MapRepair: Mapping and Repairing under Policy Views

Angela Bonifati, Ugo Comignani, Efthymia Tsamoura

MapRepair: Mapping and Repairing under Policy Views

Angela Bonifati
Lyon 1 University & Liris CNRS
angela.bonifati@univ-lyon1.fr

Ugo Comignani
Lyon 1 University & Liris CNRS
ugo.comignani@univ-lyon1.fr

Efthymia Tsamoura
University of Oxford
efthymia.tsamoura@cs.ox.ac.uk

ABSTRACT

Mapping design is overwhelming for end users, who have to check at par the correctness of the mappings and the possible information disclosure over the exported source instance. In this demonstration, we focus on the latter problem by proposing a novel practical solution to ensure that a mapping faithfully complies with a set of privacy restrictions specified as source policy views. We showcase MapRepair, that guides the user through the tasks of visualizing the results of the data exchange process with and without the privacy restrictions. MapRepair leverages formal privacy guarantees and is inherently *data-independent*, i.e. if a set of criteria are satisfied by the mapping statement, then it guarantees that both the mapping and the underlying instances do not leak sensitive information. Furthermore, MapRepair also allows to automatically *repair* an input mapping w.r.t. a set of policy views in case of information leakage. We build on various demonstration scenarios, including synthetic and real-world instances and mappings.

CCS CONCEPTS

• **Information systems → Data exchange**.

KEYWORDS

privacy-preserving data exchange; mappings; repairs

1 INTRODUCTION

We consider the problem of exchanging data between a source schema S and a target schema T via a set of *source-to-target* (s-t) dependencies Σ_{st} expressed as *tuple-generating dependencies* (tgds) [5].

Our work considers a privacy-conscious variant of the data exchange problem, in which the source schema comes with a set of privacy constraints (under the form of policy views), representing the data that is *safe* to expose to the target schema over *all instances* of the source. As a worst-case scenario, when wanting to protect the information of the source instance we assume that all users, both the malicious and the non-malicious ones, might know the source and the target schemas, the target instances along with the s-t tgds within the mapping.

Inspired by prior theoretical work on privacy preservation [2, 7], we define a set of s-t tgds to be safe w.r.t. the policy views if every information, both in term of values and joins between attributes, that is kept secret by the source policy views is also kept secret by the s-t tgds during the data exchange process.

Contributions. We demonstrate MapRepair, a system allowing a user to : (i) visualize the information leakage of policy views defined on the source schema of her data, (ii) visualize how a mapping violates the privacy rules defined by the policy views over its source schema, and (iii) automatically repair a mapping that does not respect the policy views over its source schema.

Both the privacy compliance and the repairing algorithms implemented by MapRepair are *data-independent*. Instead, these tasks only need to be performed over the schemas, allowing MapRepair to provide mapping with strong privacy preservation guarantees over all instances of the sources.

During our demonstration, we will showcase MapRepair both on real-life mapping scenarios borrowed from a real hospital in the UK allowing to better experience the way the algorithms behind MapRepair work, and on complex synthetically generated scenarios showing the performance of MapRepair on such cases.

To the best of our knowledge, MapRepair is the first system to show the underpinnings of privacy-preserving data exchange, while classical data exchange [4, 5] focused on the privacy-unaware case.

Modifications of the underlying source and target instances by leveraging probabilistic approaches and anonymization techniques [6, 8] are orthogonal to our approach, which is inherently data-independent. The source code along with the experimental scenarios are publicly available .

Paper layout. We present an overview of MapRepair in Section 2 and the details of our demonstration in Section 3.

2 SYSTEM OVERVIEW

Our system can be used either to *visualize the information leakage* occurring when exchanging data from source to target across the mappings or to *automatically repair* the mappings in order to make them compliant with a given set of source policy views.

In this section, we illustrate the key concepts of our approach (fully detailed in [3]), then we describe the implementation of our system and give an illustration of its scalability.

Main algorithmic concepts. We illustrate the main algorithmic concepts underlying our system by using the running example shown in Figure 1. Precisely, Figure 1(i) and (ii) show the source and target schemas of a real-life mapping scenario describing two hospitals in the UK, while Figure 1(iii) shows the set of policy views $\mathcal{V}$

(i) **Source schema S :**
$Oncology(idInsurance, treatment, evolution)$
$Patient(idInsurance, name, ethnicity, country)$
$Hospital(idInsurance, pathology)$
$Student(idInsurance, name, ethnicity, country)$

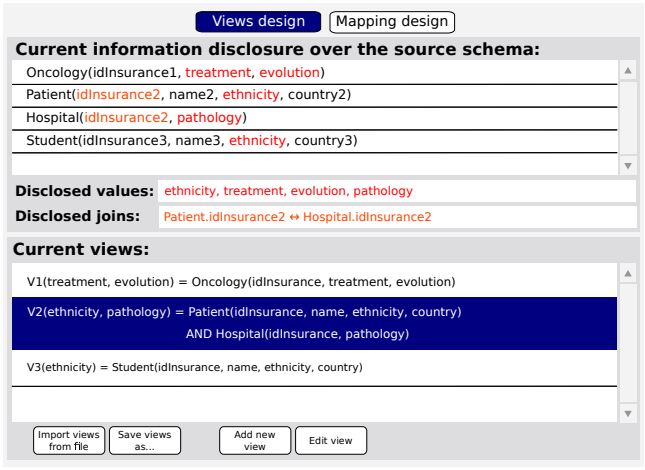
(ii) **Target schema T :**
$SO(ethnicity)$
$EthDis(ethnicity, pathology)$
$CountryDis(country, pathology)$

(iii) **Policy views $\mathcal{V}$ over S :**
$V_1(treatment, evolution) = Oncology(idInsurance, treatment, evolution)$
$V_2(ethnicity, pathology) = Patient(idInsurance, name, ethnicity, country) \wedge Hospital(idInsurance, pathology)$
$V_3(ethnicity) = Student(idInsurance, name, ethnicity, country)$

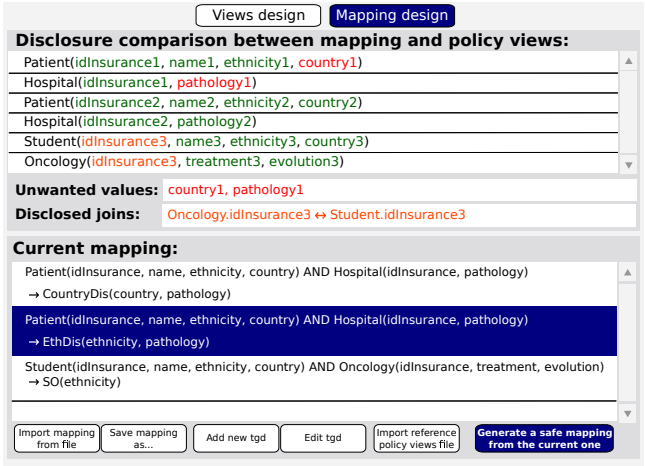
(iv) **Mapping s-t tgds Σ from S to T :**
$Student(idInsurance, name, ethnicity, country) \wedge Oncology(idInsurance, treatment, evolution) \rightarrow SO(ethnicity)$
$Patient(idInsurance, name, ethnicity, country) \wedge Hospital(idInsurance, pathology) \rightarrow EthDis(ethnicity, pathology)$
$Patient(idInsurance, name, ethnicity, country) \wedge Hospital(idInsurance, pathology) \rightarrow CountryDis(country, pathology)$

(v) **A repaired mapping s-t tgds Σ' for Σ w.r.t. $\mathcal{V}$:**
$Patient(idInsurance, name, ethnicity, country) \wedge Hospital(idInsurance, pathology) \rightarrow \exists n, EthDis(n, pathology)$
$Patient(idInsurance, name, ethnicity, country) \wedge Hospital(idInsurance, pathology) \rightarrow CountryDis(country, pathology)$

Figure 1: Running example : policy views and mapping over a hospital schema.



(a) Views design using information disclosure.



(b) Automatic repair of a mapping w.r.t. policy views.

Figure 2: Workflow of MapRepair.

associated with the source schema. Figure 1(iv) shows the schema mapping Σ for which we would like to detect information disclosure, whereas Figure 1(v) shows a possible repair for Σ w.r.t. $\mathcal{V}$ as output by our system `MapRepair`.

In the following, we are now going to introduce the basic ingredients of our approach by using the running example as a basis for the presentation.

• **Critical instance and visible chase.** In order to detect information disclosure of a set of s-t tgds, we rely on the visible chase algorithm over the critical instance of the source schema, as defined in previous work [2] for the simple case of boolean queries. Since we handle non-boolean conjunctive queries in our mappings, we extended this notion accordingly. The *critical instance* of a schema S is an instance such that, for each tuple over S, its domain contains exactly one constant. For example, borrowing the source schema from Figure 1(i), and assuming we choose the constant $*$, the corresponding critical instance Crt_S for S is as follows :

$$Student(*,*,*,*) \qquad Oncology(*,*,*)$$
$$Patient(*,*,*,*) \qquad Hospital(*,*)$$

Running the *visible chase* algorithm over such an instance and a set of dependencies Σ will work in two steps. The first step of the visible chase aims at detecting information leakage over each dependency taken separately, i.e., interactions between dependencies are not considered at this step. For this first step, the chase is run over Σ then the result is chased over Σ^{-1}, allowing to see the disclosed information for each tgd. As an example, running this step over the mapping Σ in Figure 1(iv) will first give an instance :

$$I_S = \text{chase}(\Sigma, \text{Crt}_S) = \{V_1(*,*); V_2(*,*); V_3(*)\}$$

then the chase of I_S over Σ^{-1} will give an instance I_0 with the following tuples :

$$Student(n_i, n_n, *, n_c) \qquad Oncology(n_i, n_t, n_e)$$
$$Patient(n'_i, n'_n, *, n'_c) \qquad Hospital(n'_i, *)$$
$$Patient(n''_i, n''_n, n''_e, *) \qquad Hospital(n''_i, *)$$

with the variables n being labeled nulls. From this first step, we can see which values are disclosed by our mapping by looking at the $*$ values. When values are not disclosed, we can still see which joins are disclosed by looking at the labeled nulls such as n_i showing that, in our mapping Σ, students tuples are linked to oncology appointments tuples.

The next visible chase step aims at finding which information can be deduced from the interactions between the dependencies. To this end, the visible chase algorithm checks whether a homomorphism from the dependencies bodies into I_0 allowing to unify labeled nulls to the value $*$ exists. In our example, this will lead to unify n'_c and n''_e to $*$, thus obtaining the following tuples as output of the visible chase :

$$Student(n_i, n_n, *, n_c) \qquad Oncology(n_i, n_t, n_e)$$
$$Patient(n'_i, n'_n, *, *) \qquad Hospital(n'_i, *)$$

• **Repairing of mappings.** Based on the visible chase algorithm, it can be shown that a mapping Σ preserves the privacy of a set of source policy views $\mathcal{V}$ if, and only if, there exists a homomorphism from the visible chase results over Σ into the visible chase results over $\mathcal{V}$.

Thus, the repairing algorithm implemented in MapRepair allows to ensure that the output mapping respects such a constraint. It also aims at maximizing the disclosure of non-sensitive information at the same time in order to let the data exchange process seamlessly occur. Our algorithm works in two fundamental steps. The first step (F-repair) will ensure that each tgd of the repaired mapping Σ is safe w.r.t. the set of policy views $\mathcal{V}$, leading to a so-called partially safe mapping. This algorithm will produce repairs by breaking joins in the bodies or hiding exported target variables, and choose the best repair based on a preference function. The second step (S-repair) aims at ensuring that the repaired mapping is safe, despite the possible interactions between its tgds. This is done by focusing on homomorphisms similar to the second step of the visible chase algorithm described above. This step tracks which tgds lead to break the safety and corrects them by either hiding exported variables or breaking joins in their bodies. As in the first step, the output mapping is chosen among multiple rewritings by using a preference function.

On our running example, the repaired mapping w.r.t. $\mathcal{V}$ will be the mapping shown in Figure 1(v).

Data exchange privacy-aware workflow. We present the workflow of our system as the transition between four steps. A high-level architecture of MapRepair is provided in Figure 3 to illustrate the communication involved in our workflow.

- **Initial state : waiting to load policy views.** At first, MapRepair waits for the user to load a set of policy views, by either using an input file or by entering the views manually through the graphical user interface. The user might want to add new policy views at this step by using the corresponding 'Add New View' button as shown in Figure 2a. Next, the user can choose to visualize the results of the visible chase over the set of policy views as shown in Figure 2a, and interactively update them to meet our desired privacy restrictions. In this figure, we show how our system will represent information disclosure of the view $\mathcal{V}$ from our running example, by representing the disclosed values highlighted in red and the disclosed joins over labeled nulls highlighted in orange, respectively. The underlying chase engine of MapRepair allows us to obtain in real time this information about the color-coded items illustrated in the interface.
- **Waiting to load mapping to repair.** Once the policy views have been validated by the user, the system is kept on hold for a mapping as input. Like for the policy views, it can be provided either by using an input file or by entering the mappings by hand through the graphical interface using the corresponding 'Add New Tgd' button as shown in Figure 2b. As shown in this figure, the GUI also allows a comparison of information leakage between the mapping Σ and the set of policy views $\mathcal{V}$ (shown for our running example in the screenshot). The interface will highlight in red a value that is disclosed by the mapping whereas it should be hidden according to the policy views. Analogously, if the mapping discloses a join over labeled nulls and that should be hidden according to the policy views, then this join will be color-coded in orange. Finally, if an information disclosed by the mapping is allowed according to the policy views, then this value will be in green. At this point, the user can update manually the tgds in her mapping and visualize the effect of her modification on the disclosed information.

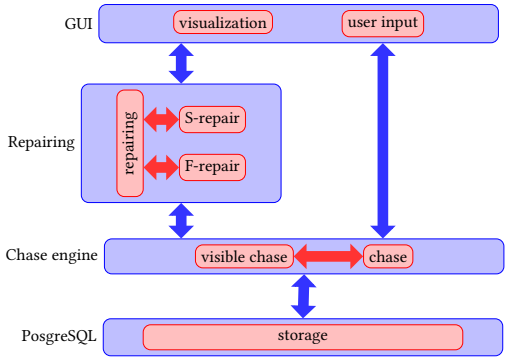


Figure 3: MapRepair architecture.

- **Repairing in progress.** Pressing the dedicated button triggers the generation of a repaired mapping. This will lead to the computation as describe previously, with the use of the chase engine and the repairing module of MapRepair.
- **Final state : repairing complete.** At the end of the repairing process, the output mapping is loaded in the interface. As soon as the mapping is safe w.r.t. the policy views, all variables in the mapping will be highlighted in green in the interface. At this point, the user can choose to manually update this mapping, and MapRepair will show her how the applied modifications affect the disclosed information. The obtained mapping can be exported as an xml file and reused in case by the system.

System implementation and assessment. The core algorithms behind MapRepair are implemented using Java 8. The chase engine uses a PostgreSQL 10.6 database to store tuples during computations. The graphical user interface is coded in JavaScript and allows users to load their views and mappings either by providing a xml file or writing them directly in the interface. The interface also allows to update input views/mappings and the output repaired mapping in order to allow users to gauge the effect of their modifications on information disclosure.

Our demonstration scenarios also show that, despite the complexity of homomorphism detection, our repair algorithm efficiently scales in the presence of mappings including hundreds of s-t tgds. Figure 4 shows several execution times obtained by repairing scenarios (that were generated with iBench [1]) with mappings containing up to 300 s-t tgds, these tgds containing up to five atoms in their bodies. We can easily notice that the runtimes of our repairing algorithms are small and suitable for an interactive demonstration.

3 DEMONSTRATION OVERVIEW

We now discuss the capabilities of MapRepair that we will demonstrate on a set of mapping scenarios.

Showcased scenarios. We borrowed mapping scenarios both from real-life use cases and synthetically generated mappings obtained with the state-of-the-art benchmark iBench[1].

- **Scenario 1** is the simplest scenario, containing 3 policy views and a mapping of 3 s-t tgds, both defined over schemas with relations containing up to 4 attributes. This is the running example used to illustrate our approach in the previous section.

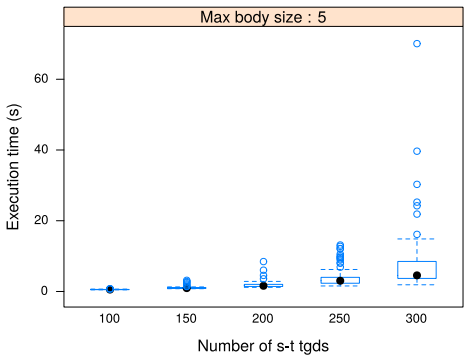


Figure 4: Repairing times for mappings up to 300 tgds.

• **Scenario 2** is a more complex real-life healthcare scenario describing the exchange of data between two hospitals in the UK. This scenario consists of a set of 6 policy views and a mapping of 8 s-t tgds, both defined over schemas with relations containing up to 51 attributes.
• **Scenario 3** is a medium-complexity scenario synthetically generated with `iBench`. This scenario consists of a set of 20 policy views and a mapping of 20 s-t tgds, both defined over schemas with relations containing up to 20 attributes.
• **Scenario 4** is a highly complex scenario synthetically generated with `iBench`. This scenario includes a set of 300 policy views and a mapping of 300 s-t tgds, both defined over schemas with relations containing up to 50 attributes.

Notice that the first two scenarios will allow the attendee to have a better grasp of the underpinnings of our approach due to their reasonable size, whereas the larger third and fourth mapping scenarios will be used to show the scalability of the system (as shown in Figure 4).

Showcased features. Through the scenarios described in the previous section, we will showcase the following:
• **Design of policy views.** We will demonstrate how `MapRepair` can guide a mapping designer through the specification of privacy-preserving views and schema mappings that do not violate the policy views over their source schema.
• **Detection of information leakage in the mappings.** We will demonstrate how `MapRepair` can help to spot mapping violations wrt. policy views, and manually modify the mappings to recover privacy compliance.
• **Automatic reparation of mappings.** Finally, we will show how `MapRepair` can automatically repair a mapping that violate policy views.
• **Performances.** We will showcase the scalability of our system in presence of highly complex scenarios. To illustrate that, we will particularly emphasize the repairing of Scenarios 3 and 4 and their runtimes.

REFERENCES

[1] P.C. Arocena, B. Glavic, R. Ciucanu, and R. Miller. 2015. The iBench integration metadata generator. In *VLDB*.
[2] M. Benedikt, B. Cuenca Grau, and E. Kostylev. 2017. Source Information Disclosure in Ontology-Based Data Integration.. In *AAAI*.
[3] A. Bonifati, U. Comignani, and E. Tsamoura. 2019. Repairing mappings under policy views. *CoRR* arXiv:1903.09242 (2019).
[4] Angela Bonifati, Ioana Ileana, and Michele Linardi. 2016. Functional Dependencies Unleashed for Scalable Data Exchange. In *SSDBM*.
[5] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336, 1 (2005).
[6] G. Miklau and D. Suciu. 2007. A formal analysis of information disclosure in data exchange. *J. Comput. Syst. Sci.* 73, 3 (2007).
[7] A. Nash and A. Deutsch. 2007. Privacy in GLAV Information Integration. In *ICDT*.
[8] L. Sweeney. 2002. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 5 (2002), 557–570.