Comparing Incomplete Database Instances

(Discussion Paper)

Boris Glavic[1], Giansalvatore Mecca[2], Renée J. Miller[3], Paolo Papotti[4], Donatello Santoro[2] and Enzo Veltri[2,*]

[1]Illinois Inst. of Technology, Chigaco, IL, USA
[2]Università degli Studi della Basilicata (UNIBAS), Potenza, Italy
[3]Northeastern University, Boston, MA, USA
[4]EURECOM, Biot, France

Abstract

Comparing incomplete database instances is crucial in various applications, including dataset evolution, evaluating data cleaning solutions, and comparing instances from data exchange systems. We present a framework designed to compute similarity among instances containing labeled null values, even in the absence of primary keys. A notable outcome of our approach is the generation of a mapping between tuples in the instances, which explains the similarity score. Computing the similarity of two incomplete instances is NP-hard in the instance size. To compare instances of realistic size we present an approximate PTIME algorithm that approximates the exact score with an error always smaller than 1% but it significantly speedup the computation up to three orders of magnitude w.r.t. the exact algorithm.

Keywords

Labelled Nulls, Incomplete Databases, Instance Similarity

1. Introduction

Organizations use "data lakes" for storing their data in schema-on-read storage systems [1]. The reliance on data lakes is driving new techniques for organizing datasets [2, 3]. In this setting, an important task is to compare datasets. Comparing instances could have multiple uses. First, finding datasets that are similar to an already known dataset (e.g., find more census data or medical records [4, 5]), 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, evaluate instances produced by different data exchange and constraint-based data repair algorithms. 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, comparison of incomplete datasets is difficult: i) in general, we cannot rely on metadata – such as keys – to determine the correspondence between the tuples of two incomplete

SEBD 2024: 32nd Symposium on Advanced Database Systems, June 23-26, 2024, Villasimius, Sardinia, Italy
*Corresponding author.
bglavic@uic.edu (B. Glavic); giansalvatore.mecca@unibas.it (G. Mecca); miller@northeastern.edu (R. J. Miller); papotti@eurecom.fr (P. Papotti); donatello.santoro@unibas.it (D. Santoro); enzo.veltri@unibas.it (E. Veltri)
0000-0003-2887-2452 (B. Glavic); 0000-0002-1189-1481 (G. Mecca); 0000-0003-0651-4128 (P. Papotti); 0000-0002-5651-8584 (D. Santoro); 0000-0001-9947-8909 (E. Veltri)

Conference $\mathbf{I}$

	Name	Year	Place	Org
t_{01}	VLDB	1975	Framingham	VLDB End.
t_{02}	VLDB	1976	$Null$	$Null$
t_{03}	SIGMOD	1975	San Jose	ACM

(a)

Conference $\mathbf{I_1}$

	Name	Year	Place	Org
t_{07}	SIGMOD	1975	San Jose	ACM
t_{08}	VLDB	$Null$	Framingham	VLDB End.
t_{09}	$Null$	1976	Brussels	IEEE
t_{10}	VLDB	$Null$	$Null$	VLDB End.

(b)

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.

(c)

Conference $\mathbf{I_3}$

	Name	Year	Place	Org
t_{21}	VLDB	1975	Framingham	N_1
t_{22}	VLDB	1976	Brussels	N_1
t_{22}	N_2	1975	San Jose	ACM

(d)

Figure 1: Three versions of instance I (a,b,c) where missing values are denoted with $Null$. In (d) the version of I obtained with data cleaning, containing labeled nulls (N_1).

instances (key values may be missing); and $ii)$ many datasets are inherently incomplete due to unknown values encoded as nulls in the dataset creation 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 [6], a constraint-repair algorithm may have replaced conflicting values with labeled nulls [7], or outliers may have been replaced with nulls.

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 [8] and Dolt [9]. Such systems provide version management features for datasets. However, they do not support comparing versions of incomplete datasets.

Consider the relational schema T describing database conferences: Conference(Name, Year, Place, Org). Figure $1(a)$ 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. Figures $1(b, c)$ 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_{09} and t_{16}), two *Null* values in I (t_{02}) has 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_{01} or t_{03}; both t_{09} and t_{10} can be mapped to t_{02}.

Empirical Evaluation of Data Cleaning and Integration. Empirical evaluation is important in data integration and data cleaning [10]. ST-Benchmark [11], IQ-Meter [12] and iBench [13] are examples of frameworks for data-exchange evaluation, while BART [14] is an error-generation tool for data repair. In data cleaning and integration systems differ not just in their runtime efficiency but also in terms of the quality of the produced results. 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 use *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 [15]. In constraint-based data repair, labeled nulls are used by systems to mark conflicts among values that require user intervention [16, 7, 17, 18, 19, 20, 21].

Labeled nulls encode incompleteness [22] and turn the instances we need to compare into *representation systems* of incomplete databases. For example, in instance I_3 in Figure $1(d)$, 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 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. All these settings share a common problem, which is the one of *comparing incomplete instances without keys*, or *instance-comparison problem* [23] for short. This problem is challenging for two reasons. First, we demonstrate that the instance comparison problem is NP-hard [23]. 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 [23].

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 [24, 25].

We formalize in the next section the instance-comparison problem and present some experiments to show the proposed solution's scalability and accuracy. A detailed evaluation can be found in the full paper [23].

2. The Instance Comparison Problem

Instances with Labeled Nulls. 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 labeled nulls or nulls for short.

An *instance* $I = (I_1, \ldots, I_k)$ of R consists of finite relations $I_i \subset (\mathsf{Consts} \cup \mathsf{Vars})^{n_i}$, for $i \in [1, k]$. We denote by $\mathsf{Consts}(I)$ and $\mathsf{Vars}(I)$ the set of constants and nulls in I, respectively. The *active domain* of I is $\mathsf{adom}(I) = \mathsf{Consts}(I) \cup \mathsf{Vars}(I)$.

We assume the presence of *unique tuple identifiers* in an instance; by t_{id} we denote the tuple with identifier "*id*" in I.

A *cell* is a location in I specified by a tuple id/attribute pair $t_{id}.A_i$. We denote by $\mathsf{ids}(I)$ the set of tuple ids of instance I. When comparing two instances I and I', we will assume that

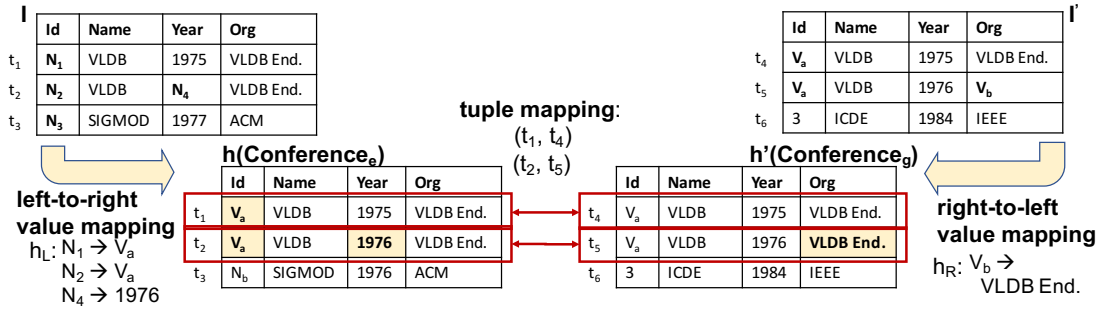

Figure 2: A Sample Instance Match.

$\text{ids}(I) \cap \text{ids}(I') = \emptyset$.

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'.

Figure 1(d) shows an instance I_3 that contains constants from Consts (for example, "1975" or "San Jose") and contains nulls from Vars (N_1, N_2). This might be the result of repairing an instance of the database that is dirty wrt. the functional dependency (FD): Conference : Name $\rightarrow$ 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 (N_1) to mark the conflict so that a human expert solves it using domain knowledge [26]. The same applies in data-exchange where the instance might be the result of mapping a source database into the target schema T. Some of the mappings leave unspecified values of some attributes introducing labeled nulls as placeholders for human experts.

The Instance-Comparison Problem. The instance-comparison problem takes as input two instances I and I' and outputs the similarity of the two instances, i.e. a value between 0 and 1, where 0 indicates a total dissimilarity and 1 indicates a total similarity. Intuitively, to compare two instances we need: 1) a way to map tuples from I and I' (and vice-versa), i.e. we need to find tuples in I that *match* to tuples I'. We call this step *instance match*; and 2) we need to compute a *score* that takes into account tuples that match but also tuples that do not match.

Fig. 2 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. Among, all the possible mappings, we are interested in finding the best mappings, i.e. the ones that maximize the matches and thus the similarity.

The similarity $\text{similarity}(I, I')$ of I and I' is defined as: $\text{similarity}(I, I') = \max_{M \in \mathcal{M}}(\text{score}(M))$, where $\text{score}(M)$ takes into account the best mappings. The *instance-comparison problem* takes as input instances I and I' and outputs $\text{similarity}(I, I')$.

Instance Match. To match tuples from two instances I and I' we first should define how to match cells among the tuples. 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 $c \in \text{Consts}(I)$, i.e., is a mapping that preserves

constants. 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. We do not allow a constant to be mapped to a different constant. For instance, t_{16} in Fig. 1 is not mapped to any tuple in instance I.

Given two instances I and I' for the same schema R, a *tuple mappings* m is a subset of $I \times I'$. This design choice permits to consider not only functional, total mappings – like homomorphisms – but also non-functional mappings [23]. It is clear, that given I and I' there could be multiple tuple mappings, i.e. multiple combinations to match tuples from I to tuples in I'. Consider Fig. 2, depending on the tuple mappings configuration (functional, non-functional), tuple t_2 in I could be mapped only to t_4, or only to t_5, or could be mapped to both t_4 and t_5 in I' (in total three possible tuple mappings for t_2).

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' since it is clear that there could be multiple instance matches depending on the tuple mapping configuration.

Match Score. 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. We first define how to score cells among tuples in match, then we define how to score the two instances.

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 should measure 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 $v \in \mathsf{Vars}(I)$, and $v \in \mathsf{Vars}(I')$ as shown in Eq.1. Then, the score for the same attribute A of two tuples $t \in I$ and $t' \in I'$ that are in match is defined as shown in Eq. 2, where we assume a parameter $0 \leq \lambda < 1$, which defines the penalty for mapping a variable to a constant, given as input. Now the score of the two tuples $t \in I$ and $t' \in I'$ in matches is the sum of the scores of each attribute (Eq. 3).

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 ($\mathsf{score}(M, t)$). 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. (Eq. 4). 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

number of cells in the instance,i.e. $size(I)$ (Eq. 5).

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

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

$$\mathsf{score}(M, t, t') = \sum_{A \in R} \mathsf{score}(M, t, t', A) \tag{3}$$

$$\mathsf{score}(M, t) = \frac{\sum_{t_m \in m(t)} \mathsf{score}(M, t, t_m)}{size(m(t))} \tag{4}$$

$$\mathsf{score}(M) = \frac{\sum_{t \in I} \mathsf{score}(M, t) + \sum_{t' \in I'} \mathsf{score}(M, t')}{size(I) + size(I')} \tag{5}$$

Exact and Signature Algorithm. To calculate the $\mathsf{similarity}(I, I')$ of two instances I and I' we need to discover $\mathcal{M}$ (the set of all complete instance matches for I and I') and for each tuple mappings $m \in \mathcal{M}$ we need to compute the score and return the m that has the highest score.

The *exact-algorithm* works in two steps. First, it builds 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.

The *Signature Algorithm* is 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 than 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. 2: $t_5 : \langle V_b, \textit{VLDB}, \textit{1976}, V_c \rangle$. One signature of t_5 is: *[Name: VLDB, Year: 1975]*. We use a greedy algorithm: 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.

Given a tuple t over schema R, we associate with it a number of *signatures*, i.e. all the possible signatures that could be generated for t. Our search for compatible tuples relies on signatures. We construct all maximal signatures (i.e. the ones with the highest number of constants) 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. This allow us to find

Table 1
Score results for Exact (Ex) and Signature (Sig).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	-

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, so we execute the same step starting from I' and scanning tuples in I.

We have derived an instance match M that contains signature-based matches, but these do not cover all possible tuple matches. Consider tuples $t_2 = \langle N_2, \textit{VLDB}, N_4, \textit{VLDB End.}\rangle$ and $t_5 = \langle V_b, \textit{VLDB}, \textit{1976}, V_c\rangle$ in Fig. 2. Despite the two tuples are compatible (they are matched in Fig. 2), 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 idea of the *exact-algorithm* in discovering compatible tuples. 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. 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 pseudocode of both algorithms is presented in the full paper [23].

3. Experimental Results and Conclusions

We evaluate our approach around two 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?

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 obtained programmatically by introducing changes (constants to

Table 2
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 Sig.-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

nulls, nulls to constants, contants to different constants, nulls to different nulls) in the instances a keep track of the changes. (see the full paper for more details [23]). Notice that the gold mapping could be used to compute the exact score of the two instances. We use three datasets: Doctors (Doct) is a synthetic dataset with constants and nulls [27]; Bikeshare (Bike) [28] and GitHub (Git) [29] are real datasets with constants. For each original dataset we generate different scenarios of different sizes and changes.

Table 1 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. 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. The highest score difference for Sig algorithm is 0.005. In five cases the difference is zero. In terms of execution time, the Sig. algorithms is faster up to three orders of magnitude wrt Ex. algorithm.

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, and also the number of attributes containing nulls affect it.

Table 2 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.

An extensive evaluation of our framework can be found in the full paper [23].

Conclusions. We presented the problem of comparing incomplete instances in the absence of shared keys. In addition to an exact algorithm, we presented an efficient approximate instance comparison algorithm based on signatures. We demonstrated in our experimental evaluation, the 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.

References

[1] E. Kandogan, M. Roth, P. M. Schwarz, J. Hui, I. Terrizzano, C. Christodoulakis, R. J. Miller, Labbook: Metadata-driven social collaborative data analysis, in: IEEE Big Data, 2015, pp.

431–440.
[2] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, S. E. Whang, Goods: Organizing google's datasets, in: SIGMOD, 2016, pp. 795–806.
[3] Y. Zhang, 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, 2020, pp. 1951–1966.
[4] P. Lapadula, G. Mecca, D. Santoro, L. Solimando, E. Veltri, Humanity is overrated. or not. automatic diagnostic suggestions by greg, ml (extended abstract), Communications in Computer and Information Science 909 (2018) 305 – 313. doi:10.1007/978-3-030-00063-9_29.
[5] P. Lapadula, G. Mecca, D. Santoro, L. Solimando, E. Veltri, Greg, ml – machine learning for healthcare at a scale, Health and Technology 10 (2020) 1485 – 1495. doi:10.1007/s12553-020-00468-9.
[6] A. A. Qahtan, A. K. Elmagarmid, R. C. Fernandez, M. Ouzzani, 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, ACM, 2018, pp. 2100–2109. URL: https://doi.org/10.1145/3219819.3220109. doi:10.1145/3219819.3220109.
[7] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, N. Tang, NADEEF: a Commodity Data Cleaning System, in: SIGMOD, 2013, pp. 541–552.
[8] A. P. Bhardwaj, A. Deshpande, A. J. Elmore, D. R. Karger, S. Madden, A. G. Parameswaran, H. Subramanyam, E. Wu, R. Zhang, Collaborative data analytics with datahub, PVLDB 8 (2015) 1916–1919.
[9] Dolt, online https://github.com/dolthub/dolt, 2023. URL: https://github.com/dolthub/dolt.
[10] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, E. Tsamoura, Benchmarking the chase, in: PODS, ACM, 2017, pp. 37–52. URL: https://doi.org/10.1145/3034786.3034796. doi:10.1145/3034786.3034796.
[11] B. Alexe, W. Tan, Y. Velegrakis, Comparing and Evaluating Mapping Systems with STBenchmark, PVLDB 1 (2008) 1468–1471.
[12] G. Mecca, P. Papotti, S. Raunich, D. Santoro, What is the IQ of your Data Transformation System?, in: CIKM, 2012, pp. 872–881.
[13] P. C. Arocena, B. Glavic, R. Ciucanu, R. J. Miller, The ibench integration metadata generator, PVLDB 9 (2015) 108–119. URL: http://www.vldb.org/pvldb/vol9/p108-arocena.pdf.
[14] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, D. Santoro, Messing-Up with BART: Error Generation for Evaluating Data Cleaning Algorithms, PVLDB 9 (2015) 36–47.
[15] R. Fagin, P. Kolaitis, R. Miller, L. Popa, Data Exchange: Semantics and Query Answering, TCS 336 (2005) 89–124.
[16] W. Fan, F. Geerts, Foundations of Data Quality Management, Morgan & Claypool, 2012.
[17] G. Beskales, I. F. Ilyas, L. Golab, Sampling the Repairs of Functional Dependency Violations under Hard Constraints, PVLDB 3 (2010) 197–207.
[18] F. Geerts, G. Mecca, P. Papotti, D. Santoro, The Llunatic Data-Cleaning Framework, PVLDB 6 (2013) 625–636.
[19] S. Kolahi, L. V. S. Lakshmanan, On Approximating Optimum Repairs for Functional Dependency Violations, in: ICDT, 2009.

[20] X. Chu, I. F. Ilyas, P. Papotti, Holistic Data Cleaning: Putting Violations into Context, in: ICDE, 2013, pp. 458–469.
[21] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, 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, Association for Computing Machinery, New York, NY, USA, 2016, p. 893 – 907. doi:10.1145/2882903.2915242.
[22] T. Imieliński, W. Lipski, Incomplete Information in Relational Databases, J. of the ACM 31 (1984) 761–791.
[23] B. Glavic, G. Mecca, R. J. Miller, P. Papotti, D. Santoro, E. Veltri, Similarity measures for incomplete database instances, in: Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28, OpenProceedings.org, 2024.
[24] R. Shraga, R. J. Miller, Explaining dataset changes for semantic data versioning with explain-da-v, Proc. VLDB Endow. 16 (2023) 1587–1600. URL: https://doi.org/10.14778/3583140.3583169. doi:10.14778/3583140.3583169.
[25] T. Bleifuß, L. Bornemann, D. V. Kalashnikov, F. Naumann, 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. URL: http://cidrdb.org/cidr2019/papers/p65-bleifuss-cidr19.pdf.
[26] F. Geerts, G. Mecca, P. Papotti, D. Santoro, Cleaning data with llunatic, VLDB J. 29 (2020) 867–892. URL: https://doi.org/10.1007/s00778-019-00586-5. doi:10.1007/s00778-019-00586-5.
[27] F. Geerts, G. Mecca, P. Papotti, D. Santoro, Mapping and Cleaning, in: ICDE, 2014, pp. 232–243.
[28] Bikeshare dataset, online https://s3.amazonaws.com/capitalbikeshare-data/, 2023. URL: https://s3.amazonaws.com/capitalbikeshare-data/.
[29] Github dataset, online https://cloud.google.com/bigquery/public-data, 2023. URL: https://cloud.google.com/bigquery/public-data.