instances. For incomplete instances, we will take renaming of variables into account and also account for the fact that under some interpretation a null can be equal to a particular constant. We do this by relaxing the requirement that matched tuples have to be equal. Instead we will require them to be equal under some appropriate mapping of the variables from both instances into constants and variables (we refer to such mappings as *value mappings*). Note that there may exist many possible such mappings, some of which may not match tuples that are quite similar. Thus, we will define the instance comparison problem as the optimization problem of finding a match that maximizes the number of matched tuples while preserving the instances as much as possible, i.e., we will penalize equating two distinct nulls from an instance and prefer renaming of nulls to mapping of nulls to constants.

Example 3.1 (Example Comparison). Fig. 6 shows two instances. We can map tuple t_1 to t_4 and t_2 to t_5 by mapping nulls $N_1 \rightarrow V_a$, $N_2 \rightarrow V_a$, and $N_4 \rightarrow$ *1976* for I and $V_b \rightarrow$ *VLDB End.* for I'. Note that this is the *best* mapping we could apply. If we map $N_4 \rightarrow$ *1975* and $N_1, N_2 \rightarrow V_a$ then we can map t_2 to t_4 but we miss to map t_1 and t_5.

Given a pair of instances I and I' being compared, we will call I the left instance and I' the right instance. We refer to a combination of a *value mapping* h_l for the left instance I, a *value mapping* h_r for the right instance I', and a *tuple mapping* $m \subseteq I \times I'$ such that for any $(t, t') \in m$ we have $h_l(t) = h_r(t')$ as an *instance match* and use $\mathcal{M}$ to denote the set of all such matches for the input instances I and I'.

We postpone the formal definition of instance matches to Sec. 4 and will define the score $\text{score}(M)$ of a match M in Sec. 5. Given $\mathcal{M}$ and score we define the instance-comparison problem as shown below.

Definition 3.2 (Instance-Comparison Problem). Let I and I' be two instances of a schema R. The similarity $\text{similarity}(I, I')$ of I and I' is defined as:

$$\text{similarity}(I, I') = \max_{M \in \mathcal{M}} (\text{score}(M))$$

The **instance-comparison problem** takes as input instances I and I' and outputs $\text{similarity}(I, I')$

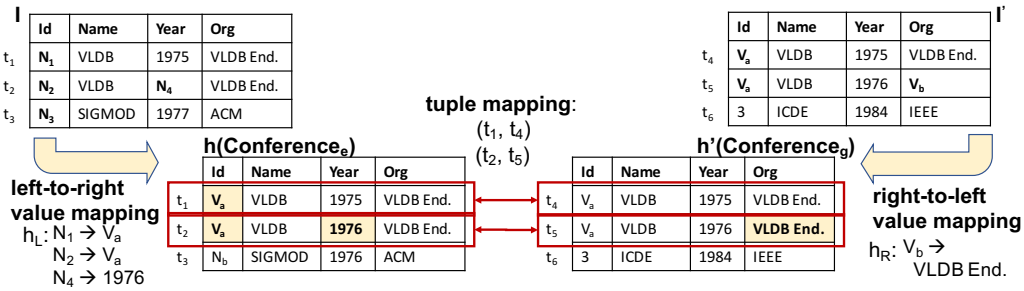

Figure 6: A Sample Instance Match.

In addition to being required for computing the similarity similarity(I, I'), the *"optimal"* instance match also provides further information about the differences between the two instances, namely: (i) how matched tuples are related to each other (by substituting nulls with other nulls or constants) and (ii) which tuples cannot be matched, e.g., tuples t_3 and t_6 in Fig. 6.

4 INSTANCE MATCHES

We now proceed with the formalization of the notion of an *instance match*. In the following, without loss of generality, we assume that we are given instances I, I' of the same relational schema R with disjoint nulls and tuple identifiers, i.e., such that $\text{Vars}(I) \cap \text{Vars}(I') = \emptyset$ and $\text{ids}(I) \cap \text{ids}(I') = \emptyset$. Notice that this is not a limiting assumption as (i) we can always generate such tuple identifiers as we do not require them to be predictive of what tuples are related across the two instances and (ii) we can rename labeled nulls in an instance without changing its semantics as long as we do not equate nulls that are different before the renaming. In addition, later in this section we discuss how to relax the requirement on the same relational schema.

4.1 Value Mappings

To start, let us first formalize the notion of a *value mapping* as a mapping h of the values in $\text{adom}(I)$ into $\text{Vars} \cup \text{Consts}$ that preserves constants:

Definition 4.1 (Value Mapping). Let I be an instance. A value mapping h for I is a total function $\text{adom}(I) \rightarrow \text{Vars} \cup \text{Consts}$ such that $h(c) = c$ for each $c \in \text{Consts}(I)$. We use $h(t)$ to denote the application of value mapping h to the attribute values of a tuple t and $h(I)$ to denote the application of h to all tuples in I.

As constants are fixed across all ground instances represented by an instance with nulls, we do not allow a constant to be mapped to a different constant. For instance, t_{20} in Fig. 1 is not mapped to any tuple in instance I.

As a notational conventional, we will sometimes specify a value mapping h as a partial function and assume that h is the identity on all other values of $\text{adom}(I)$.

4.2 Tuple Mappings

Given instances I, I', *tuple mappings* specify the pairs of tuples from I, I' we want to match to each other.

Definition 4.2 (Tuple Mapping). Given two instances I and I' for the same schema R, a tuple mappings m is a subset of $I \times I'$.

Notice how we design tuple mappings as relations, not functions. In this way we may take into account not only functional, total mappings – like homomorphisms – but also non-functional mappings. In fact, we classify tuple mappings as follows:

- *left injective* iff $\forall t \in I : \nexists t_1 \neq t_2 \in I' : (t, t_1) \in m \wedge (t, t_2) \in m$;
- *right injective* $\forall t \in I' : \nexists t_1 \neq t_2 \in I : (t_1, t) \in m \wedge (t_2, t) \in m$;
- *fully injective* iff it is both left injective and right injective;
- *right (left) total* iff it is right (left) surjective.

4.3 The Notion of an Instance Match

We are now ready to introduce the notion of an *instance match*. As discussed in Sec. 3, an instance match is composed of a tuple mapping and two value mappings:

Definition 4.3 (Instance Match). Let I and I' be two instances over schema R. An *instance match* is a triple $M = (h_l, h_r, m)$ where h_l is a value mapping for I, h_r is a value mapping for I', and m is a tuple mapping for I and I'. An instance match M is a **complete match** iff

$$\forall (t_1, t_2) \in m : h_l(t_1) = h_r(t_2)$$

We use $\mathcal{M}$ to denote the set of all complete instance matches for I and I'.

Note that the notion of a complete instance match is a generalization of the notion of a homomorphism between instances with nulls. Specifically, if m is total on I (I') and is left (right) injective, and h_r (h_l) is the identity, then M is a homomorphism from I to I' (I' to I). If m is total on both I and I' and m is fully injective, then M is an isomorphism.

The rationale for defining a large number of properties for tuple, value and instance matches is that our instance similarity measure can be tailored to specific applications by restricting value mappings, tuple mappings, and/or instance matches. We discuss some use cases in the following.

Data Versioning. In data versioning, we may use our instance similarity measure to test how likely it is that instance I evolved into I'. In such a scenario, some old tuples may no longer exist in I' and some new tuples may be inserted (do not exist in I). Assuming that tuples represent unique entities, we should require that the tuple mapping is *fully injective* but does not require it to be total on either side. If we are dealing with a domain where tuples may get merged, e.g., we have multiple patient records for a person with missing information that get merged into a complete record, then we should only require the tuple mapping to be *left injective*. Our discussion has primarily centered on matching instances with identical relational schemas. However, our approach can be modified for instances with different schemas. For example, if instance I has an attribute A_i not in I', then our instance similarity measure can be used by adding a column to I' with distinct null values for each row. This allows mapping tuples in I to tuples in I' without constraints over attribute A_i. Finally, if we drop the requirement for complete matches, similar tuples may be matched to each other (e.g., two people with the

same attributes except different salaries). However, such partial instance matches further increase the size of the search space of instance matches: even given the value mappings, there are multiple possible tuple mappings as a tuple can be matched against any other tuple. For the remainder of the paper, we focus on complete matches and discuss partial instance matches in Sec. 6.3.

Data Exchange. Given an instance I for a source schema S, a target schema T, and a schema mapping Σ which is a set of logical constraints relating these two schemas, data exchange systems [23] generate an instance J of a target schema such that $(I, J) \models \Sigma$. There are typically many possible target instances that are solutions for a data exchange scenario. Most approaches produce so-called *universal solutions* which have some desirable properties including being the only solutions over which certain answers to unions of conjunctive queries (answers that are in the result to the query for every possible solution) can be computed by simply evaluating the query. This is due to the fact that for a universal solution J and any other solution J', there exists a homomorphism from J to J'. All universal solutions are homomorphically equivalent, but typically not unique. Some approaches produce so-called *core* solutions which are unique up to isomorphism. Evaluation of data exchange systems may require comparing a produced (universal) solution to a (universal) gold standard solution, such as the core solution or one provided by a benchmark [6, 7]. Comparing two universal solutions requires a *non injective* mapping since the same source information can be exchanged into multiple tuples in an instance (e.g., (VLDB, 1976, N_1), (VLDB, N_2, Brussels)) or merged in a single tuple in another instance (e.g., (VLDB, 1976, Brussels)) and vice versa. As another example, we may want to compare a universal solution produced by a data exchange system against a core solution to measure the amount of redundancy in the universal one. Based on the properties of cores and universal solutions, we can require tuple mappings to be *left and right total* since all tuples need to be mapped to one or more tuples in the other instance. Furthermore, given that if J is a universal solution for I, there is a one-to-one homomorphism h from J to the core solution J_0 [25, Corollary 3.5], then the tuple mapping has to be *left injective*. If we know that we are comparing two universal solutions, tuple mappings have to be total, but we cannot require them to be left injective.

Constraint-based Data Repair. Consider repairing an instance I that violates a set of integrity constraints Σ. Some data repair systems update cell values to repair the instance. A constant value can be changed to another constant value to satisfy a constraint [34], or if there is an ambiguity, some tools introduce variables to identify conflicts that need to be resolved by further intervention, e.g., by a human [10, 19, 20, 26, 29, 39]. To compare repairs produced by two systems or compare a repair against a gold standard repair, we need to use *complete and full injective* mapping. In addition to the similarity score, the instance match can be used to explain the repair by highlighting non-matching tuples and by identifying how tuples were repaired.

Note that these are just a few of the many applications of some restricted versions of our instance similarity measure. Our final goal is to support a wide variety of scenarios by using general relations and adding in restrictions (injectivity or totality) as required by the problem setting.

5 SCORING INSTANCE MATCHES

We formalize how instance matches are scored. We first define a tuple score and then use it to define an instance score. We then present complexity results for the instance matching problem.

5.1 Match Score

Recall that we require our definition of similarity to be symmetric (see Eq. (5)), i.e., the score of an instance match should not depend on the order in which instances I, I' are considered. As a consequence, we need a symmetric function to calculate scores.

Given an instance match $M = (h_l, h_r, m)$, we will define the similarity measure by assigning scores to each tuple based on what tuples in the other instance it is matched with by the tuple matching m. Each tuple t will be assigned a score between $[0, n]$ where n is the arity of t. To achieve a similarity score in $[0, 1]$ we will normalize the sum of the tuple scores by the sizes of the instances defined below.

Definition 5.1 (Size of an Instance). Let I be an instance of a schema R

$$size(I) = \Sigma_{t \in I}(arity(R)) = |I| \cdot arity(R)$$

We first define a tuple score $\text{score}(M, t)$ and then use this, together with the size of instances, to define an instance score.

As a tuple matching m may not be injective, we have to decide how to calculate a score for a tuple based on the tuples it is matched to by m. For that, we define the *image* of a tuple according to a tuple mapping m. For a tuple $t \in I$ we define the *image* of t as $m(t) = \{t_m \mid (t, t_m) \in m\}$, and for $t' \in I'$ the *image* of t' as $m(t') = \{t_m \mid (t_m, t') \in m\}$. We then calculate the score of a tuple t as the average score for the pairs (t, t') for every tuple t' in the image of t. Finally, we define $\text{score}(M, t, t')$, the score of a pair of tuples such that $(t, t') \in m$, as the sum of scores for each cell $\text{score}(M, t, t', A)$ for attribute A in t (t'), which will be discussed below.

Definition 5.2 (Tuple Score). Let I and I' be two instances of a schema R and M be an instance match of I and I'. Given a tuple $t \in I$ (or $t \in I'$), we define the score of t with respect to M as:

$$\text{score}(M, t) = \frac{\sum_{t_m \in m(t)} \text{score}(M, t, t_m)}{size(m(t))}$$

We are now ready to define an instance match score.

Definition 5.3 (Instance Match Score). Given an instance match M between instances I and I', the score $\text{score}(M)$ of M is:

$$\text{score}(M) = \frac{\sum_{t \in I} \text{score}(M, t) + \sum_{t' \in I'} \text{score}(M, t')}{size(I) + size(I')}$$

Recall the four requirements Eq. (1), (2), (4) and (5) for $\text{similarity}(I, I')$. Given the definitions for tuple (pair) scores and instance match scores above, these enforce the following constraints on the assignment of cell scores for any tuple pair $(t, t') \in m$:

Lemma 5.4 (Cell Score Properties). *Given the definitions for scores shown above, unless a function* $\text{score}(M, t, t', A)$ *fulfills the following conditions then* $\text{similarity}(I, I')$ *violates at least one of Eq.* (1), (2), (4) *and* (5)

(1) If $t.A = t'.A$ *and* $t.A \in Consts$, *then* $\text{score}(M, t, t', A) = 1$.

(2) If I *and* I' *are isomorphic and* $h_l(t.A) = h_r(t'.A)$ *for* $(t, t') \in m$, *then* $\text{score}(M, t, t', A) = 1$.

(3) If I *and* I' *are not isomorphic then there has to exist at least one pair* $(t, t') \in I \times I'$ *and attribute* A *such that* $h_l(t.A) = h_r(t'.A)$ *for* $(t, t') \in m$ *and* $\text{score}(M, t, t', A) < 1$.

(4) $\text{score}(M, t, t', A) = \text{score}(M^{-1}, t', t, A)$ *where* $M = (h_l, h_r, m)$, $M^{-1} = (h_r, h_l, m^{-1})$, *and* $m^{-1} = \{(t', t) \mid (t, t') \in m\}$.

Proof. Condition (1) is necessary to ensure that comparing a ground truth instance with itself has score 1 (Eq. (1)). Condition (2) is necessary to ensure that isomorphic instances have score q (Eq. (2)). Condition (3) is required to ensure that non-isomorphic instances have a similarity strictly less than 1. Finally, condition (4) is necessary for ensuring symmetry. The full proof in [32]. □

While Lem. 5.4 places some restrictions on $\text{score}(M, t, t', A)$ it does not uniquely define it. We now motivate additional design decisions for $\text{score}(M, t, t', A)$. First, observe that Lem. 5.4 does not restrict scores for mapping nulls to constants. As a null represents a different value in each ground instance represented by an instance with nulls, intuitively, mapping a null to a constant should get a score less than 1 (the score for matched constants). Furthermore, we will ensure conditions (2) and (3) by measuring the degree of non-injectivity for value mappings for a null in I (I') and penalize scores for cells which contain nulls with larger degrees of non-injectivity. This ensures that for isomorphic instances where h_l and h_r will be injective on nulls, there is no penalty, and for non-isomorphic instances either some tuples do not match or both value mappings are not injective on all nulls.

Towards this goal, we define a function $\sqcap$ for a value v in I, I', that measures that level of *"non-injectivity"* of the value mappings h_l, h_r for v. We distinguish the case of a constant from the one of a null. For constants, $\sqcap$ is always equal to 1 – this captures the fact that constants can only be mapped to themselves and therefore cannot be the source of non-injectivity. This is due to the mapping of nulls, for which we distinguish the case of $v \in \text{Vars}(I)$, and $v \in \text{Vars}(I')$:

$$\sqcap(v) = \begin{cases} 1 & \text{if } v \in \text{Consts} \\ |\{v' | h_l(v') = h_l(v)\}| & \text{if } v \in \text{Vars}(I) \\ |\{v' | h_r(v') = h_r(v)\}| & \text{if } v \in \text{Vars}(I') \end{cases} \quad (6)$$

Based on the discussion so far, for cells $t.A, t'.A$, which are both nulls, we set their score to $2/(\sqcap(t.A) + \sqcap(t'.A))$. Thus, if both h_l is injective on $t.A$ and h_r is injective on $t'.A$, as is the case for isomorphic instances, the score is 1. We use $\sqcap(t.A, t'.A)$ to denote $\sqcap(t.A) + \sqcap(t'.A)$. We now define tuple pair scores.

Definition 5.5 (Tuple Pair Score). Let I and I' be two instances of a schema R and M be an instance match of I and I'. Given a pair $t \in I$ and $t' \in I'$, we define the score of (t, t') wrt. M as shown below. We assume a parameter $0 \leq \lambda < 1$, which defines the penalty for mapping a variable to a constant, given as input.

$$\text{score}(M, t, t') = \sum_{A \in R} \text{score}(M, t, t', A)$$

$$\text{score}(M, t, t', A) =$$

$$\begin{cases} 0 & \text{if } h_l(t.A) \neq h_r(t'.A) \\ 1 & \text{if } t.A, t'.A \in \text{Consts} \wedge t.A = t'.A \\ \frac{2}{\sqcap(t.A, t'.A)} & \text{if } t.A, t'.A \in \text{Vars} \wedge h_l(t.A) = h_r(t'.A) \\ \frac{2 \times \lambda}{\sqcap(t.A, t'.A)} & \text{otherwise, with } h_l(t.A) = h_r(t'.A) \end{cases}$$

Theorem 5.6 (Correctness). *Similarity measure* $\text{similarity}(I, I')$ *fulfills Eq.* (1) *to* (5).

Proof. It is easy to see that $\text{score}(M, t, t', A)$ fulfills the conditions of Lem. 5.4. Substituting the definitions of instance match score, tuple score, and tuple pair score, defined above, this implies that $\text{similarity}(I, I') = \max_{M \in \mathcal{M}}(\text{score}(M))$ fulfills Eq. (1) to (5). For the full proof please see [32]. □

5.2 Examples

Let us illustrate the score function by means of a few examples.

Example 5.7. Consider the following instances I and I':

I

	Id	Year	Org
t_1	N_1	1975	VLDB End.
t_2	N_2	1976	VLDB End.

I'

	Id	Year	Org
t_3	N_a	1975	VLDB End.
t_4	N_b	1976	VLDB End.

A total fully-injective tuple mapping m maps tuple t_1 in t_3 and tuple t_2 in t_4, with the left-to-right value mapping h_l defined as $N_1 \rightarrow N_a$, $N_2 \rightarrow N_b$. Since all nulls are properly renamed, $\sqcap$ is equal to 1 for all values. In this case, $\text{score}(M, t_1, t_3, Id) = 2/(\sqcap(N_1) + \sqcap(N_a)) = 1$, and similarly for tuple pair t_2 and t_4. Hence the total score of the mapping is equal to 1.

Example 5.8. Assume now we compare instance I with I'':

I''

	Id	Year	Org
t_3	N_a	1975	V_1
t_4	N_b	1976	V_1

We still map both tuples in I to I'' with the same left-to-right value mapping h_l in Ex. 5.7. In addition, we have a right-to-left value mapping h_r defined as $V_1 \rightarrow$ *VLDB End.*. Since also in this case all nulls are properly renamed, $\sqcap$ is equal to 1. The final score is $(8 + 4\lambda)/12$, which is less than 1 if $\lambda < 1$, since we penalize that value *VDLB End.* was approximated with the variable V_1.

Example 5.9. Consider the instance match in Fig. 6. Since all nulls are properly renamed, the score is $(12 + 4\lambda)/24$.

Example 5.10. Consider the following instances S, S', S'':

S

	Dept	Name
t_1	A	Mike
t_2	A	Laure

S'

	Dept	Name
t_3	A	N_1
t_4	A	N_2

S''

	Dept	Name
t_5	A	N_3

For S,S', there is a total fully-injective tuple mapping that maps tuple t_1 in t_3 and tuple t_2 in t_4, with the right-to-left value mapping defined as $N_1 \rightarrow Mike$, $N_2 \rightarrow Laure$. Since all nulls are properly renamed, $\sqcap$ is equal to 1 for all values. In this case, score $(M, t_1, t_3, Name) = (2\lambda)/(\sqcap(Mike) + \sqcap(N_1)) = \lambda$, and similarly for tuple pair t_2, t_4. Hence the total score is equal to $(4 + 4\lambda)/8$.

For S,S'', the same null N_3 is mapped to only one constant (as a function, by definition of Value mapping). Therefore score$(M, t_1, t_3, Name) = (2\lambda)/(1 + 1)$, and there is no tuple pair matching t_2 and t_5. The final score of the match is $(1 + \lambda + 1 + \lambda)/6$, which is lower than the score for S and S'.

5.3 Complexity Analysis

We now analyze the complexity of the instance comparison problem. Unfortunately, the instance comparison problem is intractable in general. It is tractable if both instances are ground.

Theorem 5.11 (Complexity Results). *The instance-comparison problem is NP-hard in terms of data complexity. The decision version of the problem (is* $\text{similarity}(I, I') \geq k$ *for a threshold* k*) is NP-complete. The problem remains hard even if one of the two instances is ground. The problem has* PTIME *data complexity if both instances are ground.*

Proof. We show the hardness result through a reduction from the NP-complete 3-colorability problem. We provide a PTIME verification procedure to show that the decision version is NP-complete. The PTIME runtime for the constant-only case is proven constructively, i.e., we provide a PTIME algorithm for the problem under this restriction. For the full proof, please see [32]. □

6 ALGORITHMS

We now present an *exact algorithm* for the intractable instance-comparison problem from Def. 3.2. Unsurprisingly, this algorithm is expensive. We then introduce the *signature algorithm*, a more efficient algorithm that returns approximate similarity scores.

6.1 The Exact Algorithm

The exact algorithm (sketched in Alg. 1) works in two steps. First, we build a set of *candidate tuple pairs* by looking for *compatible tuples*. We say that (t, t') from I, I' are *compatible* if it is possible to construct value mappings h_l, h_r such that $h_l(t) = h_r(t')$. Then, we combine these candidate tuple pairs in all possible ways to construct candidate instance matches, compute their scores, and return the instance match with the highest score.

Algorithm 1: EXACT(I, I')

Input: Two instances I and I' for the same schema R
Output: The best instance match $M = (h_l, h_r, m)$.

```
1  compatible ← CompatibleTuples(I, I')
2  m ← ∅                                   // candidate tuple mappings
3  if searching for non left injective mapping then
4      NFM = GenNonFunctionalMapping(compatible)
5      m ← P(NFM)    // powerset of non-functional mappings
6  else
7      FM ← GenerateFunctionalMappings(compatible)
8      foreach m ∈ FM do // union powersets of func. mappings
9          m ← m ∪ P(m)
10 M ← ∅                                   // instance matches
11 foreach m ∈ m do                  // construct instance matches
12     M ← FindCompleteInstanceMatch(m)
13     if M exists then     // can extend m into instance match?
14         M ← M ∪ {M}
15 return argmax_{M∈M} score(M)
```

Algorithm 2: COMPATIBLETUPLES(T_l, T_r)

Input: Sets of tuples T_l and T_r for the schema R
Output: Dictionary for every $t \in T_l$ to compatible ones in T_r

```
1  foreach A ∈ R do
2      V_A ← ∅                              // Init empty dictionary
3  foreach t' ∈ T_r do
4      foreach A ∈ R do
5          if t'.A ∈ Consts then
6              V_A[t'.A] ← V_A[t'.A] ∪ {t'_id}     // add as t'.A
7          else
8              V_A[*] ← V_A[*] ∪ {t'_id}           // add as null
9  compatible ← ∅
10 foreach t ∈ T_l do
11     compatible[t] = ids(T_r)
12     foreach A ∈ R do                  // compute c-compatibles
13         if t.A ∈ Consts then      // intersect attribute maps
14             compatible[t] ←
                 compatible[t] ∩ (V_A[t.A] ∪ V_A[*])
15     foreach t' ∈ compatible[t] do  // remove non-compatible
16         if ¬ t ≃ t' then
17             compatible[t] ← compatible[t] − {t'}
18 return compatible
```

Step 1: Finding Compatible Tuples. A straightforward way to implement the first step is to compare every tuple t in I to every tuple t' in I'. This has a quadratic cost that can be avoided (assuming that there is an upper bound on the number of tuples a tuple is compatible with) by relying on the following property:

Definition 6.1. Tuples t, t' from I, I' are *c-compatible*, written as $t \sim t'$ iff they do not contain conflicting constant values, i.e., there is no attribute A for which $t.A, t'A \in$ Consts and $t.A \neq t'A$. Furthermore, t and t' are *compatible*, written as $t \simeq t'$ iff there exists value mappings h_l and h_r such that $h_l(t) = h_r(t')$.

Note that c-compatibility is a necessary, but not sufficient, condition for the compatibility of two tuples. Consider, for example, $t = \langle a1, b1, c1\rangle$ and $t' = \langle a1, N_1, N_1\rangle$ are c-compatible, yet they are not compatible: in fact, no pair of value mappings can map null N_1 to both constants $b1$ and $c1$. We use c-compatibility to prune candidate tuple pairs early on as follows:

- *Step 1.1.* for each attribute A find all sets of tuples in I, I' that are c-compatible on A, i.e., they have the same value for A, or a null value;
- *Step 1.2.* given tuple $t \in I$, compute all sets from I' that are c-compatible with t on some attributes, and their intersection.
- *Step 1.3* try to construct value mappings h_l, h_r for all non-pruned candidates to determine compatible tuples.

The main advantage of this approach comes from the fact that we can use hashing to solve step 1.1. and 1.2 in linear time. Furthermore, step 1.3 also just requires linear time.

More precisely, for each attribute A in R we build a hash-based index of the tuples in I', denoted as $\mathcal{V}_A$ (Alg. 2 lines 3-8). Index $\mathcal{V}_A$ maps each constant value c that appears in attribute A to the set of all tuple ids id in I' such that $t_{id}[A] = c$ (Alg. 2 line 6). To record the position of nulls, we introduce a special, constant value, say $*$, that does not appear elsewhere in I, I', and map it to all tuples that contain a null for A (Alg. 2 line 8). Fig. 7 shows the entire value map for instance I' (Step 1.1). For brevity, we use $A[c]$ to denote the set of tuples t for which $t[A] = c$.

Using $\mathcal{V}_A$, we determine pairs of compatible tuples. Given $t \in I$, the set of tuples of I' compatible with t, denoted by $compatible(t)$, is computed as follows (Alg. 2 lines 9-17):

- For each attribute A of R, consider $t[A] = v$. We find the set of tuples c-compatible with t wrt. A. If $v \in$ Vars, these are all tuples in I'. If $v \in consts$, this is the union of two sets: (i) $\mathcal{V}_A[v]$, i.e., all tuples $t' \in I'$ such that $t'[A] = v$; (ii) $\mathcal{V}_A[*]$, i.e., all tuples $t' \in I'$ such that $t'[A] \in$ Vars.
- Then $compatible(t)$ is obtained as the intersection of all sets $compatible(t, A)$, for each $A \in R$, discarding all tuples t' for which $\neg t \simeq t'$ (it is not possible to construct value mappings h_l, h_r s.t. $h_l(t) = h_r(t')$). This check is linear in the arity of R.

Consider, for example, tuple $t_2 = \langle a1, N_3, c1\rangle$ in I. To find tuples c-compatible with t_2, we intersect the sets: (i) $\mathcal{V}_A[a1] \cup \mathcal{V}_A[*]$; (ii) $\mathcal{V}_C[c1] \cup \mathcal{V}_C[*]$. We disregard attribute B since all tuples in I' are potentially compatible with null N_3. Then, we check each of the tuples in the intersection for compatibility. As a result, t_2 is compatible with t'_1, t'_2, as shown in Fig. 7. Once we have determined all compatible tuple pairs, we construct a value mapping, h_l, h_r for each such pair (t, t') such that $h_l(t) = h_r(t')$.

Step 2: Finding Instance Matches. Once we have discovered the pairs of compatible tuples, we combine these in all possible ways to generate instance matches. Notice that if t and t' are compatible with each other, this only guarantees that it is possible to build some value mappings h_l and h_r that maps nulls into constants or into each other, such that $h_l(t) = h_r(t')$. However,

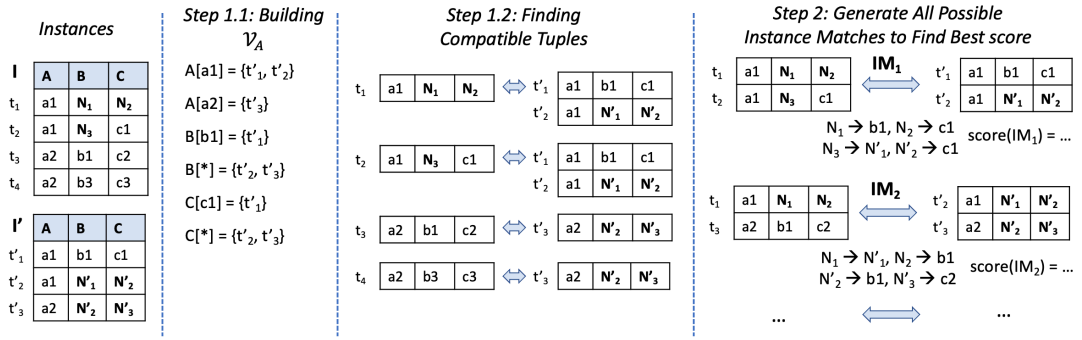

Figure 7: The Exact Algorithm.

as soon as we consider two pairs of compatible tuples, say (t_1, t'_1), (t_2, t'_2), each with an associated pair of value mappings, h^1_l, h^1_r, h^2_l, h^2_r, we may discover that these are incompatible – for example, because h^1_l, h^1_r map a null N to a constant c, while h^2_l, h^2_r maps N to a different constant c'.

In our example in Fig. 7, for the mapping of t_2 into t'_2, we map null N'_2 to $c1$. This makes it impossible to map t_3 to t'_3, which would require to map N'_2 to $b1$. Therefore, we construct all possible tuple mappings that consist of compatible tuples and check if value mappings are consistent. This step is combinatorial: for each tuple $t \in I$ we need to consider all possible mappings, i.e., the powerset of $compatible(t)$. Then, we combine the possible mappings for tuples in I in all possible ways and finally check if value mappings are consistent to generate the instance match.

Notice that the number of possible tuple mappings can be reduced if we restrict our attention to left-injective mappings, i.e., those that are functional on I. For each tuple in t, possible mappings consist of a single tuple t' from the set of $compatible(t)$ – i.e., no need to compute powersets. Still, this does not affect the asymptotic complexity of the algorithm. As we find a candidate tuple mapping that is total on I' – i.e., we have associated a tuple of I' with each tuple of I – we still must consider all possible subsets, i.e., all possible non-total tuple mappings. This is necessary, because our scoring function may assign a higher score to a subset than to the total mapping.

Once all candidate tuple mappings have been constructed, we check if any of these actually represent a complete instance match – i.e., all value mappings are indeed compatible with each other. To do this, for each candidate tuple mapping m, we consider tuple pairs one by one, and try to *"grow"* the final value mappings h_l and h_r such that $h_l(t) = h_r(t')$ for each $(t, t') \in m$, based on the partital value mappings for these pairs.

Finally, we consider the set of instance matches generated above (Step 2 in Fig. 7). If the set is empty, we fail. Otherwise, we return the instance match with the highest score.

6.2 The Signature Algorithm

We now present a scalable approximate algorithm, that we show empirically to often obtain optimal or near optimal results for real use cases. The intuition is that finding mappings between tuples sharing the same constant values is easier that finding mappings between tuples that have no conflicting constant values. To do that, we introduce the concept of a *signature* of a tuple t, as a positional encoding of some of the constants in the tuple. Consider for example tuple t_5 in Fig. 6: $t_5 : \langle V_b, VLDB, 1976, V_c \rangle$. One signature of t_5 is: *[Name: VLDB, Year: 1975]*.

Algorithm 3: SIGNATURE(I, I')

Input: Two instances I and I' for the same schema R
Output: An instance match $M = (h_l, h_r, m)$

```
1  T_l = I , T_r = I'
2  M = (h_l, h_r, m) ← ∅
3  FindSigMatches(T_l, T_r, M)     // determine signature matches
4  FindSigMatches(T_r, T_l, M)
5  compatible = CompatibleTuples(T_l, T_r)  // determine compat.
6  foreach t ∈ T_l do        // greedy instance match generation
7      foreach t' ∈ compatible[t] do
8          if IsCompatible(t, t', M) then
9              UpdateInstanceMatch(M, t, t')
10             if searching for right injective mapping then
11                 Remove(t', compatible)
12             if searching for left injective mapping then
13                 goto 6
14 return M
```

The pseudo-code of the signature algorithm is in Alg. 3. The algorithm is greedy: as soon as it finds a compatible mapping of two tuples based on their signatures, it uses it to construct the instance match. The intuition is to start with very promising matches, i.e., tuples that share most constant values, and then move to less promising ones. We discuss the algorithm next.

Algorithm 4: FINDSIGMATCHES(T_l, T_r, M)

Input: Sets of tuples T_l and T_r and instance match $M = (h_l, h_r, m)$

```
1  sigmap ← ∅
2  foreach t ∈ T_l do          // compute maximal signatures
3      sigmap[S_max[t]] ← sigmap[S_max[t]] ∪ {t}
4  foreach t' ∈ T_r do
5      A_ground = {A | t'[A] ∈ Consts}
6      foreach A ∈ P(A_ground) do
7          foreach t ∈ sigmap[S[t', A]] do   // use Property 1
8              if IsCompatible(t, t', M) then
9                  UpdateInstanceMatch(M, t, t')
10                 if searching for left injective mapping then
11                     Remove(t, sigmap)
12                     Remove(t, T_l)
13                 if searching for right injective mapping then
14                     Remove(t', T_r)
15                     goto 4
```

Step 1: Building Signatures. Given a tuple t over schema R, we associate with it a number of *signatures*.

Definition 6.2 (Signature). Given a tuple t over R, a *signature* for t is any string of the form $[A_{i_1} : v_{i_1}, \ldots, A_{i_k} : v_{i_k}]$, where:

- for each $j \in \{1, \ldots, k\}$, A_{i_j} is an attribute of R such that $t[A_{i_j}] \in \text{Consts}$, i.e., t has constant values on all A_{i_j};
- attributes $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ appear in lexicographic order.

We use $\mathcal{S}[t, \mathbf{A}]$ to denote the signature of tuple t on a set of attributes $\mathbf{A}$. The *maximal signature* $\mathcal{S}_{max}[t]$ for t is the signature on $\mathbf{A}_{max,t} = \{A \mid t[A] \in consts\}$.

It is easy to see that the following property holds:

Property 1: *Consider tuples t and t'. If $\mathcal{S}_{max}[t] = \mathcal{S}[t', \mathbf{A}_{max,t}]$, then $t \sim t'$.* □

Consider, for example, our tuple t_5 in Fig 6, and its maximal signature, $\mathcal{S}_{max}[t]$ = *[Name: VLDB, Year: 1975]*. Based on the maximal signature, if a tuple t' over the same schema has values *VLDB* and *1975* for attributes Name and Year, respectively, then t' is c-compatible with t. This means that if t' has a signature $\mathcal{S}[t', \mathbf{A}_{max,t}]$ – not necessarily the maximal one – equal to $\mathcal{S}_{max}[t]$, then $t \sim t'$. We call a tuple match that satisfies Property 1 a *signature-based* match.

Step 2: Finding Signature-Based Matches. Our search for compatible tuples relies on signatures. Based on Property 1, we construct all maximal signatures for tuples in one of the instances – say I – and store them in an appropriate hash-based data structure, called a *signature map*. Then, we scan the tuples of the other instance – I' in our example – and for each of them consider all of its signatures to find possibly-matching tuples on the other side. In doing this, we greedily construct our instance match.

Notice that, given tuple $t \in I$, by Property 1 we can only find candidate tuples $t' \in I'$ that have at least as many constants. To identify candidates with less constants, we need to reverse the direction of the check. Thus, given the symmetric nature of our notion of an instance match, the algorithm runs in two steps:

- it first scans tuples in I to find candidate tuples in I' based on their maximal signatures;
- then, it does the opposite, i.e., it scans the tuples of I' to find candidates in I.

More precisely:

(1) we start with an empty instance match, M;
(2) we populate this instance match M by calling the procedure FindSignatureMatches on I and I' (Alg. 3 line 3)
(3) in this procedure, we generate the signature map of all tuples in I (Lines 2-3 in Alg. 4);
(4) we scan tuples in I'; for each tuple $t' \in I'$: (i) we consider the set of attributes $\mathbf{A}_{ground}$ that have constant values in t'; (ii) we progressively generate the powerset of $\mathbf{A}_{ground}$, starting with subsets of the largest size (Alg. 4 line 6); (iii) for each subset $\mathbf{A}$ we generate the corresponding signature $\mathcal{S}[t', \mathbf{A}]$; (iv) using the signature map, we obtain all tuples $t \in I$ such that $\mathcal{S}_{max}[t] = \mathcal{S}[t', \mathbf{A}]$ (Alg. 4 line 7); (v) for each of these tuples, we check whether $t \simeq t'$ are indeed compatible with each other (considering labeled nulls) and with the current instance match, M; if this is the case, we update M to include (t, t') in the tuple mapping (Alg. 4 line 9).

After this first run, we repeat steps 2–4 above with I' in place of I and vice-versa (Alg. 3 line 4). Consider our example in Fig. 7. All pairs of compatible tuples satisfy Property 1, thus can be found with their signatures. Therefore, the generation of compatible tuples is much faster than with the exact algorithm.

Step 3: Completing the Instance Match. We have derived an instance match M that contains signature-based matches, but these do not cover all possible tuple matches. Consider tuples $t_2 = \langle N_2, VLDB, N_4, VLDB\ End. \rangle$ and $t_5 = \langle V_b, VLDB, 1976, V_c \rangle$ in Fig. 6. Despite the two tuples are compatible (they are matched in Fig. 6), they have no signature-based match. This is due to the different positions of the nulls, that prevent from using maximal signatures to identify the match. Therefore, we complete the process by adding non-signature-based matches. This step relies on the same procedure CompatibleTuples in the exact algorithm (Alg. 3 line 5). However, instead of trying all powersets, we adopt a greedy approach: as soon as an extension of M exists for two compatible tuples, t and t', the match is confirmed (Alg. 3 line 9).

Since signature-based matches are typically a majority of the matches to identify, the number of tuples in the final step of the algorithm is lower than the original size of I. The worst-case runtime of the signature-based mapping algorithm is quadratic in the number of tuples and combinatorial in the number of attributes with labeled nulls.

We distinguish four cases for the signature algorithm.

Case 1: General Instance Matches. For the most general form of the instance comparison problem, as defined in Sec. 3, the algorithm brings an improvement over the exact solution, despite the final step (find matches that are not signature-based) requires to check every compatible tuple-pair. However, since the combinatorial aspect of the exact algorithm is avoided, this step is feasible for large instances, as shown in Sec. 7.

Case 2: Fully Signature-Based Matches. At the opposite end of the complexity spectrum, when instance matches are fully signature-based, the algorithm is extremely fast. In this setting, it runs in linear time wrt. the instance size, and combinatorial wrt. the number of columns that contain labeled nulls.

Case 3: Functional Matches. The algorithm is still considerably faster than the exact one in many typical cases. For example, it brings a considerable speed-up when looking for left-injective – i.e., functional tuple mappings. In this case, as soon as we have matched a tuple t from I to some tuple t' from I', we may remove it from the ones under consideration, since we do not test further matches for t (while it is needed for Case 1).

Case 4: Fully-Injective Matches. The benefit is even more apparent with fully-injective mappings, i.e., functional mappings that are also injective. In this case, in addition to removing tuples from I from the ones under consideration as soon as they have been matched, we do the same for the tuples in I', thus further decreasing the time needed to execute the last step.

6.3 Partial Mappings

The algorithms presented so far generate instance matches that are complete. In some cases, one might be interested in partial matches, where a tuple is matched to a similar tuple, having a different constant value for one or more attributes. The framework presented in Sec. 4 is general enough to handle these mappings. However, from a practical point of view, this setting further increases the size of the search space of instance matches. Both the exact and the signature algorithms exploit the c-compatibility to aggressively reduce the search space. The exact algorithm can be adapted to support partial mappings by implementing a more flexible variant of alg. 2. This could involve using string similarity thresholds to assess the compatibility of two tuples. However, such a version would be substantially slower, as it cannot use hashing for comparisons. Moreover, an increase in compatible

tuples leads to a corresponding rise in the number of powersets that must be managed. As discussed, the signature algorithm works by greedily finding mappings between tuples sharing the same constant values. With partial mappings, the Property 1 is not longer valid, and should be revised in:

Property 2: *If* $\mathcal{S}[t, \mathrm{A}] = \mathcal{S}[t', \mathrm{A}]$, *then* $t \sim t'$. □

Partial matches between two tuples occur when they share at least one signature, which is not necessarily the maximal one. For the *signatureMap* construction in alg. 4, line 3, it is necessary to consider every possible signature of a tuple. Consequently, the same tuple may be listed multiple times in *signatureMap*. Therefore, when a left injective mapping is identified (alg. 4, line 11), it is essential to remove all instances of the tuples involved. These modifications impact the algorithm in two ways. First, constructing and accessing the signature map becomes slower due to the increased number of entries. Second, the impact of the greedy choice on the quality of matches is more pronounced, as a tuple t in I can be matched with a tuple t' in I' even if they share only a single constant. We leave the development of an optimized solution for partial matching for future research.

7 EXPERIMENTAL RESULTS

We evaluate our approach around three questions: 1) what is the signature quality vs. the exact algorithm? i.e., what is the difference in terms of the computed similarity scores?; 2) can the signature algorithm scale up to higher instances? i.e., can we run the signature algorithm on instances with thousands of tuples?; and 3) is our approach useful for empirical evaluation of data curation? i.e., can we use the score to evaluate the solutions produced by various systems? All experiments are executed on a MacBook Pro with Intel i9@2.9GHz and 32GB RAM. Code and datasets are available at https://github.com/dbunibas/Instance-Comparison.

7.1 Signature VS Exact

In this section, we evaluate the score of the Signature algorithm and its execution time in comparison with the exact solution.

Ground Truth. Using the Exact algorithm, we obtain the similarity score of the two instances. We then compare such a score with the one obtained by using our Signature algorithm. This comparison, however, is feasible only for very small instances due to the computational complexity of the Exact algorithm.

For settings with bigger instances, we rely on a gold mapping between the two instances in the comparison. We programmatically modify a given table to generate source and target instances with the known tuple mappings. Starting from a table I, we clone it into a source instance I_s and a target instance I_t. By construction, the mapping for I_s and I_t is an isomorphism, i.e., a mapping from I_s to I_t s.t. their tuples are in the same position. We then introduce cell value modifications using labeled nulls or new random constants in both I_s and I_t, updating the mappings according to these changes. We add redundant and random tuples to I_s and I_t for cases with non-functional and non-injective mappings. Finally, the instances are shuffled. We obtain the instances I_s and I_t with their mappings so that we compute the exact similarity score to evaluate our Signature algorithm.

Datasets. We start with three datasets: Doctors (Doct) is a synthetic dataset with constants and nulls [30]; Bikeshare (Bike) [1] and GitHub (Git) [4] are real datasets with constants only. Statistics about the datasets are in Tab. 1. For each original dataset, we generate two scenarios with different source and target instances:

Table 1: Statistics for the original datasets.

	Doct	Bike	Git	Bus	Iris	Nba
Rows	20000	10000	10000	20000	120	9360
#Distinct val.	44600	23974	39142	29930	76	2823
Attrs	5	9	19	25	5	11

Table 2: Score results for Exact (Ex) and Signature (Sig). Noise: 5%, modCell, functional and injective (1 to 1). For each dataset, #T, #C, #V are the number of tuples, constants and nulls. * indicates score by construction.

	Source			Target			Ex	Sig		Sig	Ex
Data	#T	#C	#V	#T	#C	#V	Score	Score	Diff	T (s)	T (s)
Doct	.5k	2.4k	600	.5k	2.4k	600	.759	.759	**.000**	.1	10
Doct	1k	4.8k	1.1k	1k	4.8k	1.1k	.771	.771	**.000**	.1	40
Doct	5k	24k	6k	5k	25k	5k	.768*	.768	**.000**	.5	-
Doct	10k	49k	11k	10k	50k	10k	.775*	.775	**.000**	.9	-
Doct	100k	491k	109k	100k	520k	98k	.779*	.779	**.000**	30.7	-
Bike	.5k	4.9k	.1k	.5k	4.9k	.1k	.583	.583	**.000**	.1	101
Bike	1k	9.8k	.2k	1k	9.8k	.2k	.564	.564	**.000**	.4	569
Bike	5k	49k	1k	5k	49k	1k	.577*	.576	.001	3.6	-
Bike	10k	98k	1.9k	10k	97.5k	2.3k	.576*	.574	.002	7.2	-
Bike	100k	987k	13k	100k	976k	25k	.572*	.569	0.03	58.7	-
Git	.5k	9.7k	.2k	.5k	9.7k	.2k	.351	.349	.002	1.2	1450
Git	1k	19.7k	.2k	1k	19.4k	.5k	.333	.331	.002	3.7	2100
Git	5k	98k	2k	5k	97k	3k	.320*	.318	.002	52.9	-
Git	10k	196k	4k	10k	195k	5k	.320*	.311	.009	109.6	-
Git	100k	1.96M	33k	100k	1.95M	53.k	.315*	.314	.001	4539	-

- *modCell*: modify $C_{\%}$ cells with a null or a constant value (equal probability) in both source and target instances. Notice that the same null might have multiple occurrences;
- *addRandomAndRedundant*: run *modCell*, then generate $Rnd_{\%}$ new tuples (with random values) and duplicate $Red_{\%}$ tuples both in source and target.

Intuitively, changing cell values is used to check functional and injective mappings, while adding extra tuples is used to check non-functional and non-injective mappings.

Results. Tab. 2 reports the statistics about the source and target instances in terms of the number of tuples (#T), constants (#C), and nulls (#V). We use different tuple sizes for each dataset and start with the *modCell* scenario with $C_{\%}$=5. We measure the score of Exact (Ex) and Signature (Sig), and the execution time in seconds. When Ex exceeds a timeout of *8 hours*, we use the score computed by constructing the instances as described in the previous section. The highest score difference for Sig is 0.009. In six cases, the difference is zero. In terms of execution time, the Sig algorithm is faster up to three orders of magnitude wrt Ex.

Tab. 3 reports the same results for the *addRandomAndRedundant* scenario with $C_{\%}$=5 and 10 for both $Rnd_{\%}$ and $Red_{\%}$. The errors committed by Sig are low also in this more challenging scenario. The execution time increases for both Sig and Ex, while for Sig is still much lower.

Results confirm that Ex can be used only on small instances, while Sig scales up to thousands of tuples with a low error in the computed score. Results on Git shows that Sig is affected by the increasing size of the attributes, e.g., we observe two order of magnitude difference between Doct (5 attributes) and Git (19 attributes) on the same instance sizes. We detail how the number of attributes containing nulls affect Sig in the report [32].

Table 3: Score results for Exact (Ex) and Signature (Sig). Noise: 5%, addRandomAndRedundant, non-functional and non-injective (n to m). For each dataset, #T, #C, #V are the # of tuples, constants, nulls. * indicates score by construction.

	Source			Target			Ex	Sig		Sig	Ex
Data	#T	#C	#V	#T	#C	#V	Score	Score	Diff	T (s)	T (s)
Doct	.6k	2.7k	700	.6k	2.7k	670	.724	.721	.003	.1	15.6
Doct	1.1k	5.5k	1.3k	1.1k	5.5k	1.3k	.722	.720	.002	.2	55.3
Doct	5.6k	27.6k	6k	5.6k	28k	5k	.754*	.751	.003	2.3	-
Doct	11k	55k	12k	11k	55k	11k	.763*	.761	.002	7.0	-
Doct	110k	544k	120k	110k	556k	108k	.776*	.771	.005	18.8	-
Bike	.6k	5.6k	.3k	.6k	5.6k	.2k	.535	.535	**.000**	.5	147.5
Bike	1.1k	11k	.5k	1.1k	11k	.5k	.543	.543	**.000**	1.4	688.3
Bike	5.8k	56k	2k	5.7k	55k	2k	.549*	.549	**.000**	20.1	-
Bike	11k	111k	4k	11k	111k	4k	.544*	.543	.001	45.0	-
Bike	115k	1.12M	34k	115k	1.11M	46k	.543*	.54	.003	279	-
Git	.6k	11k	.7k	.6k	11k	.8k	.290	.290	**.000**	3.4	1870
Git	1.2k	22k	1.4k	1.2k	22k	1.4k	.317	.316	.001	8.8	8552
Git	6k	113k	6.2k	6k	111k	6.4k	.294*	.293	.001	211.0	-
Git	12k	225k	12k	12k	223k	12k	.298*	.295	.002	498.5	-
Git	117k	2.2M	97k	116k	2.2M	107k	.297*	.297	**.000**	42k	-

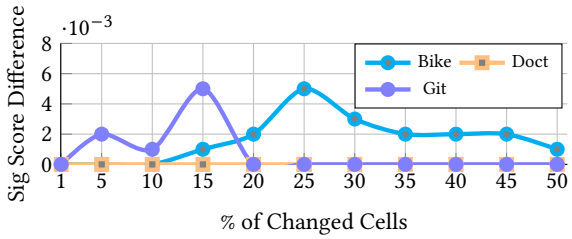

Figure 8: Impact of the $C_{\%}$ on the Signature Algorithm score difference wrt. Exact Algorithm; instances of 1k.

Table 4: Impact of CompatibleTuples in the Signature Algorithm. We report the % of the matches discovered in the Signature-Based search step (SB); the % discovered in Exact search step (Ex); the score using only Signature Based step (SB); and the overall Score (Final Score).

Dataset	% Matches SB	% Matches Ex	Score SB	Score Final
Doct 1k	98.69	1.31	.712	.720
Bike 1k	99.85	0.15	.542	.543
Git 1k	99.74	0.26	.315	.316

Impact of the % of Cell Changes. We measure the impact of the injected cell changes over the score difference between the two algorithms. We generate sources and target scenarios with different values of $C_{\%}$, then obtain the scores. In Figure 8, the highest difference for Sig is 0.005. For Doct, the error is always zero. With percentages of changes higher than 25%, Sig generate less erroneous mappings: the more we perturb the original instance, the lower the number of possible mappings.

Ablation of the Signature Algorithm. Tab. 4 reports the % of tuple mapping discovered in the two steps of Sig. Almost all the matches are discovered in the first step, i.e., Signature-Based Matches, and only a small percentage in the second, exhaustive step. This explains why Sig is much faster than Ex: most of the mappings are discovered in the first step, drastically reducing the number of tuples in the expensive check.

Table 5: Data Cleaning, comparison among F1, F1 Instance and Score computed with Signature.

Dataset	System	F1	F1 Inst.	Sig Score
Bus	Holistic [19]	0.853	0.999	0.994
Bus	HoloClean [48]	0.857	0.999	0.998
Bus	Llunatic [31]	0.997	0.999	0.999
Bus	Sampling [10]	0.406	0.998	0.964

Table 6: Data Exchange, comparison between Wrong (W) mapping, two correct user mapping (U1, U2) and the core solution (Gold).

	Solution			Gold			Miss.	Row	Sig
Scenario	#T	#C	#V	#T	#C	#V	Rows	Score	Score
Doct-W	5627	30526	0	5627	30019	543	5627	1.0	.00
Doct-U1	8959	43003	4351	5627	30019	543	0	.63	.95
Doct-U2	6827	34819	1743	5627	30019	543	0	.82	.98
Doct-W	21981	111086	0	21981	110676	410	21981	1.0	.00
Doct-U1	47281	211786	25800	21981	110676	410	0	.46	.92
Doct-U2	32781	153876	11210	21981	110676	410	0	.67	.95

7.2 Applications

In this section, we first use the Signature (Sig) algorithm for evaluating Data Cleaning and Data Exchange generated solutions. While there exist metrics in the literature, they do not consider incomplete instances. We then evaluate Sig as a tool for data versioning of incomplete instances without keys.

Data Cleaning. Cleaning systems take an input dataset and output a "clean" instance, oftentimes called solution. The standard method to qualitatively evaluate an algorithm is to compare its generated solution with a ground truth (gold) one. We report results for four systems that clean instances with constant values and variables, that we model as labeled nulls [10, 19, 31, 48]. We use the same input for all systems and evaluate their solutions with three metrics: 1) *F1*: the standard metric used in data cleaning, i.e., the f-measure calculated only on cells with errors from the gold solution; 2) *F1 Inst.*: the f-measure calculated on all the cells in generated solution wrt. the gold; 3) the *Signature score* computed with our algorithm. Metrics *F1* and *F1 Inst.* do not consider nulls: so a system introduces a null, such null is counted as an error as it differs from the constant value in the gold solution. Researchers have adopted different metrics to handle this problem, but we argue that a standard instance comparison framework will help reproducibility.

Tab. 5 reports the results. *F1* score suffers from the presence of nulls introduced by the systems, indeed Sampling has a very low *F1*, even though 99.8% of the cells are clean in the instance as reported by *F1 Inst.* Our *Sig score* represents a fair metric that considers also the nulls introduced. Indeed, *Sig Score* maintains the same ranking that can be obtained by *F1*, but considers nulls.

Data Exchange. In a Data Exchange scenario, users write source-to-target rules to integrate different sources into a target schema. A generated solution could vary depending on the used rules, the chase algorithm, and the Skolemization strategy [9]. We evaluate the generated solution using a core solution (gold). We consider three settings: wrong (W), where mapping rules are incorrect and two user-provided (U1, U2) mappings rules. As a baseline, we measure the quality of the solution by calculating a Row score as the fraction of generated solution rows/gold solution rows. We compare such a baseline with our Sig score.

Table 7: Data Versioning. Comparison with Diff tool. Comparing original dataset, shuffled (S) version, removed (R) rows, removed rows and shuffled (RS) and removed columns (C). We report the number of tuple matches (#M), left/right non matching tuples (#LNM /#RNM).

				File Diff			Signature		
Orig.	Mod.	#TO	#TM	#M	#LNM	#RNM	#M	#LNM	#RNM
Iris	Iris-S	120	120	17	103	103	120	0	0
Iris	Iris-R	120	99	99	21	0	99	21	0
Iris	Iris-RS	120	99	18	102	81	99	21	0
Iris	Iris-C	120	120	0	120	120	120	0	0
NBA	Nba-S	9360	9360	125	9235	9235	9360	0	0
NBA	Nba-R	9360	9043	9043	317	0	9043	317	0
NBA	Nba-RS	9360	9043	112	9248	8931	9043	317	0
NBA	Nba-C	9360	9360	0	9360	9360	9360	0	0

Tab. 6 reports the results on two different sizes of the Doctor dataset. We also report the number of missing rows from the gold solution. The wrong mapping refers to a different table in the source instances and produces a solution that contains constants not present in the core solution (non-universal solution).

In this scenario, our approach offers two main contributions: 1) it is the first scalable system that can be used to check homomorphism. The state of the art is a brute-force algorithm [9]. 2) it offers a more robust metric to compare solutions than measuring the row score, which fails to capture non-universal solutions.

Data Versioning. We compare two incomplete instances without key attributes. Our goal is to find differences between the two instances, such as the number of tuples that are in common and the number of tuples that differ. We use two datasets from the data versioning literature [50]: Iris and NBA. As a baseline, we adopt the command line tool DIFF to identify differences among different versions of the same dataset. While DIFF was not conceived to detect modification in terms of schema or to handle placeholders, it is the best available baseline for the problem at hand. Given an instance, we generate different versions of it: shuffling the rows (S), removing some rows (R), removing and shuffling the rows (RS), and removing some columns (C).

Tab. 7 reports comparisons between the original version (Orig.) and the modified version (Mod). We report the statistics about the size of the instances (#TO, #TM), number of matching tuples (#M), left and right non-matching tuples (#LNM, #RNM) for both DIFF and Signature. DIFF returns the same results as Signature only in the variant generated by simply removing tuples. In all other cases, DIFF fails to match tuples. This confirms that even when a dataset contains only constant values, existing tools fail to correctly evaluate its evolution in terms of new tuples added or removed, columns dropped or inserted, or a simple shuffling. Our approach can be used to compute the similarity of two versions of the same dataset while explaining the changes.

8 RELATED WORK

Comparing incomplete instances is related to computing their homomorphism, a problem with implications for applications such as query containment [17], schema mapping equivalence [24, 43], and benchmarking instances in general [9, 10, 19, 20, 23, 26, 29, 37, 39]. A homomorphism between two database instances is a mapping from the domain of one instance to the domain of the other that preserves the structures. However, finding such a homomorphism has exponential complexity in general. Our score formulation subsumes this problem, as discussed in Sec. 3.

Data versioning focuses on the creation of tools to store, retrieve, and analyze iterations of large datasets [49]. For example, DataHub offer a Git-like interface for efficient version control with a directed graph approach to manage the versions [11, 13]. Research directions focus on exploring [15] and explaining [50] the differences between dataset versions. While these tools explain versions, they rely on the availability of preexisting mappings between grounded data instances, i.e., no labeled nulls or variables are considered. Our framework compute such mappings as a side-product, thereby enabling the comparison in scenarios with incomplete instances and missing keys. Moreover, our tuple mappings can serve as explanations for the computed similarities.

Our work is related to entity resolution (ER) which matches records that refer to the same real-world entities [16, 18, 22, 42, 46]. This problem is also studied under the assumption that unique keys or identifiers are not shared across instances and the signature of a tuple, i.e. the hashing of the constants, can remind the blocking function in ER. The clean-clean ER can be modeled with an injective and functional mapping, while the non-injective and non-functional mapping generalizes the dirty ER case [46]. However, we provide a similarity score between entire instances in the presence of labeled nulls. It is a global comparison that takes into consideration incompleteness, while ER could be seen as a possible component in the overall process of instance comparison. Finally, ER can be a source of labeled nulls in the merge step [21]. If two tuples, from different instances, are aligned with ER methods, they may have conflicting constant values that are replaced with a labeled null [16].

Our work is different from approaches that compute the update distance between two databases, defined as the minimum number of insert, delete, and modification operations that transform one database into the other [45]. Tree edit distance is a widely studied metric [14], however, instances with variables should be see as graphs, rather than trees as edges across variable can be seen as edges among leaves, thus making each instance a graph [44]. Graph edit distance is more complex than tree edit distance [28].

9 CONCLUSIONS AND FUTURE WORK

In this work, we formalized the problem of comparing incomplete instances in the absence of shared keys and have shown this problem to be NP-hard. In addition to an exact algorithm, we introduced an efficient approximate instance comparison algorithm based on signatures. As we demonstrated in our extensive experimental evaluation, our approximate algorithm can compute the similarity of large instances and closely approximates the similarity computed using the exact algorithm. Our framework provides a flexible, efficient, and comprehensive addition to the existing data versioning ecosystem, with its capacity to calculate similarity scores and mappings between incomplete instances. Extending the algorithm to support string similarity metrics to provide a more fine-grained view on the similarity of different constants is an interesting avenue for future work. Finally, there might be scenarios where it is desirable to match tuples that conflict on constants. Our definition is general enough to capture such a setting, as discussed in Section 6.3. However, developing a scalable algorithm for partial tuple matches is an open problem.

ACKNOWLEDGMENTS

Glavic and Miller were supported in part by NSF award number IIS-2107248; Miller by IIS-1956096 and IIS-2325632. Papotti was supported in part by the projects ANR ATTENTION (ANR-21-CE23-0037) and CHIST-ERA CIMPLE (CHIST-ERA-19-XAI-003).

REFERENCES

[1] Bikeshare dataset. online https://s3.amazonaws.com/capitalbikeshare-data/, 2023.

[2] Datahub. online https://datahub.io/, 2023.

[3] Dolt. online https://github.com/dolthub/dolt, 2023.

[4] Github dataset. online https://cloud.google.com/bigquery/public-data, 2023.

[5] B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. *PVLDB*, 1(2):1468–1471, 2008.

[6] B. Alexe, W. Tan, and Y. Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *PVLDB*, 1(1):230–244, 2008.

[7] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[8] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing-Up with BART: Error Generation for Evaluating Data Cleaning Algorithms. *PVLDB*, 9(2):36–47, 2015.

[9] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, pages 37–52. ACM, 2017.

[10] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *PVLDB*, 3(1):197–207, 2010.

[11] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[12] A. P. Bhardwaj, A. Deshpande, A. J. Elmore, D. R. Karger, S. Madden, A. G. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with datahub. *PVLDB*, 8(12):1916–1919, 2015.

[13] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, 2015.

[14] P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005.

[15] T. Bleifuß, L. Bornemann, D. V. Kalashnikov, F. Naumann, and D. Srivastava. Dbchex: Interactive exploration of data and schema change. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[16] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1335–1349, New York, NY, USA, 2020. Association for Computing Machinery.

[17] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[18] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. An overview of end-to-end entity resolution for big data. *ACM Comput. Surv.*, 53(6):127:1–127:42, 2021.

[19] X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, pages 458–469, 2013.

[20] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a Commodity Data Cleaning System. In *SIGMOD*, pages 541–552, 2013.

[21] D. Deng, W. Tao, Z. Abedjan, A. K. Elmagarmid, I. F. Ilyas, G. Li, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Unsupervised string transformation learning for entity consolidation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 196–207. IEEE, 2019.

[22] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate Record Detection: A Survey. *IEEE TKDE*, 19(1):1–16, 2007.

[23] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[24] R. Fagin, P. Kolaitis, A. Nash, and L. Popa. Towards a Theory of Schema-Mapping Optimization. In *PODS*, pages 33–42, 2008.

[25] R. Fagin, P. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *ACM TODS*, 30(1):174–210, 2005.

[26] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool, 2012.

[27] R. C. Fernandez, P. Subramaniam, and M. J. Franklin. Data market platforms: Trading data assets to solve data problems. *arXiv preprint arXiv:2002.01047*, 2020.

[28] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13:113–129, 2010.

[29] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The Llunatic Data-Cleaning Framework. *PVLDB*, 6(9):625–636, 2013.

[30] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, pages 232–243, 2014.

[31] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Cleaning data with llunatic. *VLDB J.*, 29(4):867–892, 2020.

[32] B. Glavic, G. Mecca, R. J. Miller, P. Papotti, D. Santoro, and E. Veltri. A Fast Algorithm to Compare Database Instances with Variables. Technical Report TR 1-2023 - https://github.com/dbunibas/Instance-Comparison/blob/main/instanceTR.pdf, 2023.

[33] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *SIGMOD*, pages 795–806, 2016.

[34] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning: A tossed stone raises a thousand ripples. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26-June-2016, page 893 – 907, New York, NY, USA, 2016. Association for Computing Machinery.

[35] T. Imieliński and W. Lipski. Incomplete Information in Relational Databases. *J. of the ACM*, 31(4):761–791, 1984.

[36] E. Kandogan, M. Roth, P. M. Schwarz, J. Hui, I. Terrizzano, C. Christodoulakis, and R. J. Miller. Labbook: Metadata-driven social collaborative data analysis. In *IEEE Big Data*, pages 431–440, 2015.

[37] A. Khatiwada, R. Shraga, W. Gatterbauer, and R. J. Miller. Integrating data lake tables. *Proc. VLDB Endow.*, 16(4):932–945, 2022.

[38] M. Koch, M. Esmailoghli, S. Auer, and Z. Abedjan. Duplicate table discovery with xash. In B. König-Ries, S. Scherzinger, W. Lehner, and G. Vossen, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings*, volume P-331 of *LNI*, pages 367–390. Gesellschaft für Informatik e.V., 2023.

[39] S. Kolahi and L. V. S. Lakshmanan. On Approximating Optimum Repairs for Functional Dependency Violations. In *ICDT*, 2009.

[40] P. Lapadula, G. Mecca, D. Santoro, L. Solimando, and E. Veltri. Humanity is overrated. or not. automatic diagnostic suggestions by greg, ml (extended abstract). *Communications in Computer and Information Science*, 909:305 – 313, 2018.

[41] P. Lapadula, G. Mecca, D. Santoro, L. Solimando, and E. Veltri. Greg, ml – machine learning for healthcare at a scale. *Health and Technology*, 10(6):1485 – 1495, 2020.

[42] Y. Li, J. Li, Y. Suhara, A. Doan, and W.-C. Tan. Deep entity matching with pre-trained language models. *Proc. VLDB Endow.*, 14(1):50–60, sep 2020.

[43] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings. In *SIGMOD*, pages 655–668, 2009.

[44] G. Mecca, P. Papotti, S. Raunich, and D. Santoro. What is the IQ of your Data Transformation System? In *CIKM*, pages 872–881, 2012.

[45] H. Müller, J.-C. Freytag, and U. Leser. Describing differences between databases. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, page 612–621, New York, NY, USA, 2006. Association for Computing Machinery.

[46] G. Papadakis, E. Ioannou, E. Thanos, and T. Palpanas. The four generations of entity resolution. *Synthesis Lectures on Data Management*, 16(2):1–170, 2021.

[47] A. A. Qahtan, A. K. Elmagarmid, R. C. Fernandez, M. Ouzzani, and N. Tang. FAHES: A robust disguised missing values detector. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 2100–2109. ACM, 2018.

[48] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, 2017.

[49] M. E. Schüle, J. Schmeißer, T. Blum, A. Kemper, and T. Neumann. Tardisdb: Extending sql to support versioning. In *Proceedings of the 2021 International Conference on Management of Data*, page 2775–2778. Association for Computing Machinery, 2021.

[50] R. Shraga and R. J. Miller. Explaining dataset changes for semantic data versioning with explain-da-v. *Proc. VLDB Endow.*, 16(6):1587–1600, apr 2023.

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